



$\wedge \vee \neg$



$\binom{n}{k}$



УНИВЕРСИТЕТ ИТМО

ΔΙΑΚΡΙΤΑ ΜΑΘΗΜΑΤΙΚΑ

Дискретная математика

конспект лекций

Константин Чухарев

MMXXVI-MMXXVII



$g \circ f$



$\Gamma \vdash e : \tau$



$Yf = f(Yf)$



Содержание

Введение · · · · ·	13
I Логика и доказательства · · · · ·	16
1.1 Логика высказываний · · · · ·	18
1.2 Методы доказательств · · · · ·	31
1.3 Логика и корректность программ · · · · ·	45
1.4 Итоги главы · · · · ·	47
1.5 Упражнения · · · · ·	48
II Дедукция и системы вывода · · · · ·	51
2.1 Что такое система вывода · · · · ·	53
2.2 Деревья вывода · · · · ·	54
2.3 Системы Гильберта · · · · ·	55
2.4 Натуральная дедукция · · · · ·	57
2.5 Нотация Фитча · · · · ·	59
2.6 Правила для связок · · · · ·	60
2.7 Косвенное доказательство и классические правила · · · · ·	63
2.8 Производные правила · · · · ·	64
2.9 Нормализация · · · · ·	65
2.10 Секвенциальное исчисление Гентцена · · · · ·	67
2.11 Резолюция · · · · ·	70
2.12 Проверка и поиск доказательств · · · · ·	72
2.13 Кванторы в натуральной дедукции · · · · ·	74
2.14 Корректность и полнота · · · · ·	78
2.15 Итоги главы · · · · ·	80
2.16 Упражнения · · · · ·	81
III Логика предикатов · · · · ·	84
3.1 Предикаты · · · · ·	86
3.2 Кванторы · · · · ·	87
3.3 Отрицание кванторов · · · · ·	88
3.4 Несколько кванторов · · · · ·	89
3.5 * Сколемизация · · · · ·	90
3.6 Ограниченные кванторы · · · · ·	91
3.7 Перевод на язык логики предикатов · · · · ·	92
3.8 * Сорта и типизированные переменные · · · · ·	93
3.9 * Силлогистика Аристотеля · · · · ·	94
3.10 Термы и формулы · · · · ·	98
3.11 Семантика: модели и истинность · · · · ·	100
3.12 Полнота и компактность · · · · ·	102
3.13 Итоги главы · · · · ·	105
3.14 Упражнения · · · · ·	106
IV Наивная теория множеств · · · · ·	109
4.1 Множество и его элементы · · · · ·	111
4.2 Равенство и подмножества · · · · ·	113
4.3 Булеан · · · · ·	115

4.4	Операции над множествами	116
4.5	Соответствие операций над множествами и логических связок	117
4.6	Диаграммы Венна	120
4.7	Декартово произведение	121
4.8	Семейства и разбиения	124
4.9	Парадокс Рассела и аксиоматическая теория множеств	125
4.10	Реляционная алгебра и SQL	127
4.11	Типы данных как множества	128
4.12	Битовые маски	130
4.13	Хеширование	131
4.14	* Аксиоматическая теория множеств	132
4.15	* Кардинальные числа	134
4.16	Итоги главы	135
4.17	Упражнения	136
V	Отношения	139
5.1	Бинарные отношения	141
5.2	Отношение эквивалентности	152
5.3	Теорема о разбиении	156
5.4	Примеры отношений эквивалентности	157
5.5	Приложения отношений эквивалентности	157
5.6	Минимизация ДКА через отношение неразличимости	158
5.7	Частичные порядки	159
5.8	Принцип Дирихле	160
5.9	* Ультраметрика и иерархическая кластеризация	161
5.10	Итоги главы	168
5.11	Упражнения	169
VI	Функции	172
6.1	Определения и базовые понятия	174
6.2	Инъекция	179
6.3	Сюръекция	181
6.4	Биекция	183
6.5	Критерии мощности	185
6.6	Композиция функций	186
6.7	Алгебра образов и прообразов	188
6.8	Асимптотические обозначения	190
6.9	Характеристическая функция	195
6.10	Пол, потолок и нотация Айверсона	196
6.11	Лямбда-нотация	197
6.12	* Каррирование и λ -исчисление	197
6.13	Итоги главы	198
6.14	Упражнения	199
VII	Мощность и бесконечность	203
7.1	Равномощность	204
7.2	Отель Гильберта	206
7.3	Счётные и несчётные множества	208
7.4	Теорема Кантора	216
7.5	Теорема Кантора–Бернштейна–Шрёдера	218

7.6	Кардинальные числа и арифметика	221
7.7	Две иерархии: алефы и беты	221
7.8	* Недостижимые кардиналы	225
7.9	Итоги главы	226
7.10	Упражнения	227
VIII	Отношения порядка	230
8.1	Частичный порядок	232
8.2	Диаграммы Хассе	236
8.3	* Ширина порядка и теорема Дилуорса	238
8.4	Экстремальные элементы	240
8.5	Грани, супремум, инфимум	242
8.6	Решётки	244
8.7	Топологическая сортировка	251
8.8	Лексикографический порядок	253
8.9	Приложения отношений порядка	258
8.10	Неподвижные точки в решётках	260
8.11	Первый взгляд на абстрактную интерпретацию	262
8.12	Итоги главы	263
8.13	Упражнения	264
IX	Графы	267
9.1	Основные определения	269
9.2	* Реализуемые последовательности степеней: Хавел–Хакими	275
9.3	Пути, циклы, связность	278
9.4	Обходы графа	281
9.5	Топологическая сортировка и сильная связность	285
9.6	Деревья	289
9.7	Минимальные остовные деревья	294
9.8	Эйлеровы графы	298
9.9	* Произвольно вычерчиваемые графы	301
9.10	Гамильтоновы графы	302
9.11	Кратчайшие пути	305
9.12	Обходы с оптимизацией: почтальон и коммивояжёр	310
9.13	Двудольные графы	311
9.14	Сетевые потоки	315
9.15	Непересекающиеся пути и надёжность сети	324
9.16	Паросочетания и вершинное покрытие	326
9.17	Планарность	329
9.18	Раскраска графов	333
9.19	Применения графов	337
9.20	* Случайные графы	339
9.21	Итоги главы	340
9.22	Упражнения	341
X	Булева алгебра	345
10.1	Булевы функции	347
10.2	* Клоны и замыкание	358
10.3	Литералы, минтермы, макстермы	361
10.4	ДНФ и КНФ	362

10.5	СДНФ, АНФ и смена базиса	365
10.6	Минимизация	366
10.7	Полином Жегалкина	370
10.8	Диаграммы решений (BDD)	374
10.9	* Теорема Стоуна о представлении	379
10.10	Итоги главы	384
10.11	Упражнения	385
XI	Схемы	389
11.1	Системы счисления	391
11.2	Логические вентили и схемы	393
11.3	Сумматоры	399
11.4	Код Грея	407
11.5	Верификация схем через SAT	409
11.6	Схемная сложность	411
11.7	* R/poly и монотонные схемы	414
11.8	Итоги главы	416
11.9	Упражнения	417
XII	Алгебраические структуры	421
12.1	Операция	422
12.2	Полугруппа и моноид	426
12.3	Группа	428
12.4	Подгруппы и теорема Лагранжа	434
12.5	Действие группы на множестве	438
12.6	Кольцо	442
12.7	Поле	445
12.8	Конечные поля	447
12.9	Гомоморфизм и изоморфизм	451
12.10	Векторные пространства над полем	456
12.11	* Полурешётки и полукольца	460
12.12	Приложения	461
12.13	Связь с кодами	469
12.14	Итоги главы	469
12.15	Упражнения	471
XIII	Коды и информация	473
13.1	Расстояние Хэмминга и геометрия кода	475
13.2	Код Хэмминга (7, 4)	478
13.3	Расширенный код Хэмминга (8, 4, 4)	482
13.4	Линейные коды и $GF(2)$	483
13.5	Конечные поля $GF(p^m)$	486
13.6	Граница Шеннона	488
13.7	Код Хаффмана (сжатие без потерь)	491
13.8	* Словарное сжатие: LZ77 и LZ78	494
13.9	Циклические коды	496
13.10	Коды Рида–Соломона	498
13.11	Коды и комбинаторика	501
13.12	* Обобщённые RS-коды и связь с БЧХ	503
13.13	* Свёрточные коды и алгоритм Витерби	504

13.14	* Пакетная передача и протоколы	505
13.15	Итоги главы	506
13.16	Упражнения	507
XIV	SAT	511
14.1	Задача SAT	513
14.2	Резолюция	514
14.3	Кодирование задач в SAT	521
14.4	От DPLL к CDCL	526
14.5	Применения SAT-решателей	530
14.6	Локальный поиск: WalkSAT	531
14.7	Граф импликаций: от 2-SAT до CDCL	533
14.8	SAT и NP	533
14.9	Итоги главы	537
14.10	Упражнения	538
XV	SMT	541
15.1	Многосортная логика первого порядка	543
15.2	Теории первого порядка	546
15.3	Какие теории решает SMT	555
15.4	Разностная логика	558
15.5	Равенство с неинтерпретируемыми функциями	564
15.6	Линейная арифметика	567
15.7	Битовые векторы	571
15.8	Теория массивов	572
15.9	Theory-солверы и DPLL(T)	577
15.10	Комбинация теорий (Nelson-Oppen)	580
15.11	SMT-LIB и солверы	583
15.12	Итоги главы	585
15.13	Упражнения	586
XVI	Логическое программирование	589
16.1	Термы	590
16.2	Подстановки	591
16.3	Унификация	593
16.4	Программы из фактов и правил	595
16.5	SLD-резолюция	598
16.6	Поиск и бэктрекинг	602
16.7	Списки	603
16.8	Рекурсия и терминация	604
16.9	* Арифметика: $is/2$	605
16.10	* Отрицание как провал и cut	606
16.11	Итоги главы	607
16.12	Упражнения	608
XVII	Матроиды	611
17.1	Задача выбора и жадный алгоритм	612
17.2	Три задачи, где жадный алгоритм оптимален	614
17.3	Свойства допустимых семейств	615
17.4	Определение матроида	617

17.5	Примеры матроидов	619
17.6	Корректность жадного алгоритма	622
17.7	Обратное утверждение	624
17.8	Матроиды в приложениях	625
17.9	* Дальше в матроиды	628
17.10	Итоги главы	629
17.11	Упражнения	630
XVIII	Комбинаторика	633
18.1	Правила подсчёта	635
18.2	Суммы и замкнутые формы	639
18.3	Перестановки, размещения, сочетания	640
18.4	Биномиальные коэффициенты и тождества	647
18.5	Принцип включений-исключений	650
18.6	Рекуррентные соотношения	653
18.7	Числа Каталана	656
18.8	Код Прюфера	658
18.9	Числа Стирлинга и Белла	659
18.10	Двенадцатеричный путь	662
18.11	Лемма Бёрнсайда и теория Пойа	669
18.12	* Коды Грея и комбинаторная генерация	672
18.13	Начала теории Рамсея	674
18.14	Комбинаторные игры (Ним)	676
18.15	Итоги главы	678
18.16	Упражнения	679
XIX	Теория чисел и криптография	683
19.1	Взлом как проверка безопасности	685
19.2	Делимость и алгоритм Евклида	686
19.3	Модулярная арифметика	688
19.4	Малая теорема Ферма и функция Эйлера	689
19.5	Тесты простоты	691
19.6	Китайская теорема об остатках	693
19.7	Шифрование с открытым ключом (RSA)	694
19.8	Протокол Диффи–Хеллмана	698
19.9	Эллиптические кривые и криптография	702
19.10	Итоги главы	705
19.11	Упражнения	706
XX	Дискретная вероятность	709
20.1	Вероятностное пространство	711
20.2	Условная вероятность и независимость	713
20.3	Формула Байеса	716
20.4	Случайные величины	719
20.5	Линейность матожидания	721
20.6	Основные распределения	723
20.7	Неравенства концентрации	725
20.8	* Неравенство Чернова	727
20.9	Центральная предельная теорема	729
20.10	Парадокс дней рождения	732

20.11	Применения в алгоритмах	735
20.12	Вероятностный метод	736
20.13	Случайные графы	737
20.14	* Марковские цепи	738
20.15	* Энтропийный метод в комбинаторике	740
20.16	* Аксиомы Колмогорова	741
20.17	Итоги главы	746
20.18	Упражнения	747
XXI	Производящие функции	751
21.1	Обычные производящие функции	752
21.2	Экспоненциальные производящие функции	765
21.3	Применения производящих функций	769
21.4	* Комбинаторные виды	773
21.5	* Формула Кэли	778
21.6	* Асимптотика коэффициентов	779
21.7	Итоги главы	780
21.8	Упражнения	782
XXII	Конструкции чисел	785
22.1	Аксиомы Пеано	786
22.2	Алгебры Дедекинда	788
22.3	Натуральные числа по фон Нейману	792
22.4	Построение целых чисел	795
22.5	Построение рациональных чисел	796
22.6	Сечения Дедекинда	798
22.7	Что утверждает аксиома выбора	802
22.8	Эквиваленты аксиомы выбора	803
22.9	Парадокс Банаха–Тарского	804
22.10	Множество Витали	806
22.11	Континуум-гипотеза	810
22.12	Недостижимые кардиналы и аксиомы больших кардиналов	813
22.13	Итоги главы	815
22.14	Упражнения	816
XXIII	Конечные автоматы и регулярные языки	819
23.1	Алфавит, слова и формальные языки	821
23.2	Детерминированные конечные автоматы	822
23.3	Недетерминированные конечные автоматы	826
23.4	Конструкция подмножеств	829
23.5	Регулярные выражения	833
23.6	От выражения к автомату	838
23.7	Свойства замкнутости	839
23.8	Лемма о накачке	842
23.9	Теорема Майхилла–Нерода	845
23.10	Минимизация ДКА	849
23.11	* Минимизация ДКА (алгоритм Хопкрофта)	851
23.12	* Минимализация Бржозовски	852
23.13	* Автомат Кнута–Морриса–Пратта	855
23.14	* Регулярные выражения на практике: ДКА против backtracking	859

23.15	Об автомате можно узнать всё · · · · ·	861
23.16	Верификация программ · · · · ·	862
23.17	Итоги главы · · · · ·	863
23.18	Упражнения · · · · ·	864
XXIV	Контекстно-свободные языки · · · · ·	867
24.1	Контекстно-свободные грамматики · · · · ·	869
24.2	Автоматы с магазинной памятью · · · · ·	872
24.3	Структура контекстно-свободных языков · · · · ·	877
24.4	Нормальная форма Хомского · · · · ·	881
24.5	Замкнутость КС-языков · · · · ·	883
24.6	От грамматики к парсеру · · · · ·	885
24.7	Контекстно-зависимые языки · · · · ·	888
24.8	Иерархия Хомского · · · · ·	890
24.9	Итоги главы · · · · ·	890
24.10	Упражнения · · · · ·	891
XXV	Машины Тьюринга · · · · ·	896
25.1	Определение машины Тьюринга · · · · ·	898
25.2	Примеры машин Тьюринга · · · · ·	902
25.3	Стандартные приёмы программирования · · · · ·	906
25.4	Устойчивость и варианты · · · · ·	907
25.5	Тезис Чёрча–Тьюринга · · · · ·	909
25.6	Универсальная машина Тьюринга · · · · ·	910
25.7	Итоги главы · · · · ·	913
25.8	Упражнения · · · · ·	914
XXVI	Разрешимость и неразрешимость · · · · ·	916
26.1	Разрешимость · · · · ·	919
26.2	Проблема остановки · · · · ·	920
26.3	Сведения · · · · ·	923
26.4	Теорема Райса · · · · ·	927
26.5	За пределами машин Тьюринга · · · · ·	928
26.6	Арифметическая иерархия · · · · ·	929
26.7	Теоремы Гёделя о неполноте · · · · ·	932
26.8	Функция занятого бобра · · · · ·	934
26.9	* Теорема Райса и статический анализ программ · · · · ·	937
26.10	* Десятая проблема Гильберта · · · · ·	938
26.11	* Арифметика Пресбургера · · · · ·	940
26.12	Итоги главы · · · · ·	942
26.13	Упражнения · · · · ·	944
XXVII	Бестиповое λ-исчисление · · · · ·	947
27.1	Три способа строить термы · · · · ·	949
27.2	β -редукция · · · · ·	954
27.3	Программирование в λ -исчислении · · · · ·	959
27.4	Неподвижные точки и рекурсия · · · · ·	962
27.5	λ -исчисление и вычислимость · · · · ·	964
27.6	* Можно ли обойтись без переменных? · · · · ·	966
27.7	Итоги главы · · · · ·	967

27.8	Упражнения	968
XXVIII	Теория типов	972
28.1	Простые типы	975
28.2	Правила типизации	976
28.3	Что можно и что нельзя типизировать	979
28.4	Свойства типизированного λ -исчисления	980
28.5	Соответствие Карри–Ховарда	982
28.6	За пределы $\lambda \rightarrow$	986
28.7	Итоги главы	990
28.8	Упражнения	991
XXIX	За пределами натуральных чисел	994
29.1	Теорема компактности и нестандартные модели	996
29.2	Ординалы за пределами ω	1000
29.3	Сюрреальные числа	1007
29.4	Нестандартный анализ	1011
29.5	Итоги главы	1014
29.6	Упражнения	1015
XXX	Сложность и NP-полнота	1018
30.1	Класс P	1021
30.2	Класс NP	1022
30.3	Полиномиальные сведения и NP-полнота	1025
30.4	Стратегии решения NP-трудных задач	1033
30.5	Параметризованная сложность	1036
30.6	Ядро задачи	1038
30.7	Классы сложности за пределами NP	1041
30.8	* Полиномиальная иерархия	1049
30.9	Итоги главы	1050
30.10	Упражнения	1051
XXXI	Абстрактная интерпретация	1054
31.1	Знаки вместо чисел	1056
31.2	Вычисления в абстрактном мире	1059
31.3	Домены на практике	1061
31.4	Неподвижная точка и расходямость	1064
31.5	Как заставить итерацию сойтись (widening)	1067
31.6	Цена аппроксимации	1069
31.7	Применения абстрактной интерпретации	1070
31.8	Символьное исполнение	1071
31.9	Итоги главы	1072
31.10	Упражнения	1073
XXXII	Формальная верификация	1077
32.1	Тройки и правила вывода	1079
32.2	Инварианты и тотальная корректность	1087
32.3	От аннотаций к SMT-солверу	1091
32.4	Итоги главы	1096
32.5	Упражнения	1097

XXXIII	Модальная логика	1100
33.1	Фреймы и операторы	1102
33.2	Свойства отношения и аксиомы	1104
33.3	Модальные системы и куб модальностей	1106
33.4	Задачи модальной логики	1110
33.5	Знание, обязанность, убеждение	1111
33.6	Время как модальность	1112
33.7	Алгоритмы model checking	1116
33.8	Model checking на практике	1118
33.9	Итоги главы	1120
33.10	Упражнения	1121
XXXIV	Интуиционистская логика	1124
34.1	Семантика интуиционизма	1126
34.2	* Промежуточные логики	1138
34.3	* Линейная логика	1139
34.4	Компактность и существование без построения	1140
34.5	* Доказательные ассистенты (Coq и Lean)	1141
34.6	Итоги главы	1142
34.7	Упражнения	1143
XXXV	Нечёткие множества	1146
35.1	Функция принадлежности	1148
35.2	Нечёткие числа	1152
35.3	Операции над нечёткими множествами	1153
35.4	Нечёткая логика	1158
35.5	Применения и связь с остальной книгой	1161
35.6	* Интерполяция и схема Сугено	1162
35.7	Итоги главы	1163
35.8	Упражнения	1165
XXXVI	Теория категорий	1168
36.1	Категория и её стрелки	1170
36.2	Стрелки вместо элементов	1176
36.3	Функторы и перенос структуры	1179
36.4	Естественные преобразования	1183
36.5	Универсальные свойства	1185
36.6	Пределы и копределы	1188
36.7	Лемма Йонеды	1190
36.8	Сопряжённые функторы	1192
36.9	Монады и композиция эффектов	1195
36.10	* Стринг-диаграммы	1198
36.11	* Алгебры и коалгебры	1199
36.12	* Метрики, вероятности и нечёткость	1200
36.13	* Топосы и внутренняя логика	1201
36.14	Итоги главы	1202
36.15	Упражнения	1203

Закключение · · · · ·	1206
-----------------------	------

Указания к упражнениям · · · · ·	1208
----------------------------------	------

36.15 Глава 1. Логика и доказательства · · · · ·	1208
36.15 Глава 2. Дедукция · · · · ·	1208
36.15 Глава 3. Логика предикатов · · · · ·	1209
36.15 Глава 4. Множества · · · · ·	1209
36.15 Глава 5. Отношения · · · · ·	1209
36.15 Глава 6. Функции · · · · ·	1210
36.15 Глава 7. Кардиналы · · · · ·	1210
36.15 Глава 8. Отношения порядка · · · · ·	1211
36.15 Глава 9. Графы · · · · ·	1212
36.15 Глава 10. Булева алгебра · · · · ·	1212
36.15 Глава 11. Схемы · · · · ·	1213
36.15 Глава 12. Алгебраические структуры · · · · ·	1213
36.15 Глава 13. Коды · · · · ·	1214
36.15 Глава 14 SAT · · · · ·	1215
36.15 Глава 15. SMT · · · · ·	1215
36.15 Глава 15 SMT · · · · ·	1215
36.15 Глава 16. Логическое программирование · · · · ·	1216
36.15 Глава 17. Матроиды · · · · ·	1216
36.15 Глава 18. Комбинаторика · · · · ·	1216
36.15 Глава 19. Теория чисел и криптография · · · · ·	1217
36.15 Глава 20. Вероятность · · · · ·	1217
36.15 Глава 21. Производящие функции · · · · ·	1218
36.15 Глава 22. Конструкции чисел · · · · ·	1218
36.15 Глава 23. Конечные автоматы и регулярные языки · · · · ·	1219
36.15 Глава 24. Контекстно-свободные языки · · · · ·	1220
36.15 Глава 25. Машины Тьюринга · · · · ·	1220
36.15 Глава 26. Разрешимость и неразрешимость · · · · ·	1221
36.15 Глава 27. Лямбда-исчисление · · · · ·	1221
36.15 Глава 28. Теория типов · · · · ·	1222
36.15 Глава 29. За пределами натуральных чисел · · · · ·	1222
36.15 Глава 30. Сложность вычислений · · · · ·	1223
36.15 Глава 31. Абстрактная интерпретация · · · · ·	1223
36.15 Глава 32. Формальная верификация · · · · ·	1224
36.15 Глава 33. Модальная и темпоральная логика · · · · ·	1224
36.15 Глава 34. Интуиционизм · · · · ·	1225
36.15 Глава 35. Нечёткие множества · · · · ·	1225
36.15 Глава 36. Категории · · · · ·	1225

Словарь терминов · · · · ·	1227
----------------------------	------

36.16 Язык дискретных структур · · · · ·	1227
36.17 Алгебра и инженерия · · · · ·	1232
36.18 Счёт, случайность и бесконечность · · · · ·	1239
36.19 Вычислимость и её границы · · · · ·	1243

Введение

Мир, каким его описывают вычисление и физика, — *дискретный*. Всё, с чем работает вычисление, распадается на отдельные элементы: картинка — на пиксели, звук — на отсчёты, текст — на символы, программа — на команды. И природа устроена так же: вещество — из атомов, свет — из квантов, энергия меняется порциями. Интуиция рисует мир иначе: прямая на чертеже непрерывна, время течёт без делений, поверхность сплошная. Но и с *непрерывным* на практике обращаются как с *дискретным*: кривую строят точками, интеграл заменяют суммой по шагам, дифференциальное уравнение решают сеткой, синус — конечным рядом.

Замечание Почему дискретность — это точность

Непрерывное описывать трудно: между любыми двумя точками есть третья, и чтобы задать объект, нужны пределы. *Дискретный* объект устроен проще: каждый элемент отделён от соседних, и про каждый можно сказать, «входит он в рассматриваемое множество или нет». Отсюда и сила метода: точное описание заменяет *непрерывное* набором элементов и делает рассуждение проверяемым. Непрерывное — часто лишь удобная модель: сплошная среда физики оказывается сгущением атомов.

Дискретность при этом не значит *конечность*. Натуральные числа бесконечны, но дискретны: между соседними нет ничего, каждый шаг отделён от следующего. Дискретная математика начинается с этой пары: отдельные элементы — и бесконечное, которое отдельности не отменяет.

Пифагор учил, что всё есть число, и дискретная математика принимает это всерьёз. В 1874 году Георг Кантор показал, что бесконечности бывают разного размера: вещественных чисел больше, чем натуральных, $|\mathbb{N}| < |\mathbb{R}|$. Часть современников отвергла открытие: бесконечность, которую нельзя занумеровать, казалась противоречием самой мысли. *Счётное* множество можно занумеровать, *несчётное* — нет. Дальше идут *кардиналы* — меры бесконечности, ответ на вопрос «сколько». Рядом — *ординалы*, ответ на вопрос «который по счёту»: счёт продолжается и за всеми конечными номерами, и первое из этих чисел — ω .

Пример Отель Гильберта

Представьте отель с бесконечным числом номеров, и все они заняты. Приезжает новый гость. Управляющий просит каждого постояльца переехать в следующий номер, и первый номер становится свободным. Полный конечный отель нового гостя не примет: мест нет. Полный бесконечный — примет: свободный номер найдётся, и бесконечность от этого не изменится.

Конечное при этом умеет быть *обманчиво большим*: перестановок шестидесяти предметов больше, чем атомов в наблюдаемой Вселенной — это $60! \approx 8 \cdot 10^{81}$ вариантов. Задача с конечным числом вариантов может не иметь быстрого способа решения: маршрут, проходящий все города, упирается в *перебор*, который не закончится за жизнь. Существование можно утверждать и не предъявляя объекта: Эрдёшу хватало показать, что случайный объект обладает нужным свойством с положительной вероятностью.

От перебора спасает *структура*. Граф, отношение порядка, алгебраическая операция — это способы увидеть в массе отдельных случаев закон, позволяющий не перебирать. В упорядоченном множестве не нужно сравнивать каждый элемент с каждым.

Эйлер открыл теорию графов на задаче о семи мостах Кёнигсберга. Буль превратил логику рассуждений в алгебру нулей и единиц. Лейбниц мечтал о языке, в котором спор разрешался бы вычислением.

Пример Структура вместо перебора

На карте дорог кратчайший путь между двумя городами находится быстро, хотя число возможных маршрутов огромно: структура графа позволяет отбрасывать целые классы путей, не просматривая их по одному. А вот маршрут, обходящий все города, так быстро не ищется — это отдельная, трудная задача.

Объекты — не всё: важно ещё, *как* о них рассуждать. Естественный язык для рассуждений плохо приспособлен: «или» означает то одно, то другое, «если» меняет смысл от контекста. Математика долго полагалась на интуицию и авторитет, но консенсус ошибается: убедительное для сообщества доказательство может оказаться с дырой.

Замечание Почему формальный метод?

Решение — формализация: для рассуждений строится искусственный язык, в котором каждое правило задано явно. Тогда доказательство проверяется механически — шаг за шагом, по точным правилам, без опоры на авторитет. Корректность рассуждения перестаёт быть вопросом убедительности и становится вопросом формы: каждый шаг либо разрешён, либо нет. *Логика* — первый такой язык, и с неё курс начинается.

Проверить — не значит *понять*: чтобы знание стало своим, доказательство нужно продумать самому, от первого шага до последнего.

Из механики проверки вырастает *вычисление*, а из вычисления — алгоритмы, программы, автоматы. Вычисление измеряет задачи шагами, и у этого измерения есть *цена*: время, память, а иногда и случайность. Сколько шагов нужно, чтобы решить задачу, — отдельный вопрос, и на него отвечает *теория сложности*.

ИСТОРИЯ Границы алгоритмического метода

В 1931 году Курт Гёдель доказал: в любой непротиворечивой и достаточно выразительной системе аксиом найдётся утверждение, которое она не может ни доказать, ни опровергнуть. В 1936 году Алан Тьюринг показал, что вопрос «остановится ли программа» не имеет алгоритма, который отвечал бы на него всегда. Эти границы следуют из свойств самих задач, и увидеть их — значит узнать предел алгоритмического метода.

Рядом с точным дискретным живёт то, что строгому описанию сопротивляется: *случайность* и *нечёткость*. Для случайности есть вероятности, для нечёткости — *степени принадлежности*. Принадлежность бывает *частичной*: между «да» и «нет» лежат промежуточные степени.

Точность достигается явными решениями: что считать элементом, где провести разрез, какие определения принять. Это язык со своими возможностями и границами, и курс — о том, как им владеть.

Инструменты этого языка строятся по порядку: сначала идут *точное описание* и *точное рассуждение*, за ними счёт, потом вычислимость и её цена, а в конце — то, что сопротивляется точному описанию, и язык, собирающий всё пройденное в одну рамку. По пути накопится и общий словарь программиста, инженера и теоретика: биты, графы, автоматы, выполнимость, матроиды, сложность, коды. Но словарь — не цель: по итогу останется умение мыслить точно, рассуждать строго, считать и понимать, где точность недостижима.

Книга состоит из некоторого количества глав, разбитых на четыре блока. Определения всех глав собраны в словаре терминов в конце книги. Главы разные: одни читаются сами по себе, другие опираются на соседние, а финальные собирают в одно целое всё, что было до них.

Ответы к упражнениям не приводятся: упражнение имеет смысл, только когда оно проделано на бумаге. К задачам со звёздочкой — «на подумать» — приложены указания после заключения. Разделы, отмеченные звёздочкой, содержат дополнительный материал и могут быть пропущены при первом чтении.

I

Логика и доказательства

Что значит «доказать утверждение»?

Рассмотрим утверждение: «сумма двух чётных чисел всегда чётна». Истинно ли оно? Проверим: $2 + 4 = 6$, $10 + 6 = 16$, $100 + 48 = 148$ — чётно.

Сколько таких проверок достаточно, чтобы считать утверждение доказанным? Ответ: никакое конечное число проверок не даёт доказательства — миллион удачных примеров всё ещё не доказательство, ведь следующая пара чисел может оказаться исключением. Пример описывает одну конкретную ситуацию, а утверждение обещает истинность во *всех* ситуациях сразу.

Доказательство — рассуждение, которое покрывает все случаи разом и не может быть опровергнуто новой проверкой. Отличать доказательство от набора примеров — первый шаг.

Чтобы вести такие рассуждения, нужен точный язык. Естественный язык для этого плохо приспособлен: слово «или» в повседневной речи означает то одно, то другое, «если» меняет смысл от контекста к контексту. Математика столетиями полагалась на интуицию и авторитет: доказательством считалось рассуждение, которое убеждает сообщество экспертов. Критерием истинности было согласие, а согласие ошибается. В 1879 году Альфред Кемпе опубликовал доказательство теоремы о четырёх красках, и одиннадцать лет спустя Перси Хивуд нашёл в нём неустранимую ошибку.

Решение — формализация. Для рассуждений строится искусственный язык, в котором каждое правило задано явно и проверяется механически. Корректность рассуждения перестаёт быть вопросом убедительности и становится вопросом формы: каждый шаг либо разрешён, либо нет. Проверка доказательства сводится к вычислению — столь же однозначному, как арифметическая операция.

Словарь такого языка — высказывания и логические связки: «и», «или», «не», «если... то». Инструмент — таблицы истинности, вычисляющие значение любого составного утверждения. Приёмы — методы доказательств: прямое рассуждение, рассуждение от противного, разбор случаев, индукция. Вместе они дадут ответ на исходный вопрос: что делает рассуждение доказательством и как это проверить.

Программиста эти приёмы касаются напрямую. Спецификация функции — утверждение о связи входа и выхода. Инвариант цикла — утверждение, которое обязано держаться на каждой итерации. Вопрос «корректна ли программа» — это вопрос об истинности утверждения обо всех возможных выполнениях. Падающий тест

— *контрпример* к утверждению о корректности. Доказательство показывает, что контрпримеров нет вовсе. Различие между проверкой на конечном числе входов и доказательством для всех входов — то же, что между примером и доказательством, и оно пронизывает весь курс.

Как записать рассуждение, чтобы его правильность проверялась вычислением?

ИСТОРИЯ Мечта Лейбница о логическом исчислении

Аристотель в «Органоне» (IV век до н.э.) заложил первую формальную теорию рассуждений — силлогистику, классифицировав правильные умозаключения по фигурам и модусам. Стоики, прежде всего Хрисипп (III век до н.э.), развили логику высказываний — анализ составных утверждений через связки. Но античная логика оставалась философским инструментом: она описывала, как люди рассуждают, а не как вычислять истинность механически.

Готфрид Вильгельм Лейбниц (1646–1716) — пожалуй, первый мыслитель, всерьёз вообразивший, что рассуждение можно свести к вычислению. Он называл эту идею *calculus ratiocinator* — «исчисление рассуждений» — и *characteristica universalis* — «универсальная характеристика», формальный язык для записи любых понятий. Лейбниц надеялся, что однажды философские споры можно будет разрешать вычислением: спорщики записывают аргументы на этом языке и вычисляют, кто прав.

Замысел не был реализован — математический аппарат XVII века был для этого недостаточен. Через двести лет Джордж Буль алгебраизировал логику: конъюнкция стала умножением, дизъюнкция — сложением.¹ Готлоб Фреге ввёл логику предикатов с кванторами — первую формальную систему, достаточно выразительную для записи произвольных математических утверждений.² Его двумерная нотация не прижилась, но логическая конструкция стала стандартом. В 1920-е годы Давид Гильберт выдвинул программу полной формализации математики: все утверждения записываются в формальном языке, а доказательства сводятся к синтаксическим манипуляциям символами по фиксированным правилам. Программа столкнулась с ограничениями — теоремами Гёделя о неполноте (1931) — но породила современную математическую логику и теорию доказательств. Современные автоматические доказатели теорем (Coq, Lean, Isabelle) — прямые наследники *calculus ratiocinator*: они проверяют доказательства механически, без обращения к смыслу. Лейбницево *Calculemus* — «давайте вычислим» — стало неофициальным девизом формальной верификации. Путь от аристотелевых силлогизмов до этого девиза занял два тысячелетия.

¹G. Boole. «The Laws of Thought». 1854.

²G. Frege. «Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens». 1879.

ОБЗОР ГЛАВЫ

Мы построим язык, на котором рассуждение проверяется вычислением. Из простых утверждений соберём составные, таблицы истинности научат нас вычислять их значение, и мы увидим, почему из пяти привычных связок достаточно двух. Затем мы перейдём к методам доказательств: прямому рассуждению, контрапозиции, рассуждению от противного, разбору случаев и индукции. Индукции мы посвятим отдельное внимание, доведя её от свойств натуральных чисел до корректности циклов и рекурсивных функций. В конце главы мы применим логику к программам: тройки Хоара запишут корректность как формулу, и вы осознаете, чем доказательство отличается от тестирования.

После этой главы вы сможете:

1. вычислять значение формулы таблицей истинности и строить таблицу по любому выражению;
2. доказывать утверждения о числах и структурах — прямо, от противного, контрапозицией, разбором случаев;
3. проводить индукцию по натуральным числам и применять её к корректности циклов;
4. распознавать ошибки рассуждения — утверждение консеквента, отрицание антецедента — и объяснять, почему они не доказательства;
5. записывать корректность простой программы тройкой Хоара и понимать, чем доказательство отличается от набора тестов.

§ 1.1 Логика высказываний

Первый слой формального языка — логика высказываний: правила, по которым истинность составных утверждений вычисляется из истинности их частей. Сначала вводятся атомы и связки, затем — вычисление истинности по таблицам.

1.1.1 Высказывания и связки

Высказывание (или *утверждение*) — это повествовательное предложение, которое либо истинно, либо ложно, но не то и другое одновременно. Вопросы, команды и самореференциальные парадоксы исключаются. Допустимы только замкнутые, однозначные утверждения.

ОПРЕДЕЛЕНИЕ 1.1 Высказывание

Высказывание — это повествовательное утверждение с определённым истинностным значением: истина (**T**) или ложь (**F**).

Пример Высказывания и не-высказывания

- « $2 + 2 = 4$ » — высказывание (истинное).
- « $2 + 2 = 5$ » — высказывание (ложное).
- «Который час?» — не высказывание (вопрос).
- « $x > 5$ » — не высказывание: истинность зависит от значения переменной. Это предикат — утверждение с переменной, его мы разберём в главе 3.

Составные высказывания строятся из атомарных высказываний с помощью логических связок. *Атомарное высказывание* — это высказывание, которое не раскладывается на более мелкие. В логике высказываний атомы обозначаются переменными p, q, r . Стандартные связки, перечисленные в порядке убывания силы связывания. Пусть p и q — высказывания.

ОПРЕДЕЛЕНИЕ 1.2 Отрицание

Отрицание высказывания p — высказывание $\neg p$ («не p »), истинное, когда p ложно, и ложное, когда p истинно.

ОПРЕДЕЛЕНИЕ 1.3 Конъюнкция

Конъюнкция высказываний p и q — высказывание $p \wedge q$ (« p и q »), истинное, когда истинны оба.

ОПРЕДЕЛЕНИЕ 1.4 Дизъюнкция

Дизъюнкция высказываний p и q — высказывание $p \vee q$ (« p или q »), истинное, когда истинно хотя бы одно из них.

ОПРЕДЕЛЕНИЕ 1.5 Импликация

Импликация высказываний p и q — высказывание $p \rightarrow q$ («если p , то q »), ложное, когда p истинно, а q ложно, и истинное во всех остальных случаях.

ОПРЕДЕЛЕНИЕ 1.6 Эквивалентность

Эквивалентность высказываний p и q — высказывание $p \leftrightarrow q$ (« p тогда и только тогда, когда q »), истинное, когда p и q имеют одно и то же истинностное значение.

Знак импликации пережил несколько нотаций: в «Principia Mathematica» Уайтхеда и Рассела она обозначалась «подковой» $\supset$, и такая запись встречается в старых текстах до сих пор.³

³Выбор знака опирался на аналогию с классами: « p влечёт q » означает, что класс ситуаций, где истинно p , содержится в классе, где истинно q . Но $\supset$ — тот же знак, которым теория множеств

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

В быту «или» часто означает выбор: к стейку подаётся салат или суп — в цену входит что-то одно. Логическая дизъюнкция $p \vee q$ включающая: она истинна и тогда, когда истинны оба члена. Гостиница подсказывает, когда «или» понимается именно так: бесплатный Wi-Fi или парковка — гость вправе воспользоваться и тем, и другим.

Пример Вычисление истинности

Пусть p — «сегодня дождь» (истинно). Пусть q — «я взял зонт» (ложно). Тогда:

- $\neg p$ ложно.
- $p \wedge q$ ложно (зонт не взят, несмотря на дождь).
- $p \vee q$ истинно (дождь идёт).
- $p \rightarrow q$ ложно (дождь идёт, а зонт не взят — обещание нарушено).
- $p \leftrightarrow q$ ложно. Значения p и q различны.

В импликации $p \rightarrow q$ левая часть — это *антецедент* (посылка). Правая часть — это *консеквент* (закключение). В *таблице истинности* для импликации есть строка, требующая пояснения: $F \rightarrow T$ истинно. Сравнение с нарушенным обещанием помогает: «Если пойдёт дождь, я возьму зонт» ложно ровно в одной строке — дождь идёт, а зонт не взят. В остальных трёх строках обещание не нарушено:

- дождь идёт и зонт взят.
- дождя нет, а зонт взят.
- дождя нет и зонта нет.

Строка $F \rightarrow T$ (дождя нет, зонт взят) — одна из них, поэтому импликация истинна. Точнее, в двух нижних строках обещание не вступило в силу: условие, при котором оно должно было исполниться, не наступило. Нельзя нарушить договор, который не был активирован.

Примечание Парадокс материальной импликации

Импликация не требует причинной связи между антецедентом и консеквентом. Истинностное значение определяется исключительно таблицей: $F \rightarrow T$, $F \rightarrow F$ истинны. Из этого следует контр-интуитивное свойство: импликация с ложной посылкой истинна при любом заключении — вакуумная истинность.

Альтернативные прочтения «если... то» существуют. Строгая импликация Льюиса объявляет, что p строго влечёт q , когда *невозможно*, чтобы p было истинно,

записывает «содержит», то есть включение в обратную сторону, отсюда постоянная путаница. Современная стрелка $\rightarrow$ свободна от этой двусмысленности.

⁴C. I. Lewis. «Implication and the Algebra of Logic». Mind, 1912.

а q ложно.⁴ Релевантная логика Андерсона и Белнапа требует, чтобы посылка действительно использовалась в выводе заключения.⁵ Обе системы ближе к интуитивному «если... то» и обе жертвуют главным достоинством материальной импликации: ни одна не допускает механической проверки формулы конечным перебором строк таблицы истинности. Модальные системы, выросшие из строгой импликации, разбираются в главе 33.

Пример Противоречие доказывает всё

Теория, в которой доказаны $\varphi, \neg\varphi$, — не теория с одной лишней ошибкой. Из неё выводимо *любое* утверждение ψ — например, что Луна сделана из зелёного сыра.

Раз φ истинно, дизъюнкция $\varphi \vee \psi$ истинна при любом ψ . Вместе с $\neg\varphi$ она даёт ψ . Из двух членов дизъюнкции один ложен, значит, истинным остаётся второй.⁶

Поэтому противоречие в теории — фатальный дефект: одна ошибка делает теорию способной доказать всё, что угодно, включая бессмыслицу.

Замечание Почему импликация именно такая?

Естественно ожидать, что импликация $p \rightarrow q$ выражает содержательную связь: причина влечёт следствие, условие гарантирует результат. Но таблица истинности материальной импликации не различает «Сократ — человек, поэтому Сократ смертен» и «трава зелёная, поэтому дважды два — четыре». Оба утверждения истинны, если составляющие их высказывания истинны. Связь между ними для логики высказываний не существует: семантика истинностно-функциональна, значение формулы зависит исключительно от значений подформул, а не от их смысла. Именно истинностная функциональность даёт таблицы истинности — исчерпывающий и конечный метод проверки. Она же гарантирует принцип подстановки: замена подформулы на логически эквивалентную сохраняет истинность всей формулы.

За эту простоту мы платим парадоксами материальной импликации. Вакуумная истинность разобрана выше. Зеркальное правило *verum ex quolibet sequitur* делает импликацию с истинным заключением истинной при любом антецеденте. Это структурное следствие композициональности: строка (F, T) обязана иметь *какое-то* значение, иначе соответствие не было бы функцией. Для материальной импликации в этой строке выбрана истина. Правильность определения зависит от задачи, которую решает инструмент. Для эффективно проверяемого формального языка математических доказательств и автоматического вывода истинностно-функциональная семантика оправдана. Простота проверки пере-

⁴ A. R. Anderson, N. D. Belnap. «Entailment: The Logic of Relevance and Necessity». Vol. 1, 1975.

⁶ Принцип взрыва (ex falso quodlibet) — стандартный результат классической логики высказываний. См., например: E. Mendelson. «Introduction to Mathematical Logic». 6-е изд., 2015, гл. 1.

вешивает интуитивные издержки — до тех пор, пока мы помним, *чем* именно мы пожертвовали.

1.1.2 Синтаксис формул

Из атомов и связок собираются формулы, и синтаксис языка определяется рекурсивно:

ОПРЕДЕЛЕНИЕ 1.7 Правильно построенная формула (*well-formed formula*)

- Каждая пропозициональная переменная (атом) является **правильно построенной формулой** (ППФ).
- Если φ, ψ — ППФ, то составные формулы ниже — тоже ППФ:

$$(\neg\varphi), \quad (\varphi \wedge \psi), \quad (\varphi \vee \psi), \quad (\varphi \rightarrow \psi), \quad (\varphi \leftrightarrow \psi).$$

- Ничто иное не является ППФ.

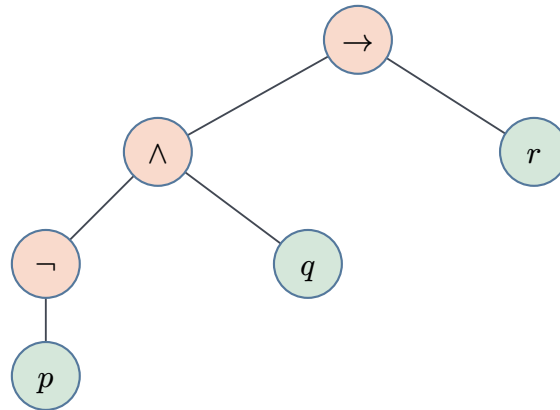
Пример ППФ и не-ППФ

- p — ППФ (атом).
- $(p \wedge q)$ — ППФ (конъюнкция двух атомов).
- $(\neg(p \wedge q) \rightarrow r)$ — ППФ (импликация отрицания конъюнкции и атома).
- $p \wedge q$ без скобок — технически не ППФ по строгому определению, но мы используем соглашения о приоритетах для опускания скобок.
- $p \neg q$ — не ППФ (отрицание требует скобок вокруг аргумента).

Соглашения о приоритетах уменьшают количество скобок:

- $\neg$ связывает сильнее всего,
- затем $\wedge$,
- затем $\vee$,
- затем $\rightarrow$,
- затем $\leftrightarrow$.

Каждая формула соответствует единственному дереву разбора, которое делает её синтаксическую структуру явной. На рис. 1 показано, как приоритеты расставляют структуру без явных скобок.

Рис. 1. Дерево разбора формулы $\neg p \wedge q \rightarrow r$.

Отрицание $\neg$ связывается с p , конъюнкция $\wedge$ соединяет результат отрицания с q , а импликация $\rightarrow$ — со всем выражением, так что формула читается как $(\neg p \wedge q) \rightarrow r$. Без соглашений о приоритетах ту же формулу пришлось бы записать как $((\neg p) \wedge q) \rightarrow r$.

1.1.3 Функциональная полнота системы связок

Пять связок — отрицание, конъюнкция, дизъюнкция, импликация, эквивалентность — выглядят произвольным набором. За этим выбором стоит конкретный факт: любой булев оператор от n переменных можно выразить через отрицание и конъюнкцию. Достаточно двух связок, а не пяти. Можно обойтись даже одной-единственной связкой:

- *Штрих Шеффера* (NAND, $\uparrow$) — функционально полон.
- *Стрелка Пирса* (NOR, $\downarrow$) — функционально полна.

Через каждую выражаются отрицание, конъюнкция и дизъюнкция. Вот как это работает для штриха Шеффера ($\uparrow$):

Пример Выражение связок через NAND

- **Отрицание:** $\neg p \equiv p \uparrow p$. Если p истинно, $p \uparrow p$ ложно. Если p ложно, $p \uparrow p$ истинно — в точности отрицание.
- **Конъюнкция:** $p \wedge q \equiv (p \uparrow q) \uparrow (p \uparrow q)$. Внутренний NAND даёт $\neg(p \wedge q)$. Применяя к нему NAND с самим собой, получаем двойное отрицание — и возвращаем $p \wedge q$.
- **Дизъюнкция:** $p \vee q \equiv (p \uparrow p) \uparrow (q \uparrow q)$. Это $\neg(\neg p \wedge \neg q)$. По закону де Моргана $\neg(p \wedge q) \equiv \neg p \vee \neg q$. Значит, равно $p \vee q$.

Пример Выражение связок через стрелку Пирса

Стрелка Пирса $p \downarrow q$ истинна ровно тогда, когда ложны оба операнда — двойственная картина к штриху Шеффера.

- **Отрицание:** $\neg p \equiv p \downarrow p$. Если p истинно, $p \downarrow p$ ложно. Если p ложно, $p \downarrow p$ истинно — в точности отрицание.

- **Дизъюнкция:** $p \vee q \equiv (p \downarrow q) \downarrow (p \downarrow q)$. Внутренний NOR даёт $\neg(p \vee q)$. Применяя NOR снова, получаем двойное отрицание — и возвращаем $p \vee q$.
- **Конъюнкция:** $p \wedge q \equiv (p \downarrow p) \downarrow (q \downarrow q)$. Это $\neg(\neg p \vee \neg q)$. По закону де Моргана $\neg(\neg p \vee \neg q) \equiv p \wedge q$.

Двух связок достаточно, а мы используем пять, потому что они формализуют распространённые паттерны рассуждений. «И», «или», «если... то» — естественно-языковые обороты, которые логика делает точными. Выбор правил — часть проекта языка. Затем система проверяется на корректность и полноту — об этом глава 2.

Функциональная полнота системы связок — это вопрос о минимальном наборе операций, из которых можно собрать все остальные. В главе 11 мы вернёмся к нему при проектировании логических схем.

1.1.4 Семантика: таблицы истинности

ОПРЕДЕЛЕНИЕ 1.8 Интерпретация

Интерпретация сопоставляет каждой атомарной пропозициональной переменной истинностное значение (Т или F).

Для n атомов существует 2^n различных интерпретаций. Каждая строка таблицы истинности соответствует одной интерпретации.

ОПРЕДЕЛЕНИЕ 1.9 Таблица истинности

Таблица истинности перечисляет все интерпретации атомов и вычисляет истинностное значение составной формулы при каждой интерпретации.

Пример Таблица истинности для $p \rightarrow q$

p	q	$p \rightarrow q$
Т	Т	Т
Т	Ф	Ф
Ф	Т	Т
Ф	Ф	Т

Таблица истинности исчерпывающе описывает поведение формулы при всех интерпретациях. По этому поведению формулы распределяются по классам.

ОПРЕДЕЛЕНИЕ 1.10 Тавтология

Тавтология — формула, истинная при любой интерпретации.

ОПРЕДЕЛЕНИЕ 1.11 Противоречие

Противоречие — формула, ложная при любой интерпретации.

ОПРЕДЕЛЕНИЕ 1.12 Выполнимая формула (*satisfiable formula*)

Выполнимая формула — формула, истинная хотя бы при одной интерпретации.

Интерпретацию, при которой формула истинна, называют *моделью* этой формулы.

ОПРЕДЕЛЕНИЕ 1.13 Контингентная формула

Контингентная формула — формула, истинная при одних интерпретациях и ложная при других, то есть не тавтология и не противоречие.

Пример Тавтологии, противоречия и выполнимые формулы

- $p \vee \neg p$ — тавтология (закон исключённого третьего).
- $p \wedge \neg p$ — противоречие.
- $p \wedge q$ выполнима, но не тавтология.

Для иллюстрации посмотрим на две простейшие таблицы: тавтологию и противоречие.

p	$p \vee \neg p$	$p \wedge \neg p$
T	T	F
F	T	F

Таблица истинности для n переменных содержит 2^n строк. По одной строке на каждую возможную комбинацию значений. Тавтология, противоречие и выполнимость описывают поведение отдельной формулы при всех возможных интерпретациях — своего рода «паспорт» формулы.

Но именно сравнение формул между собой — главная работа логики. Разные синтаксические выражения могут описывать одно и то же логическое содержание. Этот вопрос выводит нас на понятие логической эквивалентности.

ОПРЕДЕЛЕНИЕ 1.14 Логическая эквивалентность

Две формулы φ и ψ называются **логически эквивалентными**, обозначается $\varphi \equiv \psi$, если они имеют одинаковое истинностное значение при любой интерпретации.

Эквивалентность означает, что $\varphi \leftrightarrow \psi$ является тавтологией. Эквивалентные формулы взаимозаменяемы в любом контексте: подстановка сохраняет эквивалентность.

Пример Импликация через дизъюнкцию

$p \rightarrow q \equiv \neg p \vee q$ — одна из стандартных эквивалентностей. Проверяется таблицей истинности: обе формулы ложны только при $p = T, q = F$ и истинны в остальных трёх строках. Эта эквивалентность даёт расчётное объяснение вакуумной истинности. Если p ложно, то $\neg p$ истинно, и вся дизъюнкция истинна независимо от q .

Эквивалентность — отношение между формулами: они ведут себя одинаково при всех интерпретациях. Но часто нас интересует более слабая связь: истинность одних формул *гарантирует* истинность другой, без обратного.

ОПРЕДЕЛЕНИЕ 1.15 Логическое следствие

Формула ψ называется логическим следствием формул $\varphi_1, \dots, \varphi_n$, обозначается $\varphi_1, \dots, \varphi_n \models \psi$, если каждая интерпретация, в которой истинны все посылки φ_i , делает истинной и ψ .

Пример Логическое следствие: конъюнкция

Из $p \wedge q$ логически следует p . Запись: $p \wedge q \models p$. Всякая интерпретация, в которой $p \wedge q$ истинно, делает p истинным. А вот $p \models p \wedge q$ неверно. При $p = T, q = F$ посылка истинна, а заключение ложно.

ТЕОРЕМА 1.1 Семантическая теорема о дедукции

$\varphi_1, \dots, \varphi_n \models \psi$ тогда и только тогда, когда импликация является тавтологией:

$$(\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \psi.$$

Набросок доказательства

Докажем эквивалентность, переходя от семантического следования к условию на импликацию.

По определению, запись $\varphi_1, \dots, \varphi_n \models \psi$ означает, что любая интерпретация, делающая все φ_i истинными, делает истинным и ψ . Это равносильно отсутствию интерпретации, в которой $\varphi_1 \wedge \dots \wedge \varphi_n$ истинно, а ψ ложно. Но импликация $(\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \psi$ ложна ровно в этом случае. Значит, отсутствие таких интерпретаций — в точности тавтологичность импликации.

Семантическая теорема о дедукции соединяет следование ($\models$) и материальную импликацию. Одноимённая теорема для синтаксической выводимости $\vdash$ появится в теории вывода (глава 2). Чтобы доказать $p \rightarrow q$, предполагаем p и выводим q . Символ $\rightarrow$ и слова «если... то» обозначают одну и ту же связку.

Проверка Семантика: таблицы истинности

1. Сколько строк в таблице истинности формулы от пяти переменных и почему именно столько?
2. Формула ложна при любой интерпретации. Как называется такой класс и есть ли у неё модель?
3. Верно ли, что из $p \vee q$ следует p ? Какая интерпретация опровергает следование?

1.1.5 Законы и тождества

Логика высказываний удовлетворяет алгебраическим законам, позволяющим упрощать формулы — аналогично упрощению арифметических выражений.

УТВЕРЖДЕНИЕ 1.2 Законы логики высказываний

Для всех высказываний p, q, r :

Закон	Конъюнктивная форма	Дизъюнктивная форма
Коммутативность	$p \wedge q \equiv q \wedge p$	$p \vee q \equiv q \vee p$
Ассоциативность	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Дистрибутивность	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
Идемпотентность	$p \wedge p \equiv p$	$p \vee p \equiv p$
Поглощение	$p \wedge (p \vee q) \equiv p$	$p \vee (p \wedge q) \equiv p$
Двойное отрицание	$\neg\neg p \equiv p$	—
Тождество	$p \wedge \text{Т} \equiv p$	$p \vee \text{Ф} \equiv p$
Доминирование	$p \wedge \text{Ф} \equiv \text{Ф}$	$p \vee \text{Т} \equiv \text{Т}$
Дополнение	$p \wedge \neg p \equiv \text{Ф}$	$p \vee \neg p \equiv \text{Т}$

Набросок доказательства

Каждый закон проверяется таблицей истинности: вычисляем истинностные значения левой и правой частей при всех интерпретациях переменных и убеждаемся, что они совпадают. Для законов с двумя переменными (p, q) — четыре строки. С тремя — восемь. Проверка механическая: никаких дополнительных предположений не требуется.

Из таблицы законов особо выделяются два — законы де Моргана: они связывают отрицание с конъюнкцией и дизъюнкцией.

ТЕОРЕМА 1.3 Законы де Моргана⁷

- $\neg(p \wedge q) \equiv \neg p \vee \neg q$.
- $\neg(p \vee q) \equiv \neg p \wedge \neg q$.

Набросок доказательства

Проверим оба равенства таблицей истинности. Для $\neg(p \wedge q) \equiv \neg p \vee \neg q$ разберём две крайние строки.

При $p = q = \text{Т}$ левая часть ложна. Правая часть: $\text{F} \vee \text{F} = \text{F}$.

При $p = \text{Т}, q = \text{F}$ левая часть истинна. Конъюнкция $p \wedge q$ ложна, поэтому её отрицание истинно. Правая часть: $\text{F} \vee \text{Т} = \text{Т}$.

Остальные две строки симметричны. Для второго закона — двойственная проверка.

Законы де Моргана обобщаются: отрицание конъюнкции есть дизъюнкция отрицаний, и наоборот. Они проверяются таблицей истинности, но интуиция запоминается: «неверно, что оба истинны» означает «хотя бы одно ложно». Компиляторы и линтеры (ESLint, Clippy) применяют законы де Моргана для упрощения условий вида $\neg((x > 0) \wedge (y < 10))$. Такое условие превращается в $(x \leq 0) \vee (y \geq 10)$.

Законы, которые мы перечислили, дают инструмент для преобразования формул. Но преобразование осмысленно только если эквивалентность сохраняется при замене части на эквивалентную — иначе каждое применение закона требовало бы проверки всей формулы заново. Именно эту гарантию даёт следующий принцип.

ТЕОРЕМА 1.4 Принцип подстановки

Если $\varphi \equiv \psi$ и χ — формула, содержащая φ как подформулу, то замена φ на ψ в χ даёт формулу, эквивалентную χ .

Набросок доказательства

Индукция по построению формулы χ — это структурная индукция.

База: если χ совпадает с φ , то замена даёт ψ , и $\varphi \equiv \psi$ по условию.

Шаг: если χ получена из подформул с помощью связок, то по индукционному предположению замена в подформулах сохраняет эквивалентность. Таблицы истинности связок гарантируют, что эквивалентность подформул влечёт эквивалентность всей формулы.

Принцип подстановки означает, что мы можем заменить любую подформулу на эквивалентную, не меняя общего смысла. Именно так мы упрощаем булевы выражения в коде — выносим общие подвыражения, устраняем избыточные проверки и применяем правила вычисления с коротким замыканием.

⁷ А. Де Моргана. «Formal Logic: or, The Calculus of Inference, Necessary and Probable». 1847. Законы были известны в схоластической логике (XIV век, Уильям Оккам), но вошли в оборот современной логики благодаря Де Моргану и Булю.

Имея законы преобразования и гарантию сохранения эквивалентности при подстановке, мы можем поставить более амбициозную задачу: привести любую формулу к каноническому виду. Каноническое представление даёт возможность сравнивать формулы синтаксически — две формулы эквивалентны тогда и только тогда, когда их канонические формы совпадают.

Нормальные формы — ДНФ и КНФ — претендуют на роль такого канонического вида. Для канонической формы нужно, чтобы она была единственной: это гарантирует лишь *совершенная* (полная) ДНФ, где каждая конъюнкция содержит все переменные формулы. Именно её и строят по таблице истинности.

1.1.6 Нормальные формы

Каждая пропозициональная формула может быть приведена к каноническому виду. По ДНФ и КНФ две формулы сравниваются механически: привёл к форме — сравнил. Для больших формул нормализация — движок SAT-решателей (глава 14). Нормальные формы нужны и для других автоматизированных рассуждений: минимизация схем и выполнимость ограничений тоже работают с нормализованными формулами.

ОПРЕДЕЛЕНИЕ 1.16 Дизъюнктивная нормальная форма

Формула находится в ДНФ, если она является дизъюнкцией конъюнкций литералов (атомов или их отрицаний):

$$(l_{1,1} \wedge \dots \wedge l_{1,k_1}) \vee \dots \vee (l_{n,1} \wedge \dots \wedge l_{n,k_n})$$

ДНФ собирает условие истинности «снизу»: в совершенной ДНФ каждая конъюнкция кодирует ровно одну строку таблицы истинности, в которой формула истинна. КНФ, двойственно, собирает условие ложности: каждая дизъюнкция кодирует строку, которая *не должна* выполняться.

ОПРЕДЕЛЕНИЕ 1.17 Конъюнктивная нормальная форма

Формула находится в КНФ, если она является конъюнкцией дизъюнкций литералов:

$$(l_{1,1} \vee \dots \vee l_{1,k_1}) \wedge \dots \wedge (l_{n,1} \vee \dots \vee l_{n,k_n})$$

ТЕОРЕМА 1.5 Существование нормальных форм

Каждая пропозициональная формула имеет эквивалентные представления в ДНФ и КНФ.

Доказательство

Докажем, предложив явный алгоритм построения по таблице истинности.

Для построения ДНФ просматриваем все строки таблицы истинности, в которых формула истинна. Для каждой такой строки записываем конъюнкцию литералов: переменная входит без отрицания, если в этой строке она истинна, и с отрицанием, если ложна. Все полученные конъюнкции соединяем дизъюнкцией.

КНФ строится двойственно: просматриваем строки, в которых формула ложна. Для каждой записываем дизъюнкцию литералов — переменная входит с отрицанием, если в строке она истинна, и без отрицания, если ложна. Все полученные дизъюнкции соединяем конъюнкцией.

Алгоритм корректен: ДНФ истинна ровно в тех интерпретациях, которые попали в дизъюнкцию, то есть ровно в тех строках, где исходная формула истинна. Аналогично, КНФ ложна ровно в тех строках, где исходная формула ложна. Полученные формы не обязательно минимальны, но эквивалентны исходной формуле.

Существование ДНФ и КНФ для любой пропозициональной формулы — доказательство факта, который мы обсуждали в начале главы: система связок $\{\neg, \wedge, \vee\}$ функционально полна. Каждая формула выражима через три базовые операции. Законы де Моргана позволяют обойтись двумя: $\{\neg, \wedge\}$ или $\{\neg, \vee\}$. ДНФ и КНФ дают конструктивный алгоритм этого выражения: прочитал таблицу истинности — построил формулу.

Пример Построение ДНФ и КНФ по таблице истинности

Построим ДНФ и КНФ для импликации $p \rightarrow q$:

p	q	$p \rightarrow q$	ДНФ-терм	КНФ-терм
F	F	T	$\neg p \wedge \neg q$	—
F	T	T	$\neg p \wedge q$	—
T	F	F	—	$\neg p \vee q$
T	T	T	$p \wedge q$	—

ДНФ собирается из строк, где формула истинна: $(\neg p \wedge \neg q) \vee (\neg p \wedge q) \vee (p \wedge q)$. После упрощения: $(\neg p) \vee q$ — это одна из известных эквивалентностей для импликации.

КНФ собирается из строк, где формула ложна (здесь одна строка): $\neg p \vee q$. В данном случае КНФ состоит из единственной дизъюнкции литералов — формула уже была близка к КНФ.

Проверка Нормальные формы

1. Из каких строк таблицы истинности собирается совершенная ДНФ и из каких — КНФ?

2. Формула от трёх переменных истинна в трёх строках из восьми. Сколько конъюнкций в её совершенной ДНФ и сколько дизъюнкций литералов в совершенной КНФ?
3. Почему КНФ импликации $p \rightarrow q$ состоит из единственной дизъюнкции литералов?

§ 1.2 Методы доказательств

Доказательство — это формальное рассуждение, устанавливающее истинность утверждения из аксиом, определений и ранее доказанных утверждений.

Рассуждение движется по *правилам вывода* — разрешённым схемам переходов от посылок к заключению. Первое такое правило — *modus ponens* (MP): из φ и $\varphi \rightarrow \psi$ выводится ψ .

Пример Вывод по modus ponens

Пусть p — «идёт дождь», q — «улица мокрая».

Из посылок p и $p \rightarrow q$ правило modus ponens выводит q :

- | | | |
|-----|-------------------|----------------------|
| (1) | p | посылка |
| (2) | $p \rightarrow q$ | посылка |
| (3) | q | modus ponens: (1, 2) |

Это первый образец явного правила вывода из введения: каждый шаг рассуждения либо разрешён правилом, либо нет.

1.2.1 Прямое доказательство

Простейшая стратегия: чтобы доказать $P \rightarrow Q$, предполагаем P и выводим Q через цепочку логических рассуждений.

УТВЕРЖДЕНИЕ 1.6 Чётные квадраты

Если n чётно, то n^2 чётно.

Доказательство

Докажем прямо: предположим, что n чётно. Значит, $n = 2k$ для некоторого целого k .

Возводим в квадрат: $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$. Число $2k^2$ целое. Поэтому $n^2 = 2 \cdot \text{целое}$ — чётное.

1.2.2 Доказательство через контрапозицию

Импликация $P \rightarrow Q$ логически эквивалентна своей контрапозиции. Контрапозиция — это $\neg Q \rightarrow \neg P$. Иногда контрапозицию доказать легче, чем исходное утверждение.

УТВЕРЖДЕНИЕ 1.7 Нечётные квадраты

Если n^2 нечётно, то n нечётно.

Доказательство

Докажем контрапозицию: если n чётно, то n^2 чётно. (Каждое целое число либо чётно, либо нечётно, так что «не нечётно» означает «чётно».)

Это в точности доказанное выше утверждение о чётных квадратах. Контрапозиция эквивалентна исходному утверждению, так что утверждение выполняется.

Контрапозиция — это ход учёного и детектива: если теория предсказывает результат, а результат не наступил, теория неверна, и точно так же рана справа исключает убийцу-правшу. Опровержение следствия опровергает гипотезу, не требуя наблюдать саму гипотезу.

Контрапозиция особенно полезна, когда отрицание заключения ($\neg Q$) даёт конкретную отправную точку.

1.2.3 Доказательство от противного

Чтобы доказать P , предполагаем $\neg P$ и выводим противоречие. Противоречие имеет вид $R \wedge \neg R$ для некоторого R . Поскольку противоречия невозможны, $\neg P$ должно быть ложно, так что P истинно.

ТЕОРЕМА 1.8

$\sqrt{2}$ иррационально.

Доказательство

Докажем от противного, предположив, что $\sqrt{2}$ рационально. Тогда $\sqrt{2} = p/q$ для некоторых $p, q \in \mathbb{Z}^+$. Дробь несократима: $\gcd(p, q) = 1$.

Возводим в квадрат: $\sqrt{2}^2 = 2 = p^2/q^2$. Значит, $p^2 = 2q^2$. Следовательно, p^2 чётно, откуда p чётно. Если бы p было нечётным, $p = 2k + 1$. Тогда $p^2 = 2(2k^2 + 2k) + 1$ — нечётное число. Запишем $p = 2k$.

Подставляем:

$$\begin{aligned}(2k)^2 &= 2q^2 \\ 4k^2 &= 2q^2 \\ q^2 &= 2k^2.\end{aligned}$$

Таким образом, q^2 чётно, так что q чётно.

Оба p и q чётны, поэтому $\gcd(p, q) \geq 2$. Это противоречит $\gcd(p, q) = 1$. Следовательно, $\sqrt{2}$ иррационально.

ЛОВУШКА Контрапозиция $\neq$ доказательство от противного

Доказав, что из $\neg Q$ следует $\neg P$, делают вывод: «предположили P , вывели противоречие». Но контрапозиция — это прямое доказательство импликации $\neg Q \rightarrow \neg P$. Она эквивалентна $P \rightarrow Q$, противоречия в ней нет.

Доказательство от противного предположило бы $P \wedge \neg Q$. Оно вывело бы противоречие и заключило бы об истинности импликации $P \rightarrow Q$ (через снятие двойного отрицания). Различие видно в конструктивности. Контрапозиция выстраивает путь от $\neg Q$ к $\neg P$, а от противного лишь исключает отрицание, не предъявляя построения.

Доказательство от противного временно допускает ложное утверждение и выводит из него противоречие — без требования истинности посылки. Метод гибче контрапозиции: противоречие может не иметь прямого отношения к исходной гипотезе, поэтому от противного работает и там, где контрапозиция не просматривается.

1.2.4 Доказательство разбором случаев

Если гипотезу можно разбить на конечное множество взаимоисключающих и исчерпывающих случаев, то достаточно доказать заключение в каждом случае.

УТВЕРЖДЕНИЕ 1.9 Мультипликативность модуля

Для всех вещественных x, y выполняется равенство $|xy| = |x| \cdot |y|$.

Доказательство

Докажем тождество разбором случаев: знак xy определяется знаками x и y , а четыре комбинации знаков взаимоисключают друг друга и покрывают все возможности.

Случай 1. Если $x \geq 0$ и $y \geq 0$, то $|x| = x$, $|y| = y$ и $xy \geq 0$, откуда $|xy| = xy = |x| \cdot |y|$.

Случай 2. Если $x \geq 0$ и $y < 0$, то $|x| = x$, $|y| = -y$ и $xy \leq 0$, откуда $|xy| = -xy = x(-y) = |x| \cdot |y|$.

Случай 3. Если $x < 0$ и $y \geq 0$, то $|x| = -x$, $|y| = y$ и $xy \leq 0$, откуда $|xy| = -xy = (-x)y = |x| \cdot |y|$.

Случай 4. Если $x < 0$ и $y < 0$, то $|x| = -x$, $|y| = -y$ и $xy > 0$, откуда $|xy| = xy = (-x)(-y) = |x| \cdot |y|$.

Все случаи дают равенство, так что тождество выполняется для всех вещественных x, y .

Разбор случаев способен на большее, чем проверка тождеств. Иногда он доказывает *существование* объекта, не предъявляя его — потому что в каждом из случаев свидетель существует, но в разных случаях это разные объекты.

Классический пример — доказательство, приписываемое Джеффри Хантеру⁸ и популяризированное Довом Джарденом⁹.

УТВЕРЖДЕНИЕ 1.10 Иррациональная степень иррационального числа

Существуют иррациональные числа a и b , такие что a^b рационально.

Доказательство

Докажем разбором случаев: возьмём $x = \sqrt{2}^{\sqrt{2}}$, которое либо рационально, либо иррационально, и в каждом случае предъявим пару (a, b) .

Случай 1. x рационально, тогда $a = \sqrt{2}$, $b = \sqrt{2}$ — искомая пара. Оба числа иррациональны, а $a^b = x$ рационально.

Случай 2. x иррационально, тогда $a = x = \sqrt{2}^{\sqrt{2}}$ и $b = \sqrt{2}$. Оба иррациональны (по предположению случая). Возводим: $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2} \cdot \sqrt{2}} = \sqrt{2}^2 = 2$ — рационально.

В обоих случаях пара (a, b) существует. Доказательство не говорит, какая именно пара реализуется: $(\sqrt{2}, \sqrt{2})$ либо $(\sqrt{2}^{\sqrt{2}}, \sqrt{2})$. Оно устанавливает факт существования, не предъявляя свидетеля.

Такое доказательство называется *неконструктивным*: оно убеждает, что объект с требуемым свойством существует, но не даёт способа его построить. Какая из двух пар реализуется, доказательство не выясняет: вопрос, рационально ли $\sqrt{2}^{\sqrt{2}}$, для него безразличен.

Конструктивное доказательство предъявляет объект явно.

⁸G. Hunter. «Metalogic: An Introduction to the Metatheory of Standard First Order Logic». University of California Press, 1971.

⁹D. Jarden. «A simple proof that a power of an irrational number to an irrational exponent may be rational». Mathematics Magazine, 1953.

Пример Конструктивное доказательство того же утверждения

Пара $(\sqrt{2}, 2 \log_2 3)$ — искомая: оба числа иррациональны, а степень рациональна. Иррациональность $\sqrt{2}$ — классический результат (доказанный в этой главе). Иррациональность числа $2 \log_2 3$ проверяется от противного. Пусть $2 \log_2 3 = p/q$ для некоторых натуральных p, q . Тогда $\log_2 3 = \frac{p}{2q}$ и $3 = 2^{\frac{p}{2q}}$. Возведение обеих частей в степень $2q$ даёт $3^{2q} = 2^p$. Слева стоит степень тройки, справа — степень двойки, и по основной теореме арифметики (доказанной ниже в этой главе) такое равенство невозможно. При этом $\sqrt{2}^{2 \log_2 3} = 2^{\log_2 3} = 3$ — число рациональное.

Математическая практика принимает оба типа доказательств существования. Различие между ними существенно: одно дело — знать, что объект существует, другое — держать его в руках. Интуиционизм Брауэра (1907) проводит границу жёстче: существование считается доказанным, только когда объект предъявлен. Подробно — в главе 34.

1.2.5 Эквивалентность и контрпримеры

Утверждение вида $P \leftrightarrow Q$ доказывают в обе стороны: нужны обе импликации, $P \rightarrow Q$ и $Q \rightarrow P$. Другой способ — построить цепочку эквивалентностей: $P \leftrightarrow R_1 \leftrightarrow \dots \leftrightarrow Q$.

Пример Чётность числа и его квадрата

Докажем, что « n чётно тогда и только тогда, когда n^2 чётно».

($\Rightarrow$): Доказано выше (прямое доказательство).

($\Leftarrow$): Контрапозиция. Если n нечётно, то $n = 2k + 1$. Тогда $n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$ — нечётно.

Универсальное утверждение «для всех x верно $P(x)$ » опровергается одним контрпримером. Запись $\forall$ читается «для всех». Подробно — в главе 3.

Пример Контрпример: не все простые числа нечётны

- **Утверждение:** «Все простые числа нечётны».
- **Контрпример:** 2 простое и чётное.
- Следовательно, утверждение ложно.

Контрпример — логический аналог падающего теста: один опровергающий случай стоит любого числа подтверждающих.

Индукция — это компактная форма тех же рассуждений, которые программист проводит при написании циклов и рекурсивных функций.

1.2.6 Математическая индукция

Индукция доказывает утверждения вида $\forall n \in \mathbb{N} P(n)$. Это двигатель рассуждений о рекурсивно определённых объектах.

УТВЕРЖДЕНИЕ 1.11 Принцип математической индукции

Чтобы доказать $\forall n \geq n_0 \in \mathbb{N} : P(n)$:

1. **База индукции:** Доказать $P(n_0)$.
2. **Шаг индукции:** Доказать $\forall k \geq n_0 (P(k) \rightarrow P(k+1))$.

Заключение: $\forall n \geq n_0 P(n)$.

ЛОВУШКА Индукция без базы

Шаг индукции, доказанный без базы, ничего не доказывает. Уже в первом переходе $P(1) \rightarrow P(2)$ шаг опирается на $P(1)$. Эту посылку никто не установил.

Рассмотрим «доказательство» утверждения $n = n + 1$ для всех натуральных n . Шаг проходит: пусть $k = k + 1$. Прибавив к обеим частям единицу, получаем $k + 1 = k + 2$. Но база $1 = 2$ ложна — и с ней рухнет всё рассуждение.

Применим принцип к классической формуле — сумме первых n натуральных чисел.

УТВЕРЖДЕНИЕ 1.12 Сумма первых n натуральных чисел

Для любого $n \geq 1$:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Доказательство

Проведём индукцию по n .

База. При $n = 1$ левая часть равна 1. Правая часть: $\frac{1 \cdot 2}{2} = 1$.

Индукционное предположение: предположим, что $\sum_{i=1}^k i = \frac{k(k+1)}{2}$. Равенство рассматривается для некоторого $k \geq 1$.

Шаг индукции: вычислим сумму $\sum_{i=1}^{k+1} i$.

$$\sum_{i=1}^{k+1} i = \left(\sum_{i=1}^k i \right) + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}.$$

Это в точности $\frac{n(n+1)}{2}$. Формула записана для $n = k + 1$.

По индукции формула выполняется для всех $n \geq 1$.

Индукция — алгоритм проверки, а не источник понимания. Она лишь механически убеждается, что формула верна на каждой ступени, не объясняя, почему она верна. Формулу суммы первых n чисел можно увидеть, составляя пары слагаемых. Индукция лишь подтверждает результат, найденный другим путём.

Разложим сумму для $n = 6$ на пары: $1 + 2 + 3 + 4 + 5 + 6 = (1 + 6) + (2 + 5) + (3 + 4) = 3 \cdot 7 = 21$. Формула даёт тот же результат: $6 \cdot 7/2 = 21$. Для чётного n пар ровно $\frac{n}{2}$. Каждая пара даёт $n + 1$. Для нечётного n к парам добавляется средний элемент $\frac{n+1}{2}$ — итог тот же.

Обычная индукция опирает утверждение о $k + 1$ на одно значение $P(k)$. Иногда этого мало: чтобы разложить число на простые множители, нужны разложения его делителей, чтобы разменивать сумму монетами — размены меньших сумм. Для таких утверждений шагу нужна вся предыстория, и сильная индукция даёт индукционное предположение сразу обо всех предыдущих значениях.

ТЕОРЕМА 1.13 Сильная индукция

Чтобы доказать $\forall n \geq n_0 \in \mathbb{N} : P(n)$:

1. **База индукции:** Доказать $P(n_0)$.
2. **Шаг индукции:** Доказать $\forall k > n_0 ((\forall n_0 \leq i < k P(i)) \rightarrow P(k))$.

Заключение: $\forall n \geq n_0 P(n)$.

Набросок доказательства

Сильная индукция сводится к обычной. Определим $Q(n)$ как « $P(k)$ выполнено для всех k от n_0 до n ».

База: $Q(n_0)$ есть $P(n_0)$.

Шаг: предполагая $Q(k)$, нужно доказать $Q(k + 1)$. То есть нужно доказать $P(k + 1)$. Но $Q(k)$ означает, что все P от n_0 до k истинны. А это ровно посылка сильной индукции для $k + 1$. Значит, $P(k + 1)$ следует. Итак, $Q(k + 1)$ доказано.

Первый пример такой зависимости — размен монет: размен суммы n опирается на размен $n - 3$.

Пример Размен монет

Утверждение: любую сумму $n \geq 8$ копеек можно набрать монетами в 3 и 5 копеек.

Докажем сильной индукцией по n .

База. Проверим $n = 8, 9, 10$:

- $8 = 3 + 5$,
- $9 = 3 + 3 + 3$,
- $10 = 5 + 5$.

Шаг индукции. Пусть $n > 10$. Для всех k от 8 до $n - 1$ утверждение верно. Рассмотрим $n - 3$. Поскольку $n > 10$, имеем $n - 3 \geq 8$. По индукционному предположению сумму $n - 3$ можно набрать монетами в 3 и 5. Добавив ещё одну трёхкопеечную монету, получаем размен для n .

По сильной индукции утверждение верно для всех $n \geq 8$.

Основная теорема арифметики — пример, где сильная индукция нужна по существу: разложение составного числа опирается на разложения его меньших множителей. Её доказательство использует лемму Евклида.

ЛЕММА 1.14 Лемма Евклида

Если простое p делит произведение ab , то p делит один из сомножителей.

Набросок доказательства

Доказательство опирается на лемму Безу, которую принимаем здесь как допущение: если $\gcd(p, a) = 1$, то найдутся целые u, v с $pu + av = 1$. Полное доказательство леммы Безу через алгоритм Евклида — в главе 19.

Пусть простое p делит ab и не делит a . Делители простого p — это 1 и само p , поэтому у чисел p и a нет общих делителей, кроме 1. Значит, $\gcd(p, a) = 1$, и по лемме Безу найдутся целые u, v с $pu + av = 1$. Умножим равенство на b : $pub + abv = b$. Оба слагаемых слева делятся на p : первое — явно, второе — потому что $p \mid ab$. Значит, $p \mid b$.

УТВЕРЖДЕНИЕ 1.15 Основная теорема арифметики

Каждое целое число $n \geq 2$ можно представить как произведение простых чисел, и это представление единственно с точностью до порядка сомножителей.

Доказательство

Проведём сильную индукцию по n .

База. $n = 2$ простое.

Индукционное предположение: предположим, что каждое m из диапазона $2 \leq m < k$ раскладывается на простые множители.

Шаг индукции: если k простое, разложение уже получено. Если k составное, то $k = ab$. При этом $2 \leq a, b < k$. По предположению, a и b раскладываются на простые множители. Перемножая, получаем разложение k на простые множители.

По сильной индукции каждое целое $n \geq 2$ раскладывается на простые множители.

Единственность.

Пусть $n = p_1 p_2 \cdots p_k = q_1 q_2 \cdots q_m$ — два разложения на простые множители.

Тогда p_1 делит правую часть равенства. Так как p_1 простое, лемма Евклида гарантирует делимость одного из сомножителей. Этим сомножителем будет некоторое q_j . Но q_j — простое число. Поэтому $p_1 = q_j$.

Сокращаем пару p_1, q_j из обоих разложений и повторяем рассуждение для оставшихся множителей. В итоге каждое p_i находит себе пару среди сомножителей, и наоборот. $k = m$, и множители совпадают с точностью до перестановки.

Пример Разложение конкретного числа

Теорему удобно проверить на примере: $84 = 2 \cdot 2 \cdot 3 \cdot 7$. Порядок сомножителей не важен: $84 = 7 \cdot 3 \cdot 2 \cdot 2$ — то же разложение. Запись $84 = 4 \cdot 21$ разложением на простые не является: и 4, и 21 — составные, и их нужно продолжать, пока не останутся только простые множители.

Индукция напоминает рост бамбука: каждое коленце вырастает из предыдущего по одному и тому же правилу, и этого достаточно чтобы достичь тридцати метров. База: $P(0)$ — семя. Шаг: если выросло до k , то вырастет и до $k + 1$. Больше ничего не нужно.

У индукции есть равносильная формулировка: каждое непустое множество натуральных чисел имеет наименьший элемент. Приём «возьмём *наименьший* контрпример» переводит рассуждение из одной формулировки в другую, и он ещё не раз пригодится в курсе.

ТЕОРЕМА 1.16 Принцип полной упорядоченности (*well-ordering principle*)

Каждое непустое подмножество $\mathbb{N}$ имеет наименьший элемент.

Набросок доказательства

Докажем эквивалентность двух принципов в обе стороны.

Покажем сначала, что из принципа полной упорядоченности следует индукция. Пусть $P(0)$ истинно и пусть выполняется $\forall k (P(k) \rightarrow P(k + 1))$. Предположим вопреки этому, что найдётся n с ложным $P(n)$. Тогда множество $S = \{n \in \mathbb{N} \mid P(n) \text{ ложно}\}$ непусто, и по принципу полной упорядоченности у него есть наименьший элемент. Обозначим его m . Так как $P(0)$ истинно, $m > 0$, а значит, $P(m - 1)$ истинно — ведь m наименьший контрпример. Но из шага индукции следует $P(m)$, что даёт противоречие. Следовательно, S пусто, и $\forall n P(n)$.

Обратно: пусть работает индукция, а $S \subseteq \mathbb{N}$ непусто и не имеет наименьшего элемента. Докажем по индукции, что $\forall n (n \notin S)$. База: $0 \notin S$, иначе 0 был бы наименьшим. Шаг: если $0, \dots, k \notin S$, то $k + 1$ не принадлежит S , иначе он был бы наименьшим, что противоречит предположению об отсутствии у S наименьшего элемента.

1.2.7 Индукция и программирование

База индукции — это пустой вход или условие завершения рекурсивной функции. Шаг индукции — это тело цикла. Предполагая корректность после k итераций, доказываем корректность после $k + 1$.

Пример Инвариант цикла: сумма чисел

Пусть цикл накапливает сумму $1 + 2 + \dots + n$:

```
s := 0
i := 1
while i <= n:
    s := s + i
    i := i + 1
```

Инвариант цикла: $s = 1 + 2 + \dots + (i - 1)$. Перед первой итерацией $i = 1, s = 0$: инвариант тривиален.

Шаг: если инвариант верен, то после обновления $s := s + i, i := i + 1$ он верен снова. Сумма пополняется слагаемым i . Индекс становится $i + 1$, и инвариант сохраняется.

По завершении цикла $i = n + 1$. Инвариант даёт искомую сумму: $s = 1 + 2 + \dots + n$.

Структурная индукция обобщает этот принцип на деревья, формулы и программы. Именно на ней держится доказательство корректности компиляторов и свойств систем типов. CompCert — верифицированный компилятор C, сертифицированный с помощью Coq¹⁰. В теории языков программирования стандартное доказательство type safety опирается на два свойства — progress и preservation. Оба доказываются структурной индукцией по дереву вывода типов¹¹.

1.2.8 Структурная индукция

Обычная математическая индукция работает на натуральных числах. Но многие объекты в дискретной математике — формулы, деревья, списки, программы — определяются рекурсивно, а не нумеруются. Для них предназначена структурная индукция: чтобы доказать свойство для всех объектов, проверяем его для базовых

¹⁰X. Leroy et al. «CompCert — A Formally Verified Optimizing Compiler». INRIA, 2005–2025.

¹¹B. Pierce. «Types and Programming Languages». Глава 8, 2002.

(атомарных) элементов. Затем доказываем, что каждая операция построения новых объектов сохраняет это свойство.

ОПРЕДЕЛЕНИЕ 1.18 Структурная индукция для пропозициональных формул

Чтобы доказать, что свойство P выполнено для всех правильно построенных формул:

1. **База:** Доказать $P(A)$ для каждой атомарной формулы (пропозициональной переменной).
2. **Шаги индукции:** Предполагая, что $P(\varphi), P(\psi)$ выполнены (индукционное предположение), доказать:
 - $P(\neg\varphi)$,
 - $P(\varphi \wedge \psi)$,
 - $P(\varphi \vee \psi)$,
 - $P(\varphi \rightarrow \psi)$,
 - $P(\varphi \leftrightarrow \psi)$.

Структурная индукция — та же математическая индукция, применённая к деревьям вместо чисел.

Пример Структурная индукция: баланс скобок

Докажем, что каждая правильно построенная формула (ППФ) содержит равное число открывающих и закрывающих скобок.

База: Атомарная формула — переменная $p, q, r, \dots$ — скобок не содержит: 0 открывающих, 0 закрывающих.

Шаг для $\neg$: Пусть φ — ППФ. По индукционному предположению φ имеет равное число открывающих и закрывающих скобок. Формула $(\neg\varphi)$ добавляет одну открывающую и одну закрывающую скобку. Баланс сохраняется.

Шаг для бинарных связок: Пусть φ, ψ — ППФ со сбалансированными скобками (индукционное предположение). Формула $(\varphi \wedge \psi)$ добавляет одну открывающую и одну закрывающую скобку. Аналогично для связок $\vee, \rightarrow, \leftrightarrow$. Скобки добавляются к сумме скобок φ и ψ . Баланс сохраняется.

По принципу структурной индукции, свойство выполнено для всех ППФ.

С программистской точки зрения структурная индукция — это свёртка (*fold*). Рекурсивная функция обходит дерево формулы: в листе она возвращает базовое значение, в узле применяет операцию к результатам поддеревьев. Доказательство структурной индукцией устроено так же: база фиксирует значение на листьях, а индукционные шаги — обработку результатов поддеревьев. Разница в том, что функция вычисляет, а доказательство убеждает.

Структурная индукция — один из основных инструментов в теории формальных языков, теории типов и верификации программ. Когда доказывается свойство для всех программ на некотором языке, наиболее прямой метод — структурная индукция по абстрактному синтаксическому дереву. Синтаксис языка задаёт индуктивную структуру, а семантическое свойство — утверждение, которое требуется доказать.

Замечание Почему индукция работает?

Принцип математической индукции часто преподносят как исходное допущение о натуральных числах — закон мышления, принимаемый без обоснования. Эта картина переворачивает логическую зависимость.

Индукция — не внешняя аксиома, наложенная на натуральный ряд. Она — прямое следствие того, как натуральные числа *построены*. Натуральный ряд определён рекурсивно: 0 — базовый элемент. Для каждого n существует следующее за ним число $n + 1$ — и никаких других натуральных чисел нет. Если объекты исчерпываются базовыми элементами и конструкторами, то утверждение, истинное для базовых и сохраняющееся всеми конструкторами, обязано быть истинным для всех объектов. Причина в том, что каждый объект строится из базовых конечным числом применений конструкторов.

То же верно для структурной индукции с той разницей, что вместо цепочки 0, 1, 2, ... перед нами синтаксические деревья. Формулы логики высказываний определены рекурсивно: атомарные переменные служат базовыми элементами, а отрицание и бинарные связки — конструкторами. Свойство, истинное для атомов и сохраняющееся при всех операциях построения формул, выполнено для любой правильно построенной формулы. Любая формула получается из атомов конечным числом шагов. Логика та же, путь построения — дерево вместо цепочки.

Эта точка зрения проясняет отношение математической индукции к индукции в естественно-научном смысле. Юм заметил, что эмпирическая индукция — переход от наблюдения конечного числа случаев к универсальному утверждению — не имеет логического оправдания. Сколько бы белых лебедей вы ни видели, это не доказывает, что все лебеди белы. Математическая индукция, вопреки названию, не делает такого перехода. Она не обобщает наблюдения. Она опирается на конструктивную структуру области определения и идёт по конструктивной структуре самого объекта. Рассуждение дедуктивно, с той же необходимостью, с какой истинна теорема, доказанная прямым рассуждением. Название «индукция» — историческая условность, а логическая структура — дедуктивная.

1.2.9 Типичные ошибки

Начинающие совершают предсказуемые ошибки в доказательствах. Распознать их рано — значит сэкономить время и избежать неловких ситуаций.

ОПРЕДЕЛЕНИЕ 1.19 Порочный круг (*circular reasoning*)

Порочный круг — это рассуждение, в котором доказываемое утверждение (возможно, в переформулированном виде) используется среди его же посылок.

Пример Порочный круг в действии

«Бог существует, потому что так сказано в Библии, а Библия — это слово Бога» — посылка уже предполагает заключение.

ОПРЕДЕЛЕНИЕ 1.20 Утверждение консеквента (*affirming the consequent*)

Из $P \rightarrow Q$ и Q делается вывод P .

Пример Утверждение консеквента

«Если идёт дождь, земля мокрая. Земля мокрая. Следовательно, шёл дождь» — земля могла быть мокрой из-за поливальной машины.

Две следующие ошибки — близнецы: обе нарушают направление импликации. Утверждение консеквента выводит P из Q . Отрицание antecedента выводит $\neg Q$ из $\neg P$.

ОПРЕДЕЛЕНИЕ 1.21 Отрицание antecedента (*denying the antecedent*)

Из $P \rightarrow Q$ и $\neg P$ делается вывод $\neg Q$.

Пример Отрицание antecedента

«Если идёт дождь, земля мокрая. Дождя не было. Следовательно, земля не мокрая» — опять же, достаточно поливальной машины.

Замечание Тест Уэйсона: поиск подтверждения вместо опровержения

Проверьте правило на четырёх карточках: на одной стороне буква, на другой число, и видны «А», «К», «4», «7». Правило: если на одной стороне гласная, то на другой — чётное число.

Импликация ложна ровно в одном случае: гласная и нечётное число. Значит, переворачивать нужно карточки «А» и «7» — гласную проверить на чётность оборота, нечётное число — на отсутствие гласной. Карточки «К» и «4» проверять

бесполезно: правило ничего не говорит ни об обороте согласной, ни об обороте чётного числа.

Большинство людей переворачивают «А» и «4» — они ищут подтверждение правила, а не его нарушение. Та же задача в социальной форме — «если человек пьёт алкоголь, ему должно быть больше 21 года» — почти не вызывает ошибок: мозг настроен ловить обманщиков, а не перебирать строки таблицы.

Своя ловушка есть и у индукции.

ОПРЕДЕЛЕНИЕ 1.22 Ошибка в шаге индукции

Шаг $P(k) \rightarrow P(k+1)$ обязан работать при всех k , начиная с базы. Если переход опирается на скрытое предположение, верное не для всех k , индукция ломается даже при верной базе.

Пример Ошибка в шаге: все лошади одного цвета

«Все лошади одного цвета». Докажем индукцией по n , что любые n лошадей одного цвета. База $n = 1$ тривиальна. Шаг: пусть верно для n . Возьмём $n + 1$ лошадь и уберём первую — останется n лошадей одного цвета. Уберём вместо неё последнюю — снова n лошадей одного цвета. Две группы по n лошадей пересекаются, значит все $n + 1$ лошадей одного цвета.

База верна, ошибка — в шаге: он опирается на непустоту пересечения двух групп, а при $n = 1$ (переход к двум лошадям) пересечение пусто. Шаг не работает при $n = 1$, индукция застревает на первом же переходе. Контрпример — пара лошадей разного цвета.

Ошибка в одном шаге обесценивает всё рассуждение: вывод строится на цепочке переходов, и один неверный переход оставляет заключение необоснованным, даже если остальные шаги корректны. Проверка каждого шага на соответствие определениям и известным фактам отлавливает большинство ошибок.

Примечание Арсенал доказательств

Индукция, контрапозиция и доказательство от противного — самые распространённые техники доказательства в дискретной математике.

- **Прямое доказательство** — первое, что стоит пробовать: идём от посылок к заключению.
- **Контрапозиция** — чище и предпочтительнее, когда заключение имеет вид отрицания.
- **От противного** — работает всегда, но тяжелее других: предполагаем отрицание желаемого и ищем любое противоречие.
- **Индукция** — для утверждений о натуральных числах и рекурсивных структурах.

Форма утверждения подсказывает метод: для $\neg Q \rightarrow \neg P$ — контрапозиция, для свойств натуральных чисел — индукция.

Проверка Методы доказательств

1. Почему в примере «все лошади одного цвета» переход от одной лошади к двум не срабатывает, хотя шаг индукции выглядит корректным?
2. Чем структурная индукция отличается от обычной, если натуральные числа — тоже индуктивная структура?

§ 1.3 Логика и корректность программ

Методы доказательств — прямое рассуждение, контрапозиция, индукция — находят прямое применение в программировании. Ассерт о программе — это формула, а проверка программы — доказательство этой формулы. Заодно различие тестирования и доказательства, о котором шла речь во введении, станет здесь точным.

1.3.1 Ассерты и тройки Хоара

ОПРЕДЕЛЕНИЕ 1.23 Ассерт

Ассерт (*assertion*) — это логическая формула, размещённая в коде, которая должна быть истинна в этой точке выполнения.

Ассерты документируют ожидания и отлавливают ошибки во время выполнения.

ОПРЕДЕЛЕНИЕ 1.24 Тройка Хоара

Тройка Хоара $\{P\} S \{Q\}$ означает следующее. Предусловие P выполняется перед оператором S . Постусловие Q выполняется после завершения S .

Пример Корректная тройка Хоара

$$\{x = 5\} x := x + 1 \{x = 6\}$$

Здесь, если x равно 5 до присваивания, то после него x равно 6.

Пример Тройка Хоара с условным оператором

$$\{x \geq 0\} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \{y = |x|\}$$

Если x неотрицательно, то после условного оператора y содержит модуль x . Когда $x > 0$, присваивается сам x . Когда $x = 0$, присваивается $-0 = 0$. Предусло-

вие гарантирует, что ветка «else» выполняется только при $x = 0$. Случай $x < 0$ исключён. Поэтому $y := -x = 0 = |x|$.

Тройка Хоара фиксирует лишь то, что постусловие выполнено *после* выполнения оператора S , — она ничего не говорит о том, что происходит *во время* выполнения.

Предусловие кодирует предположения о состоянии программы. Постусловие кодирует гарантии. Вместе они образуют контракт: если вызывающая сторона удовлетворяет P , процедура обеспечивает Q .

Тройки Хоара — это язык, а не алгоритм: они лишь записывают утверждения о корректности. Как доказывать корректность циклов (инварианты), как проверять терминируемость и как автоматизировать верификацию — в главе 32.

Пример Цикл как тройка Хоара

Цикл накопления суммы из раздела об индукции — готовая тройка Хоара:

$$\{s = 0 \wedge i = 1\} \text{ while } i \leq n \text{ do } (s := s + i; i := i + 1) \{s = 1 + 2 + \dots + n\}$$

Инвариант $s = 1 + 2 + \dots + (i - 1)$ — тот же, что в примере о сумме чисел: там проверено, что перед первой итерацией он тривиален, что тело цикла его сохраняет и что при $i = n + 1$ он даёт искомую сумму. Тройка Хоара добавляет к этой проверке только оформление: предусловие фиксирует начальное состояние, постусловие — требуемый результат. Терминирование — отдельное обязательство, инвариантом не покрываемое: i увеличивается на 1 на каждом шаге, после n шагов условие $i \leq n$ становится ложным и цикл завершается.

1.3.2 Контрпримеры и тестирование

Падающий тест — это контрпример к утверждению «программа корректна на этом входе». Тестирование может продемонстрировать *наличие* ошибок, но никогда их *отсутствие* — точно так же, как один контрпример опровергает $\forall$, но никакое количество примеров его не доказывает.

Логическая связь между тестированием и доказательством:

- Тестирование = поиск контрпримера.
- Доказательство = демонстрация того, что контрпримеров не существует.

И то, и другое необходимо: тестирование находит лёгкие ошибки, а доказательство — глубокие.

Идею теста-свойства доводит до инструмента подход *property-based testing*¹². Библиотека QuickCheck проверяет не отдельные примеры, а свойства: программист

¹²K. Claessen, J. Hughes. «QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs». ICFP, 2000.

описывает свойство функции, библиотека генерирует сотни случайных входов и ищет вход, на котором свойство нарушается. Найденный вход автоматически сокращается до минимального контрпримера и выдаётся программисту. Граница между проверкой и доказательством не стирается: сотни случайных проверок не складываются в доказательство, зато каждый найденный контрпример — готовый падающий тест.

Проверка Логика и корректность программ

1. Тройка Хоара $\{P\}S\{Q\}$ корректна. Гарантирует ли она, что S завершается?
2. Почему один падающий тест опровергает корректность, а сотни успешных прогонов её не доказывают?

§ 1.4 Итоги главы

Убедительность и обоснованность — не одно и то же, и это первое различие, которое мы теперь держим в голове. Сколько бы примеров мы ни проверили, универсальное утверждение держится только на рассуждении, охватывающем все случаи, а рушится от одного контрпримера. Миллион удачных проверок того, что сумма двух чётных чисел чётна, ничего не доказывает. Критерий истинности — то, что каждый шаг рассуждения либо разрешён, либо нет, а не убедительность и не согласие сообщества. Отсюда два занятия с разными гарантиями: тестирование ловит контрпример, а доказательство показывает, что его не существует.

Язык и механизм мы собрали заново. Высказывания и связки — конъюнкция, дизъюнкция, отрицание, импликация — дают формальную запись. Таблица истинности превращает проверку формулы в вычисление: для n переменных — 2^n строк, и значение любого составного высказывания находится механически. Мы увидели и цену этого языка: импликация с ложной посылкой истинна — так она покрывает все случаи разом. Из небольшого набора связок выражается любая булева функция (дизъюнкция — через штрих Шеффера).

Методы доказательств — ограниченный набор приёмов, и мы научились выбирать между ними. Прямое рассуждение разворачивает определение, контрапозиция доказывает $A \rightarrow B$ как $\neg B \rightarrow \neg A$. Доказательство от противного опирается на закон исключённого третьего, разбор случаев покрывает исчерпывающие ветвления. Индукция — отдельный приём: база и честный переход $P(k) \rightarrow P(k+1)$, которые доказывают переход, а не наблюдают закономерность. Сильная индукция нужна там, где шаг зависит от нескольких предыдущих значений. Отдельный навык — замечать ходы, которые выглядят доказательствами, но не являются ими: утверждение консеквента и отрицание антецедента.

Наконец, всё это приложилось к программам. Ассерт — формула, инвариант цикла — утверждение, которое обязано держаться на каждой итерации, а корректность — доказуемость. Тройка Хоара $\{P\}S\{Q\}$ упаковывает утверждение о программе в

проверяемый вид: если до S истинно P , то после — Q . Падающий тест ловит одну ошибку, доказательство исключает их все, и это различие для нас теперь точное, а не интуитивное.

Проверка Проверьте себя

1. Почему таблица истинности для формулы от n переменных содержит 2^n строк?
2. Почему импликация с ложной посылкой истинна, и чем приходится платить за это определение?
3. Почему доказательство от противного опирается на закон исключённого третьего?
4. Чем доказательство через контрапозицию отличается от доказательства от противного?
5. Почему математическая индукция не является эмпирическим обобщением?

Что дальше?

Теперь, когда язык логики и техника доказательств освоены, фиксируем правила вывода: в главе 2 будет показано, как проверка доказательства сводится к механической процедуре. Затем переходим к множествам — базовой структуре данных математики. Логические связки переходят в операции над множествами. В главе 4 мы формализуем коллекции, операции над ними и установим связь теории множеств с SQL, битовыми масками и типами в языках программирования.

§ 1.5 Упражнения

Высказывания, связки и таблицы истинности.

1. Какие из следующих предложений являются высказываниями? «Сегодня понедельник», «Иди сюда!», « $x > 5$ », «Число 7 — простое». Для каждого высказывания укажите истинностное значение.
2. Постройте таблицу истинности для формулы $(p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow r) \rightarrow r$. Является ли она тавтологией?
3. * Выразите дизъюнкцию $p \vee q$ через штрих Шеффера (NAND). Проверьте эквивалентность таблицей истинности.

Эквивалентности и нормальные формы.

4. Приведите $(p \rightarrow q) \wedge (q \rightarrow r)$ к ДНФ и КНФ по таблице истинности. Упростите результат с помощью эквивалентностей.
5. Докажите с помощью эквивалентностей: $\neg(p \vee q) \equiv \neg p \wedge \neg q$. Проверьте результат таблицей истинности.

Методы доказательств.

6. Докажите прямо: если n нечётно, то n^2 нечётно.
7. Докажите от противного: $\sqrt{3}$ иррационален.
8. Докажите разбором случаев, что для любого целого n произведение $n(n+1)(n+2)$ делится на 3.
9. * Предъявите контрпример к утверждению «если целое число делится на 6 и на 10, то оно делится на 60». Объясните, почему контрпример опровергает утверждение.

Логика и корректность программ.

10. Запишите тройку Хоара для оператора $x := x + 1$. Возьмите предусловие $x = 5$.
11. * Проверьте тройку $\{x = 0\}x := x + 1\{x = 1\}$. Корректна ли она? Сравните с тройкой $\{x = 0\}x := x + 1\{x = 2\}$.

Математическая индукция.

12. Докажите по индукции, что сумма первых n нечётных чисел равна n^2 : $1 + 3 + 5 + \dots + (2n - 1) = n^2$.
13. Докажите по индукции: $1^3 + 2^3 + \dots + n^3 = (1 + 2 + \dots + n)^2$.
14. * Найдите ошибку в индукционном «доказательстве», что все лошади одного цвета (разобрано в тексте главы). На каком именно шаге перехода $P(k) \rightarrow P(k+1)$ рассуждение ломается и почему?
15. ** Докажите сильной индукцией: любое целое $n \geq 2$ раскладывается в произведение простых чисел. Почему обычной индукции недостаточно?
16. * Найдите ошибку в «доказательстве» по индукции, что все числа Фибоначчи нечётны. Числа Фибоначчи заданы равенствами $F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2}$. База $F_1 = 1, F_2 = 1$ верна, а индукционный шаг якобы следует из того, что свойство нечётности наследуется от пары предыдущих чисел. Что на самом деле даёт шаг, и почему $F_3 = 2$ опровергает утверждение?

ПРОЕКТ Инспектор формул: таблицы истинности как проверка спецификаций

Логика высказываний из этой главы даёт язык для условий: «если a и не b , то c ». Инженеру такие условия нужны, чтобы проверять спецификации — набор требований, каждое из которых записано формулой. Сразу возникают три вопроса: выполнима ли спецификация, не противоречат ли два требования друг другу и следует ли из набора требований ожидаемый вывод. На двух-трёх переменных хватает карандаша и бумаги, но уже при шести переменных таблица истинности занимает 64 строки, и считать её вручную бессмысленно. Соберите инструмент, который делает это сам.

① Парсер формул

Научите программу читать формулы в текстовом виде: переменные, связки $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ (или их символы), скобки. Введите приоритеты: $\neg$ связывает сильнее $\wedge$, $\wedge$ — сильнее $\vee$, $\vee$ — сильнее $\rightarrow$. Постройте дерево разбора: рекурсивный спуск по приоритетам или обратная польская запись. Проверьте на формулах с вложенными скобками вроде $(a \vee b) \wedge \neg(a \wedge b)$.

② *Оценка и таблица истинности*

Пусть дерево вычисляет значение формулы при заданной интерпретации — наборе значений всех переменных. Переберите все 2^n интерпретаций в лексикографическом порядке и выведите полную таблицу: столбец на каждую переменную, последний столбец — формула. Сверьте таблицу для $(a \rightarrow b) \leftrightarrow (\neg a \vee b)$ с ручным расчётом: эквивалентности главы должны проявляться как одинаковые столбцы.

③ *Классификация и эквивалентность*

По столбцу формулы определите её тип: тавтология (все единицы), противоречие (все нули) или выполнимая формула (встречаются и те, и другие). Для двух формул сравните столбцы: совпали — формулы эквивалентны. Не совпали — предъявите контрпример, интерпретацию, на которой значения расходятся.

④ *Логическое следование*

Добавьте проверку следования: даны посылки $P_1, \dots, P_k$ и заключение Q . Заключение следует из посылок, если на каждой интерпретации, где все посылки истинны, истинно и Q . Если это не так — выведите контрпример: интерпретацию, на которой посылки истинны, а заключение ложно. Проверьте модус поненс: из p и $p \rightarrow q$ следует q , а из q и $p \rightarrow q$ — нет. Сформулируйте, чем этот перебор отличается от понятия доказательства из главы 2.

⑤ *Нормальные формы*

Как расширение: по таблице истинности постройте СДНФ (строки с единицами) и СКНФ (строки с нулями). Сгенерируйте случайные пары формул и проверьте автоматически законы алгебры логики: де Моргана, дистрибутивность, поглощение. Убедитесь, что эквивалентности главы — это следствия таблиц: вся логика высказываний уместается в перечисление строк.

Итог: инструмент, который по текстовой формуле строит таблицу истинности, классифицирует формулу, сравнивает две формулы на эквивалентность и проверяет логическое следование с выдачей контрпримера.

II

Дедукция и системы вывода

В 1998 году Томас Хейлс объявил, что доказал гипотезу Кеплера. Гипотеза, высказанная в 1611 году, утверждает, что шары в пространстве нельзя уложить плотнее, чем пирамидой, которой торговцы складывают апельсины на прилавке. Плотность такой укладки — $\frac{\pi}{\sqrt{18}} \approx 0.74048$, и вопрос в том, бывает ли что-нибудь плотнее.

Доказательство Хейлса не походило на привычное — двести пятьдесят страниц рассуждений плюс три гигабайта программ, в которых компьютер перебирал тысячи случаев. Проверить такое целиком не мог ни один человек.

Редакция журнала *Annals of Mathematics*, одного из самых строгих в математике, согласилась опубликовать доказательство — при условии, что его примет группа из двенадцати рецензентов. Рецензенты работали четыре года. В 2003 году руководитель группы Габор Фейеш Тот подвёл итог: рецензенты на 99 процентов уверены, что доказательство верно, но засвидетельствовать это целиком не берутся — за компьютерную часть они не поручаются.

Четыре года работы двенадцати специалистов дали не доказательство, а мнение.

Тогда Хейлс поставил задачу иначе — всё доказательство, страницы и вычисления, было записано заново на формальном языке и прогнано через программу-проверщик. В 2014 году, через шестнадцать лет после первого объявления, каждая строчка оказалась проверена машиной. Без процентов уверенности — каждый шаг либо разрешён правилами, либо нет.

Между 1998 и 2014 годами изменилась не математика — рассуждение осталось прежним. Изменилось то, чем стало доказательство — из текста, который нужно понять, оно превратилось в объект, который можно проверить.

Чтобы превратить рассуждение в такой объект, нужны две вещи — точный язык, записывающий рассуждение формулами, и явные правила, говорящие, какие переходы между формулами разрешены. Язык построен в главе 1. Правила и то, как они устроены и как ими пользуются, образуют систему вывода.

Систем вывода придумано несколько, и записывают выводы в них по-разному — колонкой с вложенными поддоказательствами, деревом, цепочкой. Различаются и сами правила — где-то их немного и логика спрятана в аксиомах, а где-то аксиом нет вовсе.

Что делает проверку доказательства механической процедурой, а не делом вкуса?

ИСТОРИЯ От Begriffsschrift до Робинсона

Первую полноценную систему вывода построил Готлоб Фреге в 1879 году в сочинении «Begriffsschrift» («Запись понятий»): аксиомы и правила вывода для логики предикатов. Работу почти не заметили. Вскоре к тем же идеям независимо пришли Чарльз Пирс и Джузеппе Пеано.

На рубеже веков Давид Гильберт превратил формализацию в программу: вся математика — аксиомы плюс правила вывода, и доказательства проверяются механически. Программа упиралась в вопрос о непротиворечивости арифметики — вопрос, который в 1931 году закрыл Курт Гёдель теоремой о неполноте.

Натуральную дедукцию в 1934 году изобрели дважды и независимо: Станислав Яськовский в Польше и Герхард Гентцен в Германии. Яськовский предложил запись колонкой с поддоказательствами — её называют нотацией Фитча, по имени преподавателя, который популяризовал этот стиль. Гентцен — деревья вывода и, годом позже, секвенциальное исчисление. В работе о секвенциях он доказал Hauptsatz — теорему об устранении сечений — и вывел из неё непротиворечивость логики: пустая секвенция не выводима, значит, противоречия в системе нет. Труднее оказался вопрос о непротиворечивости арифметики, который Гёдель в 1931 году объявил недоказуемым средствами самой арифметики. Гентцен доказал её в 1936 году — трансфинитной индукцией до ординала ε_0 , средством за пределами арифметики Пеано. Деревья и колонки — две записи одной и той же системы, и мы увидим обе.

В 1965 году Джон Алан Робинсон опубликовал правило резолюции — систему, пригодную для машинного поиска доказательств. Она легла в основу автоматического доказательства теорем и SAT-решателей. От рукописи Фреге до алгоритмов Робинсона прошло меньше века — срок, за который вопрос «что такое доказательство» получил ответ: объект, который проверяет машина.

ОБЗОР ГЛАВЫ

В этой главе мы выясним, что такое система вывода: аксиомы, правила, вывод — и чем одна система отличается от другой. Затем разберём четыре классические системы: системы Гильберта, натуральную дедукцию, секвенциальное исчисление Гентцена и резолюцию. Нотация Фитча — линейная запись натуральной дедукции — станет нашим рабочим инструментом: для каждой связки мы разберём правила ввода и удаления. В конце главы появятся кванторные правила, корректность и полнота — два результата, связывающих выводимость с истинностью.

После этой главы вы сможете:

1. проверять доказательство механически — по шагам, не обращаясь к смыслу формул;
2. строить выводы в нотации Фитча для высказываний и для предикатов;

3. различать системы Гильберта, натуральную дедукцию, секвенции и резолюцию и объяснять, чем они различаются;
4. отделять интуиционистские правила от классических и понимать, почему закон исключённого третьего не выводим без косвенного доказательства;
5. формулировать корректность и полноту и понимать, что каждая из них гарантирует.

§ 2.1 Что такое система вывода

Рассуждение на естественном языке трудно проверять: «следовательно» в нём означает «я так считаю». Система вывода заменяет этот переход на явное правило: переход законен, если он имеет форму, разрешённую системой, и только тогда.

ОПРЕДЕЛЕНИЕ 2.1 Система вывода

Система вывода — это набор аксиом и правил вывода.

ОПРЕДЕЛЕНИЕ 2.2 Аксиома

Аксиома — это формула, принимаемая системой без доказательства.

ОПРЕДЕЛЕНИЕ 2.3 Правило вывода

Правило вывода — это разрешённый переход от посылок заданной формы к заключению заданной формы.

Например, системой вывода будет такая пара: аксиома — формула $A \rightarrow (B \rightarrow A)$ для любых формул A и B , правило — modus ponens, которое из $A \rightarrow B$ и A выводит B .

Аксиома входит в вывод без ссылок на предыдущие строки, результат правила — со ссылками на посылки. Проверка правила смотрит только на форму посылок и заключения, поэтому проверять выводы может машина — ей не нужно понимать, что означают формулы, достаточно сверить форму.

ОПРЕДЕЛЕНИЕ 2.4 Гипотеза

Гипотеза — это формула, принимаемая в выводе без доказательства, но не входящая в систему.

Как и аксиома, гипотеза входит в вывод без ссылок, но действует лишь внутри одного вывода. Аксиома $A \rightarrow (B \rightarrow A)$ принимается в любом выводе системы, а допущение «предположим, что A » — только в том выводе, где его объявили.

ОПРЕДЕЛЕНИЕ 2.5 Логический вывод (*derivation*)

Логическим выводом формулы φ из гипотез Γ называется конечная последовательность формул, в которой каждая формула либо аксиома, либо гипотеза из Γ , либо результат применения правила вывода к предыдущим формулам. Существование такого вывода обозначается $\Gamma \vdash \varphi$ и читается «из Γ выводится φ ».

Примечание Пустой контекст

Набор гипотез Γ может быть пустым. Запись $\vdash \varphi$ означает, что формула φ выводится без допущений. Такая формула — теорема системы.

В нашей системе вывод $B \rightarrow A$ из гипотезы A занимает три строки:

1. $A \rightarrow (B \rightarrow A)$ — аксиома.
2. A — гипотеза.
3. $B \rightarrow A$ — modus ponens по строкам 1 и 2.

Каждая строка снабжена оправданием, и каждое оправдание проверяется без понимания смысла формул — сверкой с формой. Modus ponens устроен так, что в нём невозможно усомниться: система вывода обязана состоять из таких шагов, иначе проверка снова станет делом вкуса.

Разные системы по-разному распределяют нагрузку между аксиомами и правилами. На одном полюсе — системы Гильберта: много аксиом и одно правило. На другом — натуральная дедукция: аксиом нет вовсе, зато правил много, и каждое отражает один естественный ход рассуждения. Секвенциальное исчисление и резолюция — ещё две точки на этой шкале. Обзор всех четырёх — содержание ближайших разделов.

Замечание Доказательство как объект

Здесь различают три вещи: систему (правила и аксиомы), вывод (объект, построенный по правилам) и процесс поиска вывода. Проверка вывода — механическая и быстрая. Поиск вывода — творческая и трудная задача. Это разделение проходит через всю главу: научиться проверять доказательства просто, научиться их находить — ремесло, которому посвящён раздел о проверке и поиске.

§ 2.2 Деревья вывода

Такую запись придумал Герхард Гентцен в 1934 году, когда строил натуральную дедукцию, и называется она деревом вывода (*proof tree*). Применение правила рисуется чертой: над чертой — посылки, под чертой — заключение, справа — имя правила. Композиция применений даёт дерево: листья — аксиомы и гипотезы, внутренние узлы — применения правил, корень — заключение. Деревья — стандарт-

ное обозначение вывода, общее для всех систем, и с ними удобно начинать: каждое применение правила видно как отдельная черта, и структура вывода читается с одного взгляда. Простейший пример — цепочка из двух применений modus ponens, выводящая C из гипотез $A \rightarrow B$, A и $B \rightarrow C$. Она показана на рис. 2.

$$\frac{\frac{A \rightarrow B \quad A}{B} \text{MP} \quad B \rightarrow C}{C} \text{MP}$$

Рис. 2. Дерево вывода C из гипотез $A \rightarrow B$, A и $B \rightarrow C$.

Дерево строится от листьев к корню: верхнее применение MP (modus ponens) берёт гипотезы $A \rightarrow B$ и A и выводит B , а нижнее применение берёт полученный B и гипотезу $B \rightarrow C$ и выводит C . Подвывод, завершающийся на B , служит посылкой нижнему применению: заключения внутренних применений становятся посылками внешних.

ОПРЕДЕЛЕНИЕ 2.6 **Дерево вывода**

Дерево вывода формулы φ из гипотез Γ — конечное дерево, в котором:

1. листья — гипотезы из Γ или аксиомы;
2. каждый внутренний узел — применение правила вывода;
3. корень — φ .

В системах Гильберта выводы — цепочки, то есть деревья без ветвлений: каждая формула, кроме первых, получена из предыдущих по modus ponens. В натуральной дедукции, секвенциальном исчислении и резолюции деревья по-настоящему ветвятся: правила с двумя посылками раздваивают вывод, и поддереву одной ветви независимо от другой. Каждую из этих систем мы будем показывать её деревьями.

§ 2.3 Системы Гильберта

Идея систем Гильберта — сжать всё рассуждение до нескольких аксиом и одного правила: аксиом много, потому что именно в них зашита логика, а правило одно — modus ponens — потому что больше ничего не нужно.

Аксиомы задают схемами, а не по одной на формулу: шаблон, в который вместо букв можно подставлять любые формулы. Каждый экземпляр схемы, полученный подстановкой, — аксиома, и схема $A \rightarrow (B \rightarrow A)$ порождает аксиомы $P \rightarrow (Q \rightarrow P)$, $(P \rightarrow Q) \rightarrow (R \rightarrow (P \rightarrow Q))$ и бесконечно много других.

ОПРЕДЕЛЕНИЕ 2.7 **Схемы аксиом для импликации**

$$K : A \rightarrow (B \rightarrow A)$$

$$S : (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

Схема K утверждает: истинная формула следует из чего угодно — если A истинно, то A следует из любого B . Схема S — правило дистрибутивности импликации: она переставляет посылки. Вместе с *modus ponens* они порождают всю импликативную логику высказываний. Для полной логики добавляют схемы, описывающие отрицание.

Вывод в такой системе — цепочка формул, каждая из которых либо экземпляр схемы, либо результат *modus ponens* по двум предыдущим. Проверка тривиальна: для каждой строки достаточно сверить, аксиома ли это и откуда взяты две посылки.

Пример Вывод $A \rightarrow A$ в системе Гильберта

Казалось бы, $A \rightarrow A$ — простейшая тавтология. В системе Гильберта её вывод занимает пять строк.

1. $A \rightarrow ((A \rightarrow A) \rightarrow A)$ — схема K с подстановкой $B := A \rightarrow A$.
2. $(A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$ — схема S , где внешний A остаётся собой, а $B := A \rightarrow A$, $C := A$.
3. $(A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)$ — МР по строкам 1 и 2.
4. $A \rightarrow (A \rightarrow A)$ — схема K с подстановкой $B := A$.
5. $A \rightarrow A$ — МР по строкам 4 и 3.

Схема S работает «расщеплением»: она разбирает посылку $A \rightarrow ((A \rightarrow A) \rightarrow A)$ на два отдельных шага. Первый МР снимает внешнюю импликацию, второй — внутреннюю. Подстановки в схемы механические: буквы A, B, C заменяются формулами, и проверка каждой строки — сравнение с шаблоном.

Пять строк ради тривиальности — цена абстрактных схем. Каждый шаг разрешён, но увидеть за цепочкой схем знакомое утверждение трудно. Это и есть главный недостаток систем Гильберта: проверка проста, а *построение* вывода требует изобретательности, потому что каждый шаг — подстановка в абстрактную схему, а не содержательный ход.

Замечание Зачем нужна неудобная система

Неудобство систем Гильберта для построения выводов — плата за два удобства. Проверка каждого шага тривиальна, потому что правило одно. И доказывать утверждения о самой системе легко: вывод — цепочка, каждый шаг которой либо аксиома, либо *modus ponens*, и разбор сводится к этим двум случаям. Первое же такое утверждение — теорема о дедукции — появляется ниже и опирается ровно на эту бедность устройства.

Облегчает жизнь мета-теорема, впервые доказанная Жаком Эрбраном в 1930 году:

ТЕОРЕМА 2.1 Теорема о дедукции

$\Gamma, A \vdash B$ тогда и только тогда, когда $\Gamma \vdash A \rightarrow B$.

Набросок доказательства

Докажем эквивалентность в обе стороны.

($\leq$) Если $\Gamma \vdash A \rightarrow B$, то, добавив A к гипотезам, применяем МР к $A \rightarrow B$ и A и получаем B .

($\Rightarrow$) По выводу B из Γ, A построим вывод $A \rightarrow B$ из Γ индукцией по длине вывода. Если B — аксиома или гипотеза из Γ , то из B схемой K получаем $B \rightarrow (A \rightarrow B)$, и МР даёт $A \rightarrow B$. Если B — сама гипотеза A , то нужен вывод $A \rightarrow A$, показанный в примере выше. Если B получена МР из C и $C \rightarrow B$, то по индукции уже выведены $A \rightarrow C$ и $A \rightarrow (C \rightarrow B)$, и схема S склеивает их в $A \rightarrow B$.

Полученный вывод длиннее исходного — цена перевода, но он существует.

Теорема о дедукции оправдывает естественный приём: чтобы доказать $A \rightarrow B$, можно временно предположить A . Математики пользуются этим постоянно, и именно этот приём систематизирует натуральная дедукция — система, в которой временные предположения стали полноправным инструментом, а не скрытой подстановкой.

Проверка Системы Гильберта

1. Чем аксиома отличается от гипотезы, если обе входят в вывод без ссылок на предыдущие строки?
2. Почему проверка каждой строки тривиальна, а построение вывода требует изобретательности?
3. Что разрешает теорема о дедукции и почему в её доказательстве работают именно схемы K и S ?

§ 2.4 Натуральная дедукция

Системы Гильберта неудобны, потому что их шаги непохожи на человеческие рассуждения, и в 1934 году Яськовский и Гентцен независимо предложили систему, где правила соответствуют тому, как математики рассуждают на практике — натуральную дедукцию.

Идея в том, что для каждой связки есть два правила: ввод, говорящий, как доказать формулу с этой связкой, и устранение, говорящий, как использовать такую формулу. Аксиом нет вовсе: вместо них вывод начинается с гипотез, которые позже можно снять. Снять гипотезу — значит закончить рассуждение, которое на неё опиралось, и записать результат без неё, и ввод импликации делает ровно это: чтобы доказать $A \rightarrow B$, предполагаем A , выводим B , и гипотеза A больше не нужна — получили импликацию.

Все десять правил собраны на рис. 3. У каждой связки пара: ввод собирает формулу со связкой, устранение её разбирает. Снятая гипотеза помечается номером, например $[A]^1$, и правило ввода ссылается на этот номер: заключение перестаёт зависеть от A .

$$\begin{array}{c}
 \frac{[A]^1 \quad \vdots}{B} \rightarrow I \\
 \frac{A \rightarrow B \quad A}{B} \rightarrow E \\
 \\
 \frac{A \quad B}{A \wedge B} \wedge I \\
 \frac{A \wedge B}{A} \wedge E \\
 \\
 \frac{A}{A \vee B} \vee I \\
 \frac{A \vee B \quad \frac{[A]^1 \quad \vdots}{C} \quad \frac{[B]^2 \quad \vdots}{C}}{C} \vee E \\
 \\
 \frac{[A]^1 \quad \vdots}{\perp} \neg I \\
 \frac{A \quad \neg A}{\perp} \neg E \\
 \\
 \frac{\perp}{A} \perp E \\
 \frac{[\neg A]^1 \quad \vdots}{\perp} IP \\
 \frac{\perp}{A} IP
 \end{array}$$

Рис. 3. Правила натуральной дедукции: у каждой связки пара ввод/устранение, и гипотеза, снятая правилом ввода, помечается номером. Правило IP — классическое, о нём — отдельный раздел.

Выводы в натуральной дедукции записываются деревьями, как договорились в разделе 2.2: листья — гипотезы, корень — заключение, правила ввода и удаления — внутренние узлы. Посмотрим на простейший вывод — проекцию конъюнкции $A \wedge B \rightarrow A$, изображённую на рис. 4.

$$\frac{\frac{A \wedge B}{A} \wedge E}{A \wedge B \rightarrow A} \rightarrow I$$

Рис. 4. Дерево вывода проекции $A \wedge B \rightarrow A$.

Дерево читается от листа к корню: верхний узел — гипотеза $A \wedge B$, которую мы временно допускаем, а правило $\wedge E$ извлекает из конъюнкции левую половину A . Правило $\rightarrow I$ превращает подвывод «из $A \wedge B$ выведено A » в импликацию $A \wedge B \rightarrow A$ и снимает гипотезу: заключение уже не зависит ни от каких допущений. Посмотрим на вывод, где правил больше одного, — коммутативность конъюнкции $A \wedge B \rightarrow B \wedge A$ на рис. 5.

$$\frac{\frac{A \wedge B}{B} \wedge E \quad \frac{A \wedge B}{A} \wedge E}{B \wedge A} \wedge I$$

Рис. 5. Дерево вывода коммутативности конъюнкции: два устранения и один ввод.

Каждое применение $\wedge E$ извлекает одну половину посылки, и $\wedge I$ собирает их в обратном порядке. Обе ветви используют одну и ту же гипотезу $A \wedge B$ — лист дерева не обязан быть единственным.

Деревья наглядны, но для длинных выводов громоздки: ветви разъезжаются по странице, повторное использование одной формулы требует копирования целого поддерева, а вынести промежуточный результат в лемму нельзя — лемма вклеивается в дерево целиком. Яськовский придумал линейную запись тех же выводов — колонку пронумерованных строк с вложенными поддоказательствами, которую называют нотацией Фитча. В линейной записи цепочки шагов идут подряд, повторное использование формулы — ссылка на её номер вместо копирования поддерева, а лемма — отдельный вывод, на который ссылаются как на посылку. Поэтому в натуральной дедукции дальше мы работаем в основном в нотации Фитча, а деревья остаются наглядным обозначением и рабочим инструментом секвенций и резолюции.

§ 2.5 Нотация Фитча

Нотация Фитча — способ записи деревьев натуральной дедукции, а не отдельная система: доказательство — вертикальная колонка пронумерованных строк, каждая из которых либо гипотеза, либо формула, выведенная по правилу из предыдущих строк с указанием правила и ссылок. Временное предположение открывает поддоказательство — строки, выделенные отступом и вертикальной чертой слева. Когда поддоказательство завершено, его гипотеза снимается: извне на неё ссылаться нельзя, и она не входит в число допущений заключения. Правило R (реитерация) разрешает повторить формулу из внешнего поддоказательства: лист дерева, который мы использовали дважды, в линейной записи становится ссылкой на номер строки.

Пример Доказательство $A \rightarrow (B \rightarrow A)$

1	A	<i>Assumption</i>
2	B	<i>Assumption</i>
3	A	<i>R 1</i>
4	$B \rightarrow A$	$\rightarrow I$ 2-3
5	$A \rightarrow (B \rightarrow A)$	$\rightarrow I$ 1-4

Читаем снизу вверх по цели, сверху вниз по построению. Цель — импликация, поэтому ввод импликации: предполагаем A (строка 1) и ставим задачу вывести $B \rightarrow A$. Новая цель — снова импликация: предполагаем B (строка 2). Формула A (строка 3) — повторение гипотезы строки 1: она доступна внутри вложенного поддоказательства. Строка 4 снимает гипотезу B , строка 5 — гипотезу A . Заключение не зависит ни от одной гипотезы: это теорема.

Ссылки вида «2-3» означают диапазон строк поддоказательства. Правило $\rightarrow I$ ссылается на поддоказательство целиком: его гипотеза становится антецедентом, последняя строка — консеквентом.

§ 2.6 Правила для связок

Правила делятся на вводящие и устранивающие, и для каждой связки есть пара. Разберём их по очереди.

2.6.1 Конъюнкция

Конъюнкция — самая простая пара: ввод собирает две формулы, устранение разбирает одну.

Пример Ввод и устранение конъюнкции

1	$P \wedge Q$	<i>Premise</i>
2	P	$\wedge E$ 1
3	Q	$\wedge E$ 1
4	$Q \wedge P$	$\wedge I$ 3, 2

Правило $\wedge I$ из A и B выводит $A \wedge B$. Правило $\wedge E$ из $A \wedge B$ извлекает A или B по выбору. Здесь мы извлекли обе половины и собрали их в обратном порядке: доказана коммутативность конъюнкции — $P \wedge Q$ влечёт $Q \wedge P$. Ссылки строки 4 перечислены в том порядке, в каком формулы входят в заключение: сначала Q (строка 3), затем P (строка 2).

Устранение конъюнкции не извлекает ничего нового: всё уже лежит в посылке, а ценность $\wedge E$ в другом — он делает доступными части, с которыми работают дальнейшие правила.

2.6.2 Импликация

Ввод импликации мы уже видели: он снимает гипотезу. Устранение — *modus ponens*: из импликации и её антецедента выводится консеквент.

Пример Modus ponens (MP)

1	$P \rightarrow Q$	<i>Premise</i>
2	P	<i>Premise</i>
3	Q	$\rightarrow E$ 1, 2

Правило $\rightarrow E$ требует двух посылок: импликации и её антецедента. Порядок ссылок произволен: строка 1 — импликация, строка 2 — антецедент. Заключение зависит от обеих посылок. Ни одна гипотеза не снималась.

Пара $\rightarrow I$ и $\rightarrow E$ — сердце натуральной дедукции: ввод соответствует «рассуждению с допущением», устранение — применению уже доказанного следствия. Вместе они точно воспроизводят теорему о дедукции, но без подстановок в схемы: временное предположение становится видимой частью доказательства.

2.6.3 Дизъюнкция

Ввод дизъюнкции тривиален: из A можно вывести $A \vee B$ с любым вторым членом. Устранение — правило разбора случаев: дизъюнкция говорит «истинно одно из двух», и доказательство обязано покрыть оба случая.

Пример Ввод и устранение дизъюнкции (разбор случаев)

1	$P \vee Q$	<i>Premise</i>
2	P	<i>Assumption</i>
3	$Q \vee P$	$\vee I$ 2
4	Q	<i>Assumption</i>
5	$Q \vee P$	$\vee I$ 4
6	$Q \vee P$	$\vee E$ 1, 2-3, 4-5

Правило $\vee E$ ссылается на три вещи: дизъюнкцию (строка 1) и два поддоказательства — по одному на каждый случай. В первой ветви предполагаем P и вводом дизъюнкции получаем $Q \vee P$ (строка 3). Во второй ветви предполагаем Q и получаем ту же формулу (строка 5). Правило $\vee E$ снимает обе гипотезы: заключение не зависит от того, какая ветвь истинна.

Разбор случаев — единственное правило, которое работает с двумя поддоказательствами сразу, и именно оно формализует рассуждение «в любом случае»: что бы ни было истинно в дизъюнкции, результат одинаков.

2.6.4 Отрицание и противоречие

Отрицание определяется через противоречие: $\neg A$ означает, что из допущения A следует всё, включая ложь. Противоречие $\perp$ — формула, которая всегда ложна, и в системе она служит маркером невозможности.

Пример Ввод и устранение отрицания

1	$A \wedge \neg A$	<i>Assumption</i>
2	A	$\wedge E\ 1$
3	$\neg A$	$\wedge E\ 1$
4	$\perp$	$\neg E\ 3, 2$
5	$\neg(A \wedge \neg A)$	$\neg I\ 1-4$

Правило $\neg E$ из A и $\neg A$ выводит противоречие $\perp$. Правило $\neg I$: если поддоказательство из гипотезы A заканчивается на $\perp$, выводится $\neg A$, и гипотеза снимается. Здесь из допущения $A \wedge \neg A$ извлекаются обе половины (строки 2 и 3), и $\neg E$ даёт противоречие (строка 4). Ввод отрицания (строка 5) закрывает поддоказательство: $\neg(A \wedge \neg A)$ — закон непротиворечия, теорема.

В правилах для отрицания есть асимметрия: $\neg I$ *вводит* отрицание, снимая гипотезу A . Правила, которые *устраняют* отрицание, — косвенное доказательство — уже классические, и до них мы дойдём.

2.6.5 Взрыв

Из противоречия следует всё, и правило, которое это разрешает, выглядит самым странным в системе: произвольная формула выводится из лжи.

Пример Взрыв: из $\perp$ следует всё

1	$\perp$	<i>Premise</i>
2	A	$X\ 1$

Правило $\perp E$ (взрыв, ex falso quodlibet), в нотации Фитча помечаемое буквой X , из $\perp$ выводит любую формулу. Оно нужно, чтобы закрывать невозможные ветви рассуждения: если допущение привело к противоречию, эта ветвь описывает невозможный случай, и из неё можно вывести что угодно. В разборе случаев такая ветвь обязана закончиться на нужном заключении — взрыв доставляет его из ничего.

Взрыв — интуиционистски корректное правило: посылка $\perp$ никогда не бывает истинной, поэтому из неё следует любая формула. Интуиционистская логика принимает взрыв, а снятие двойного отрицания отвергает — об этом следующий раздел.

§ 2.7 Косвенное доказательство и классические правила

Все правила до сих пор конструктивны: каждое показывает, как получить заключение, а ввод отрицания из противоречия даёт только отрицание. Для положительной формулы классическая логика добавляет правило, которое выводит её напрямую:

Пример Косвенное доказательство и снятие двойного отрицания

1	$\neg\neg A$	<i>Premise</i>
2	$\neg A$	<i>Assumption</i>
3	$\perp$	$\neg E$ 1, 2
4	A	<i>IP</i> 2-3

Правило IP (косвенное доказательство, *reductio ad absurdum*): если поддоказательство из гипотезы $\neg A$ заканчивается на $\perp$, выводится A . Гипотеза $\neg A$ снимается. Здесь из $\neg\neg A$ выводится A — снятие двойного отрицания.

Косвенное доказательство не говорит, почему A истинно: оно говорит, что предположение « A ложно» ведёт к противоречию, а значит, A истинно. Это рассуждение об исключённой возможности, а не о построении.

Пример Закон исключённого третьего

1	$\neg(A \vee \neg A)$	<i>Assumption</i>
2	A	<i>Assumption</i>
3	$A \vee \neg A$	$\vee I$ 2
4	$\perp$	$\neg E$ 1, 3
5	$\neg A$	$\neg I$ 2-4
6	$A \vee \neg A$	$\vee I$ 5
7	$\perp$	$\neg E$ 1, 6
8	$A \vee \neg A$	<i>IP</i> 1-7

Закон исключённого третьего $A \vee \neg A$ — классическая тавтология, но доказываемая только через косвенное доказательство. Сначала из предположения о ложности цели выводится $\neg A$ (строка 5) — это интуиционистская часть. Но из $\neg A$ тут же получается $A \vee \neg A$ (строка 6), что противоречит предположению, и косвенное доказательство (строка 8) снимает его. Весь вывод держится на одном классическом шаге — строке 8.

Три классических принципа — косвенное доказательство IP, снятие двойного отрицания ($\neg\neg A \rightarrow A$) и закон исключённого третьего ($A \vee \neg A$) — эквивалентны по силе: каждый выводится из любого другого. Четвёртый — закон Пирса — появится ниже и тоже эквивалентен им. Ни один не имеет места в интуиционистской логике, которой посвящена глава 34. Интуиционистские правила — всё, что было до IP: они не доказывают $A \vee \neg A$, потому что не умеют рассуждать об исключённой возможности.

§ 2.8 Производные правила

Частые ходы оформляют как производные правила: они выводимы из базовых, но ссылаться на них короче.

- *Дизъюнктивный силлогизм (DS)*: из $A \vee B$ и $\neg A$ выводится B .
- *Modus tollens (MT)*: из $A \rightarrow B$ и $\neg B$ выводится $\neg A$.
- *Гипотетический силлогизм (HS)*: из $A \rightarrow B$ и $B \rightarrow C$ выводится $A \rightarrow C$.
- *Контрапозиция*: из $A \rightarrow B$ выводится $\neg B \rightarrow \neg A$.
- *Законы де Моргана*: $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$ и $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$. Одно направление первого закона ($\neg(A \wedge B) \rightarrow \neg A \vee \neg B$) требует косвенного доказательства, остальное выводимо интуиционистски.

Производное правило — сокращение, а не новое допущение: его можно развернуть в базовые шаги. Поэтому доказательства, использующие производные правила, остаются проверяемыми — верификатор разворачивает их при желании.

Пример Дизъюнктивный силлогизм через взрыв

1	$P \vee Q$	<i>Premise</i>
2	$\neg P$	<i>Premise</i>
3	P	<i>Assumption</i>
4	$\perp$	$\neg E$ 2, 3
5	Q	X 4
6	Q	<i>Assumption</i>
7	Q	R 6
8	Q	$\vee E$ 1, 3-5, 6-7

Первая ветвь разбора случаев (P) противоречитсылке $\neg P$, поэтому закрывается взрывом: из $\perp$ выводится Q . Вторая ветвь (Q) заканчивается на Q повторением. Разбор случаев собирает результат: в любом случае Q .

Пример Modus tollens (MT)

Из посылок $A \rightarrow B$ и $\neg B$ выведем $\neg A$ без классических шагов.

Предположим A (гипотеза для $\neg I$). МР из $A \rightarrow B$ и A даёт B . Противоречие с посылкой $\neg B$ правилом $\neg E$ даёт $\perp$. Ввод отрицания снимает гипотезу A : получаем $\neg A$.

Вывод в нотации Фитча:

1	$A \rightarrow B$	<i>Premise</i>
2	$\neg B$	<i>Premise</i>
3	A	<i>Assumption</i>
4	B	$\rightarrow E$ 1, 3
5	$\perp$	$\neg E$ 2, 4
6	$\neg A$	$\neg I$ 3-5

Modus tollens интуиционистски допустим: он строится вводом отрицания, без косвенного доказательства. Обратное направление — из $\neg B \rightarrow \neg A$ вывести $A \rightarrow B$ — классическое.

Законы де Моргана и контрапозиция доказываются тем же ремеслом: разбор случаев для дизъюнкции, ввод отрицания для отрицания. Производные правила не меняют силу системы, но меняют удобство: они превращают многошаговые конструкции в один ход, который можно развернуть при проверке.

§ 2.9 Нормализация

Правило ввода каждой связки зеркально её правилу устранения. Правило ввода говорит, как построить формулу со связкой, а правило устранения — как извлечь из неё части. В аккуратном доказательстве эти два шага не дублируют друг друга.

Замечание Почему у каждой связки ровно пара правил?

Гентцен предлагал читать пару правил так: правило ввода задаёт смысл связки, перечисляя способы доказать формулу с ней. Устранение — обратный ход: оно извлекает из формулы ровно то, что ввод принял как условия. Конъюнкция вводится из двух готовых половин, поэтому её устранение вправе требовать ровно эти половин и ничего сверх. Праувиц позже сформулировал это как принцип инверсии: устранение оправдано, если оно восстанавливает то, на что опирался ввод.

Число правил существенно: без устранения связку нельзя применить, а лишнее устранение доказывает больше, чем допускает заданный вводом смысл.

Дублирование возникает, когда формула вводится и тут же устраняется. Конъюнкция $A \wedge B$ собирается правилом $\wedge I$, а следующей строкой разбирается правилом $\wedge E$ обратно на A и B . Такой фрагмент — обходной путь, *локальный максимум*, и он ничего не добавляет, ведь A и B уже были в выводе.

Пример Локальный максимум

1	A	<i>Premise</i>
2	B	<i>Premise</i>
3	$A \wedge B$	$\wedge I$ 1, 2
4	A	$\wedge E$ 3

Строка 3 вводит конъюнкцию, строка 4 тут же её устраняет, и весь вывод сводится к строке 1.

Удаление всех локальных максимумов даёт *нормальный* вывод, в котором ни одна формула не вводится, чтобы немедленно быть устранённой той же связкой. Нормальный вывод составной формулы оканчивается правилом введения её главной связки, а не правилом устранения, и обладает свойством подформул — каждая его формула есть подформула заключения или открытой гипотезы. Из свойства подформул следует непротиворечивость — $\perp$ не вводится никаким правилом введения и не является гипотезой, поэтому в нормальном выводе из пустых гипотез он появиться не может.

Нормализация — двойник устранения сечений, к которому мы перейдём в следующем разделе. Там лишнюю формулу, внесённую сечением, убирает устранение сечений, здесь — пара правил ввода и устранения. В классической логике нормализация проходит не так гладко, и локальный максимум с косвенным доказательством не всегда устраняется. Это различие отделяет интуиционистскую логику от классической (глава 34).

Проверка Связки и классика

1. Как ввод импликации соотносится с теоремой о дедукции, и что именно он снимает?
2. Почему закон исключённого третьего не выводится без косвенного доказательства, хотя ввод отрицания уже есть?
3. Дизъюнктивный силлогизм выводится через взрыв — какой его шаг опирается на противоречие?

§ 2.10 Секвенциальное исчисление Гентцена

Вторая форма, предложенная Гентценом, — секвенциальное исчисление. Его единица — не формула, а секвенция $\Gamma \vdash \Delta$.

ОПРЕДЕЛЕНИЕ 2.8 Секвенция

Секвенция $\Gamma \vdash \Delta$ — это пара конечных списков формул, разделённых турникетом. Она читается так: из всех формул списка Γ следует хотя бы одна формула списка Δ .

Классическое исчисление LK разрешает справа несколько формул: $\Gamma \vdash \Delta$ истинна, когда из конъюнкции формул Γ следует дизъюнкция формул Δ , а интуиционистское исчисление LJ разрешает справа ровно одну формулу.

Разница между LK и LJ — не косметическая: возможность держать справа несколько формул и есть источник классической силы. Секвенция $\vdash A, \neg A$ утверждает «истинно A или истинно $\neg A$ » — закон исключённого третьего, записанный без единого правила. В LJ такой записи нет: справа место лишь для одной формулы.

Правила исчисления — вводы слева и справа для каждой связки: правило ввода справа показывает, как доказать формулу с данной связкой, правило ввода слева — как использовать её как гипотезу. Аксиома одна: $\Gamma, A \vdash A, \Delta$ — формула выводится из самой себя.

Аксиома	$\Gamma, A \vdash A, \Delta$
$\rightarrow R$	из $\Gamma, A \vdash B, \Delta$ выводится $\Gamma \vdash A \rightarrow B, \Delta$
$\rightarrow L$	из $\Gamma \vdash A, \Delta$ и $\Gamma, B \vdash \Delta$ выводится $\Gamma, A \rightarrow B \vdash \Delta$
$\neg R$	из $\Gamma, A \vdash \Delta$ выводится $\Gamma \vdash \neg A, \Delta$
$\neg L$	из $\Gamma \vdash A, \Delta$ выводится $\Gamma, \neg A \vdash \Delta$
$\vee R$	из $\Gamma \vdash A, \Delta$ выводится $\Gamma \vdash A \vee B, \Delta$; то же с B
$\vee L$	из $\Gamma, A \vdash \Delta$ и $\Gamma, B \vdash \Delta$ выводится $\Gamma, A \vee B \vdash \Delta$
$\wedge R$	из $\Gamma \vdash A, \Delta$ и $\Gamma \vdash B, \Delta$ выводится $\Gamma \vdash A \wedge B, \Delta$
$\wedge L$	из $\Gamma, A \vdash \Delta$ выводится $\Gamma, A \wedge B \vdash \Delta$; то же с B
контракция	из $\Gamma \vdash A, A, \Delta$ выводится $\Gamma \vdash A, \Delta$; то же слева
сечение	из $\Gamma \vdash A, \Delta$ и $\Gamma, A \vdash \Delta$ выводится $\Gamma \vdash \Delta$

Примечание Структурные правила

Три правила не разбирают связки, а переставляют формулы: ослабление добавляет лишнюю формулу в Γ или Δ , обмен меняет порядок, контракция склеивает две одинаковые. В таблице выписана только контракция: ослабление и обмен подразумеваются, их применение механично и не меняет смысла секвенции. Контракция же существенна — закон исключённого третьего выводится ровно потому, что два одинаковых члена справа можно склеить.

Структура правил — как в разделе о деревьях вывода 2.2: посылки над чертой, заключение под ней, имя правила сбоку. Вывод — дерево, растущее от целевой секвенции вниз к аксиомам, и каждое правило разбирает ровно одну связку в одной формуле, поэтому дерево вывода показывает всю структуру доказательства.

Пример Вывод $A \rightarrow (B \rightarrow A)$ в ЛК

Дерево вывода растёт от целевой секвенции к аксиоме, и каждый шаг разбирает главную связку.

1. $A, B \vdash A$ — аксиома.
2. $A \vdash B \rightarrow A$ — правило $\rightarrow R$: формула B перенесена в антецедент импликации.
3. $\vdash A \rightarrow (B \rightarrow A)$ — снова $\rightarrow R$.

Два применения одного правила — и теорема доказана. В натуральной дедукции тот же вывод требовал двух вложенных поддоказательств. Здесь снятие гипотезы становится переносом формулы через турникет.

Пример Закон исключённого третьего в ЛК

Секвенция $\vdash A \vee \neg A$ выводится пятью шагами — дерево секвенций на рис. 6.

$$\begin{array}{c}
 \frac{A \vdash A}{\vdash \neg A, A} \neg R \\
 \frac{\vdash \neg A, A}{\vdash \neg A, A \vee \neg A} \vee R_1 \\
 \frac{\vdash \neg A, A \vee \neg A}{\vdash A \vee \neg A, A \vee \neg A} \vee R_2 \\
 \frac{\vdash A \vee \neg A, A \vee \neg A}{\vdash A \vee \neg A} \text{контракция}
 \end{array}$$

Рис. 6. Дерево секвенций для закона исключённого третьего.

1. $A \vdash A$ — аксиома.
2. $\vdash \neg A, A$ — правило $\neg R$: формула A перенесена направо с отрицанием.
3. $\vdash \neg A, A \vee \neg A$ — правило $\vee R$, заменяем правый член A на $A \vee \neg A$.
4. $\vdash A \vee \neg A, A \vee \neg A$ — правило $\vee R$, заменяем $\neg A$ на $A \vee \neg A$.
5. $\vdash A \vee \neg A$ — контракция справа: два одинаковых члена склеены в один.

Классический шаг — строка 2: правило $\neg R$ даёт справа две формулы, а в LJ справа место лишь для одной. В интуиционистском LJ он недоступен, и закон исключённого третьего в LJ не выводится — в точности как в натуральной дедукции без IP.

Пример Ветвящееся дерево: коммутативность дизъюнкции

Правила с двумя посылками — $\vee L, \wedge R$, сечение — ветвят дерево: над чертой стоят два поддеревя. Такое дерево возникает при выводе коммутативности дизъюнкции $A \vee B \vdash B \vee A$, показанном на рис. 7.

$$\frac{\frac{A \vdash A}{A \vdash B \vee A} \vee R_2 \quad \frac{B \vdash B}{B \vdash B \vee A} \vee R_1}{A \vee B \vdash B \vee A} \vee L$$

Рис. 7. Дерево секвенций для коммутативности дизъюнкции $A \vee B \vdash B \vee A$.

Левая ветвь: из аксиомы $A \vdash A$ вводом правого дизъюнкта получаем $A \vdash B \vee A$. Правая ветвь: из $B \vdash B$ получаем $B \vdash B \vee A$. Правило $\vee L$ собирает ветви: заключение не зависит от того, какой дизъюнкт истинен — ровно как разбор случаев в натуральной дедукции.

Секвенциальное исчисление окупается главным результатом теории доказательств: Гентцен доказал для него теорему об устранении сечений.

ТЕОРЕМА 2.2 Устранение сечений (Гентцен)

Всякий вывод в ЛК можно перестроить так, чтобы сечение не использовалось вовсе.

Набросок доказательства

Докажем перестройкой вывода.

Спуск. Если формула A , по которой проведено сечение, вводится в одной из посылок не своим главным правилом, переставляем это правило вниз, мимо сечения, пока A не окажется главной с обеих сторон.

Главный случай. Если A вводится главным правилом в обеих посылках, разбираем по её главной связке: сечение по конъюнкции $B \wedge C$ заменяется двумя сечениями — по B и по C .

Аксиома. Если одна из посылок — аксиома, сечение исчезает.

Каждый шаг либо убирает сечение, либо заменяет его сечениями по формулам меньшей сложности, поэтому перестройка завершается бессекционным выводом.

Правило сечения — транзитивность вывода: если A выводится из Γ , а Δ — из Γ и A , то Δ выводится из Γ . Оно удобно, но вносит в вывод формулу, которая потом исчезает — как промежуточная лемма, и устранение сечений показывает: без таких лемм можно обойтись. Следствие — свойство подформул: в бессекционном выводе каждая формула — подформула целевой секвенции, и доказательства становятся локальными — каждый шаг виден, ничего не спрятано в скрытой лемме. На свойстве подформул держится и непротиворечивость — у аксиомы и у заключения каждого правила хотя бы одна сторона непуста, поэтому пустая секвенция $\vdash$ бессекционного вывода не имеет.

Деревья секвенций и деревья натуральной дедукции — две формы одного явления: Гентцен показал, что системы эквивалентны — выводимость в одной совпадает с выводимостью в другой. Секвенциальное исчисление удобнее для анализа доказательств, натуральная дедукция — для их построения.

Проверка Секвенции

1. Чем секвенция $\Gamma \vdash A$ отличается от секвенции $\Gamma \vdash A, B$, и почему вторая классична?
2. Какое правило разбирает связку слева, а какое — справа?
3. Что устраняет теорема об устранении сечений, и какое свойство из неё следует?

§ 2.11 Резолюция

Четвёртая из разобранных систем — резолюция — спроектирована для машинного поиска, а не для человеческого чтения. Её язык беден намеренно: формула приводится к конъюнктивной нормальной форме (КНФ), то есть к конъюнкции клауз, а правило вывода одно.

ОПРЕДЕЛЕНИЕ 2.9 Литерал

Литерал — это атом или отрицание атома.

ОПРЕДЕЛЕНИЕ 2.10 Клауза

Клауза — это дизъюнкция литералов.

ОПРЕДЕЛЕНИЕ 2.11 Правило резолюции

Из клауз $C_1 \vee L$ и $C_2 \vee \neg L$ выводится их резольвента $C_1 \vee C_2$:

$$\frac{C_1 \vee L \quad C_2 \vee \neg L}{C_1 \vee C_2}$$

Литерал L и его отрицание $\neg L$ образуют *контрарную пару*: в резольvente они сокращаются, и остаются только C_1 и C_2 .

ОПРЕДЕЛЕНИЕ 2.12 Пустой дизъюнкт

Пустой дизъюнкт $\square$ — символ противоречия.

Если обе клаузы состояли только из контрарной пары, резольвента пуста, и её обозначают пустым дизъюнктом $\square$.

Правило опирается на разбор случаев: клауза $C_1 \vee L$ истинна, если истинен L , а клауза $C_2 \vee \neg L$ истинна, если L ложен, поэтому в любом случае хотя бы одна из остатков C_1, C_2 обязана быть истинной. Резольвента — логическое следствие посылок: если посылки выполнимы, выполнима и резольвента.

Пример Один шаг резолюции

Резолюция работает с клаузами $P \vee Q$ и $\neg P \vee R$. Контрарная пара — P и $\neg P$. Резольвента — $Q \vee R$. Если P истинно, то из второй клаузы верен R . Если P ложно, то из первой верен Q . В обоих случаях хотя бы один из литералов Q, R истинен.

Для логики высказываний этого достаточно: резолюция полна в смысле опровержения. Множество клауз невыполнимо тогда и только тогда, когда из него резолюцией выводится пустой дизъюнкт $\square$. Теорема о корректности и полноте резолюции с наброском доказательства приведена в главе 14, где резолюция работает движком решения задач.

Опровержение — способ проверки невыполнимости: привести формулу к КНФ, добавить отрицание проверяемого следствия и резольвировать до пустого дизъюнкта. Пустой дизъюнкт означает противоречие, а противоречие — что исходное множество выполнить нельзя.

Пример Опровержение: транзитивность импликации

Проверим, что из $P \rightarrow Q$ и $Q \rightarrow R$ следует $P \rightarrow R$. Клаузы посылка: $\neg P \vee Q$ и $\neg Q \vee R$. Отрицание следствия: $\neg(\neg P \vee R)$, то есть P и $\neg R$. Резолюция:

1. $\neg P \vee Q$ — клауза 1.
2. $\neg Q \vee R$ — клауза 2.
3. P — клауза 3.
4. $\neg R$ — клауза 4.
5. Q — резольвента клауз 1 и 3 по P .
6. R — резольвента клауз 2 и 5 по Q .
7. $\square$ — резольвента клауз 4 и 6 по R .

То же опровержение — деревом на рис. 8: листья — клаузы, каждый узел — шаг резолюции, корень — пустой дизъюнкт.

$$\begin{array}{c}
 \frac{\neg P \vee Q \quad P}{Q} \text{резолюция по } P \qquad \frac{\neg Q \vee R}{R} \text{резолюция по } Q \\
 \frac{Q \quad R}{\square} \text{резолюция по } R
 \end{array}$$

Рис. 8. Дерево опровержения резолюцией.

Пустой дизъюнкт выведен — значит, отрицание следствия невыполнимо вместе с посылками, и следствие выводимо.

Резолюция минимальна — одно правило вместо пары на связку, — но платит за это формой: формула обязана быть в КНФ, а доказательство превращается в механический перебор резольвент. Именно перебор делает резолюцию пригодной для машин: стратегии поиска, ограничивающие перебор, — предмет главы о выполнении 14. Для логики предикатов правило резолюции дополняется *унификацией* — подстановкой, делающей два литерала контрарными.

Пример Резолюция с унификацией

Даны клаузы $P(a)$ и $\neg P(x) \vee Q(x)$. Литералы $P(a)$ и $\neg P(x)$ станут контрарными после подстановки $x := a$, которую называют унификатором. Применив подстановку ко второй клаузе и резольвируя по $P(a)$, получаем $Q(a)$.

Унификация ищет такую подстановку механически, и именно она делает резолюцию методом машинного доказательства в логике предикатов. Полное изложение — в главе 14.

§ 2.12 Проверка и поиск доказательств

У всех четырёх систем общее устройство: вывод — конечный объект, и его проверка — механическая процедура, линейная по размеру вывода — каждая строка сверяется с правилом за постоянное время. Именно это свойство делает формальные доказательства проверяемыми машиной, и именно на нём стоит проект главы — верификатор доказательств.

Поиск доказательства — другая задача, и она принципиально труднее проверки: дерево вывода ветвится на каждом шаге, ведь правила можно применять к разным формулам в разном порядке. Перебор всех возможных выводов экспоненциален, и вопрос о том, можно ли найти доказательство существенно быстрее — это вопрос о классах сложности (глава 30).

Перебор устроен как поиск с возвратом (backtracking): решатель применяет к текущей цели правило, спускается к подцелям, а в тупике снимает последний ход и пробует другой. Такой поиск полон: существующее доказательство он рано или поздно находит. Тот же каркас — перебор с возвратом — лежит в основе DPLL (от англ. Davis–Putnam–Logemann–Loveland) из главы 14. Практика обходит перебор эвристиками: резолюция с ограничениями, обратный поиск от цели, стратегии, к которым мы сейчас перейдём.

Поиск доказательства — работа с формой цели и посылок, а не перебор вслепую. Главный ориентир — главная связка цели: правило, которое её вводит, почти всегда первый ход.

Если цель — импликация $A \rightarrow B$, ввод импликации диктует ход: предположить A и доказывать B . Если цель — конъюнкция $A \wedge B$, её доказывают по частям: отдельно A , отдельно B . Если цель — отрицание $\neg A$, ввод отрицания предлагает

предположить A и вывести противоречие. Эти три хода работают *назад*, от цели: они разбирают цель на подцели, пока не останутся элементарные утверждения.

Работа *вперёд*, от посылок, использует правила устранения. Из $A \wedge B$ извлекают части, из $A \rightarrow B$ и A применяют МР, из $A \vee B$ готовят разбор случаев. Устранение не выбирает цель — оно накапливает материал, из которого цель может собраться.

Порядок работы:

1. Сначала назад от цели: вводы $\rightarrow$, $\wedge$, $\neg$, $\forall$.
2. Затем вперёд от посылок: устранения $\vee$, $\exists$.
3. Косвенное доказательство — последнее средство: если прямые ходы не сходятся, предположить отрицание цели и искать противоречие.

Такой порядок не произволен: вводы разбирают цель на меньшие куски, и каждый кусок доказывается проще, а устранения дизъюнкции и существования ветвят доказательство — их применяют, когда ветвление неизбежно. Косвенное доказательство меняет цель целиком — это сильный, но слепой ход, и к нему прибегают, когда конструктивный путь не виден.

Пример Контрапозиция: стратегия разбора цели

Цель — вывести $\neg B \rightarrow \neg A$ из посылки $A \rightarrow B$. Главная связка цели — импликация, поэтому ввод импликации: предполагаем $\neg B$ (строка 2) и доказываем $\neg A$. Новая цель — отрицание: ввод отрицания, предполагаем A (строка 3) и ищем противоречие. МР из посылки и A даёт B , что противоречит $\neg B$.

1	$A \rightarrow B$	<i>Premise</i>
2	$\neg B$	<i>Assumption</i>
3	A	<i>Assumption</i>
4	B	$\rightarrow E$ 1, 3
5	$\perp$	$\neg E$ 2, 4
6	$\neg A$	$\neg I$ 3-5
7	$\neg B \rightarrow \neg A$	$\rightarrow I$ 2-6

Стратегия отработала без единого отступления: каждый ход продиктован главной связкой текущей цели. Два вложенных ввода — импликации и отрицания — соответствуют двум уровням цели.

Пример Закон Пирса: когда без косвенного доказательства не обойтись

Закон Пирса $((P \rightarrow Q) \rightarrow P) \rightarrow P$ — классическая тавтология. Цель — импликация: предполагаем $(P \rightarrow Q) \rightarrow P$ (строка 1) и доказываем P . Посылка — импликация с антецедентом $P \rightarrow Q$. Чтобы применить МР и получить P , нужен

$P \rightarrow Q$. Его и добываем классически: предполагаем $\neg P$ и выводим противоречие.

1	$(P \rightarrow Q) \rightarrow P$	<i>Assumption</i>
2	$\neg P$	<i>Assumption</i>
3	P	<i>Assumption</i>
4	$\perp$	$\neg E$ 2, 3
5	Q	X 4
6	$P \rightarrow Q$	$\rightarrow I$ 3-5
7	P	$\rightarrow E$ 1, 6
8	$\perp$	$\neg E$ 2, 7
9	P	IP 2-8
10	$((P \rightarrow Q) \rightarrow P) \rightarrow P$	$\rightarrow I$ 1-9

Ход, на котором держится вывод — строка 6: из гипотезы P , противоречащей $\neg P$, через взрыв выводится произвольная Q , и ввод импликации даёт $P \rightarrow Q$. Строка 7 применяет посылку, строка 8 фиксирует противоречие, и косвенное доказательство (строка 9) снимает гипотезу $\neg P$. Классический шаг — ровно строка 9: без IP закон Пирса не выводим.

Разделение проверки и поиска — концептуальный сдвиг, а не техническая деталь: проверяемость делает доказательство общественным благом, ведь его может проверить любой, не доверяя автору. Поиск остаётся искусством — но искусством, вооружённым механической проверкой, и так устроены современные доказательные ассистенты: человек строит доказательство, машина проверяет каждый шаг.

Разделение проверки и поиска имеет и более глубокую форму. Если доказательство — объект, проверяемый по правилам, его можно читать как программу, а проверку — как типизацию: правило вывода превращается в правило типизации, и доказательство формулы A становится программой типа A . Это соответствие Карри–Ховарда связывает системы вывода с лямбда-исчислением (глава 27) и теорией типов (глава 28) и вернётся в главе об интуиционизме (глава 34). Стратегии работают и для кванторов: правила для $\forall$ и $\exists$ — в следующем разделе, и общий принцип тот же — главная связка цели выбирает правило.

§ 2.13 Кванторы в натуральной дедукции

Натуральная дедукция расширяется с высказываний на логику предикатов четырьмя правилами: пара ввод/устранение для каждого из кванторов $\forall$ и $\exists$, а нотация

Фитча и все правила для связок остаются прежними — меняется только набор допустимых шагов. Запись $\varphi(t)$ означает результат подстановки терма t вместо переменной x в формулу φ , то есть $\varphi[x := t]$, где терм — это переменная или константа предметной области, а подстановка должна быть свободной: без захвата переменной квантором.

2.13.1 Универсальное устранение

ОПРЕДЕЛЕНИЕ 2.13 Универсальное устранение ($\forall E$)

$\forall x \varphi(x) \vdash \varphi(t)$ для любого терма t :

$$\frac{\forall x \varphi(x)}{\varphi(t)}$$

Терм t подставляется вместо x свободно: ни одна переменная терма не должна оказаться связанной квантором внутри φ .

Если свойство выполнено для всех объектов, оно выполнено и для любого конкретного объекта, и ограничений на выбор t , кроме свободы подстановки, нет.

Пример Универсальное устранение

1	$\forall x P(x)$	<i>Premise</i>
2	$P(a)$	$\forall E 1$

Из «все обладают свойством P » следует «конкретный объект a обладает свойством P ». На место t можно подставить любой терм: константу, переменную, сложный терм.

Пример Захват переменной при подстановке

Применять универсальное устранение можно не с любым термом. Пусть $P(x, y)$ означает « x меньше y ». Посылка $\forall x \exists y P(x, y)$ истинна на натуральных числах: у каждого числа есть большее. Подстановка терма y вместо x даёт $\exists y P(y, y)$ — «существует число, меньшее самого себя». Такое заключение ложно, хотя посылка истинна: переменная терма попала под квантор существования и была им захвачена. Правило требует свободной подстановки: ни одна переменная терма не должна оказаться связанной квантором внутри формулы.

2.13.2 Универсальное введение

ОПРЕДЕЛЕНИЕ 2.14 Универсальное введение ($\forall I$)

$\varphi(a) \vdash \forall x \varphi(x)$, где a — свежая переменная:

$$\frac{\varphi(a)}{\forall x \varphi(x)}$$

Условие: a не встречается свободно ни в одной открытой гипотезе, от которой зависит вывод $\varphi(a)$.

Свежесть a — это и есть произвольность: если про a заранее ничего не известно и ни одна гипотеза его не упоминает, вывод верен для любого объекта, и обобщение законно.

Пример Универсальное введение

1	$\forall x (P(x) \wedge Q(x))$	<i>Premise</i>
2	a	<i>произвольное a</i>
3	$P(a) \wedge Q(a)$	$\forall E$ 1
4	$P(a)$	$\wedge E$ 3
5	$\forall x P(x)$	$\forall I$ 2-4

Строка 2 объявляет a произвольным — это не формула, а свежее имя, читаемое как «пусть a — любой объект». Из неё выводится $P(a)$, и только затем обобщение до $\forall x P(x)$. Гипотеза строки 2 снимается правилом $\forall I$ — заключение не зависит от выбора a .

2.13.3 Экзистенциальное введение

ОПРЕДЕЛЕНИЕ 2.15 Экзистенциальное введение ($\exists I$)

$\varphi(t) \vdash \exists x \varphi(x)$ для некоторого терма t :

$$\frac{\varphi(t)}{\exists x \varphi(x)}$$

Терм t подставляется вместо x свободно.

Нашли конкретный объект со свойством φ — значит, объект с таким свойством существует. Свидетель не обязан быть единственным: достаточно одного.

Пример Экзистенциальное введение

1	$P(a)$	<i>Premise</i>
2	$\exists x P(x)$	$\exists I$ 1

Из « a обладает свойством P » следует «существует объект со свойством P ». Заключение слабее посылки: оно не говорит, какой именно объект — но свидетель a остался в выводе.

2.13.4 Экзистенциальное устранение

ОПРЕДЕЛЕНИЕ 2.16 Экзистенциальное устранение ($\exists E$)

Из $\exists x \varphi(x)$ и подвывода $\varphi(a) \vdash C$ выводится C :

$$\frac{\exists x \varphi(x) \quad \varphi(a) \vdash C}{C}$$

Условие: a не встречается свободно ни в C , ни в открытых гипотезах, кроме самой $\varphi(a)$.

Правило формализует рассуждение «существует объект с нужным свойством, назовём его a , поработаем с ним и получим C ». Свежесть a гарантирует, что C не зависит от конкретного выбора: имя a — произвольная метка, а не объект, о котором уже что-то известно.

Пример Экзистенциальное устранение

1	$\exists x (P(x) \wedge Q(x))$	Premise
2	$P(a) \wedge Q(a)$	свежее a
3	$P(a)$	$\wedge E$ 2
4	$\exists x P(x)$	$\exists I$ 3
5	$\exists x P(x)$	$\exists E$ 1, 2-4

Гипотеза $P(a) \wedge Q(a)$ вводит свежую переменную a — имя, о котором ничего не известно. Из неё выводится $\exists x P(x)$. Правило $\exists E$ снимает гипотезу: заключение не зависит от конкретного a .

Четыре кванторных правила вместе со связочными дают натуральную дедукцию для логики предикатов. Проверка условий на переменные — свежесть, свобода подстановки — часть механической проверки вывода, и верификатор из проекта главы обязан их контролировать.

Проверка Кванторы в натуральной дедукции

1. Почему правило $\forall I$ требует, чтобы обобщаемая переменная не встречалась свободно ни в одной открытой гипотезе?

2. Какое условие правило $\exists E$ накладывает на свежую переменную a и что пошло бы иначе без него?

§ 2.14 Корректность и полнота

Вывод и семантическое следование — два разных отношения: $\vdash$ живёт в синтаксисе, $\models$ — в семантике. Связывает их пара теорем: корректность и полнота.

ТЕОРЕМА 2.3 Корректность натуральной дедукции (soundness)

$\Gamma \vdash \varphi$ влечёт $\Gamma \models \varphi$: всё выводимое — логическое следствие.

Доказательство

Докажем индукцией по дереву вывода.

База. Гипотеза из Γ следует из Γ по определению следования: любая интерпретация, в которой истинны все формулы Γ , делает истинной и саму гипотезу.

Шаг. Покажем, что каждое правило сохраняет следование: если все посылки правила следуют из Γ , то и его заключение следует из Γ .

Modus ponens. Если $\Gamma \models A \rightarrow B$ и $\Gamma \models A$, то в интерпретации с истинным Γ истинны $A \rightarrow B$ и A . По таблице истинности импликации из истинных $A \rightarrow B$ и A следует истинность B .

Ввод импликации. Пусть $\Gamma, A \models B$. Возьмём интерпретацию с истинным Γ и рассмотрим $A \rightarrow B$. Если A истинно, то B истинно по предположению, а если A ложно, импликация истинна тривиально. В обоих случаях $A \rightarrow B$ истинно, то есть $\Gamma \models A \rightarrow B$.

Разбор случаев. Пусть $\Gamma \models A \vee B$, $\Gamma, A \models C$ и $\Gamma, B \models C$. В интерпретации с истинным Γ истинно хотя бы одно из A , B , и в каждом из двух случаев соответствующая ветвь даёт истинное C .

Остальные связочные правила. Ввод и устранение конъюнкции, ввод дизъюнкции, ввод и устранение отрицания и взрыв проверяются по таблицам истинности связок: конъюнкция истинна ровно при истинных членах, дизъюнкция — хотя бы при одном, отрицание — при ложности формулы, а $\perp$ не истинно нигде, поэтому из него следует любая формула. Косвенное доказательство корректно в классической семантике: если $\Gamma, \neg A \models \perp$, то допущение $\neg A$ несовместимо с истинным Γ , и, поскольку каждая формула истинна или ложна, отсюда следует истинность A .

Кванторы. Универсальное устранение из истинной формулы $\forall x \varphi(x)$ даёт истинную $\varphi(t)$, если подстановка свободна. Универсальное введение из $\varphi(a)$ со

свежим a даёт $\forall x \varphi(x)$, потому что про a ничего не предполагалось. Экзистенциальные правила зеркальны. Точное определение истинности в модели — предмет главы 3.

По индукции корень дерева следует из Γ , то есть $\Gamma \models \varphi$.

Корректность — обязательное свойство: система, выводящая ложь из истины, бесполезна. Без неё механическая проверка теряет смысл — проверенный вывод ничего не гарантирует.

ТЕОРЕМА 2.4 Полнота натуральной дедукции (теорема Гёделя о полноте)

Всё, что следует семантически, выводимо: $\Gamma \models \varphi$ влечёт $\Gamma \vdash \varphi$.

Набросок доказательства

Докажем контрапозицией: если $\Gamma \not\models \varphi$, построим интерпретацию, в которой все формулы Γ истинны, а φ ложна.

Первый шаг. Расширим Γ до максимального непротиворечивого множества Δ (лемма Линденбаума). Нумеруем все формулы языка и, проходя их по очереди, добавляем формулу в Δ , если добавление не делает множество противоречивым. По построению Δ максимально непротиворечиво: оно содержит ровно одну формулу из каждой пары $\psi, \neg\psi$ — иначе недостающую формулу можно было бы добавить, не нарушив непротиворечивость.

Второй шаг. Построим модель по Δ . В пропозициональном случае атом p объявляем истинным тогда и только тогда, когда $p \in \Delta$, а в первопорядковом случае нужны свидетели Генкина: для каждой формулы $\exists x A(x)$ из Δ добавляем новую константу c с $A(c) \in \Delta$, а носителем модели берём термы языка. Без свидетелей экзистенциальная формула могла бы не найти объекта, на котором выполняется.

Третий шаг. Индукция по сложности формулы: $\Delta \models \psi$ тогда и только тогда, когда $\psi \in \Delta$. База — определение истинности атомов. Шаг — по главной связке, и каждое направление опирается на максимальность Δ : формула с главной связкой содержится в Δ ровно тогда, когда в нём содержатся её компоненты, требуемые семантикой.

Осталось собрать отрицание: если $\Gamma \not\models \varphi$, то $\Gamma, \neg\varphi$ непротиворечиво, и лемма Линденбаума расширяет его до Δ с $\neg\varphi \in \Delta$. В построенной модели все формулы Γ истинны, а φ ложна, то есть $\Gamma \not\models \varphi$, и контрапозиция даёт полноту.

Замечание **Корректность и полнота — две стороны одной медали**

Корректность говорит: вывод не обманывает, всё выводимое истинно, а полнота — вывод не упускает, всё истинное выводимо. Вместе они утверждают, что $\vdash$ и $\models$ — одно и то же отношение, записанное синтаксически и семантически.

Оговорка: доказательство полноты неконструктивно. Оно гарантирует, что вывод существует, но не даёт способа его найти, поэтому проверка остаётся механической, а поиск — трудным.

Проверка **Корректность и полнота**

1. Что гарантирует корректность, а что — полнота, и почему нужны оба свойства?
2. Почему доказательство полноты неконструктивно, и как это связано с трудностью поиска вывода?

§ 2.15 **Итоги главы**

Вопрос, с которого началась глава, получил ответ: проверка доказательства становится механической, когда доказательство записано как объект — конечная последовательность формул, в которой каждый переход разрешён явным правилом вывода. Спор о корректности рассуждения сводится к проверке этого объекта, и проверка смотрит только на форму формул, а не на их смысл. С доказательством гипотезы Кеплера произошло ровно это: четыре года работы двенадцати рецензентов дали мнение, а формальная запись — проверку, с которой машина справилась без процентов уверенности.

Четыре системы, разобранные в главе, — четыре способа организовать правила. Системы Гильберта сжимают логику в аксиомы и оставляют одно правило, натуральная дедукция раздаёт каждой связке пару правил ввода и устранения, секвенциальное исчисление превращает шаг вывода в преобразование секвенций, резолюция — в единственное правило над клаузами. При всей разнице устройства общее у них одно: каждый шаг либо разрешён, либо нет, и это свойство, а не вкус, делает доказательства проверяемыми.

Сквозная линия главы — разделение проверки и поиска. Проверка линейна и механична, поиск ветвится и требует ремесла, а корректность и полнота связывают синтаксис вывода с семантикой истинности. Корректность делает проверку осмысленной, полнота — достаточной, и вместе они показывают, что $\vdash$ и $\models$ — одно отношение, записанное двумя способами. Семантика, стоящая за $\models$, — предмет главы 3. Машинный поиск доказательств — тема главы 14, а интуиционистский вывод без косвенного доказательства — главы 34.

Проверка Проверьте себя

1. Чем доказательство в системе Гильберта отличается от доказательства в натуральной дедукции, и в чём сильные стороны каждой?
2. Что такое поддоказательство в нотации Фитча, и какое действие снимает гипотезу?
3. Зачем нужна теорема о дедукции, если правила вывода работают с формулами?
4. Чем секвенция Гентцена отличается от формулы системы Гильберта?
5. В чём разница между корректностью и полнотой, и почему для практики важны оба свойства?
6. Как резолюция сводит проверку выводимости к поиску противоречия?

Что дальше?

В этой главе кванторные правила уже появились, но их семантика осталась за скобками: мы говорили о выводимости, а не об истинности в моделях. Глава о логике предикатов 3 вводит модели, интерпретации и общезначимость — то, что стоит за символом $\models$ — и доводит теорему о полноте до полной силы для логики первого порядка.

§ 2.16 Упражнения

Системы вывода.

1. В системах Гильберта ровно одно правило вывода. Какое? Зачем нужны аксиомы?
2. Чем натуральная дедукция отличается от систем Гильберта?
3. Почему резолюция ограничена клаузуальной формой?

Выводы в нотации Фитча.

4. Докажите в нотации Фитча: $A \wedge B \rightarrow A$ (проекция).
5. Докажите в нотации Фитча: $A \rightarrow B \vdash \neg B \rightarrow \neg A$ (контрапозиция).
6. Докажите в нотации Фитча: $(A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$ (цепочка).
7. Докажите в нотации Фитча: $A \vee (B \wedge C) \vdash (A \vee B) \wedge (A \vee C)$ (дистрибутивность).
8. Докажите в нотации Фитча $\neg(A \wedge \neg A)$ (закон непротиворечия).

Схемы аксиом и секвенции.

9. В системе Гильберта со схемами K и S выведите $A \rightarrow A$. Сколько шагов потребовалось?
10. В классическом секвенциальном исчислении LK выведите секвенцию $\vdash A \vee \neg A$. Какой шаг использует многоколоночность и потому недоступен в интуиционистском LJ?

11. Назовите три структурных правила секвенциального исчисления и объясните, почему в таблице выписана только контракция.

Классические принципы.

12. * Докажите в нотации Фитча закон Пирса: $((A \rightarrow B) \rightarrow A) \rightarrow A$. Почему он не выводим без косвенного доказательства?
13. * Докажите $A \rightarrow B \vdash \neg A \vee B$. Почему для этого нужно косвенное доказательство?
14. * Найдите ошибку в «доказательстве» теоремы $\neg\neg A \rightarrow A$, претендующем на интуиционистскую корректность. Предположим $\neg\neg A, \neg A$. По правилу $\neg E$ получаем $\perp$. Интуиционистское правило $\neg I$ из этого поддоказательства позволяет заключить $\neg\neg A$ — уже имеющуюся посылку. Какой классический шаг на самом деле скрывается в рассуждении «из противоречия под гипотезой $\neg A$ заключаем A »?

Нормализация и метатеоремы.

15. * Приведите вывод с локальным максимумом, где формула вводится и тут же устраняется одной связкой, и устраните его, сократив вывод.
16. Сформулируйте корректность и полноту словами. Почему корректность обязательна, а полнота желательна?

ПРОЕКТ Верификатор доказательств в нотации Фитча

Проверка доказательства — механическая процедура: каждый шаг либо разрешён правилами вывода, либо нет. Соберите верификатор из нотации Фитча и правил вывода: он должен принимать выводы и указывать первую строку, не разрешённую правилами. Тот же верификатор должен отделить классические правила от интуиционистских.

① *Формулы как данные*

Определите формулы как синтаксическое дерево (атомы, связки, кванторы) и реализуйте подстановку терма вместо свободной переменной — основу правил $\forall E$ и $\exists I$.

② *Проверка шагов*

Доказательство — последовательность строк с указанием правила и ссылок. Соберите проверку ввода и устранения конъюнкции, дизъюнкции, импликации и отрицания, взрыва из противоречия, косвенного доказательства и четырёх кванторных правил. Продумайте, как верификатор сообщает о первой неверной строке.

③ *Условия на переменные*

Проверяйте, что $\forall I$ запрещён, когда вводимая переменная свободна в открытой гипотезе, а $\exists E$ запрещён, когда вводимая переменная входит в заключение. Соберите выводы с нарушенными условиями и покажите, что верификатор отвергает их на соответствующем шаге.

④ *Двенадцать направлений*

Проведите в нотации Фитча все 12 направлений эквивалентности четырёх классических принципов — закон исключённого третьего, снятие двойного отрицания, закон Пирса, косвенное доказательство — и проверьте каждый вывод верификатором.

⑤ *Граница классики*

В режиме только интуиционистских правил покажите, что вывод закона исключённого третьего или закона Пирса не проходит без косвенного доказательства. Объясните, какой шаг при этом блокируется.

Итог: работающий верификатор, который проверяет библиотеку из 12 взаимных выводов классических принципов и указывает первую неверную строку любого вывода.

III

Логика предикатов

«Не все студенты сдали экзамен» и «все студенты не сдали экзамен» — какая между ними разница? Первое: утверждение «все студенты сдали экзамен» ложно — не сдал *хотя бы один*, а может, и все. Второе: не сдал никто — все до единого провалились. Разговорное «не все» намекает, что кто-то всё же сдал, но отрицание утверждения «все сдали» этого не обещает. Положение одного слова «не» меняет смысл утверждения, и то же делает порядок слов: «каждый любит кого-то» — у каждого свой любимый, а «кого-то любят все» — существует человек, которого любят все без исключения. Поменяли местами два слова — и из утверждения о личной жизни каждого получилось требование ко всей вселенной сразу.

Логика высказываний из главы 1 на эту разницу ответить не может, потому что она не видит структуры утверждения. Для неё «Сократ смертен» — просто атом p : буква, которой приписано истинностное значение, и ничего больше. Внутри атома она не заглядывает: ни объектов, ни слов «все» и «существует», ни порядка, в котором эти слова стоят. Она даже не может записать вопрос, чем «каждый любит кого-то» отличается от «кого-то любят все»: для неё обе фразы — один и тот же атом.

Различение таких смыслов — задача старая. Две с половиной тысячи лет назад Аристотель построил первую систему рассуждений об объектах — силлогистику, где суждения вида «все S суть P » связывались правилами вывода. Ей пользовались больше двух тысячелетий, пока в конце XIX века Фреге не заменил её логикой предикатов с явными кванторами. Смысл, спрятанный в порядке слов, формула записывает явно.

Что нужно, чтобы заглянуть внутрь атома? Две вещи: предикаты и кванторы. Предикат — это шаблон с дырой: $P(x)$ — « x сдал экзамен», $L(x, y)$ — « x любит y ». Сам по себе шаблон не истинен и не ложен, пока переменную не подставили конкретным значением. Два квантора превращают шаблон в высказывание, задавая размах: $\forall$ («для всех») требует свойства для каждого объекта, $\exists$ («существует») — хотя бы для одного. Формула с квантором — конечная строка с бесконечным смыслом.

Порядок кванторов меняет смысл формулы. $\forall x \exists y L(x, y)$ разрешает y зависеть от x — у каждого свой любимый. $\exists y \forall x L(x, y)$ требует один и тот же y для всех — одного человека, которого любят все. На этом различии держатся определение предела функции — три квантора в одном предложении — и разница между коррелированным подзапросом в SQL и заранее вычисленной константой. На практике «все» почти всегда относится к ограниченному множеству, например к «все студенты группы». Для этого служат ограниченные кванторы $\forall x \in A$ и $\exists x \in A$.

Перевод с естественного языка на язык предикатов похож на компиляцию: каждая неоднозначная фраза получает ровно одну формулу. Формула становится данными, которые можно отрицать, соединять и проверять механически.

В главе 2 кванторы уже работали: четыре правила вывода — ввод и удаление кванторов всеобщности $\forall$ и существования $\exists$. Там кванторы были инструментом доказательств. Теперь на очереди сам язык, на котором эти доказательства записаны. Как устроен язык, на котором «каждый любит кого-то» и «кого-то любят все» различимы, а «не все сдали» и «все не сдали» — две разные формулы?

ИСТОРИЯ От Аристотеля к Фреге

До Фреге логика говорила о форме «все S суть P » — одного субъекта и одного предиката. Аристотель в «*Analytica Priora*»¹³ построил из таких суждений силлогистику, и две тысячи лет она была единственной логикой. Буль в 1854 году в «*An Investigation of the Laws of Thought*»¹⁴ алгебраизировал классы, но его уравнения связывали множества, а не объекты: кванторов и отношений по-прежнему не было.

Прорыв произошёл в 1879 году, когда Фреге в «*Begriffsschrift*»¹⁵ записал «для всех» и «существует» отдельными знаками. Тогда предикат перестал быть приговором об одном субъекте и стал шаблоном с местами для объектов, а формула — конечной строкой, но её смысл охватывает бесконечное множество объектов. Это и есть язык, на котором «каждый любит кого-то» отличим от «кого-то любят все».

ОБЗОР ГЛАВЫ

Глава заглядывает внутрь атомов, которые логика высказываний принимала как данность: предикаты и кванторы дают язык для утверждений об объектах. Предикаты — шаблоны с переменными, и кванторы $\forall$ и $\exists$ превращают их в высказывания. Затем язык собирается по правилам: сигнатура фиксирует символы и их арности, термы становятся именами объектов, а формулы строятся индуктивно — от атомарных через связи и кванторы. Затем рассматриваются отрицание кванторов, несколько кванторов подряд и ограниченные кванторы $\forall x \in A$, $\exists x \in A$. Дальше глава занимается переводом с естественного языка на язык логики предикатов: каждая неоднозначная фраза получает ровно одну формулу. В звёздных разделах глава рассматривает сорта — типизированные переменные — и силлогистику Аристотеля как исторический предшественник. Во второй половине глава переходит

¹³ Аристотель. «*Analytica Priora*» («Первая аналитика»). Около 350 г. до н. э.

¹⁴ G. Boole. «*An Investigation of the Laws of Thought*». 1854.

¹⁵ G. Frege. «*Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*» («Исчисление понятий»). 1879.

к семантике: моделям, истинности, теореме Гёделя о полноте и её следствию — компактности.

После этой главы вы сможете:

1. записывать утверждения о свойствах объектов и отношениях формулами с кванторами;
2. строить термы и формулы по индуктивным правилам сигнатуры и отличать правильно построенные выражения от неправильно построенных;
3. различать порядки кванторов и объяснять, чем $\forall x \exists y$ отличается от $\exists y \forall x$;
4. отрицать формулы с кванторами по законам де Моргана и переводить с естественного языка на язык предикатов и обратно;
5. пользоваться ограниченными кванторами и объяснять, почему $\forall$ соединяется с импликацией, а $\exists$ — с конъюнкцией;
6. определять истинность формулы в модели, а также формулировать полноту и компактность логики первого порядка.

§ 3.1 Предикаты

Разберём атом «Сократ смертен». Логика высказываний видит в нём лишь букву p . Логика предикатов заглядывает внутрь атома и находит объект и свойство.

Пример Сократ смертен

Пусть $P(x)$ — « x смертен», предметная область — люди. Тогда «Сократ смертен» — это $P(\text{Сократ})$: переменную заменили конкретным человеком. $P(\text{Сократ})$ истинно, и внутри него виден объект (Сократ) и свойство (смертность).

Зачем заглядывать внутрь атома? Утверждение «каждое натуральное число либо чётно, либо нечётно» в логике высказываний пришлось бы записать как бесконечную конъюнкцию: «0 чётно или нечётно, и 1 чётно или нечётно, и 2...». Его истинность зависит от отношений между объектами и их свойствами — этого логика высказываний не видит.

ОПРЕДЕЛЕНИЕ 3.1 Предикат

Предикат $P(x)$ — это утверждение, содержащее переменную x , которое становится высказыванием, когда x заменяется конкретным значением из предметной области.

ОПРЕДЕЛЕНИЕ 3.2 Предметная область (*domain of discourse*)

Предметная область — это множество значений, которые может принимать переменная.

Термин «область определения» здесь не используется: он занят областью определения функции, глава 6.

Предикат — это шаблон, функция из предметной области в истинностные значения.

Пример Предикат с предметной областью $\mathbb{N}$

Пусть $P(x)$ — « $x > 5$ » с предметной областью $\mathbb{N}$. Тогда $P(7)$ истинно, а $P(3)$ ложно.

Предикаты могут содержать несколько переменных: $P(x, y)$ — « $x < y$ » с предметной областью $\mathbb{R}$. Предикат с k переменными задаёт k -арное отношение на предметной области — слово станет точным термином в главе 5.

Задача кажется решённой: мы умеем говорить об объектах и их свойствах. Но предикат — это шаблон, а не высказывание. $P(x)$ не истинно и не ложно, пока x не заменён конкретным значением. Перебрать все элементы предметной области по одному невозможно для бесконечных областей.

Решение — кванторы: механизм, связывающий переменную и превращающий шаблон в высказывание. Например, «для всех x верно $P(x)$ » или «существует x , для которого верно $P(x)$ ». Вся логика предикатов держится на этой конструкции: предикат даёт шаблон, квантор задаёт размах, вместе они дают высказывание.

§ 3.2 Кванторы

Кванторы превращают предикаты в высказывания, указывая, сколько элементов предметной области удовлетворяет предикату.

ОПРЕДЕЛЕНИЕ 3.3 Квантор всеобщности

$\forall x P(x)$ утверждает, что $P(x)$ истинно для каждого x из предметной области. Читается «для всех x верно $P(x)$ ».

Квантор всеобщности — сильное утверждение: свойство должно выполняться для всех элементов без исключения. Часто нам достаточно более слабого утверждения — что свойство выполнено хотя бы для одного элемента. Для этого служит парный квантор существования.

ОПРЕДЕЛЕНИЕ 3.4 Квантор существования

$\exists x P(x)$ утверждает, что $P(x)$ истинно хотя бы для одного x из предметной области. Читается «существует x такой, что $P(x)$ ».

Пример Кванторы над натуральными числами

С предметной областью $\mathbb{N}$:

- $\forall x \, x \geq 0$ истинно.
- $\exists x \, x < 0$ ложно.
- $\forall x \exists y \, y > x$ истинно (натуральные числа неограничены).



Для конечной предметной области кванторы не добавляют выразительной силы — логика предикатов над конечной областью сводится к логике высказываний.

$$\forall x \, P(x) \equiv \bigwedge_{i=1}^n P(a_i)$$

$$\exists x \, P(x) \equiv \bigvee_{i=1}^n P(a_i)$$

Кванторы дают лишь компактность записи, а не новую логическую силу. Вся подлинная сила кванторов — в бесконечных областях.

Переменная называется *связанной*, если она стоит под квантором. В противном случае она *свободна*. Область действия квантора — это подформула, к которой он применяется.

Пример Свободные и связанные переменные

В формуле $\exists x (P(x) \wedge Q(y))$ переменная x связана квантором существования, а y свободна. Область действия квантора $\exists x$ — это подформула, к которой он применяется: $P(x) \wedge Q(y)$. Если подставить вместо y конкретное значение, формула станет высказыванием, а значение x квантор уже зафиксировал.

§ 3.3 Отрицание кванторов

На естественном языке отрицание «все сдали» звучит как «хотя бы один не сдал»: отрицание проходит сквозь квантор и меняет его на парный. Для формул верны те же два закона.

ТЕОРЕМА 3.1 Отрицание кванторов

- $\neg \forall x \, P(x) \equiv \exists x \, \neg P(x)$.
- $\neg \exists x \, P(x) \equiv \forall x \, \neg P(x)$.

Набросок доказательства

Докажем оба равенства, раскрывая определения кванторов и отрицания.

Первое равенство. $\neg\forall x P(x)$ истинно, когда неверно, что $P(x)$ выполнено для всех x . Это значит, что существует x , для которого $P(x)$ ложно, а это равно $\exists x \neg P(x)$.

Второе равенство. $\neg\exists x P(x)$ истинно, когда не существует x со свойством P . То есть каждый x свойством P не обладает, а это равно $\forall x \neg P(x)$.

Отрицание «все любят пиццу» — «существует тот, кто не любит пиццу». Отрицание «существует единорог» — «всё является не-единорогом».

Проверка Отрицание кванторов

1. Запишите без внешнего отрицания и переведите на естественный язык: $\neg\exists x P(x)$.
2. Верно ли равенство $\neg(\forall x P(x)) \equiv \forall x \neg P(x)$? Если нет — укажите предметную область и предикат, на которых формулы расходятся.
3. Что отрицание $\neg\forall x (\text{студент}(x) \rightarrow \text{сдал}(x))$ обещает о сдавших экзамен и чего оно не обещает?

§ 3.4 Несколько кванторов

Когда формула содержит более одного квантора, порядок критически важен.

Пример Порядок кванторов

Пусть $P(x, y)$ — « y является матерью x » с предметной областью «люди».

- $\forall x \exists y P(x, y)$: «У каждого есть мать» — истинно.
- $\exists y \forall x P(x, y)$: «Существует человек, являющийся матерью всех» — ложно.

Смысл перестановки в том, кто выбирает свидетеля: при $\forall x \exists y$ выбор повторяется для каждого x , при $\exists y \forall x$ он сделан один раз и обязан годиться всем. Это логический аналог вложенности циклов: $\forall x \exists y$ соответствует внутреннему циклу, зависящему от внешнего индекса, а $\exists y \forall x$ — предвычисленному значению для всей итерации. На диаграмме 9 слева каждому x_i отвечает свой y_i , который может зависеть от x . Справа один и тот же y служит всем x_i сразу.

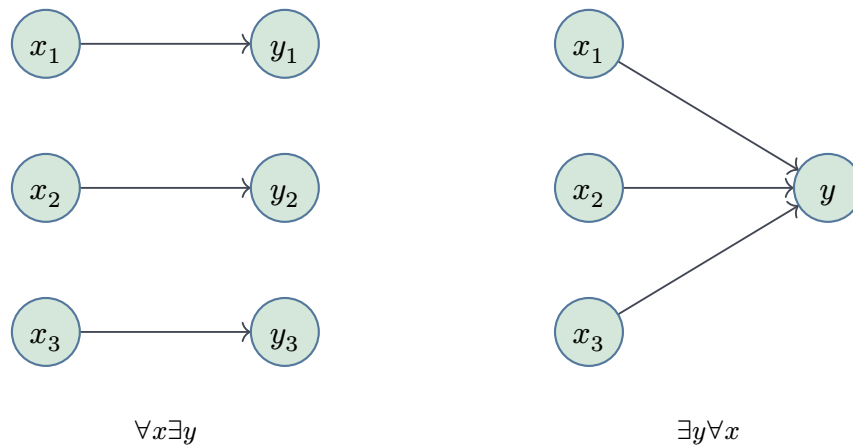


Рис. 9. Порядок кванторов меняет смысл формулы.

§ 3.5 * Сколемизация

Различие между «зависит от x » и «общий для всех» можно сделать точным: значение y — функция от x . Это наблюдение превращается в технику, убирающую кванторы существования.

ОПРЕДЕЛЕНИЕ 3.5 Сколемизация

Сколемизация — замена каждого квантора существования новой **сколемовской функцией** от всех кванторов всеобщности, в области действия которых он стоит, а при их отсутствии — новой **константой**.

Сколемовская функция фиксирует зависимость свидетеля от уже выбранных объектов: формула $\forall x \exists y P(x, y)$ переходит в $\forall x P(x, f(x))$. Если же квантор существования не лежит ни под каким квантором всеобщности, свидетелю не от чего зависеть, и вместо функции вводится константа: $\exists y \forall x P(x, y)$ переходит в $\forall x P(x, c)$. Квантор существования описывает выбор, а выбор может зависеть от уже сделанного: каждый x выбирает своего свидетеля y . Сколемовская функция записывает этот выбор явно, и квантор существования исчезает.

Пример Функция или константа

Пусть $P(x, y)$ — « y является матерью x ». Сколемизация «у каждого есть мать» даёт $\forall x P(x, f(x))$: функция f подбирает мать каждому. Сколемизация «некто является матерью всех» даёт $\forall x P(x, c)$: свидетель один на всех, ему не от чего зависеть.

Полученная формула выполнима тогда и только тогда, когда выполнима исходная. Эквивалентности эта замена не сохраняет.

ОПРЕДЕЛЕНИЕ 3.6 Предварённая нормальная форма

Формула находится в **предварённой нормальной форме** (далее — ПНФ), если все кванторы вынесены в начало и она имеет вид $Q_1 x_1 \dots Q_k x_k \psi$, где каждый Q_i — квантор $\forall$ или $\exists$, а ψ не содержит кванторов.

Любая формула приводится к ПНФ: связанные переменные переименовываются так, чтобы у каждого квантора была своя, затем кванторы выносятся в начало эквивалентностями переноса квантора через связи. Для формулы в ПНФ сколемизация применяется механически: каждый квантор $\exists$ заменяется функцией от стоящих левее кванторов $\forall$. Кванторы на этом исчезают, но бескванторная часть — матрица — ещё не обязана быть конъюнкцией дизъюнкций: в такую форму её раскладывает дистрибутивность $\vee$ относительно $\wedge$. Полученные дизъюнкты — материал для резолюции.

На Сколемизации строится автоматическое доказательство в логике первого порядка: чтобы показать невыполнимость множества формул, его отрицают, Сколемизируют, приводят к дизъюнктам и ищут противоречие резолюцией — методом из главы 14.

§ 3.6 Ограниченные кванторы

На практике большинство квантификаций происходит по ограниченному множеству.

ОПРЕДЕЛЕНИЕ 3.7 Ограниченный квантор

- $\forall x \in A P(x)$ — сокращение для $\forall x (x \in A \rightarrow P(x))$.
- $\exists x \in A P(x)$ — сокращение для $\exists x (x \in A \wedge P(x))$.

Почему $\forall$ использует импликацию, а $\exists$ — конъюнкцию? Это следствие смысла кванторов. Утверждение «все S суть P » означает, что для каждого объекта, *если* он принадлежит S , *то* он принадлежит P . Импликация ничего не утверждает об объектах вне S : для них посылка ложна, и импликация истинна автоматически — ограничение работает. Если бы мы записали $\forall x (x \in A \wedge P(x))$, это означало бы «все объекты во вселенной принадлежат A и обладают свойством P » — явно не то.

Утверждение «существует S , которое есть P » означает, что найдётся объект, который *одновременно* принадлежит S и обладает свойством P . Конъюнкция требует выполнения обоих условий. Если бы мы записали $\exists x (x \in A \rightarrow P(x))$, утверждение было бы истинно почти всегда. Любой объект вне A обращает посылку в ложь, импликация становится истинной, и такой x заведомо найдётся, если вне A есть хоть один объект.

Эта двойственность — одна из частых причин ошибок при переводе с естественного языка. Утверждение «все X обладают свойством Y » записывается через $\forall$ с импликацией. Утверждение «существует X со свойством Y » записывается через $\exists$ с конъюнкцией.

Пример Ограниченные кванторы для массивов

- «Каждый элемент массива неотрицателен»:

$$\forall i \in \{0, \dots, n-1\} A[i] \geq 0$$

- «Массив содержит хотя бы один ноль»:

$$\exists i \in \{0, \dots, n-1\} A[i] = 0$$

Пример Проверка на конкретном универсуме

Возьмём универсум $\{1, 2, 3\}$. Внутри него отметим множество $A = \{1, 2\}$. Пусть $P(x)$ — « x положительно». Тогда $\forall x \in A P(x)$ истинно: оба элемента A положительны. А запись $\forall x (x \in A \wedge P(x))$ ложна. Элемент 3 не принадлежит A , и конъюнкция для него ложна, хотя сам он положителен.

С квантором существования картина обратная. Пусть $Q(x)$ — « x отрицательно». Тогда $\exists x \in A Q(x)$ ложно: ни один элемент A не отрицателен. А запись $\exists x (x \in A \rightarrow Q(x))$ истинна. Элемент 3 вне A обращает посылку импликации в ложь. Формула получает свидетеля, хотя ни один элемент A свойством Q не обладает.

Отрицание переносится на ограниченные кванторы теми же законами:

$$\neg(\forall x \in A P(x)) \equiv \exists x \in A \neg P(x)$$

$$\neg(\exists x \in A P(x)) \equiv \forall x \in A \neg P(x).$$

Первое: если не все элементы A обладают свойством, в A найдётся элемент, который им не обладает. Второе: если в A нет элемента со свойством, каждый элемент A свойства лишён.

§ 3.7 Перевод на язык логики предикатов

Основной навык — перевод с естественного языка на язык логики предикатов и обратно. Перевод должен точно сохранять смысл — неоднозначность естественного языка должна быть устранена.

Пример Область действия отрицания: «не все» и «все не»

«Не все студенты сдали экзамен» — это отрицание утверждения «все студенты сдали экзамен»: $\neg \forall x (\text{студент}(x) \rightarrow \text{сдал}(x))$ — неверно, что все сдали, то есть *хотя бы один* не сдал. Сколько именно — один, двое или все: формула не разли-

чает. «Все студенты не сдали экзамен» — это $\forall x (\text{студент}(x) \rightarrow \neg \text{сдал}(x))$: не сдал никто.

1. В первом случае отрицание стоит перед квантором и действует на всю формулу.
2. Во втором — внутри, на предикат.

Разговорное «не все сдали» обычно подразумевает, что кто-то всё же сдал. Логическое отрицание «все сдали» этого не требует — оно лишь утверждает существование хотя бы одного несдавшего. Формула фиксирует область действия отрицания явно. Естественный язык не всегда это делает. Например, английское «All students did not pass the exam» допускает оба прочтения, и при переводе легко перепутать область действия отрицания. Запись формулой не даёт этому случиться.

Если исходная фраза не двусмысленна, перевод становится прямой подстановкой.

Пример Массив отсортирован

«Массив отсортирован» (по возрастанию):

$$\forall i \in \{0, \dots, n-2\} A[i] \leq A[i+1].$$

Математический анализ даёт следующий пример.

Пример Предел на языке кванторов

Определение предела $\lim_{x \rightarrow a} f(x) = L$:

$$\forall \varepsilon > 0 \exists \delta > 0 \forall x (0 < |x - a| < \delta \rightarrow |f(x) - L| < \varepsilon).$$

Три квантора нанизаны в одно предложение: для каждого ε существует ответ δ , работающий сразу для всех x из проколотой окрестности.

Логическая нотация фиксирует утверждение с точностью, недоступной естественному языку. Доказательство устанавливает его истинность.

§ 3.8 * Сорты и типизированные переменные

В примерах выше предметная область — одно множество: натуральные числа, люди, города. Но утверждения часто смешивают объекты разных видов: «каждый студент сдал экзамен по некоторому предмету» говорит и о студентах, и о предметах. Писать $\forall x$ без уточнения, кто такой x — студент или предмет, неудобно и опасно: квантор пробегает всё сразу.

Многосортная логика первого порядка (*many-sorted FOL*) решает это введением *сорт*ов (*sorts*) — типов переменных. Вместо одной предметной области — несколько: S_{stud}

(студенты), S_{subj} (предметы). Переменная объявляется с сортом: $x : S_{\text{stud}}$ означает, что x пробегает студентов. Кванторы становятся типизированными: $\forall x : S_{\text{stud}}. \exists y : S_{\text{subj}}. \text{Passed}(x, y)$ — «каждый студент сдал некоторый предмет».

ОПРЕДЕЛЕНИЕ 3.8 Сорт

Сорт — тип переменной: вместо одной предметной области несколько, и каждая переменная пробегает свою.

ОПРЕДЕЛЕНИЕ 3.9 Типизированный квантор

Типизированный квантор записывается через сорт и раскрывается как ограниченный квантор:

- $\forall x : S. \varphi$ — сокращение для $\forall x (x \in S \rightarrow \varphi)$.
- $\exists x : S. \varphi$ — сокращение для $\exists x (x \in S \wedge \varphi)$.

Выигрыш — в ясности и в механической проверке: типизированная формула отклоняется ещё до семантики, если сорта не сходятся (как программа, не прошедшая проверку типов).

Многосортная логика — основа *SMT-солверов* (Satisfiability Modulo Theories): решателей, которые проверяют выполнимость формул первого порядка над теориями — арифметикой, массивами, битовыми векторами. Сорт переменной указывает теорию: $x : \text{Int}$ — целочисленная арифметика, $a : \text{Array}(\text{Int}, \text{Int})$ — массив. SMT-солвер комбинирует SAT-рассуждение (глава 14) с решением уравнений в каждой теории. Примеры SMT-солверов и их применение — в главе 15.

§ 3.9 * Силлогистика Аристотеля

Примечание Исторический экскурс: традиционная логика

Традиционная логика Аристотеля — система, которой пользовались до Фреге. Современная классическая логика первого порядка (главы 1, 3, 2) строже и выразительнее: явные кванторы, явные формулы, явные правила и модели. Традиционная логика с её силлогизмами и экзистенциальными допущениями в современной математике не используется.

Нотация A, E, I, O закрепились в средневековых университетах и используется до сих пор в учебниках логики.

Пример Конкретные категорические суждения

Пусть S — «люди», P — «смертные существа».

- A (общее утвердительное): «Все люди смертны» — $\forall x (S(x) \rightarrow P(x))$.

- E (общее отрицательное): «Ни один человек не смертен» — $\forall x(S(x) \rightarrow \neg P(x))$.
- I (частное утвердительное): «Некоторые люди смертны» — $\exists x(S(x) \wedge P(x))$.
- O (частное отрицательное): «Некоторые люди не смертны» — $\exists x(S(x) \wedge \neg P(x))$.

Отношения между суждениями с одним и тем же субъектом S и предикатом P традиционно изображаются *логическим квадратом* (*square of opposition*), как показано на рис. 10. В его вершинах стоят формы A, E, I, O , а стороны и диагонали показывают отношения между ними.

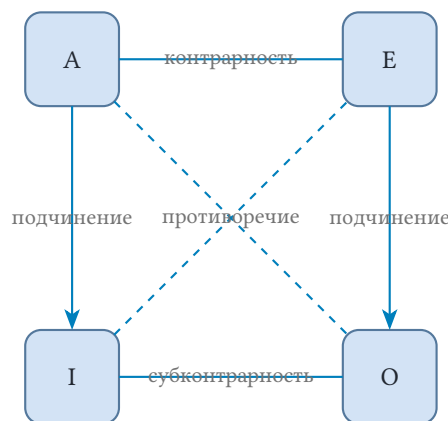


Рис. 10. Логический квадрат категорических суждений.

Отношения в квадрате:

1. **Противоречие** ($A-O, E-I$): не могут быть ни одновременно истинны, ни одновременно ложны — ровно одно истинно.
2. **Противоположность** ($A-E$): не могут быть оба истинны, но могут быть оба ложны.
3. **Частичная совместимость** ($I-O$): не могут быть оба ложны, но могут быть оба истинны.
4. **Подчинение** ($A \rightarrow I, E \rightarrow O$): если общее истинно, то и частное истинно. Если частное ложно, то и общее ложно.

Подчинение $A \rightarrow I, E \rightarrow O$ — наследие аристотелевского *экзистенциального допущения*. Оно верно, пока субъект S непуст: из того, что все S суть P , следует, что некоторый S есть P , только если хоть один S существует. В современной логике предикатов $A \rightarrow I$ из пустого S не следует. Подробнее — ниже, в разделе об экзистенциальном допущении.

3.9.1 Перевод в современную логику

Категорические суждения естественно переводятся в логику предикатов первого порядка:

Традиционная форма	Современная запись
Все S суть P (A)	$\forall x(S(x) \rightarrow P(x))$
Ни одно S не есть P (E)	$\forall x(S(x) \rightarrow \neg P(x))$
Некоторые S суть P (I)	$\exists x(S(x) \wedge P(x))$
Некоторые S не суть P (O)	$\exists x(S(x) \wedge \neg P(x))$

Таблица 1. Перевод категорических суждений в логику предикатов первого порядка.

Перевод немедленно объясняет отношения в логическом квадрате.

- A и O (противоречие): $\forall x(S(x) \rightarrow P(x))$ и $\exists x(S(x) \wedge \neg P(x))$ — отрицание квантора всеобщности даёт существование контрпримера.
- E и I (противоречие): $\forall x(S(x) \rightarrow \neg P(x))$ и $\exists x(S(x) \wedge P(x))$ — аналогично.

3.9.2 Проблема экзистенциального допущения

Здесь традиционная и современная логика расходятся.

В традиционной логике Аристотеля действует *экзистенциальное допущение* (*existential import*): каждое общее суждение подразумевает, что субъект не пуст. Из «Все S суть P » (A) логически следует «Некоторые S суть P » (I) — отношение подчинения в квадрате. Действительно, если класс S не пуст, то из того, что все его элементы обладают свойством P , следует, что некоторый элемент им обладает.

В современной логике предикатов это не так. Из $\forall x(S(x) \rightarrow P(x))$ не следует $\exists x(S(x) \wedge P(x))$. В качестве контрпримера возьмём пустое множество S . Тогда $\forall x(S(x) \rightarrow P(x))$ истинно (vacuously true — нет контрпримера), а $\exists x(S(x) \wedge P(x))$ ложно (нет элемента в S).

Пример Единороги и магия

1. «Все единороги магические» (A): $\forall x(U(x) \rightarrow M(x))$.
 - В современной логике — истинно (нет единорогов $\rightarrow$ нет контрпримера).
 - В традиционной логике — ложно (единорогов не существует, поэтому общее утверждение о них не может быть истинным).
2. «Некоторые единороги магические» (I): $\exists x(U(x) \wedge M(x))$.
 - В обеих системах — ложно (нет единорогов).
 - Но в традиционной логике A влечёт I — что приводит к противоречию.
 - Именно это противоречие и устранила современная логика, отказавшись от экзистенциального допущения.

Отказ от экзистенциального допущения — одно из определяющих достижений современной логики. Оно позволило отделить логическую форму утверждения от фактического существования объектов, сделав логику более «алгебраической» — и подготовив почву для Булевой алгебры (глава 10).

Замечание

В биологии такое случалось. Линнеевская таксономия — роды, семейства, отряды — две сотни лет была единственным способом упорядочить живое, пока не пришла молекулярная филогенетика. Она вложила систему Линнея в свою: млекопитающие остались млекопитающими, но теперь мы знаем *почему* — потому что у них общий предок, которого нет у других.

С аристотелевской логикой то же самое. Две тысячи лет она была единственной логикой, пока Буль и Фреге не построили современную, которая включила Аристотеля как частный случай. Силлогизмы работают, и теперь видно *почему* — потому что так устроена булева алгебра.

3.9.3 От Аристотеля к Булю: алгебраизация логики

Аристотелевская силлогистика прослужила две тысячи лет. Кант считал логику завершённой наукой — и под логикой понимал именно силлогистику. Основание для такой оценки было: всякое высказывание, казалось, приводится к форме « S есть P ». Однако эта форма — один субъект, один предикат — схватывает классификацию, но не схватывает отношения. Например, ни одно из следующих высказываний не сводится к форме « S есть P »:

- « x больше y »
- «прямая a параллельна прямой b »
- « A влечёт B »

Отношения могут быть транзитивными, симметричными, рефлексивными — силлогистический аппарат не различает этих свойств, потому что не может выразить их носитель. Бинарный предикат требует иной логической формы, и эта форма оставалась неосвоенной до середины XIX века.

Буль изменил вопрос. Аристотель спрашивал: «какие формы силлогизма правильны?» Буль спросил: «какие уравнения следуют из каких?»

Классы стали переменными: x обозначает совокупность объектов, обладающих свойством X . Высказывание «Все S суть P » записывается как $s = sp$: S -объекты суть в точности те S -объекты, которые являются также P -объектами. Дополнение класса выражается как $1 - x$. Если x есть класс S , то $1 - x$ есть класс не- S .

Силлогизм «Все S суть M , все M суть P , следовательно все S суть P » стал последовательностью подстановок:

$$\begin{aligned} s = sm, m = mp &\Rightarrow s = sm \\ sm = s &\Rightarrow s = sp \end{aligned}$$

Правильная форма вывода стала алгебраическим тождеством. Это переход от описания к исчислению — от грамматики правильных рассуждений к алгебре логических операций.

Второй шаг Буля был иным. Та же система операций — умножение, сложение, дополнение — работает для пропозиций, если интерпретировать 1 как истину, а 0 как ложь. Одна алгебраическая структура обслуживает две разные интерпретации: классы и высказывания. Это Булева алгебра, которую мы рассмотрим в главе 10.

После Буля оставался последний шаг — включить кванторы и отношения, чего не было ни у Аристотеля, ни у Буля. Это сделал Готлоб Фреге в «Begriffsschrift»¹⁶, создав логику предикатов — ту самую, которую мы используем сегодня.

§ 3.10 Термы и формулы

Примеры этой главы до сих пор брали формулы готовыми. Теперь соберём язык по правилам: у формального языка, как у естественного, есть алфавит и грамматика, и выражение имеет право на существование, только если собрано по ним. Раздел сознательно стоит перед семантикой: сначала договоримся, что считается формулой, а затем — что означает её истинность. Алфавит задаёт *сигнатуру* — набор знаков, каждому из которых указано, сколько аргументов он принимает.

ОПРЕДЕЛЕНИЕ 3.10 Сигнатура

Сигнатура — это набор символов языка, каждому из которых указана **арность** — число аргументов:

- константы — символы отдельных объектов, арность 0;
- функциональные символы — символы операций над объектами, арность $n \geq 1$;
- предикатные символы — символы свойств и отношений, арность $n \geq 1$.

Арности в этой главе уже работали без имён: $P(x)$ — « x сдал экзамен» — предикат арности 1, а $L(x, y)$ — « x любит y » — арности 2. Одноместный предикат выражает свойство, двуместный — отношение между парами объектов, а предикат арности k — свойство кортежей длины k (глава 5).

Пример Сигнатура с двумя предикатами

Возьмём сигнатуру из четырёх знаков: константа c , функция f арности 2, одноместный предикат P и двуместный предикат Q . Интерпретация назначит знакам смысл, но договориться о замысле можно заранее: $P(x)$ — « x сдал экзамен», $Q(x, y)$ — « x дружит с y ». Арность — это договор о числе аргументов: $Q(x, y)$ в этом языке выражение, а $Q(x)$ — уже нет, потому что двуместному предикату не хватает второго места.

¹⁶G. Frege. «Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens» («Исчисление понятий»). 1879.

Примечание Сигнатуру диктует задача

Сигнатура — не предмет вкуса, а словарь под задачу. Для базы данных предикаты — это таблицы: таблица «студент сдал предмет» с двумя столбцами даёт предикат арности 2, и порядок столбцов задаёт порядок аргументов. Для программы предикаты описывают отношения между значениями переменных, а функциональные символы — вычисления: f арности 1 — это функция одного аргумента. Выбор сигнатуры определяет, о чём вообще можно что-то сказать на этом языке.

Имена объектов в этом языке называются *термами*.

ОПРЕДЕЛЕНИЕ 3.11 Терм

Термы строятся по правилам:

1. каждая переменная — терм;
2. каждая константа — терм;
3. если $t_1, \dots, t_n$ — термы и f — функциональный символ арности n , то $f(t_1, \dots, t_n)$ — терм.

Ничто другое термом не является.

Терм называет объект и ничего о нём не утверждает, поэтому терм не бывает истинным или ложным. Функциональный символ собирает новое имя из старых: $f(x, c)$ называет объект, построенный из объектов x и c . Школьная запись собирается теми же правилами: $x + y$ — терм, если $+$ объявлен функциональным символом арности 2.

ОПРЕДЕЛЕНИЕ 3.12 Формула

Формулы строятся по правилам:

1. если P — предикат арности n и $t_1, \dots, t_n$ — термы, то $P(t_1, \dots, t_n)$ — **атомарная формула**;
2. если φ — формула, то $\neg\varphi$ — формула;
3. если φ и ψ — формулы, то $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ и $(\varphi \rightarrow \psi)$ — формулы;
4. если φ — формула и x — переменная, то $\forall x \varphi$ и $\exists x \varphi$ — формулы.

Ничто другое формулой не является.

Скобки в правилах обязательны, но на практике, как и в логике высказываний, часть из них опускают по приоритетам: $\neg$ связывает сильнее $\wedge$, $\wedge$ — сильнее $\vee$, $\vee$ — сильнее $\rightarrow$. Формула без свободных переменных называется *замкнутой*: лишь о ней имеет смысл спрашивать, истинна ли она в интерпретации, ведь свободные переменные оставляют формуле статус шаблона, ждущего назначений. Замкнутые формулы в прежних разделах уже встречались: $\forall x \exists y L(x, y)$ и $\exists x \forall y L(x, y)$ замкнуты, и потому о них можно было спрашивать, истинны ли они. Формула $\exists x (P(x) \wedge Q(y))$ из прежнего раздела замкнутой не была: её значение зависело от того, кто такой y .

Пример Разбор выражений

Сигнатура прежняя: константа c , функция f арности 2, предикаты P арности 1 и Q арности 2. Выражения x , c , $f(x, c)$, $f(f(c, x), y)$ — термы: последнее собрано правилом из двух меньших термов. Выражение $f(c, c)$ — терм из одних констант: он называет один конкретный объект. Выражение $f(x, c)$ при этом — терм, а не формула: истину оно высказать не может. Выражения $P(x)$, $Q(c, f(x, c))$, $\forall x P(f(x, x))$ — формулы: предикат получил ровно столько термов, какова его арность, а связки и кванторы построили из формул новые формулы. Выражение $Q(x)$ неправильно построено: арность Q равна двум, а аргумент один. Выражение $P(Q(x, y))$ неправильно построено по категории: аргументом предиката может быть только терм, а $Q(x, y)$ — формула.

Пример Сборка по шагам

Всякая формула представима как результат пошаговой сборки. Из переменной x и константы c правило термов строит $f(x, c)$. Предикат Q арности 2 превращает пару термов x и $f(x, c)$ в атомарную формулу $Q(x, f(x, c))$. Отрицание даёт $\neg Q(x, f(x, c))$, и квантор всеобщности завершает сборку: $\forall x \neg Q(x, f(x, c))$. Читается результат как «для каждого x объекты x и $f(x, c)$ не в отношении Q ». Перестановка шагов ломает сборку: квантор не может стоять над термом, а предикат не может принять формулу как аргумент.

Каждая формула — дерево построения: в корне последнее применённое правило, в листьях — переменные и константы, а атомарная формула стоит над своими термами, ведь предикат принимает именно термы. Все формулы прежних разделов — от $\neg \forall x P(x)$ до определения предела — собраны по этим правилам, и определение лишь наводит порядок задним числом. Дерево разбора — то, что хранит компилятор после успешного синтаксического анализа, и логика поступает с формулой так же.

§ 3.11 Семантика: модели и истинность

До сих пор мы строили *язык* логики предикатов: термы, формулы, кванторы. Чтобы говорить, истинна ли формула, нужна *интерпретация* языка — её называют также моделью или структурой.

ОПРЕДЕЛЕНИЕ 3.13 Интерпретация

Интерпретация I для языка с константами, функциональными символами и предикатами — назначение значений знакам языка:

- непустое множество D — **носитель** (предметная область);
- каждой константе c — значение $c^I \in D$;
- каждому функциональному символу f арности n — функция $f^I : D^n \rightarrow D$;

- каждому предикату P арности n — отношение $P^I \subseteq D^n$.

Истинность формулы в интерпретации определяется индукцией по построению формулы. Атом $P(t_1, \dots, t_n)$ истинен, если кортеж значений термов принадлежит P^I . Отрицание, конъюнкция и дизъюнкция — как в логике высказываний. Для кванторов $\forall x A(x)$ истинен, если A истинен для каждого элемента носителя, а $\exists x A(x)$ — если хотя бы для одного.

Схема истинности выше — не произвольное соглашение. Значение каждой сложной формулы вычисляется из значений её непосредственных частей, а значение атома назначает интерпретация. Требование вычислять значение целого из значений частей называют *композициональностью*, и основание у него точное: правил построения конечное число, а формул бесконечно много, и покрыть их можно только индукцией по построению.

Обходной путь недостижим: назначить истинность каждой формуле как целой строке — значит описать бесконечное множество формул бесконечным текстом. Всякий проект, избегающий рекурсии, воспроизводит её в скрытом виде — той же индукцией, только не названной. В главе 1 таблицы истинности уже работали композиционально, и на кванторах схема делает единственный новый шаг: значение $\forall x A$ собирается из значений A при всех назначениях объекта x из носителя.

Посмотрим, как работает эта схема, на знакомом материале — школьной математике.

Пример Интерпретация школьной математики

Сигнатура — это лишь набор знаков: константа 0, функциональный символ $+$, предикат $\leq$. Сами по себе знаки ничего не означают. Смысл им придаёт интерпретация. Возьмём носитель $D = \mathbb{Z}$ (целые числа) и назначим значения: $0^I = 0$, $+^I$ — обычное сложение, $\leq^I$ — обычный порядок.

Теперь формулы можно прочесть. $\forall x \exists y : x \leq y$ — для каждого целого найдётся целое, которое не меньше его. Это верно: подойдёт $y = x$. $\exists x \forall y : x \leq y$ — существует целое, не превосходящее всех остальных. Это неверно: у целых нет наименьшего элемента.

Оставим ту же сигнатуру, но сменим носитель на $D = \mathbb{N}$ (натуральные числа). Теперь вторая формула истинна: наименьшее натуральное число есть 0. Первая формула верна и на целых, и на натуральных числах.

Так одна и та же формула в разных интерпретациях ведёт себя по-разному: истинность — свойство пары «язык и интерпретация».

ОПРЕДЕЛЕНИЕ 3.14 Выполнимость

Формула **выполнима**, если существует интерпретация, в которой она истинна.

ОПРЕДЕЛЕНИЕ 3.15 **Общезначимость**

Формула **общезначима**, обозначается $\models A$, если она истинна в каждой интерпретации.

Пример **Общезначима, выполнима, невыполнима**

Формула $\forall x P(x) \rightarrow \exists x P(x)$ общезначима: носитель непуст, поэтому если P выполнена на всех элементах, то она выполнена хотя бы на одном, и посылка влечёт заключение. Формула $\forall x P(x)$ выполнима, но не общезначима: на носителе $\{a\}$ с $P^I = \{a\}$ она истинна, а на том же носителе с $P^I = \emptyset$ ложна. Формула $\forall x (P(x) \wedge \neg P(x))$ невыполнима: носитель непуст, конъюнкция ложна на каждом его элементе, и модели у формулы нет.

Запись $\Gamma \models \varphi$ (« Γ семантически влечёт φ ») означает, что в каждой интерпретации, где истинны все формулы Γ , истинна и φ . Выполнимость и общезначимость двойственны: A общезначима тогда и только тогда, когда $\neg A$ невыполнима. Для формул без кванторов общезначимость — это тавтология из главы 1: атомов конечное число, интерпретаций ровно 2^n , и проверка сводится к таблице истинности.

Примечание **С кванторами таблица не поможет**

С кванторами таблица истинности не работает: носитель бывает бесконечным, и перебрать его нельзя. Проверка общезначимости формулы логики первого порядка — неразрешимая задача (глава 26).

Проверка Семантика

1. Что назначает интерпретация символам сигнатуры и что задаёт носитель?
2. Формула истинна в одной интерпретации и ложна в другой. Что это говорит о её общезначимости и о её выполнимости?
3. Почему таблица истинности, спасавшая в логике высказываний, не проверяет общезначимость формулы с кванторами?

§ 3.12 Полнота и компактность

Синтаксис и семантика — две половины логики: формулы можно *выводить* и можно *интерпретировать*. Систему вывода для логики предикатов — аксиомы, правила и кванторные правила главы 2 — называют *исчислением первого порядка*.

ОПРЕДЕЛЕНИЕ 3.16 Выводимость

Вывод в исчислении — конечная последовательность формул, каждая из которых либо аксиома, либо получена правилом из предыдущих. Запись $\Gamma \vdash \varphi$ (« φ выводима из Γ ») означает, что такой вывод существует.

Между выводимостью и семантическим следованием есть два обязательных моста:

- *Корректность* (soundness): если $\Gamma \vdash \varphi$, то $\Gamma \models \varphi$. Каждое правило вывода сохраняет истинность, поэтому вывод не может «обмануть».
- *Полнота* (completeness): если $\Gamma \models \varphi$, то $\Gamma \vdash \varphi$. Всё, что истинно во всех моделях, можно вывести — ничего не упущено.

Для логики высказываний оба свойства доказаны в главе 2. У первого порядка корректность проверяется так же тривиально, а полнота — нетривиальный результат: чтобы связать вывод с истинностью во всех моделях, теорема Гёделя строит модель из термов самого языка.

ТЕОРЕМА 3.2 Теорема Гёделя о полноте

Исчисление первого порядка корректно и полно: $\Gamma \models \varphi$ тогда и только тогда, когда $\Gamma \vdash \varphi$.¹⁷

Доказательство полноты — та же конструкция Генкина, что и в главе 2, с одним дополнением. Максимальное непротиворечивое множество расширяется так, что для каждой формулы $\exists x A(x)$ в нём лежит и $A(c)$ для некоторой новой константы c — свидетеля.

Тогда носителем модели служат сами термы языка, и индукция по сложности формул завершается.

Из полноты следует главный результат метатеории:

ТЕОРЕМА 3.3 Теорема компактности

Множество формул логики первого порядка выполнимо тогда и только тогда, когда выполнимо каждое его конечное подмножество.

Доказательство

Докажем утверждение в обе стороны.

Прямое направление. Если Γ выполнимо, то выполнимо и каждое его конечное подмножество. Пусть M — модель Γ . Для любого конечного $\Delta \subseteq \Gamma$ та же M удовлетворяет всем формулам Δ — все они принадлежат Γ . Значит, Δ выполнимо.

¹⁷K. Gödel. «Die Vollständigkeit der Axiome des logischen Funktionenkalküls» («О полноте аксиом логического функционального исчисления»). Monatshefte für Mathematik und Physik, 1930.

Обратное направление. Докажем контрапозицией. Предположим, что Γ невыполнимо, то есть у Γ нет модели. Тогда Γ семантически влечёт что угодно — в частности, противоречие: $\Gamma \models \perp$. По теореме о полноте из Γ выводится противоречие: $\Gamma \vdash \perp$. Вывод конечен, поэтому в нём участвует лишь конечное множество формул $\Gamma_0 \subseteq \Gamma$. Значит, $\Gamma_0 \vdash \perp$, а по корректности $\Gamma_0 \models \perp$, то есть Γ_0 невыполнимо: конечное подмножество без модели найдено.

Мы показали, что из невыполнимости Γ следует невыполнимость некоторого конечного подмножества. Это контрапозиция обратного направления, поэтому оно доказано: если каждое конечное подмножество выполнимо, то и Γ выполнимо.

Из компактности следуют три результата.

СЛЕДСТВИЕ 3.4 Конечное следствие

Если $\Gamma \models \varphi$, то φ следует уже из конечного подмножества Γ .

Доказательство

Докажем от противного: пусть φ не следует ни из какого конечного подмножества Γ . Тогда для каждого конечного $\Sigma \subseteq \Gamma$ существует модель, в которой истинны все формулы Σ , а φ ложна. Значит, выполнимо каждое конечное подмножество множества $\Gamma \cup \{\neg\varphi\}$: его конечная часть из Γ покрыта такой моделью, а добавленная $\neg\varphi$ в ней уже истинна. По теореме компактности выполнимо и всё $\Gamma \cup \{\neg\varphi\}$ — вопреки $\Gamma \models \varphi$.

СЛЕДСТВИЕ 3.5 Нестандартные модели

Теория натуральных чисел с константой c и аксиомами $c > 0, c > 1, c > 2, \dots$ выполнима. Каждое конечное подмножество имеет модель в $\mathbb{N}$: аксиом в нём конечное число, и c берётся больше всех упомянутых в них чисел. Значит, модель есть и у всей теории, и в ней c больше каждого стандартного натурального числа — это и есть нестандартные модели арифметики (глава 29).

СЛЕДСТВИЕ 3.6 Второй порядок

В логике второго порядка компактность неверна: там можно выразить «универсум конечен», и конечные универсумы сколь угодно большого размера не складываются в бесконечный. Оговорка: кванторы второго порядка связывают переменные по предикатам, а это уже вне языка первого порядка, где переменные пробегают только объекты носителя. Насколько различаются такие языки, измеряет теория мощностей из главы 7.

§ 3.13 Итоги главы

Логика высказываний не видела структуры атома: целое утверждение было для неё просто буквой p , и порядок слов — неразличим. Предикаты и кванторы заглянули внутрь атома: предикат — шаблон с дырой, который становится высказыванием при подстановке значения, а кванторы $\forall$ и $\exists$ задают размах шаблона — для всех или хотя бы для одного. Главное, что мы теперь видим, — порядок кванторов меняет смысл: $\forall x \exists y L(x, y)$ разрешает y зависеть от x , а $\exists y \forall x L(x, y)$ требует одного общего y для всех.

Механику языка мы собрали целиком. Отрицание кванторов подчиняется законам де Моргана: $\neg \forall x P(x)$ равносильно $\exists x \neg P(x)$, а $\neg \exists x P(x) = \forall x \neg P(x)$. Ограниченные кванторы $\forall x \in A$ и $\exists x \in A$ оказались сокращениями: первый раскрывается в импликацию, второй — в конъюнкцию. Перевод с естественного языка мы научились делать как компиляцию — каждая неоднозначная фраза получает ровно одну формулу, которую можно отрицать, соединять и проверять механически.

Сорты типизировали переменные: вместо одной предметной области — несколько, по типу объекта, и именно сорта лежат в основе SMT-солверов, сочетающих логику первого порядка с теориями. Вторая половина — семантика: модель, интерпретация и истинность формулы в модели. Она согласована с выводимостью из главы 2: теорема Гёделя о полноте связывает истинность и выводимость в точную пару.

Из полноты следует компактность: если каждое конечное подмножество теории имеет модель, то модель есть и у всей теории. Это источник нестандартных моделей арифметики, к которым мы вернёмся в главе 29.

Проверка Проверьте себя

1. Почему порядок кванторов меняет смысл формулы? Приведите пример пары формул с разным порядком.
2. Запишите отрицания $\forall x P(x)$ и $\exists x P(x)$.
3. Что такое сорт и чем многосортная логика удобнее односортной?
4. Почему в ограниченном кванторе $\forall x \in A$ стоит импликация, а в $\exists x \in A$ — конъюнкция?
5. Что утверждает теорема Гёделя о полноте и что из неё следует для нестандартных моделей?

Что дальше?

Логика даёт язык для утверждений об объектах. В главе 4 мы перейдём от утверждений к самим объектам и изучим множества — фундамент дискретной математики.

§ 3.14 Упражнения

1. Переведите на язык логики предикатов: «Каждый студент, изучающий дискретную математику, умеет программировать». Запишите отрицание этого утверждения — формулой и на естественном языке.
2. Пусть $P(x, y)$ означает « x знает y », а предметная область — все люди. Запишите формулами: «Каждый знает кого-то» и «Существует тот, кого знают все». Какая из формул истинна при вашей интерпретации, а какая — нет? Объясните разницу в порядке кванторов.
3. Приведите пример предиката и предметной области, при которых $\forall x \exists y P(x, y)$ истинно, а $\exists y \forall x P(x, y)$ ложно.
4. Почему в ограниченном кванторе $\forall x \in A$ стоит импликация, а в $\exists x \in A$ — конъюнкция?
5. * Запишите утверждение «каждый студент сдал хотя бы один экзамен» в многосортной логике с сортами студентов и экзаменов. В чём преимущество сортов перед одной предметной областью?
6. Общезначима ли формула $\forall x (P(x) \vee \neg P(x))$? А выполнима ли $\exists x \forall y R(x, y)$?
7. * Выведите следствие конечности из компактности: если $\Gamma \models \varphi$, то φ следует из конечного подмножества Γ .
8. Какие вхождения переменных в формулу $\forall x (P(x) \rightarrow \exists y Q(x, y)) \vee R(x, z)$ свободны, а какие связаны?
9. Для сигнатуры с константой c , функцией f арности 2 и предикатами P арности 1 и Q арности 2 определите, какие из выражений $f(x, c)$, $P(f(x, c))$, $Q(x, P(y))$, $P(Q(x, y))$, $\forall x P(f(x, x))$ являются термами, какие — формулами, а какие неправильно построены. Для каждого неправильно построенного укажите нарушение: категория или арность.
10. Выпишите шаги сборки формулы $\forall x Q(c, f(x, x))$ из сигнатуры предыдущей задачи: какое правило из определений термов и формул работает на каждом шаге.
11. Приведите к предварённой нормальной форме формулу $\neg \forall x P(x) \vee \exists y Q(y)$.
12. Запишите формулой «ровно один студент сдал все экзамены» через $\exists$ и $\forall$, и выпишите её отрицание.
13. * Докажите общезначимость «парадокса пьющего»: $\exists x (P(x) \rightarrow \forall y P(y))$ истинна в любой непустой области. Объясните, почему это не утверждает существование конкретного «пьющего».
14. Пошагово вычислите отрицание формулы $\forall x \exists y \forall z P(x, y, z)$.
15. * Докажите, что формула $\forall x \forall y (P(x, y) \rightarrow P(y, x))$ не общезначима: предъявите модель из двух элементов, где она ложна.

ПРОЕКТ Экзекьютор формул: перебор моделей и контрпримеры

Формула с кванторами говорит обо всех интерпретациях сразу, и вручную это не проверишь. Экзекьютор — программа, которая по формуле и сигнатуре строит

конечные интерпретации заданного размера и отвечает на вопросы: общезначима ли формула, выполнима ли она, а если нет — в какой модели она ломается. Название подсказывает происхождение: исполнитель формул делает то, что в логике высказываний делала таблица истинности.

① *Парсер формул*

Задайте сигнатуру: константы, функциональные символы и предикаты с арностями. Научите программу читать термы и формулы: переменные, связки $\neg, \wedge, \vee, \rightarrow$, кванторы $\forall x$ и $\exists x$, скобки. Стройте дерево разбора рекурсивным спуском и отклоняйте выражения, где связка применена к терму, аргумент нарушает арность или квантор остался без тела. Проверьте на формулах главы: $\forall x(P(x) \rightarrow \exists y Q(x, y))$ разбирается, а $P(Q(x))$ отвергается с указанием места ошибки.

② *Оценка в интерпретации*

Конечная интерпретация — это носитель из n элементов, таблица значений на кортежах для каждого предиката и таблица значений для каждой функции. Вычислите значение формулы в интерпретации обходом дерева разбора, храня назначение: какой элемент носителя стоит за каждой свободной переменной. Значение $\forall x A$ собирается из значений A при всех элементах носителя, а значение $\exists x A$ — хотя бы при одном.

③ *Перебор моделей*

Для замкнутой формулы переберите все интерпретации на носителях из 2 и 3 элементов: все подмножества D^k для предиката арности k , все таблицы $D^n \rightarrow D$ для функции арности n . На трёх элементах у одного двуместного предиката $2^9 = 512$ вариантов, а у двуместной функции $D^2 \rightarrow D = 3^9 = 19683$, и счёт растёт быстро. Формула общезначима, если истинна в каждой построенной модели, и выполнима, если истинна хотя бы в одной. Сверьте с ответами: $\forall x(P(x) \vee \neg P(x))$ общезначима, $\forall x P(x)$ выполнима, но не общезначима, $\forall x (P(x) \wedge \neg P(x))$ невыполнима.

④ *Отчёт-контрпример*

Если формула не общезначима, выведите модель, в которой она ложна: носитель, таблицы предикатов и функций и элемент, на котором формула принимает значение «ложь». Если модель не нашлась ни при одном размере, сообщите о невыполнимости в пределах перебора. Проверьте формулу $\forall x \forall y (P(x, y) \rightarrow P(y, x))$: контрпримером служит несимметричное отношение на двух элементах.

⑤ *Оптимизация*

Замерьте, как растёт число моделей с арностями и числом символов, и уберите лишнюю работу: ранний выход при первом опровергающем назначении, переиспользование вычисленных подформул, отсеивание моделей до полного построения таблиц. Сравните время до первого контрпримера до и после.

Итог: инструмент, который по формуле и сигнатуре перебирает конечные модели, различает общезначимость и выполнимость и опровергает ложную общезначимость явным контрпримером — моделью из двух-трёх элементов.

IV

Наивная теория множеств

Запись $\{1, 2, 3\} = \{3, 1, 2\} = \{1, 1, 2\}$ выглядит ошибкой: у массива есть порядок и длина, и все три массива различны. Для множества это одно и то же — три имени одного объекта. Порядок и повторения не существуют: множество отвечает ровно на один вопрос, входит ли в него объект. Такая экономия кажется слишком бедной, чтобы на ней что-то построить.

Главы 1 и 3 построили язык: связки, кванторы, правила вывода. Язык описывает высказывания, но объекты, о которых высказываются, остались без формы. Множество — первый материал для них: контейнер, в котором объект либо внутри, либо снаружи. Именно из этого материала собирается всё: граф — множества вершин и рёбер, автомат — множество состояний, функция — множество пар. Прежде чем строить, нужно разобраться в самом контейнере.

Программист и так каждый день работает с множествами, часто не называя их. Таблица базы данных — подмножество декартова произведения, и каждый SQL-запрос — цепочка теоретико-множественных операций. Тип данных — множество значений: `bool` — это $\{T, F\}$, а `uint8` — числа от 0 до 255. Права доступа Unix — три бита на класс пользователей, и `chmod` складывает множества побитовым «или». Множество кодируется числом, а операции над множествами становятся побитовыми операциями процессора.

За привычным словом прячется точность. Два разных описания задают одно и то же множество: $\{1, 2, 3\} = \{x \in \mathbb{N} \mid 0 < x < 4\}$. *Экстенциональность* ставит на этом точку: множество определяется содержимым, а не способом описания. Родство с логикой здесь буквальное: объединение повторяет дизъюнкцию, пересечение — конъюнкцию, дополнение — отрицание, и глава 10 покажет эту алгебру целиком.

Аксиома выбора, континуум-гипотеза и парадокс Банаха–Тарского отложены до главы 22. Для повседневной работы достаточно прагматической части: множества строятся в фиксированном универсуме явными операциями.

А декартово произведение станет материалом следующей главы: бинарное отношение — множество упорядоченных пар (глава 5).

Как из единственного различия «внутри или снаружи» вырастает вся математика — и где это различие ломается?

ИСТОРИЯ Рождение теории множеств и кризис оснований

Георг Кантор в работе 1874 года «Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen» доказал несчётность действительных чисел: существует бесконечность, принципиально большая, чем бесконечность натуральных чисел. Это открытие положило начало теории множеств. В последующие два десятилетия Кантор развил понятия мощности, кардинальных и ординальных чисел, операции над бесконечными множествами.

В 1901 году Бертран Рассел обнаружил парадокс: множество всех множеств, не содержащих себя в качестве элемента, приводит к противоречию. В июне 1902 года он написал Фреге, который как раз завершал второй том «Grundgesetze der Arithmetik» — труд всей жизни, построенный на наивной теории множеств. Фреге добавил в предисловие фразу: «Вряд ли что-то более нежелательное может случиться с учёным, чем обнаружить, что один из фундаментов его труда пошатнулся, когда работа уже завершена». Теория, созданная Кантором, оказалась под угрозой: если любое свойство определяет множество, то теория противоречива.

Эрнст Цермело в 1908 году предложил первую аксиоматизацию теории множеств: множество определяет только свойство, ограниченное уже существующим множеством (аксиома выделения). Абрахам Френкель и Торальф Сколем в 1922 году независимо добавили аксиому подстановки. Возникла система ZFC — аксиоматика Цермело–Френкеля с аксиомой выбора — которая стала стандартным фундаментом математики. Кризис был преодолён: теория множеств выжила, но ценой отказа от наивной интуиции «набора всего, что удовлетворяет свойству».

ОБЗОР ГЛАВЫ

Глава строит язык теории множеств с нуля: три отношения — принадлежность, включение, равенство — и пять операций, наглядных на диаграммах Венна. Из множеств вырастут две конструкции, на которых стоят все структуры курса: декартово произведение и булеан. Следом разберутся семейства и разбиения. Наивная интуиция «набор всего, что удовлетворяет свойству» разобьётся о парадокс Рассела, и из кризиса выйдет аксиома выделения — первый камень аксиоматики ZFC. Затем множества покажут себя в деле: таблицы баз данных, типы в языках программирования, битовые маски и хеширование. В дополнительных разделах глава рассматривает аксиоматическую теорию множеств целиком и кардиналы — бесконечные мощности, которые начинаются с $\aleph_0$.

Множества лежат в основе курса: из них соберутся отношения (глава 5), функции (глава 6) и кардиналы (глава 7), а связь с логикой закрепится в булевой алгебре (глава 10).

После этой главы вы сможете:

1. записывать множества перечислением и свойством и понимать, когда два описания задают одно и то же множество;
2. проверять принадлежность, включение и равенство и пользоваться булеаном и декартовым произведением;
3. применять пять операций над множествами и читать их на диаграммах Венна;
4. объяснять парадокс Рассела и роль аксиомы выделения;
5. видеть множества в инженерии: таблицы баз данных, типы, битовые маски и хеширование.

§ 4.1 Множество и его элементы

Множество — это неупорядоченный набор различных объектов. Объекты в множестве называются его *элементами* (или *членами*). В множестве нет понятия кратности или порядка: набор различим только по составу.

ОПРЕДЕЛЕНИЕ 4.1 Принадлежность множеству

Говорят, что элемент a **принадлежит** множеству A , и пишут $a \in A$, если a входит в его состав. Если a не входит в A , пишут $a \notin A$.

Пример Принадлежность элементу

Пусть $A = \{1, 3, 5, 7\}$. Тогда:

- $3 \in A$ (истинно),
- $4 \notin A$ (истинно),
- $\{3\} \in A$ (ложно: элементами A являются числа, а не множество $\{3\}$).

Множество можно задать несколькими способами.

ОПРЕДЕЛЕНИЕ 4.2 Перечисление

Множество задано **перечислением**, если все его элементы выписаны в фигурных скобках: $\{1, 2, 3, 4\}$.

ОПРЕДЕЛЕНИЕ 4.3 Нотация порождающего свойства (*set-builder notation*)

Множество задано **нотацией порождающего свойства** — записью $\{x \in U \mid P(x)\}$, обозначающей множество всех элементов универсума U , удовлетворяющих свойству P .

Пример Нотация порождающего свойства

- $\{x \in \mathbb{N} \mid x < 5\} = \{0, 1, 2, 3, 4\}$.
- $\{x \in \mathbb{N} \mid x \text{ является простым} \wedge x < 10\} = \{2, 3, 5, 7\}$.

ОПРЕДЕЛЕНИЕ 4.4 Рекурсивное задание множества

Множество задано **рекурсивно**, если указан его базовый элемент и правила, порождающие из уже построенных элементов новые.

Пример Рекурсивное определение множества

Множество чётных натуральных чисел задаётся рекурсивно. База: $0 \in E$. Шаг: $n \in E \rightarrow n + 2 \in E$. Никакие другие числа в E не входят. Тем же приёмом задаются формальные языки и деревья в следующих главах.

Налей ещё больше чаю, — весьма серьёзно сказал Алисе Мартовский Заяц.

— Я ещё *ничего* не налила, — обиженно ответила Алиса, — так что «больше» налить не получится.

— Ты хотела сказать — *меньше*? — заметил Шляпник. — Налить *больше*, чем ничего — проще простого.

— Льюис Кэрролл

ОПРЕДЕЛЕНИЕ 4.5 Пустое множество

Множество без элементов называется **пустым** и обозначается $\emptyset$.

Пример Пустое множество

- $\emptyset = \{\}$ — пустое множество.
- $|\emptyset| = 0$, и вообще $|A|$ — число элементов множества A .
- $\emptyset \in \{\emptyset\}$ (истинно — пустое множество является элементом этого множества).
- $\emptyset \subseteq \{\emptyset\}$ (тоже истинно — пустое множество является подмножеством любого множества).

Примечание Зачем нужно пустое множество

$\emptyset$ — «ноль» теории множеств: объект без элементов, сам являющийся полноценным множеством — в точности как ноль есть число, обозначающее отсутствие количества. Без него алгебра множеств не замкнута: пересечение непересекающихся множеств оставалось бы неопределённым, а разность $A \setminus A$ не имела бы значения. Ноль в арифметике играет ту же роль: без него вычитание $a - a$ не определено.

Мы определили, что значит быть элементом множества, и научились записывать множества. Но описание не тождественно объекту. Одно и то же множество может быть задано бесконечно многими способами: $\{1, 2, 3\} = \{x \in \mathbb{N} \mid 0 < x < 4\} = \{x \in \mathbb{N} \mid x^2 < 16, x > 0\}$ — три разных текста, один и тот же набор элементов. Если

множество отождествить с его описанием, каждое новое описание порождает новое множество — бесконечное число неразличимых копий одного и того же.

Возникает проблема тождества: два описания могут задавать одно и то же множество, и нужен критерий, чтобы это установить. Синтаксическое сравнение описаний бесполезно: $\{1, 2, 3\}$ и $\{3, 1, 2\}$ — разные строки. Семантическое сравнение — проверка логической эквивалентности задающих свойств — в общем случае не сводится к конечной процедуре. Для конечных множеств вопрос решает сверка по элементам, и этот критерий стоит закрепить как закон теории.

Критерий равенства — *сознательное решение*: мы выбираем, чем задаётся множество. Альтернатива — интенциональное определение, где множества отождествляются с задающими их свойствами — сделала бы равенство либо чисто синтаксическим, либо алгоритмически неразрешимым. Выбранный критерий даёт разрешимую проверку равенства для конечных множеств и однозначно фиксирует объекты теории. В аксиоматической системе ZFC этот принцип закреплён как аксиома объёмности.

§ 4.2 Равенство и подмножества

ОПРЕДЕЛЕНИЕ 4.6 Экстенциональность

Два множества называются **равными** ($A = B$), если они содержат в точности одни и те же элементы:

$$\forall x \quad (x \in A \leftrightarrow x \in B)$$

Пример Равенство множеств: порядок и повторы

- $\{1, 2, 3\} = \{3, 1, 2\}$ (порядок не важен).
- $\{1, 2, 2\} = \{1, 2\}$ (повторения игнорируются — множество содержит каждый элемент не более одного раза).
- $\{x \in \mathbb{N} \mid x < 3\} = \{0, 1, 2\}$ (разные способы задания, одно и то же множество).

Равенство — самое сильное отношение между множествами: полное совпадение состава. Но часто нужно более тонкое сравнение: когда одно множество целиком помещается внутри другого, не обязательно исчерпывая его.

Замечание Тождество неразличимых

Лейбниц (1686) сформулировал принцип тождества неразличимых (*identitas indiscernibilium*): два объекта тождественны, если и только если они обладают одними и теми же свойствами. Экстенциональность — специализация этого принципа: единственное «свойство» элемента, значимое для тождества множе-

ства, — это принадлежность. Множество — «чёрный ящик»: его идентичность полностью определяется содержимым.

ОПРЕДЕЛЕНИЕ 4.7 Подмножество

Множество A называется **подмножеством** множества B (обозначается $A \subseteq B$), если каждый элемент A является элементом B :

$$\forall x \quad (x \in A \rightarrow x \in B)$$

ОПРЕДЕЛЕНИЕ 4.8 Собственное подмножество

Множество A называется **собственным подмножеством** множества B (обозначается $A \subset B$), если $A \subseteq B$, но $A \neq B$.

Пример Подмножество и собственное подмножество

Пусть $A = \{1, 2\}$, $B = \{0, 1, 2, 3\}$.

- $A \subseteq B$ (все элементы A содержатся в B).
- $A \subset B$ (собственное подмножество: A не равно B).
- $B \subseteq A$ ложно (0 и 3 не принадлежат A).
- $\{1, 2\} \subseteq \{1, 2\}$ истинно. Строгое включение $\{1, 2\} \subset \{1, 2\}$ ложно: это не собственное подмножество, множества равны.

ЛОВУШКА Принадлежность или подмножество?

Знаки принадлежности и включения работают на разных уровнях. Запись $x \in A$ утверждает: x — элемент множества A . Запись $B \subseteq A$ утверждает: множество B целиком лежит внутри A . Поэтому $1 \in \{1, 2\}$ — истинное высказывание. $A \subseteq \{1, 2\}$ бессмысленно: в наивной теории множеств число не является множеством. (В аксиоматике ZFC числа фон Неймана — множества, см. раздел об аксиоме бесконечности.)

Подмножеством $\{1, 2\}$ будет $\{1\}$ — множество из одного элемента. Запомнить помогает само слово: в «подмножестве» живёт «множество». Подмножество всегда само является множеством, а элемент множеством быть не обязан.

Оба уровня видны одновременно на множестве $\{\emptyset, \{\emptyset\}\}$. $\emptyset \in \{\emptyset, \{\emptyset\}\}$ истинно: пустое множество здесь элемент. Истинно и $\emptyset \subseteq \{\emptyset, \{\emptyset\}\}$: пустое множество — подмножество любого множества. Обе записи осмысленны и означают разное.

ЛОВУШКА Уровни вложенности: $\emptyset \neq \{\emptyset\}$

$\emptyset$ — множество без элементов. $\{\emptyset\}$ — множество, содержащее пустое множество. Мощности разные: $|\emptyset| = 0$, а $|\{\emptyset\}| = 1$. В $\{\emptyset, \{\emptyset\}\}$ ровно два элемента: пустое множество и множество, которое его содержит.

ТЕОРЕМА 4.1 Равенство через подмножества

$$A = B \leftrightarrow A \subseteq B \wedge B \subseteq A.$$

Доказательство

Докажем эквивалентность в обе стороны.

Прямое направление. Если $A = B$, то по определению экстенсинальности $x \in A \leftrightarrow x \in B$. Отсюда $x \in A \rightarrow x \in B$ и $x \in B \rightarrow x \in A$. Первое означает $A \subseteq B$, второе — $B \subseteq A$.

Обратное направление. Если $A \subseteq B$ и $B \subseteq A$, то обе импликации дают $x \in A \leftrightarrow x \in B$. По определению экстенсинальности это означает $A = B$.

Эта теорема даёт стандартный метод доказательства равенства множеств: показать, что каждая сторона содержится в другой.

§ 4.3 Булеан

Мы умеем сравнивать множества и проверять, лежит ли одно внутри другого. Соберём все подмножества данного множества в одно новое множество — это и есть *булеан*.

ОПРЕДЕЛЕНИЕ 4.9 Булеан (*power set*)

Булеан множества A — множество всех его подмножеств, обозначается $\mathcal{P}(A) = 2^A = \{X \mid X \subseteq A\}$.

ТЕОРЕМА 4.2 Мощность (*cardinality*) булеана

Если $|A| = n$ и A конечно, то

$$|\mathcal{P}(A)| = 2^n$$

Набросок доказательства

Докажем подсчётом.

Перенумеруем элементы множества $A: a_1, a_2, \dots, a_n$. Каждое подмножество $X \subseteq A$ однозначно задаётся ответами на n вопросов. Для каждого a_i вопрос один: входит ли элемент в X . Для каждого элемента два варианта — «да» или «нет» — и выборы независимы. Получаем 2^n различных комбинаций и столько же различных подмножеств.

Пример Булеан множества из двух элементов

Для $A = \{a, b\}$: $\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Действительно, $|\mathcal{P}(A)| = 4 = 2^2$.

Пример Булеан пустого множества

Крайний случай — пустое множество: $\mathcal{P}(\emptyset) = \{\emptyset\}$. Единственное подмножество пустого множества — оно само. Формула мощности даёт $2^0 = 1$ — согласовано.

Добавление одного элемента к A удваивает размер булеана.

- Для $n = 10$ имеем $2^{10} = 1024$ подмножества.
- Для $n = 20$ — уже более миллиона.
- Для $n = 30$ — миллиард.

Именно поэтому задача «найти подмножество с заданным свойством» решается полным перебором только для малых n — экспоненциальный рост делает перебор безнадёжным при увеличении размера.



Теорема гарантирует мощность булеана: 2^n . Но перечислить все подмножества невозможно уже при умеренных n . Из всех операций булеан — единственная, где существование и конструктивная перечислимость расходятся так рано. «Существует» и «можно предъявить» — не одно и то же.

§ 4.4 Операции над множествами

Для множеств A, B , взятых из универсального множества U , определены пять операций.

ОПРЕДЕЛЕНИЕ 4.10 Объединение

Объединение множеств A и B — множество $A \cup B = \{x \mid x \in A \vee x \in B\}$.

ОПРЕДЕЛЕНИЕ 4.11 Пересечение

Пересечение множеств A и B — множество $A \cap B = \{x \mid x \in A \wedge x \in B\}$.

ОПРЕДЕЛЕНИЕ 4.12 Разность

Разность множеств A и B — множество $A \setminus B = \{x \mid x \in A \wedge x \notin B\}$.

ОПРЕДЕЛЕНИЕ 4.13 Симметрическая разность

Симметрическая разность множеств A и B — множество $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

ОПРЕДЕЛЕНИЕ 4.14 Дополнение

Дополнение множества A — множество $\bar{A} = U \setminus A = \{x \in U \mid x \notin A\}$, определённое относительно фиксированного универсума U .

Дополнение определено только относительно универсума U : дополнения «вообще» не существует. Говоря о «всех не-овчарках», мы имеем в виду не-овчарок среди собак, а не среди всех животных — универсум фиксируется контекстом.

Два множества называются *непересекающимися*, если $A \cap B = \emptyset$.

Пример Операции над множествами

Пусть $A = \{1, 2, 3\}$, $B = \{2, 3, 4\}$, $U = \{1, 2, 3, 4, 5\}$.

- $A \cup B = \{1, 2, 3, 4\}$
- $A \cap B = \{2, 3\}$
- $A \setminus B = \{1\}$
- $A \Delta B = \{1, 4\}$
- $\bar{A} = \{4, 5\}$

Пять операций — заметное количество. Каждая операция зеркально отражает логическую связку: $\cup$ повторяет $\vee$, $\cap$ — $\wedge$, $\bar{X}$ — $\neg$, $\setminus$ — $\wedge \neg$, Δ — $\oplus$. Даже $\subseteq$ работает как $\rightarrow$, а $=$ — как $\leftrightarrow$.

Пять операций — это пять проявлений трёх базовых: объединения, пересечения и дополнения — с удобными сокращениями для часто встречающихся комбинаций.

§ 4.5 Соответствие операций над множествами и логических связок

Зафиксируем это соответствие таблицей, а затем выведем из него законы алгебры множеств.

Операция над множествами	Логический аналог
$A \cup B$	$x \in A \vee x \in B$
$A \cap B$	$x \in A \wedge x \in B$

Операция над множествами	Логический аналог
$\bar{A}$	$\neg(x \in A)$
$A \setminus B$	$x \in A \wedge \neg(x \in B)$
$A \Delta B$	$x \in A \oplus x \in B$
$A \subseteq B$	$x \in A \rightarrow x \in B$
$A = B$	$x \in A \leftrightarrow x \in B$

ОПРЕДЕЛЕНИЕ 4.15 Изоморфизм

Две структуры (множества с операциями) называются **изоморфными**, если существует биекция между их элементами, сохраняющая все операции. Такая биекция называется **изоморфизмом**.

Сохранение операций означает, что результат операции не зависит от того, в какой из двух структур её выполнять. Биекция — это взаимно однозначное соответствие (подробнее — глава 6).

Именно так устроена таблица выше: алгебра подмножеств $(2^U, \cup, \cap, \bar{})$ изоморфна алгебре характеристических функций. Характеристическая функция — это отображение $U \rightarrow \{T, F\}$, равное T на элементах множества и F вне его (подробно — глава 6). Операции над функциями покомпонентные: $\vee, \wedge, \neg$. Каждое подмножество $A \subseteq U$ отождествляется с предикатом принадлежности $x \in A$. Тогда объединение превращается в дизъюнкцию, пересечение — в конъюнкцию, дополнение — в отрицание.

Например, возьмём $U = \{1, 2, 3\}$. Множество $A = \{1, 3\}$ записывается тройкой ответов (T, F, T) : первый элемент универсума входит, второй нет, третий входит. Множество $B = \{1\}$ — тройкой (T, F, F) . Тогда $A \cup B = \{1, 3\}$ даёт (T, F, T) — ровно покомпонентная дизъюнкция $(T, F, T) \vee (T, F, F)$.

Алгебра подмножеств и логика высказываний — две реализации одной и той же структуры. Такую структуру называют *булевой алгеброй*, и в главе 10 мы изучим её в общем виде.

УТВЕРЖДЕНИЕ 4.3 Законы алгебры множеств

Для любых множеств A, B, C :

- **Коммутативность:** $A \cup B = B \cup A, A \cap B = B \cap A$.
- **Ассоциативность:** $(A \cup B) \cup C = A \cup (B \cup C), (A \cap B) \cap C = A \cap (B \cap C)$.
- **Дистрибутивность:** $A \cap (B \cup C) = (A \cap B) \cup (A \cap C), A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
- **Идемпотентность:** $A \cup A = A, A \cap A = A$.
- **Тождество:** $A \cup \emptyset = A, A \cap U = A$.
- **Дополнение:** $A \cup \bar{A} = U, A \cap \bar{A} = \emptyset$.
- **Двойное дополнение:** $\bar{\bar{A}} = A$.

Набросок доказательства

Докажем каждый закон переводом в соответствующую логическую эквивалентность из таблицы выше.

Например, коммутативность объединения: $x \in A \cup B \leftrightarrow x \in A \vee x \in B \leftrightarrow x \in B \vee x \in A \leftrightarrow x \in B \cup A$. Аналогично, дистрибутивность пересечения относительно объединения следует из дистрибутивности конъюнкции относительно дизъюнкции: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$. Законы де Моргана доказаны отдельно ниже.

Эти законы — алгебраический базис: с их помощью любое выражение из операций над множествами можно упрощать и преобразовывать, как арифметическое. Два из них, однако, выделяются из общего ряда: они управляют тем, как дополнение взаимодействует с объединением и пересечением.

Законы де Моргана связывают дополнение с объединением и пересечением. Без них $\cup$, $\cap$ и $\bar{}$ существовали бы независимо: первые две соединяют, третья отрицает. Де Морган показывает, как дополнение «протаскивается» сквозь объединение и пересечение, меняя одно на другое. Отрицание объединения есть пересечение отрицаний, отрицание пересечения есть объединение отрицаний. Это та же логика, что в тождествах $\neg(p \vee q) \equiv \neg p \wedge \neg q$, только записанная для множеств.

Пример Законы де Моргана: конкретная проверка

Пусть $U = \{1, 2, 3, 4, 5\}$, $A = \{1, 2, 3\}$, $B = \{2, 3, 4\}$. Тогда $A \cup B = \{1, 2, 3, 4\}$, $\overline{A \cup B} = \{5\}$. С другой стороны, $\bar{A} = \{4, 5\}$, $\bar{B} = \{1, 5\}$, $\bar{A} \cap \bar{B} = \{5\}$. Результат совпадает: дополнение объединения равно пересечению дополнений. Аналогично, $\overline{A \cap B} = \{1, 4, 5\} = \bar{A} \cup \bar{B}$ — второй закон де Моргана выполняется.

ТЕОРЕМА 4.4 Законы де Моргана для множеств

- $\overline{A \cup B} = \bar{A} \cap \bar{B}$.
- $\overline{A \cap B} = \bar{A} \cup \bar{B}$.

Доказательство

Докажем методом взаимного включения.

Включение $\subseteq$. Пусть $x \in \overline{A \cup B}$, тогда $x \notin (A \cup B)$, откуда $x \notin A$, $x \notin B$. Здесь применён закон де Моргана для логики из главы 1 — в этом весь смысл соответствия из таблицы выше. Следовательно, $x \in \bar{A} \cap \bar{B}$.

Обратное включение $\supseteq$. Пусть $x \in \bar{A} \cap \bar{B}$, тогда $x \notin A$, $x \notin B$, откуда $x \notin (A \cup B)$. Следовательно, $x \in \overline{A \cup B}$.

Второй закон де Моргана доказывается дуально.

Мы построили алгебру множеств: пять операций, пятнадцать законов. Всё это выводится из трёх определений — объединения, пересечения и дополнения — и логических эквивалентностей из предыдущей главы. Алгебра множеств не добавляет новых правил. Она применяет правила логики к новому типу объектов. Это видно на первой же задаче: при упрощении выражения $(A \cup B) \cap (\overline{A \cap B})$ работают ровно те же законы, что и с булевыми формулами:

$$(A \cup B) \cap \overline{(A \cap B)} = (A \cup B) \cap (A \cup \overline{B}) = A \cup (B \cap \overline{B}) = A.$$

Первый шаг — закон де Моргана, второй — дистрибутивность, третий — дополнение $B \cap \overline{B} = \emptyset$.

§ 4.6 Диаграммы Венна

Диаграммы Венна изображают множества как перекрывающиеся области, обычно внутри ограничивающего прямоугольника, представляющего универсум U . Каждая операция над множествами получает наглядную геометрическую интерпретацию, как на рис. 11:

1. объединение — все точки, попавшие хотя бы в один круг.
2. пересечение — точки, попавшие в оба круга одновременно.
3. дополнение — все точки, не попавшие в данный круг.
4. разность — точки первого круга, не попавшие во второй.
5. симметрическая разность — точки, попавшие ровно в один из двух кругов.
6. подмножество — один круг целиком внутри другого.

Диаграммы Венна эффективны для двух или трёх множеств. Для большего числа множеств рассуждают алгебраически. Все шесть операций — объединение, пересечение, дополнение, симметрическая разность, разность и включение — наглядно представлены на рис. 11.

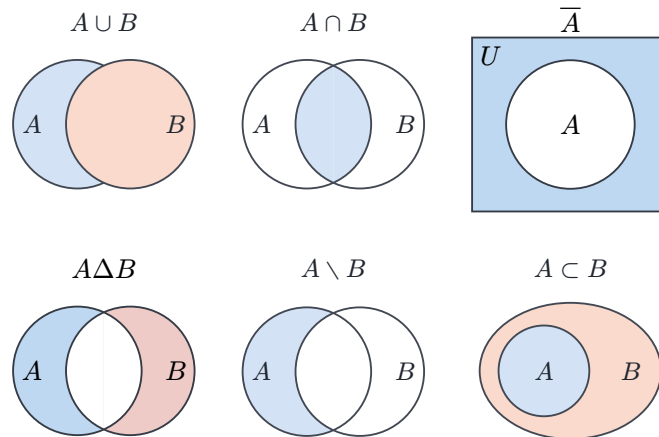


Рис. 11. Операции над множествами на диаграммах Венна.

Примечание

Диаграммы Венна — это инструмент для размышления, а не техника доказательства. Корректное доказательство равенства множеств требует алгебраических законов или поэлементного рассуждения.

ИСТОРИЯ Эйлер, Венн и визуальное мышление

Круги для изображения отношений между понятиями первым использовал Леонард Эйлер (1768). До Венна кругами Эйлера пользовались *ad hoc*: нарисовать то, что нужно в данный момент. Систематический метод разработал Джон Венн в 1880 году: его полная диаграмма явно представляет все возможные комбинации принадлежности элемента к n множествам. Любое соотношение между множествами можно выразить, заштриховав пустые регионы и пометив непустые.

Сам Венн разрабатывал диаграммы для логики — как наглядный инструмент проверки корректности силлогизмов. Чарльз Доджсон (более известный как Льюис Кэрролл) примерно в то же время изобрёл свой вариант логических диаграмм — прямоугольную сетку, которую использовал в преподавании и в книге «Символическая логика»¹⁸.

Визуальное мышление в математике — от геометрических доказательств Евклида до коммутативных диаграмм в современной алгебре — сопровождает предмет на всём его протяжении. Диаграмма Венна — один из немногих случаев, когда визуальный образ настолько точен, что полностью отражает логическую структуру утверждения. Но у визуального подхода есть предел. Для трёх множеств диаграмма ещё обозрима. Для четырёх требуются эллипсы вместо кругов (круги не могут представить все $2^4 = 16$ комбинаций). Для пяти и более диаграмма становится нечитаемой, и остаётся лишь алгебраический метод.

Диаграмма Венна с n множествами делит плоскость на 2^n регионов: для трёх множеств это восемь областей. Каждая точка плоскости получает n -битную подпись: в первом круге или нет, во втором или нет, в третьем или нет. Комбинаций ответов ровно 2^n , и все они представлены регионами диаграммы.

§ 4.7 Декартово произведение

До сих пор операции над множествами не меняли природу элементов: объединяя множества чисел, мы снова получаем числа. Декартово произведение строит из двух множеств новые объекты — *упорядоченные пары*, у которых первый компонент

¹⁸L. Carroll. «Symbolic Logic». 1896.

взяты из A , а второй — из B . Совокупность всех таких пар и называется декартовым произведением.

ОПРЕДЕЛЕНИЕ 4.16 Декартово произведение

Декартово произведение множеств A, B — множество всех упорядоченных пар (a, b) , обозначается $A \times B = \{(a, b) \mid a \in A, b \in B\}$. Две упорядоченные пары $(a, b), (c, d)$ называются **равными**, если и только если $a = c, b = d$.

Пример Декартово произведение двух множеств

Пусть $A = \{1, 2\}, B = \{a, b\}$. Тогда $A \times B = \{(1, a), (1, b), (2, a), (2, b)\}$. Порядок в паре существен: $(1, a) \neq (a, 1)$. Пара $(a, 1)$ вообще не принадлежит $A \times B$, поскольку a не принадлежит A .

До сих пор пара (a, b) была новым объектом, введённым без объяснения: мы опирались на неё, когда строили декартово произведение. Её можно собрать из множеств — и тогда пара перестаёт быть отдельной сущностью, а становится сокращением.

ОПРЕДЕЛЕНИЕ 4.17 Упорядоченная пара как множество

Упорядоченная пара элементов a и b — множество $(a, b) = \{\{a\}, \{a, b\}\}$.

Кодировку пары предложил Казимеж Куратовский в 1921 году. Первая кодировка принадлежала Винеру (1914): $\{\{\{a\}, \emptyset\}, \{\{b\}\}\}$. Смысл кодировки в том, что пара, собранная из множеств, ведёт себя как упорядоченный объект: из $\{\{a\}, \{a, b\}\} = \{\{c\}, \{c, d\}\}$ следует $a = c$ и $b = d$, то есть ровно равенство пар из определения декартова произведения.

Дальше тем же приёмом из множеств собираются и другие объекты, что кажутся самостоятельными: отношения (глава 5), функции (глава 6), порядки (глава 8), графы (глава 9), автоматы и языки (глава 23), числа (глава 22). Всякий раз новый объект оказывается множеством, оставаясь самостоятельной сущностью. Это тезис «всё есть множество»: как у Пифагора всё есть число, а в Unix всё есть файл, так и здесь всё есть множество — в том смысле, что строится из множеств. Что «строится» здесь означает «реализуется», а не «тождественно», — прояснится, когда дойдём до чисел (глава 22).

Замечание Явное определение пары

Пользоваться парами можно и без знания их внутреннего устройства: для этого достаточно равенства $(a, b) = (c, d)$ при $a = c$ и $b = d$. Явное определение нужно, чтобы пара была множеством — а значит, были множествами отношения, функции и всё, что из них строится. Тогда $A \times B$ существует как множество: каждая пара $(a, b) = \{\{a\}, \{a, b\}\}$ лежит в $\mathcal{P}(\mathcal{P}(A \cup B))$, и $A \times B$ выделяется из него аксиомой выделения.

Для конечных множеств размер декартова произведения вычисляется немедленно — это просто произведение размеров.

ТЕОРЕМА 4.5 Мощность декартова произведения

Для конечных множеств $|A \times B| = |A| \cdot |B|$.

Набросок доказательства

Докажем подсчётом.

Каждый элемент $a \in A$ образует пару с каждым элементом множества B . Для фиксированного a получаем $|B|$ пар. Общее число пар равно произведению: $|A| \cdot |B|$.

Декартово произведение не ассоциативно: $(A \times B) \times C$ и $A \times (B \times C)$ — разные множества, что видно уже в раскрытии пары. Между ними есть каноническая биекция $((a, b), c) \rightarrow (a, (b, c))$, и она позволяет отождествлять расстановки скобок: пишем $A \times B \times C$, а элементы читаем как кортежи.

Пример Скобки в декартовом произведении

Пусть $A = \{1\}, B = \{2\}, C = \{3\}$. Тогда $((1, 2), 3) \in (A \times B) \times C$. И $(1, (2, 3)) \in A \times (B \times C)$. Перестановка скобок взаимно однозначна: $((1, 2), 3) \rightarrow (1, (2, 3))$. Поэтому запись $A \times B \times C$ однозначна с точностью до этой биекции.

Примечание Вложенная пара или плоский кортеж

Отождествление — математическая договорённость, опирающаяся на каноническую биекцию. В программировании вложенная пара $((a, b), c)$ и плоский кортеж (a, b, c) — разные типы и не взаимозаменяемы без явного преобразования. Математика отождествляет их с точностью до биекции, а язык типов требует выбирать структуру.

ОПРЕДЕЛЕНИЕ 4.18 n -арное декартово произведение

n -арное декартово произведение множеств $A_1, \dots, A_n$ — множество всех n -кортежей $(a_1, \dots, a_n)$, где каждый $a_i \in A_i$:

$$A_1 \times A_2 \times \dots \times A_n = \{(a_1, \dots, a_n) \mid a_i \in A_i\}$$

ОПРЕДЕЛЕНИЕ 4.19 Декартова степень

Декартова степень множества A — произведение $A^n = A \times A \times \dots \times A$ из n одинаковых сомножителей A .

Пример Декартова степень

$\mathbb{R}^2$ — евклидова плоскость. $\{0, 1\}^n$ — все n -битные строки.

Декартово произведение — математическая основа типов-произведений (struct, record) и функций от нескольких параметров.

§ 4.8 Семейства и разбиения**ОПРЕДЕЛЕНИЕ 4.20 Индексированное семейство**

Индексированное семейство — набор множеств, проиндексированных элементами множества I , обозначается $\{A_i\}_{i \in I}$.

Объединение семейства:

$$\bigcup_{i \in I} A_i = \{x \mid \exists i \in I \quad x \in A_i\}$$

Пересечение семейства:

$$\bigcap_{i \in I} A_i = \{x \mid \forall i \in I \quad x \in A_i\}$$

Пример Индексированное семейство

Пусть $A_n = \{n, n+1, n+2\}$ для $n = 0, 1, 2$.

Тогда $\{A_n\}_{n \in \{0, 1, 2\}} = \{\{0, 1, 2\}, \{1, 2, 3\}, \{2, 3, 4\}\}$.

Объединение: $\bigcup_{n=0}^2 A_n = \{0, 1, 2, 3, 4\}$.

Пересечение: $\bigcap_{n=0}^2 A_n = \{2\}$.

Семейство — способ собрать в один объект много множеств. *Разбиение* — частный случай семейства: непересекающиеся подмножества, которые вместе покрывают всё множество.

ОПРЕДЕЛЕНИЕ 4.21 Разбиение

Разбиение множества X — набор непустых, попарно непересекающихся подмножеств, объединение которых равно X .

Разбиение — это классификация: каждый объект попадает ровно в одну категорию. Всюду один и тот же принцип — непустые, непересекающиеся классы, в сумме дающие всё множество:

- разбиение студентов группы по году рождения,

- разбиение колоды карт по мастям,
- разбиение книг на полке по языку.

Пример Пример разбиения

Пусть $X = \{1, 2, 3, 4, 5, 6\}$.

- Разбиение по чётности: $\{\{1, 3, 5\}, \{2, 4, 6\}\}$ — две части, попарно не пересекаются, объединение равно X .
- Разбиение по остатку от деления на 3: $\{\{1, 4\}, \{2, 5\}, \{3, 6\}\}$ — три части.
- Разбиение всех целых чисел на классы вычетов по модулю m :

$$\mathbb{Z} = \bigcup_{r=0}^{m-1} \{n \in \mathbb{Z} \mid n \equiv r \pmod{m}\}$$

Разбиения тесно связаны с отношениями эквивалентности — и эта связь работает в обе стороны. Каждое разбиение множества X определяет отношение эквивалентности на X : два элемента эквивалентны, если лежат в одном блоке. Обратно: каждое отношение эквивалентности на X задаёт разбиение этого множества на классы эквивалентности. Доказательство мы разберём в следующей главе.

Проверка Определения и операции над множествами

1. Когда $A \setminus B = A \Delta B$?
2. Почему разбиение требует, чтобы части были непустыми и попарно непересекающимися?

§ 4.9 Парадокс Рассела и аксиоматическая теория множеств

Георг Кантор создал теорию множеств в период 1874–1895 годов. В основе теории лежал *принцип свёртки* (comprehension): для любого свойства $P(x)$ существует множество $\{x \mid P(x)\}$ — набор всех объектов, удовлетворяющих P . Определение предельно естественно: если можно сказать «все x , которые...», значит множество существует. На этом принципе держалась вся теория.

Бытовой двойник этого противоречия — парадокс бороды. В деревне бородой бреет тех и только тех жителей, кто не бреется сам. Бреет ли бородой самого себя? Если да — он не подпадает под своё правило. Если нет — подпадает. Любой ответ противоречит условию, хотя само условие выглядит безупречно. Та же конструкция, только вместо жителей — множества, и даёт парадокс Рассела.

В 1901 году Бертран Рассел обнаружил, что принцип свёртки ведёт к противоречию.

Рассмотрим свойство $x \notin x$ — « x не содержит самого себя в качестве элемента». Большинство множеств этому свойству удовлетворяет: $\{1, 2\} \notin \{1, 2\}$ — множество чисел не является числом. Пустое множество тоже: $\emptyset \notin \emptyset$. Но есть и множества, содержащие себя: множество всех множеств (если оно существует) содержит себя,

потому что само является множеством. Если принцип свёртки верен, то существует множество $R = \{x \mid x \notin x\}$ — множество всех множеств, не содержащих себя.

Классическая логика оставляет два случая: $R \in R$ и $R \notin R$.

УТВЕРЖДЕНИЕ 4.6 Множество Рассела противоречиво

Множество $R = \{x \mid x \notin x\}$ не существует: каждое из предположений $R \in R$ и $R \notin R$ ведёт к противоречию.

Доказательство

Докажем разбором случаев.

Случай 1. Если $R \in R$, то по определению R имеем $R \notin R$ — противоречие.

Случай 2. Если $R \notin R$, то R удовлетворяет свойству $x \notin x$, следовательно, $R \in R$ — снова противоречие.

Это доказательство того, что наивное понятие «множество как набор всего, что удовлетворяет свойству» внутренне противоречиво. Свойство $x \notin x$ грамматически корректно, но определяемый им объект не может существовать — как «наибольшее натуральное число».

ИСТОРИЯ Парадокс лжеца, Рассел и Гёдель

Парадокс Рассела имеет древнюю родословную. За две с половиной тысячи лет до Рассела Эпименид Критский (VI век до н. э.) сформулировал парадокс лжеца: «Все критяне лгут». Если Эпименид, сам критянин, говорит правду — то он лжёт. Если лжёт — то говорит правду. Структура та же: самореференция порождает противоречие. Разница в том, что Эпименид оставил философскую загадку, а Рассел показал, что самореференция способна разрушить математический фундамент.

Парадокс лжеца, парадокс Рассела и теорема Гёделя о неполноте¹⁹ — три проявления одного феномена: достаточно богатая формальная система не может быть одновременно непротиворечивой и полной.

Эрнст Цермело в 1908 году предложил решение: ограничить принцип свёртки. Вместо «для любого свойства существует множество» — «для любого свойства и любого уже существующего множества можно выделить подмножество элементов, удовлетворяющих свойству».

¹⁹K. Gödel. «Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme». Monatshefte für Mathematik und Physik, 1931.

ОПРЕДЕЛЕНИЕ 4.22 Аксиома выделения

Для любого множества A и любого свойства (предиката) $P(x)$ существует множество $\{x \in A \mid P(x)\}$. Это подмножество элементов A , удовлетворяющих P .

Свойство $x \notin x$ больше не определяет множество R , потому что нет объемлющего множества, из которого можно было бы выделить R . Аксиома выделения блокирует исходное определение: нельзя сказать «множество всех x , таких что...» без указания объемлющего множества.

На аксиоме выделения аксиоматика не закончилась: достройка до полной системы ZFC — предмет звёздного раздела ниже, в этой же главе.

Парадокс Рассела не угрожает повседневной математике. Но его историческая роль существенна: он заставил математику перейти от наивной интуиции к формальной аксиоматике — и в этом переходе родилась современная математическая логика.

Аксиома выбора (AC) — отдельная тема, тесно связанная с трансфинитной теорией. Мы рассмотрим её в главе 22 наряду с континуум-гипотезой и парадоксом Банаха–Тарского.



Аксиома выделения блокирует не только парадокс Рассела. Она блокирует и множество всех множеств: если бы оно существовало, к нему можно было бы применить аксиому выделения с условием $x \notin x$ — и парадокс вернулся. Поэтому абсолютного дополнения не существует. Каждое дополнение в этой главе — относительно явно заданного универсума. Алгебра множеств всегда относительна.

Проверка Парадокс Рассела и аксиома выделения

1. Чем аксиома выделения отличается от принципа свёртки?
2. Почему из отказа от принципа свёртки следует, что множество всех множеств не существует?

§ 4.10 Реляционная алгебра и SQL

Таблица базы данных — это подмножество декартова произведения.

ОПРЕДЕЛЕНИЕ 4.23 Операции реляционной алгебры

- **Выборка:** $\sigma_{\text{условие}}(R) = \{t \in R \mid \text{условие}(t)\}$.
- **Проекция** $\pi_{\text{столбцы}}(R)$: сохранить указанные столбцы.
- **Объединение, пересечение, разность:** теоретико-множественные операции над кортежами.
- **Декартово произведение:** сцепить каждую строку R с каждой строкой S .

Пример Запрос SELECT как множество

`SELECT DISTINCT Name FROM Employees WHERE Age > 30` = $\{e.Name \mid e \in \text{Employees}, e.Age > 30\}$.

Пример JOIN как декартово произведение с условием

Таблица `Students(id, name)` и таблица `Courses(sid, course)`.

Запрос «имена студентов, записанных на дискретную математику»:

```
SELECT DISTINCT s.name
FROM Students s JOIN Courses c ON s.id = c.sid
WHERE c.course = 'Дискретная математика'
```

В терминах множеств: результат — это

$$\pi_{\text{name}}(\sigma_{\text{course} = \text{Дискр. мат.}}((\text{Students}) \times_{\{\text{id} = \text{sid}\}} (\text{Courses}))),$$

где JOIN с условием P определяется как $R \bowtie_P S = \sigma_P(R \times S)$.

JOIN строит декартово произведение (каждая строка `Students` с каждой строкой `Courses`), затем `ON` отбирает пары с совпадающими `id`, `WHERE` фильтрует по названию курса, а `SELECT DISTINCT name` берёт проекцию на столбец `name`.

Каждый шаг — операция над множествами кортежей.

SQL — это порождающая запись множеств с синтаксическим сахаром. Каждый JOIN — это декартово произведение с последующей выборкой (фильтрацией по условию). Каждый UNION — это объединение множеств кортежей. `SELECT DISTINCT` возвращает множество (без дубликатов). `SELECT` без `DISTINCT` возвращает коллекцию, в которой строки могут повторяться — в отличие от математического множества, но ближе к реальности баз данных.

Трёхзначная логика с `NULL` добавляет третье истинностное значение — «неизвестно» — расширяя классическую двузначную логику. Для большинства запросов, однако, теоретико-множественная модель даёт верную интуицию: выборка, проекция, объединение, разность — всё это операции над множествами кортежей.²⁰

§ 4.11 Типы данных как множества

Та же идея — отождествлять объекты с множествами — работает и в программировании: тип данных задаёт множество допустимых значений. Для базовых типов соответствие прямое.

²⁰E. F. Codd. «A Relational Model of Data for Large Shared Data Banks». Communications of the ACM, 1970.

ОПРЕДЕЛЕНИЕ 4.24 Типы как множества

- $\text{bool} = \{T, F\}$.
- $\text{uint8} = \{0, \dots, 255\}$.
- $\text{string} \subseteq \Sigma^*$ (конечные последовательности над алфавитом Σ).
- $A \rightarrow B$ — множество всех функций из A в B , в теории множеств обозначается B^A .

Ср. с булеаном 2^A — множество функций из A в булевы значения.

Из этих элементов новые типы собираются двумя главными конструкторами. Первый — сумма: значение нового типа — это один из нескольких заранее заданных вариантов.

ОПРЕДЕЛЕНИЕ 4.25 Тип-сумма

Тип-сумма типов A и B — дизъюнктное объединение их множеств значений: каждое значение принадлежит ровно одной ветви и несёт её тег.

Второй конструктор — произведение: значение составного типа — кортеж из значений компонентов.

ОПРЕДЕЛЕНИЕ 4.26 Тип-произведение

Тип-произведение типов A и B — декартово произведение их множеств значений: значение составного типа — кортеж из значений компонентов.

Пример Типы данных как множества

$\text{Option}\langle T \rangle = \{\text{None}\} \cup \{\text{Some}(v) \mid v \in T\}$ — дизъюнктное объединение.

$\text{struct Point } \{ x: \text{f64}, y: \text{f64} \}$ — это $\mathbb{R} \times \mathbb{R}$.

Замечание

Груда кирпичей и архитектурный чертёж описывают одно и то же — но по-разному. Груда говорит: вот что здесь есть. Чертёж говорит: вот как это устроено и что с этим можно делать.

Множество — это груда: экстенциональность фиксирует, что два множества равны, если и только если состоят из одних и тех же элементов, и никакая дополнительная структура в сравнении не участвует. Тип данных — это чертёж: два типа с одинаковым набором значений, но разными именами полей или разными допустимыми операциями — разные типы, хотя как множества они неразличимы. Отождествление «тип = множество значений» продуктивно как первое приближение, но оно стирает ровно ту структурную информацию, которая делает типы полезными. Именно поэтому языки программирования

разделяют множества (set data structure) и типы (type system): одни хранят элементы, другие накладывают структурные ограничения, невыразимые на языке одной лишь принадлежности.

§ 4.12 Битовые маски

Множество с элементами из небольшого универсума кодируется одним машинным словом: бит i отвечает за элемент с номером i . На таких масках теоретико-множественные операции превращаются в побитовые операции процессора.

УТВЕРЖДЕНИЕ 4.7 Соответствие битовых масок

Для $X, Y \subseteq \{1, \dots, n\}$, закодированных как битовые маски:

- $b_{X \cup Y} = b_X \mid b_Y$
- $b_{X \cap Y} = b_X \& b_Y$
- $b_{X \Delta Y} = b_X \oplus b_Y$
- $b_{\bar{X}} = \sim b_X$

Набросок доказательства

Докажем побитово: сравним, когда бит i равен 1 в левой и правой частях каждого равенства.

Бит i маски b_X равен 1 тогда и только тогда, когда $i \in X$. $b_{X \cup Y}$ содержит 1 в позиции i , если $i \in X \vee i \in Y$. Это совпадает с побитовым OR: $(b_X \mid b_Y)[i] = 1 \leftrightarrow b_{X[i]} = 1 \vee b_{Y[i]} = 1$. Аналогично для пересечения (AND), симметрической разности (XOR) и дополнения (NOT).

Битовые маски — прямой вычислительный аналог характеристической функции:

- права доступа Unix (rwx)
- флаги функций (O_RDONLY | O_CREAT)
- перебор подмножеств циклом по маске

Проверка Множества в инженерии

1. Какую цепочку операций реляционной алгебры выполняет JOIN с условием?
2. Почему дополнение битовой маски определено только относительно фиксированного универсума?
3. Как через побитовые операции проверить включение $X \subseteq Y$?

§ 4.13 Хеширование

Хеш-функция $h : K \rightarrow \{0, \dots, m - 1\}$ отображает большое пространство ключей в фиксированное число ячеек. По принципу Дирихле, как только ключей становится больше m , двум ключам придётся попасть в одну ячейку — коллизии неизбежны.

ОПРЕДЕЛЕНИЕ 4.27 Коллизия

Коллизия хеш-функции — ситуация, в которой $h(k_1) = h(k_2)$, $k_1 \neq k_2$.

Пример Коллизия хеш-функции

Хеш-функция $h(k) = k \bmod 5$ применяется к ключам $\{2, 7, 12, 17\}$: все четыре дают $h(k) = 2$ и коллидируют в одной ячейке. Здесь виновата неудачная функция: те же четыре ключа в пяти ячейках могли бы разместиться и без столкновений — принцип Дирихле запрещает лишь бесколлизийное хеширование, когда ключей больше, чем ячеек.

Хорошая хеш-функция распределяет ключи по ячейкам равномерно — в языке теории множеств она ведёт себя как случайное отображение, и в каждой ячейке ожидаемое число ключей равно n/m , где n — число сохранённых ключей. Это отношение называют *коэффициентом загрузки*.

С коллизиями таблица живёт одним из двух способов. При *цепочках* каждая ячейка хранит список пар, попавших в неё, и ключ ищется по своему короткому списку. При *открытой адресации* столкнувшийся ключ укладывается в следующую свободную ячейку по заранее заданному правилу обхода. Пока коэффициент загрузки мал, и цепочка, и обход находят ключ за несколько шагов — константу, не зависящую от n .

Когда таблица переполняется, её перестраивают: число ячеек удваивается, и все ключи раскладываются заново. Перестройка дорога, но она случается редко — в серии из n вставок суммарная работа остаётся пропорциональной n . Отсюда обещание словарей: операция за *амортизированную* константу — отдельная вставка может заплатить за перестройку, средняя не платит.

Примечание

Словарь (хеш-таблица) — это частичная функция $f \subseteq K \times V$, определённая не для всех ключей (подробно — глава 6). Хеширование — эффективный способ реализовать эту теоретико-множественную структуру.

§ 4.14 * Аксиоматическая теория множеств

Мы видели, как парадокс Рассела разрушает наивную теорию множеств. Цермело предложил лекарство — аксиому выделения — и на этом можно было бы остановиться: парадокс заблокирован, повседневная математика спасена. Но математику мало заблокировать противоречие. Ему нужно знать, на чём она стоит.

По аналогии: мы строим мост. Мы знаем, что одна из балок треснула бы при нагрузке — мы её заменили. Но чтобы мост стоял, нужно проверить *все* несущие конструкции: фундамент, пролёты, подвесы. Аксиома выделения — это замена треснувшей балки. Остальные аксиомы ZFC — это полный чертёж моста.

Система Цермело–Френкеля с аксиомой выбора (ZFC) формулируется на языке первого порядка с единственным нелогическим символом — отношением принадлежности $\in$. Всё, что существует в универсуме ZFC, — это множества. Числа, функции, графы, векторные пространства строятся как множества специального вида. Никаких других объектов нет.

Две аксиомы глава уже ввела в основном тексте. Аксиома объёмности — это экстенциональность: множества равны тогда и только тогда, когда состоят из одних и тех же элементов. Аксиома выделения определена после парадокса Рассела: из существующего множества выделяется подмножество по любому выразимому свойству. Первая фиксирует, что такое «быть множеством». Вторая блокирует парадокс.

Остальные аксиомы отвечают на вопрос: откуда вообще берутся множества? Если можно только выделять подмножества из уже существующих, нужно с чего-то начать. И нужны операции, строящие новые множества.

Вся система ZF сводится в одну таблицу.

Аксиома	Что постулирует
Объёмность	множества с одними и теми же элементами равны
Выделение	из существующего множества выделимо подмножество по любому выразимому свойству
Пара	для любых элементов a и b существует множество $\{a, b\}$
Объединение	для любого множества A существует $\bigcup A$ — множество всех элементов элементов A
Степень	для любого множества A существует булеан $\mathcal{P}(A)$
Бесконечность	существует множество, содержащее $\emptyset$ и замкнутое относительно $x \rightarrow x \cup \{x\}$
Подстановка	образ множества при функциональном свойстве F — снова множество
Регулярность	всякое непустое множество A содержит элемент x с $x \cap A = \emptyset$

Из аксиомы пары и выделения получается синглетон $\{a\} = \{a, a\}$, а из пар собираются упорядоченные пары и декартово произведение. Аксиома объединения разрешает «сплющивать» множество на один уровень: обычное объединение вы-

ражается как $A \cup B = \bigcup \{A, B\}$, где $\{A, B\}$ дано аксиомой пары. Булеан, которым глава пользуется с самого начала, для бесконечных множеств требует отдельной аксиомы существования.

Пока что аксиомы производят конечное из конечного: пара из элементов даёт конечное множество, объединение конечного семейства конечных множеств конечно, булеан конечного множества конечен. Чтобы выйти за пределы конечного, нужна аксиома, постулирующая существование хотя бы одного бесконечного множества. Операция $x \cup \{x\}$ моделирует функцию следования. Если обозначить $0 = \emptyset$, $1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$, то аксиома бесконечности утверждает: существует множество, содержащее все такие объекты. Это натуральный ряд, построенный из пустоты — точнее, из пустого множества и фигурных скобок. Определение натуральных чисел принадлежит Джону фон Нейману (1923): каждое натуральное число — это множество всех предыдущих, $n = \{0, 1, \dots, n-1\}$. Такой подход унифицирует числа и множества в одной онтологии.

Аксиома подстановки — самая техническая и самая поздняя. Это *схема*, а не одно утверждение: по одному утверждению на каждое функциональное свойство F (для каждого x не более одного y), выразимое в языке теории множеств. Добавлена Френкелем и Сколемом в 1922 году. Без неё нельзя доказать существование множеств мощности $\aleph_\omega$ и выше. Она превращает ЗС (систему Цермело) в ZF (систему Цермело–Френкеля).

Аксиома регулярности исключает аномалии: множества, содержащие самих себя ($x \in x$), и бесконечно убывающие цепочки принадлежности ($x_2 \in x_1, x_3 \in x_2, \dots$). Универсум множеств строится «снизу вверх» — от пустого множества к более сложным конструкциям. Каждый шаг опирается на уже построенное, и в универсуме нет ни петель, ни спусков в бесконечность.

Восемь аксиом — объёмность, выделение, пара, объединение, степень, бесконечность, подстановка, регулярность — образуют систему ZF. Этого достаточно для построения почти всей математики. «Почти» — потому что остаётся ещё одна аксиома, стоящая особняком.

ОПРЕДЕЛЕНИЕ 4.28 Аксиома выбора

Для любого семейства непустых множеств $\{A_i\}_{i \in I}$ существует функция выбора:

$$f : I \rightarrow \bigcup_{i \in I} A_i, \quad f(i) \in A_i \quad \forall i \in I.$$

Функция f «выбирает» по одному элементу из каждого множества семейства. Для конечного I никакой аксиомы не нужно: выбор можно сделать явно, перебрав все множества. Для бесконечного I перебора недостаточно — и AC постулирует существование функции выбора, не предъявляя её.

Замечание Почему аксиома выбора спорна?

Аксиома выбора — самая спорная аксиома системы: она утверждает существование функции выбора, не предъявляя её. Аксиома бесконечности тоже постулирует объект без построения, но её объект — натуральный ряд — нужен всей математике, и споры вокруг неё быстро угасли. Функция выбора на произвольном бесконечном семействе оказалась предметом более деликатным: её следствия, вроде полного упорядочения континуума, не сопровождаются никаким явным описанием.

В 1904 году Цермело использовал АС для доказательства теоремы о полном упорядочении. Доказательство вызвало ожесточённые споры — Эмиль Борель, Анри Лебег и другие французские математики отвергали неконструктивное рассуждение. В 1938 году Гёдель доказал: если ZF непротиворечива, то ZFC тоже непротиворечива. В 1963 году Пол Коэн доказал: если ZF непротиворечива, то $ZF + \text{not AC}$ тоже непротиворечива. АС не зависит от остальных аксиом ZF. Её нельзя ни доказать, ни опровергнуть — можно только принять или отвергнуть.

Большинство математиков принимают АС. Причина не философская — прагматическая. Без АС рухнет слишком много полезных утверждений. В ZF следующие три утверждения эквивалентны АС:

1. *Лемма Цорна*: если в частично упорядоченном множестве всякая цепь имеет верхнюю грань, то существует максимальный элемент.
2. *Теорема о полном упорядочении* (Цермело, 1904): любое множество может быть вполне упорядочено.
3. *Теорема о базисе*: всякое векторное пространство имеет базис.

Лемма Цорна — стандартный инструмент в алгебре и функциональном анализе. Без неё нельзя доказать, что всякое поле имеет алгебраическое замыкание, а всякое векторное пространство — базис. Отказ от АС означает отказ от этих результатов.

Система $ZF + AC$ обозначается ZFC. Это — стандартный фундамент современной математики. Когда математик говорит «докажем теорему», он предполагает ZFC (или систему эквивалентной силы).

Мы работаем в ZFC. Аксиома выбора понадобится нам в главе о конструкциях чисел (глава 22). Но для повседневной работы с конечными структурами — графами, автоматами, булевыми формулами — все аксиомы остаются «под капотом». Мы строим множества явно, из конечного числа элементов, и никакие парадоксы нам не угрожают.

§ 4.15 * Кардинальные числа

Для конечного множества мощность — это количество элементов: $|\{a, b, c\}| = 3$, $|\emptyset| = 0$. Размеры сравниваются пересчётом, и результат — натуральное число. Сам

пересчёт — это построение биекции: нумеруя элементы, мы взаимно однозначно сопоставляем множеству отрезок натурального ряда.

Слово «счётное» обещает больше, чем может дать: бесконечное множество нельзя сосчитать до конца. Его точный смысл — «перечислимое»: элементы расставляются в бесконечный список и нумеруются натуральными числами. Счётны чётные числа, целые, рациональные — для каждого из этих множеств список существует, хотя интуиция упирается в то, что «чисел всё больше».

Для бесконечного множества пересчёт не работает, но идея сравнения через биекцию остаётся: два множества называются *равномощными*, если между ними существует биекция. Тогда «размер» перестаёт быть числом и становится классом равномощных множеств — такой класс называется *кардинальным числом* (или *кардиналом*). Наименьшая бесконечная мощность — мощность натурального ряда, она обозначается $\aleph_0 = |\mathbb{N}|$.

Булеан тянет мощности вверх. Подмножество отождествляется со своей характеристической функцией $A \rightarrow \{T, F\}$, и это сопоставление переносится с конечных множеств на бесконечные, поэтому $|\mathcal{P}(A)| = 2^{|A|}$ при любом A . Теорема Кантора утверждает большее: мощность булеана всегда строго больше мощности самого множества. Идею доказательства подсказывает уже встреченный парадокс Рассела: множество элементов, не принадлежащих своему образу, не совпадает ни с одним образом, и потому биекции не существует.

Бесконечные «количества» выстраиваются в иерархию

$$|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| < |\mathcal{P}(\mathcal{P}(\mathbb{N}))| < \dots$$

Как формализуется сравнение мощностей, почему $|\mathbb{R}|$ уходит от $|\mathbb{N}|$ и есть ли ступень между ними (континуум-гипотеза) — предмет главы о мощности (7).

§ 4.16 Итоги главы

Множество отвечает ровно на один вопрос: что в него входит, а что нет. Порядок и повторения для него не существуют: $\{1, 2, 3\} = \{3, 1, 2\} = \{1, 1, 2\}$. Принцип экстенциональности ставит на этом точку — множество определяется содержимым, а не способом описания, поэтому $\{1, 2, 3\} = \{x \in \mathbb{N} \mid 0 < x < 4\}$. Три отношения — принадлежность, включение, равенство — сравнивают объекты и контейнеры, а пять операций — объединение, пересечение, разность, симметрическая разность и дополнение — порождают из множеств новые.

Наивная точка зрения — набор всего, что удовлетворяет свойству, — рассыпалась на парадоксе Рассела: свойство быть множеством, которое не содержит себя, никакого множества не задаёт. Аксиома выделения ограничивает построение уже существующими множествами, и в повседневной работе мы действуем в фиксированном универсуме, собирая множества явными операциями.

Алгебра множеств оказалась изоморфна логике высказываний: объединение повторяет дизъюнкцию, пересечение — конъюнкцию, дополнение — отрицание. Поэтому законы из главы 1 переносятся на множества целиком — имена меняются, структура остаётся. Тот же словарь переводится на язык машин: множество кодируется битовой маской, а операции становятся побитовыми.

Две конструкции — декартово произведение и булеан — строят из множеств новые объекты, на которых держатся дальнейшие главы. Булеан дал размер: $|\mathcal{P}(A)| = 2^{|A|}$ для множества из $|A|$ элементов, а бесконечные мощности выстраиваются в иерархию $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| < \dots$. Прямая польза — в инженерии: таблица базы данных — подмножество декартова произведения, SQL-запрос — цепочка теоретико-множественных операций, тип — множество, флаги функций — битовая маска. Декартово произведение подводит вплотную к связям между элементами — бинарным отношениям.

Проверка Проверьте себя

1. Чем принадлежность отличается от включения?
2. Почему два разных описания могут задавать одно и то же множество, и как это разрешает принцип экстенциональности?
3. Почему $|\mathcal{P}(A)| = 2^n$ для множества из n элементов?
4. Чем пустое множество отличается от множества $\{\emptyset\}$?
5. Почему аксиома выделения блокирует парадокс Рассела?
6. Почему алгебра множеств и логика высказываний подчиняются одним и тем же законам?

Что дальше?

От объектов мы переходим к связям между ними. В главе 5 мы изучим бинарные отношения — формализацию связей между элементами. Разберём их алгебраические свойства и два основных частных случая: отношения эквивалентности и частичные порядки.

§ 4.17 Упражнения

Принадлежность, включение и равенство.

1. Даны $A = \{1, 2, 3\}$, $B = \{2, 3, 4\}$. Найдите $A \cup B$, $A \cap B$, $A \setminus B$, $B \setminus A$, $A \Delta B$.
2. Верно ли, что $\emptyset \in \{\emptyset\}$? Верно ли, что $\emptyset \subseteq \{\emptyset\}$? Объясните на этом примере разницу между принадлежностью ($\in$) и включением ($\subseteq$).
3. Докажите, что $A = B \leftrightarrow A \subseteq B \wedge B \subseteq A$. Примените этот критерий, чтобы доказать $A \cup B = B \cup A$.

Операции над множествами.

4. Проверьте законы де Моргана на примере: $A = \{1, 3, 5\}, B = \{2, 3, 4\}, U = \{1, 2, 3, 4, 5, 6\}$.
5. Упростите выражение $(A \cap B) \cup (A \cap \overline{B})$, используя законы алгебры множеств. Какому множеству оно эквивалентно?
6. * Докажите методом взаимного включения: $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$. Проверьте равенство на конкретном примере.

Булеан и декартово произведение.

7. Перечислите все элементы $\mathcal{P}(\{1, 2, 3\})$. Сколько их? Проверьте формулу $|\mathcal{P}(A)| = 2^{|A|}$.
8. Докажите или опровергните: $\mathcal{P}(A \cup B) = \mathcal{P}(A) \cup \mathcal{P}(B)$. Приведите контрпример в случае неверности.
9. * Покажите, что $A \times (B \cup C) = (A \times B) \cup (A \times C)$. Верно ли аналогичное равенство для пересечения?

Парадокс Рассела и аксиома выделения.

10. Воспроизведите парадокс Рассела: покажите, что из предположения о существовании «множества всех множеств, не содержащих себя», вытекает противоречие.
11. Чем аксиома выделения отличается от принципа свёртки? Почему она блокирует парадокс Рассела?

Приложения: битовые маски, SQL и типы.

12. Множества $X = \{1, 3, 5\}, Y = \{2, 3, 4\}$ закодированы битовыми масками. Универсум: $U = \{1, 2, 3, 4, 5, 6\}$. Вычислите $b_{X \cup Y}, b_{X \cap Y}$ через побитовые операции и проверьте результат.
13. * Тип $\text{Option}\langle T \rangle = \{\text{None}\} \cup \{\text{Some}(v) \mid v \in T\}$. Какой теоретико-множественной конструкции это соответствует? Как записать через неё тип $\text{Result}\langle T, E \rangle$?

ПРОЕКТ Исчисление множеств: проверка тождеств и контрпример

Булева алгебра подмножеств — «те же связки, другие буквы»: $\cup$ соответствует `or`, $\cap$ — `and`, дополнение — отрицанию. Тождества алгебры множеств (дистрибутивность, законы де Моргана, дополнение) доказываются механически, если перебрать все подмножества конечной вселенной. Соберите инструмент, который проверяет любое тождество и находит элемент, где оно ломается.

① Парсер выражений

Научите программу читать выражения множеств: $\emptyset$, универсум, переменные — заглавные буквы, $\cup, \cap, \setminus$, дополнение (постфикс), скобки. Введите приоритеты: $\setminus$ и $\cap$ связывают сильнее $\cup$. Дополнение — постфикс на переменной или группе. Постройте дерево разбора рекурсивным спуском. Проверьте на выражениях вроде $(A \cup B)^c$ и $A \setminus B \cap C$.

② Битовые подмножества

Представьте подмножество конечной вселенной битовой маской: $\cup$ — побитовое or, $\cap$ — and, $\setminus$ — and not, дополнение — побитовое xor с маской всего универсума. Перебор элементов — по установленным битам. Убедитесь, что $|\mathcal{P}(A)| = 2^{|A|}$: подмножеств ровно $1 \ll |A|$.

③ Проверка тождеств

Переберите все присваивания переменных подмножествами вселенной из n элементов при v переменных (ровно $2^{n \cdot v}$ присваиваний), вычислите обе стороны и сравните. Если расходятся — предъявите конкретный элемент, на котором выражения различаются. Проверьте: дистрибутивность, законы де Моргана, $A \cup A^c = U$, $A \setminus B = A \cap B^c$. Убедитесь, что ложное тождество вроде $A \setminus B = A$ даёт контрпример, а верное — нет.

④ Диаграммы Венна

Как расширение: выведите диаграмму Венна в SVG для двух-трёх множеств и подсветите на ней область, соответствующую заданному выражению.

⑤ Генератор и тренажёр

Сгенерируйте случайные выражения над случайными подмножествами и автоматически проверьте, верно ли тождество. Соберите тренажёр: программа предъявляет выражение, вы отвечаете, верно ли оно, и при ошибке получаете контрпример-элемент.

Итог: инструмент, который по текстовому выражению строит дерево, по любой проверке тождества над конечной вселенной выносит вердикт (верно всегда или ложно), и для ложного выводит конкретный контрпример-элемент.



Отношения

Телефонная книга — список пар: имя и номер. Карта метро — список пар: две станции, соединённые линией. Родословная, расписание поездов, граф зависимостей сборки — всё выписывается парами. Пара — самый дешёвый способ зафиксировать связь.

Глава 4 научилась описывать объекты: множества, элементы, операции. Но объекты редко живут поодиночке: студент записан на курс, город соединён дорогой с городом, число делится на число. Множество отвечает на вопрос «что есть», а нужен ещё инструмент для вопроса «как связано».

Инструментом оказывается *бинарное отношение* — просто множество упорядоченных пар. Декартово произведение собирает все возможные пары, отношение выбирает из них нужные: связь есть, если пара принадлежит отношению. В этой конструкции нет ничего, кроме «принадлежит или нет», — а из неё строится разветвлённая теория.

Свойства классифицируют связи: рефлексивность, симметричность, транзитивность — каждое из них читается и в матрице, и в графе. Замыкания достраивают отношение до нужного свойства, добавляя ровно те пары, которых не хватает. Композиция превращает связи в цепочки: «родитель родителя» — это дед, а транзитивное замыкание разворачивает «родителя» в «предка». Две комбинации свойств дают две большие структуры: эквивалентность разбивает множество на классы неразличимых, порядок выстраивает иерархию, в которой не всё сравнимо.

Язык работает далеко за пределами математики. Коллизии хеш-функций — это ключи, попавшие в один класс эквивалентности. Порядок сборки проекта — частичный порядок зависимостей. Минимизация автомата — склеивание неразличимых состояний. Реляционная база данных — таблицы-отношения, а SQL — язык запросов над ними. Математика одна, меняются только множества.

Что общего у телефонной книги, таблицы базы данных и схемы метро — и как из простой пары вырастают классы эквивалентности и иерархии?

ИСТОРИЯ От сравнений по модулю до реляционных баз данных

Карл Фридрих Гаусс в «Disquisitiones Arithmeticae»²¹ ввёл сравнения по модулю: $a \equiv b \pmod{m} \leftrightarrow m \mid (a - b)$. Это были первые явно сформулированные отно-

²¹C. F. Gauß. «Disquisitiones Arithmeticae». 1801.

шения эквивалентности — разбиение целых чисел на классы вычетов —, хотя сам термин появился позже. Гаусс показал, что арифметические операции корректно переносятся на классы: результат не зависит от выбора представителей.

В XIX веке понятие отношения развивалось прежде всего в логике: де Морган, Пирс и Шрёдер строили алгебру отношений, а в алгебре Дедекинд изучал соответствия между структурами. Общая теория отношений как подмножеств декартова произведения оформилась лишь после создания теории множеств Кантором. Решающий шаг — формализация бинарного отношения как множества упорядоченных пар — позволил перенести алгебраические понятия (рефлексивность, симметричность, транзитивность) на произвольные множества.

В 1970 году Эдгар Кодд опубликовал «A Relational Model of Data for Large Shared Data Banks»²², где предложил реляционную модель баз данных. Кодд пришёл в IBM в 1949 году программистом-математиком. Степень PhD по компьютерным и коммуникационным наукам он получил в Мичиганском университете. Таблица в его модели — буквально отношение: подмножество декартова произведения доменов. Операция выборки SELECT с условием P возвращает $\{t \in R \mid P(t)\}$ — в точности порождающую запись множества. Реляционная алгебра — выборка, проекция, соединение — это операции над математическими отношениями.

IBM поначалу сопротивлялась: компания продавала иерархическую СУБД IMS, которой реляционная модель угрожала. Реляционная СУБД System R — первая реализация SQL — разрабатывалась внутри IBM как исследовательский проект, а коммерческие системы — Oracle в конце 1970-х и DB2 в начале 1980-х — сделали SQL промышленным стандартом. Сегодня миллиарды запросов ежедневно вычисляют выражения реляционной алгебры над отношениями.²³

ОБЗОР ГЛАВЫ

В этой главе мы выберем способ записи отношения: список упорядоченных пар, матрица смежности или ориентированный граф. Потом разберём свойства — рефлексивность, симметричность, транзитивность — и научимся видеть их в каждом из трёх представлений. Замыкания добавляют недостающие пары, а композиция склеит связи в цепочки, из которых вырастет транзитивное замыкание — достижимость. Отношения эквивалентности разобьют множество на классы, и теорема о разбиении покажет: классы и разбиения — одна структура, увиденная с двух сторон. Вложенные разбиения — иерархии — зададут расстояние, ультраметрику, из которой вырастет дерево — дендрограмма. Частичный порядок получит первое определение здесь, а подробное изучение продолжится в главе 8. В конце мы при-

²²E. F. Codd. «A Relational Model of Data for Large Shared Data Banks». Communications of the ACM, 1970.

²³C. J. Date. «The Database Relational Model: A Retrospective Review and Analysis». 2000.

меним принцип Дирихле — простой инструмент с неожиданными следствиями, полная сила которого раскроется в главе 18.

Отношения составляют язык связей: на них строятся функции (глава 6), порядки (глава 8) и графы (глава 9), а реляционные базы данных и минимизация конечных автоматов (глава 23) используют отношение неразличимости.

После этой главы вы сможете:

1. записывать отношение тремя способами — парами, матрицей, графом — и переходить между ними;
2. проверять рефлексивность, симметричность и транзитивность в любом представлении;
3. строить замыкания отношений, в том числе транзитивное, и объяснять его связь с достижимостью;
4. доказывать, что отношение является эквивалентностью, и строить классы эквивалентности;
5. понимать соответствие между отношениями эквивалентности и разбиениями и видеть его в базах данных, хешировании и минимизации автоматов.

§ 5.1 Бинарные отношения

5.1.1 Определения и способы представления

Связь между объектами записывается парой. Бинарное отношение — это и есть такой список пар, и ничего больше. Если пара $(a, b) \in R$, мы говорим, что a находится в отношении R с b , и пишем aRb .

ОПРЕДЕЛЕНИЕ 5.1 Бинарное отношение

Подмножество $R \subseteq A \times A$ называется **бинарным отношением** на множестве A . Факт $(a, b) \in R$ обозначается aRb .

Отношение может быть задано тремя эквивалентными способами, каждый из которых удобен в своей ситуации.

ОПРЕДЕЛЕНИЕ 5.2 Множество пар

Отношение задано **множеством пар**, если оно перечислено явным списком связанных пар:

$$R = \{(a_1, b_1), (a_2, b_2), \dots\}.$$

ОПРЕДЕЛЕНИЕ 5.3 Матрица смежности

Матрицей смежности отношения R на элементах $a_1, \dots, a_n$ называется матрица M размера $n \times n$, в которой

$$M_{ij} = \begin{cases} 1 & \text{при } (a_i, a_j) \in R \\ 0 & \text{иначе} \end{cases}$$

ОПРЕДЕЛЕНИЕ 5.4 Ориентированный граф

Ориентированным графом (англ. directed graph) отношения R называется структура, в которой вершины — элементы A , а ориентированное ребро $a \rightarrow b$ проводится, когда $(a, b) \in R$.

Пример Отношение на пяти элементах

$A = \{1, 2, 3, 4, 5\}$. $R = \{(1, 1), (1, 2), (1, 5), (2, 3), (2, 4), (3, 1), (4, 2), (5, 3), (5, 5)\}$.

Отношение содержит две петли (в вершинах 1 и 5), цикл $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ и несколько дополнительных рёбер.

Матрица смежности (строки и столбцы соответствуют элементам 1, 2, 3, 4, 5 в порядке возрастания):

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

В позиции (i, j) стоит 1, если $(i, j) \in R$, и 0 иначе. Матрица смежности и граф несут одну и ту же информацию. Выбор представления диктуется удобством.

Читать матрицу удобно по строкам и столбцам: строка перечисляет, с кем связан элемент, столбец — кто связан с ним. Единичная матрица — матрица отношения равенства: единицы стоят ровно на диагонали, потому что различные элементы в равенство не входят.

Орграф на рис. 12 показывает третью форму представления — ориентированный граф с петлями и рёбрами, соответствующими парам отношения.

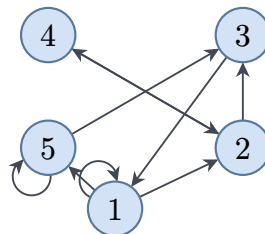


Рис. 12. Орграф отношения.

5.1.2 Свойства отношений

Отношения классифицируют по их алгебраическим свойствам. Эти свойства — «кирпичики», из которых строятся более сложные структуры: эквивалентности и порядки.

Пусть $R \subseteq A \times A$ — бинарное отношение.

ОПРЕДЕЛЕНИЕ 5.5 Рефлексивность

R называется **рефлексивным**, если

$$\forall a \in A, aRa.$$

Связь элемента с самим собой — первое, что различает отношения. Требование рефлексивности можно обратить: запретить такие связи вовсе.

ОПРЕДЕЛЕНИЕ 5.6 Иррефлексивность

R называется **иррефлексивным**, если

$$\forall a \in A, \neg(aRa).$$

Эти два свойства описывают поведение отношения на диагонали: рефлексивность требует петли у каждого элемента, иррефлексивность запрещает петли вовсе. Следующие три — направление связей между разными элементами.

ОПРЕДЕЛЕНИЕ 5.7 Симметричность

R называется **симметричным**, если

$$aRb \rightarrow bRa.$$

Симметричность требует, чтобы каждая связь имела обратную. Вопрос о том, когда взаимность допустима, приводит к двум следующим свойствам.

ОПРЕДЕЛЕНИЕ 5.8 Антисимметричность

R называется **антисимметричным**, если

$$(aRb \wedge bRa) \rightarrow a = b.$$

ОПРЕДЕЛЕНИЕ 5.9 Асимметричность

R называется **асимметричным**, если

$$aRb \rightarrow \neg(bRa).$$

Разница между антисимметричностью и асимметричностью — в допущении равенства: антисимметричность разрешает aRa , асимметричность — запрещает. Знаки $<$

$\leq$ различаются ровно этим: из $a < b$ и $b < a$ следует противоречие, а из $a \leq b$ и $b \leq a$ — лишь $a = b$.

ОПРЕДЕЛЕНИЕ 5.10 Транзитивность

R называется **транзитивным**, если

$$(aRb \wedge bRc) \rightarrow aRc.$$

Транзитивность отвечает за передачу связи по цепочке. Последнее свойство — тотальность: вопрос о том, обязана ли любая пара элементов быть сравнимой.

ОПРЕДЕЛЕНИЕ 5.11 Тотальность

R называется **тотальным**, если

$$a \neq b \rightarrow (aRb \vee bRa).$$

Пример Проверка свойств на конкретном отношении

Пусть $A = \{1, 2, 3\}$. Возьмём $R = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)\}$.

- Рефлексивно: $(1, 1), (2, 2), (3, 3) \in R$ — каждый элемент связан с собой.
- Симметрично: $(1, 2), (2, 1) \in R$, остальные пары диагональны.
- Транзитивно: каждая цепочка из двух пар замыкается парой из R :

$$(1, 2), (2, 1) \rightarrow (1, 1)$$

$$(2, 1), (1, 2) \rightarrow (2, 2)$$

- Не антисимметрично: $1R2$ и $2R1$, но $1 \neq 2$ — два различных элемента связаны в обе стороны.
- Не тотально: $(1, 3), (3, 1) \notin R$ — элементы 1 и 3 несравнимы.

ЛОВУШКА Симметричность и антисимметричность — не противоположности

Отношение может быть симметричным и антисимметричным одновременно. Равенство на числах — такой пример: из $a = b$ следует $b = a$, и из $a = b$ и $b = a$ следует $a = b$. Антисимметричность — ограничение на взаимные связи: aRb и bRa возможны, только когда $a = b$.

Три пары свойств, которые полезно различать:

- **Иррефлексивность** сильнее, чем «не рефлексивно»: «не рефлексивно» означает лишь, что для *некоторого* элемента $\neg(aRa)$. Иррефлексивность требует этого для *всех* a . Контрпример: на множестве $\{1, 2\}$ отношение $R = \{(1, 1)\}$ не рефлексивно (нет $(2, 2)$), но и не иррефлексивно (есть $(1, 1)$).

- **Асимметричность** влечёт иррефлексивность: если aRa , то из асимметричности следует $\neg(aRa)$ — противоречие. Значит, aRa невозможно.
- **Антисимметричность** допускает рефлексивность (и даже требует её в частичных порядках). Например, $\leq$ на числах антисимметрично: из $a \leq b$ и $b \leq a$ следует $a = b$. При этом $\leq$ рефлексивно: $a \leq a$ — каждый элемент связан с собой.

Каждое из этих свойств имеет наглядную интерпретацию в матричном и графовом представлении.

УТВЕРЖДЕНИЕ 5.1 Определение свойств по представлениям

- **Рефлексивность**: все диагональные элементы матрицы равны 1. Каждая вершина графа имеет петлю.
- **Иррефлексивность**: все диагональные элементы матрицы равны 0. Петель в графе нет.
- **Симметричность**: матрица симметрична:

$$M = M^T.$$

Каждое ребро в графе двунаправлено.

- **Антисимметричность**: нет симметричных единиц вне диагонали матрицы. Нет двунаправленных рёбер в графе.
- **Транзитивность**: путь длины 2 ($a \rightarrow b \rightarrow c$) требует прямого ребра $a \rightarrow c$.

Набросок доказательства

Каждое утверждение — прямой перевод определения свойства на язык матриц и графов.

Рефлексивность требует aRa для каждого элемента: в матрице это $M_{ii} = 1$, в графе — петля при каждой вершине. Иррефлексивность — отрицание: $M_{ii} = 0$, петель нет. Симметричность: взаимные пары aRb, bRa встречаются вместе. В матрице $M_{ij} = M_{ji}$, в графе каждое ребро двунаправлено.

Антисимметричность: взаимная связь возможна лишь при $a = b$. В матрице нет пары $M_{ij} = M_{ji} = 1$ при $i \neq j$, в графе нет двунаправленных рёбер. Транзитивность: из aRb и bRc следует aRc . Это в точности означает, что путь длины 2 обязан иметь прямое ребро.

Пример Минимальные контрпримеры к свойствам

На множестве $A = \{1, 2, 3\}$ матрицы ниже нарушают ровно одно из трёх основных свойств:

- Не рефлексивно (нет $(2, 2)$), но симметрично и транзитивно:

$$M_{\text{not ref}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Не симметрично ((1, 2) есть, (2, 1) — нет), но рефлексивно и транзитивно:

$$M_{\text{not sym}} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Не транзитивно ((1, 2) и (2, 3) есть, а (1, 3) — нет), но рефлексивно и симметрично:

$$M_{\text{not trans}} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

ЛОВУШКА Транзитивность: одной тройки мало

Проверив одну тройку aRb, bRc , делают вывод о транзитивности. Но определение требует, чтобы $aRb \wedge bRc \rightarrow aRc$ выполнялось для *всех* троек — одна проверка ничего не доказывает. Отношение «сосед по номеру» на $\{1, 2, 3\}$: $1R2$ и $2R3$, но $(1, 3) \notin R$.

Проверка Свойства бинарных отношений

1. Чем антисимметричность отличается от асимметричности?
2. Почему «не рефлексивно» и «иррефлексивно» — не одно и то же?

С практикой каждое из определений начинает восприниматься как визуальный образ: петля, двунаправленное ребро, замыкание пути. Транзитивность — правило «два шага обязывают к одному»: путь длины 2 требует прямого ребра. Но прежде чем мы перейдём к главным классам отношений, разберём одну техническую конструкцию, которая встречается во всех разделах дискретной математики — от графов до формальных языков.

5.1.3 Замыкания отношений

Иногда отношение «почти» обладает нужным свойством, но ему не хватает нескольких пар. Отношение «быть начальником» на множестве сотрудников компании транзитивно по замыслу (начальник начальника — тоже начальник), но оно не обязано быть рефлексивным: сотрудник не является начальником самому себе. Если мы хотим формально считать каждого сотрудника своим собственным начальником — для единообразия рассуждений —, мы добавляем все пары (x, x) , и только их. Отношение «знает пароль от учётной записи» может быть несимметрич-

ным: администратор знает пароль пользователя, но не наоборот. Если мы строим симметричное замыкание — добавляем обратные пары.

Замыкание добавляет минимально необходимое количество пар, чтобы свойство выполнялось. Минимальность критична: нам нужны ровно те пары, которых не хватает, и ничего сверх того. Иначе вместо предсказуемой конструкции мы получим произвольное расширение, не связанное с исходным отношением никакой закономерностью.

ОПРЕДЕЛЕНИЕ 5.12 Рефлексивное замыкание

Отношение называется **рефлексивным замыканием** отношения R и обозначается R^- , если оно получено из R добавлением всех недостающих петель:

$$R^- = R \cup \{(a, a) \mid a \in A\}.$$

ОПРЕДЕЛЕНИЕ 5.13 Симметричное замыкание

Отношение называется **симметричным замыканием** отношения R , если оно получено из R добавлением обратной пары (b, a) к каждой паре $(a, b) \in R$:

$$R \cup \{(b, a) \mid (a, b) \in R\}.$$

Примечание Почему замыкание единственно

Отношений, содержащих R и обладающих нужным свойством, обычно много: подойдёт и всё $A \times A$. Требование минимальности выделяет наименьшее: пересечение всех надмножеств R с данным свойством само содержит R и обладает этим свойством. Поэтому замыкание определено однозначно. Для рефлексивного и симметричного замыканий это видно прямо, для транзитивного доказано ниже.

Пример Рефлексивное и симметричное замыкания

Пусть $A = \{1, 2, 3\}$. Возьмём $R = \{(1, 2), (2, 3)\}$.

- Рефлексивное замыкание добавляет недостающие петли: $R \cup \{(1, 1), (2, 2), (3, 3)\} = \{(1, 1), (1, 2), (2, 2), (2, 3), (3, 3)\}$.
- Симметричное замыкание добавляет обратные пары: $R \cup \{(2, 1), (3, 2)\} = \{(1, 2), (2, 1), (2, 3), (3, 2)\}$.

Рефлексивное и симметричное замыкания строятся непосредственно — добавлением недостающих пар. Транзитивное замыкание устроено сложнее, и для его построения нужен инструмент, собирающий цепочки связей, — *композиция* отношений.

Прежде чем вводить композицию, расширим определение. До сих пор мы рассматривали отношения на одном множестве — подмножества $A \times A$. Но отношения

могут связывать элементы *разных* множеств: студент записан на курс (Студенты $\times$ Курсы), город связан дорогой с городом (хотя множества могут и совпадать), число сопоставлено остатку от деления.

ОПРЕДЕЛЕНИЕ 5.14 Бинарное отношение между множествами

Бинарным отношением между двумя множествами называется подмножество $R \subseteq A \times B$. При $A = B$ получаем отношение на одном множестве — частный случай, рассмотренный выше.

ОПРЕДЕЛЕНИЕ 5.15 Композиция отношений

Композиция $S \circ R$ — отношение из всех пар, связанных через промежуточный элемент:

$$S \circ R = \{(a, c) \mid \exists b \in B : (a, b) \in R \wedge (b, c) \in S\}.$$

Примечание Порядок в композиции

Запись $S \circ R$ читается справа налево: сначала R связывает a с b , затем S связывает b с c . Это то же соглашение, что и в композиции функций: $(f \circ g)(x) = f(g(x))$ — сначала применяется g , затем f .

Пример Композиция отношений

Пусть $R = \{(1, a), (2, b), (3, a)\}$ — отношение из $A = \{1, 2, 3\}$ в $B = \{a, b\}$, а $S = \{(a, x), (b, y)\}$ — отношение из B в $C = \{x, y, z\}$. Тогда $S \circ R = \{(1, x), (2, y), (3, x)\}$: каждая пара — сквозной путь через промежуточный элемент. Пара $(1, x)$ собрана через a ($1Ra$ и aSx), пара $(2, y)$ — через b , пара $(3, x)$ — снова через a .

Отношение может состоять и в композиции с самим собой: для $R = \{(1, 2), (2, 3), (3, 4)\}$ на $\{1, 2, 3, 4\}$ получаем $R \circ R = \{(1, 3), (2, 4)\}$ — «через одно».

В матричном представлении композиция читается как булево умножение матриц. Единица на месте (i, j) стоит, если найдётся индекс k с условием $M_R(i, k) = 1, M_S(k, j) = 1$. Для $R = \{(1, 2), (2, 3)\}$ на $\{1, 2, 3\}$ получаем

$$M_{R \circ R} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

и единственная единица стоит в позиции $(1, 3)$ — единственный путь длины два.

Композиция задаёт степени отношения: $R^1 = R, R^{k+1} = R^k \circ R$. R^2 даёт пары, достижимые за два шага, R^3 — за три, и так далее. Транзитивное замыкание собирает

все такие степени в одно отношение — пары, достижимые за любое положительное число шагов.

ОПРЕДЕЛЕНИЕ 5.16 Транзитивное замыкание

Транзитивное замыкание R^+ — минимальное по включению транзитивное отношение, содержащее исходное.

Эквивалентно, R^+ состоит из всех пар (a, b) , для которых существует ориентированный путь из a в b длины хотя бы 1:

$$R^+ = \bigcup_{k=1}^{\infty} R^k$$

ОПРЕДЕЛЕНИЕ 5.17 Рефлексивно-транзитивное замыкание

Рефлексивно-транзитивное замыкание R^* — отношение пар, достижимых за любое число шагов, включая ноль:

$$R^* = R^= \cup R^+ = \bigcup_{k=0}^{\infty} R^k$$

где $R^0 = \{(a, a) \mid a \in A\}$ — тождественное (диагональное) отношение.

Тот же механизм хватает малыми степенями в рекомендациях подписок: социальная сеть предлагает «друзей друзей», то есть пары из $R \circ R$. Отношение «подписан на» не транзитивно: Алиса следит за Бобом, Боб — за Чаком, а Алиса на Чака — нет. Система предлагает Алисе подписаться на Чака — добавляет ровно недостающее ребро.

Может показаться, что транзитивное замыкание строится одним проходом: для каждой пары рёбер $a \rightarrow b$ и $b \rightarrow c$ добавить недостающее ребро $a \rightarrow c$. Но добавленные рёбра порождают новые обязанности. Возьмём $R = \{(1, 2), (2, 3), (3, 4)\}$. После первого прохода добавляются $(1, 3)$ (из цепочки $1 \rightarrow 2 \rightarrow 3$) и $(2, 4)$ (из $2 \rightarrow 3 \rightarrow 4$). Теперь в расширенном отношении есть $(1, 3)$ и $(3, 4)$ — новая цепочка, требующая добавления $(1, 4)$.

На бесконечных множествах процесс не останавливается вовсе: отношение $R = \{(n, n+1) \mid n \in \mathbb{N}\}$ требует для каждого n ребра $(1, n)$, а таких рёбер бесконечно много.

УТВЕРЖДЕНИЕ 5.2 Минимальность объединения степеней

Объединение $\bigcup_{k=1}^{\infty} R^k$ транзитивно и содержится в любом транзитивном отношении, содержащем R .

Доказательство

Докажем оба свойства: транзитивность и вложенность в любое транзитивное надмножество.

Сначала транзитивность: склейка путей даёт путь суммы длин. Если $(a, b) \in R^i$ и $(b, c) \in R^j$, то $(a, c) \in R^{i+j} \subseteq \bigcup_{k=1}^{\infty} R^k$.

Теперь вложенность: любое транзитивное $T \supseteq R$ содержит каждую степень R^k — по индукции, начиная с $R \subseteq T$ и склеивая пути внутри T . Поэтому T содержит и объединение всех степеней.

Тем самым объединение степеней — минимальное транзитивное надмножество R , то есть транзитивное замыкание: два определения — минимальность по включению и объединение степеней — эквивалентны. Минимальность удобна в доказательствах, объединение степеней — в алгоритмических построениях.

Пример Транзитивное замыкание: предок

Пусть R — отношение «быть родителем» на множестве людей. Пара $(a, b) \in R$, если a является родителем b . Тогда R^2 — отношение «быть бабушкой или дедушкой» (родитель родителя). R^3 — «быть прародителем» (родитель родителя родителя). Транзитивное замыкание R^+ — это отношение «быть предком». Пара $(a, b) \in R^+$, если a — предок b в любом поколении.

Запрос «найти всех предков человека» в генеалогической базе данных — это в точности вычисление транзитивного замыкания отношения «родитель». Классический SQL не поддерживает транзитивное замыкание напрямую. Для этого нужны рекурсивные Common Table Expressions (WITH RECURSIVE), добавленные в стандарт SQL:1999. В расширениях реляционной алгебры операция транзитивного замыкания как раз и называется *closure*.

АЛГОРИТМ Уоршелла

Алгоритм вычисляет транзитивное замыкание отношения по матрице смежности M . Время работы — $O(n^3)$ для квадратной матрицы.

1. Для каждой вершины k от 1 до n :
 - Для каждой пары (i, j) : $M_{ik} = 1 \wedge M_{kj} = 1 \rightarrow M_{ij} := 1$.
2. После завершения M является матрицей смежности транзитивного замыкания.

Интуиция: на итерации k добавляются пути через вершину k (в дополнение к уже найденным через вершины $1, \dots, k-1$). После n итераций матрица содержит все прямые и не прямые связи.

Пример Прогон алгоритма Уоршелла

Отношение $R = \{(1, 2), (2, 3)\}$ на трёх вершинах задано матрицей

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

1. $k = 1$: из вершины 1 есть путь в 2, но рёбер, ведущих в 1, нет — матрица не меняется.
2. $k = 2$: из вершины 2 есть путь в 3, а из 1 — в 2. Поэтому $M_{1,3} := 1$.
3. $k = 3$: из вершины 3 путей нет — матрица не меняется.

Итог в том, что замыкание содержит $(1, 2), (2, 3), (1, 3)$ — исходные пары плюс добавленная.

Алгоритм Уоршелла (1962²⁴) и алгоритм Флойда–Уоршелла для кратчайших путей — структурно один и тот же алгоритм: замена $\vee \rightarrow \min, \wedge \rightarrow +$ превращает тот же перебор вершин в поиск кратчайших путей²⁵.

В алгоритме Флойда–Уоршелла роль матрицы смежности играет таблица расстояний: на пересечении строки и столбца — длина кратчайшего пути.

Примечание Какое замыкание какой алгоритм вычисляет

Диагональ таблицы расстояний заполнена нулями: это пути длины ноль — диагональ R^0 , которая отличает R^* от R^+ . Сам тройной цикл при этом один: булева матрица на входе даёт R^+ , матрица с единичной диагональю — R^* , таблица расстояний — длины кратчайших путей.

Термин «замыкание» — калька с английского *closure*, пришедшая, вероятно, из топологии: замыкание множества — наименьшее замкнутое множество, его содержащее. В алгебре и логике замыканием относительно операции называют наименьшее множество, которое содержит исходное и замкнуто относительно этой операции. Замыкания отношений встраиваются в этот ряд на общих основаниях: объект достраивается до минимального, обладающего нужным свойством.

Мы определили язык отношений и их алгебраические свойства. Теперь посмотрим на два основных класса отношений, каждый из которых моделирует свой тип связей между объектами.

Отношения эквивалентности формализуют идею «одинаковости по некоторому признаку»: числа с одинаковым остатком от деления, строки одинаковой длины, объекты с одинаковым хешем, люди одного года рождения. Они разбивают множество на непересекающиеся классы, внутри которых все элементы «одинаковы» с точки зрения отношения.

²⁴S. Warshall. «A Theorem on Boolean Matrices». Journal of the ACM, 1962.

²⁵R. W. Floyd. «Algorithm 97: Shortest Path». Communications of the ACM, 1962.

Частичные порядки формализуют идею «предшествования» или «иерархии»: задачи в проекте, курсы в учебном плане, версии программного обеспечения, подмножества по включению. Они задают структуру подчинения — и, в отличие от эквивалентностей, не все элементы обязаны быть сравнимы.

Формально оба класса задаются тремя свойствами: эквивалентность — это рефлексивность, симметричность и транзитивность, частичный порядок — рефлексивность, антисимметричность и транзитивность. Разница между ними — ровно одна аксиома. Остальные комбинации либо сводятся к этим двум классам, либо слишком бедны для содержательной теории: рефлексивное и симметричное отношение без транзитивности не держит классы вместе, а симметричность вместе с антисимметричностью влечёт равенство любых связанных элементов.

Замечание Три аксиомы — две структуры

Смена одной аксиомы перерождает структуру целиком, и тот же эффект встречается за пределами теории отношений.

- В алгебре группа, моноид и полугруппа различаются ровно одной аксиомой. Уберите обратимость — группа становится моноидом. Уберите ещё и нейтральный элемент — получите полугруппу.
- В топологии исключение аксиомы симметрии из определения метрики даёт квазиметрическое пространство, где расстояние от A до B не обязано равняться расстоянию от B до A .

Сами аксиомы независимы, но структура, которую они задают сообща, не сводится к их перечню.

§ 5.2 Отношение эквивалентности

Отношение эквивалентности отвечает на вопрос, что считать одним и тем же. Два элемента эквивалентны, если они неразличимы с точки зрения интересующего нас свойства. Образец — обычное равенство: число равно только самому себе, и три свойства подобраны так, чтобы равенство им удовлетворяло. Эквивалентность — то же «равно», но по выбранному признаку. Записи $10 \bmod 3$, $7 \bmod 3$, $13 \bmod 3$ выглядят по-разному, а по признаку остатка это одно и то же.

ОПРЕДЕЛЕНИЕ 5.18 Отношение эквивалентности

Бинарное отношение $\sim \subseteq A \times A$ называется **отношением эквивалентности**, если оно:

- **Рефлексивно:** $a \sim a$;
- **Симметрично:** $a \sim b \rightarrow b \sim a$;
- **Транзитивно:** $a \sim b \wedge b \sim c \rightarrow a \sim c$.

Пример Рефлексивность + симметричность $\neq$ транзитивность

Возьмём

$$A = \{1, 2, 3\}.$$

Рассмотрим

$$R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 1), (2, 3), (3, 2)\}.$$

Отношение рефлексивно (все три диагональные пары присутствуют) и симметрично (каждая не-диагональная пара имеет обратную). Но оно *не* транзитивно: $(1, 2), (2, 3) \in R, (1, 3) \notin R$.

Матрица смежности — та же $M_{\text{not trans}}$, что в примере выше: не хватает одного ребра $1 \rightarrow 3$, чтобы замкнуть единственный путь длины 2. Достаточно добавить $(1, 3)$ и по симметрии $(3, 1)$ — и отношение становится эквивалентностью с единственным классом $\{1, 2, 3\}$.

Пример Сравнимость по модулю 3

На множестве $A = \{0, 1, 2, 3, 4, 5\}$ определим отношение: Считаем $a \sim b$, если a и b дают одинаковый остаток при делении на 3. Тогда $0 \sim 3, 1 \sim 4, 2 \sim 5$ (остатки 0, 1, 2). Отношение рефлексивно: $a \sim a$ — одинаковый остаток с собой. Симметрично: если a и b имеют одинаковый остаток, то и b с a . Транзитивно: если a эквивалентен b , и b эквивалентен c , то все три имеют одинаковый остаток.

Классы эквивалентности:

- $[0] = \{0, 3\}$
- $[1] = \{1, 4\}$
- $[2] = \{2, 5\}$

Три свойства — рефлексивность, симметричность, транзитивность — работают вместе. Рефлексивность помещает каждый элемент в его собственный класс. Симметричность делает классы «круглыми»: если a в классе b , то и b в классе a . Транзитивность не даёт классам пересекаться.

Отношение эквивалентности группирует элементы в непересекающиеся классы — множества взаимно эквивалентных элементов.

ОПРЕДЕЛЕНИЕ 5.19 Класс эквивалентности

Множество всех элементов, эквивалентных a , называется **классом эквивалентности** и обозначается $[a]$:

$$[a] = \{b \in A \mid a \sim b\}.$$

ОПРЕДЕЛЕНИЕ 5.20 Представитель класса

Элемент a (как и любой другой элемент класса) называется **представителем** класса.

Из определения немедленно следуют три свойства, которые делают классы эквивалентности удобным инструментом.

УТВЕРЖДЕНИЕ 5.3 Свойства классов эквивалентности

- $a \in [a]$ (рефлексивность гарантирует, что класс содержит своего представителя).
- $[a] = [b] \leftrightarrow a \sim b$: классы равны для эквивалентных представителей.
- $[a] \cap [b] = \emptyset$, если и только если $\neg(a \sim b)$ (классы либо совпадают, либо не пересекаются).

Набросок доказательства

Каждое свойство — прямое следствие определения класса $[a] = \{b \in A \mid a \sim b\}$ и свойств эквивалентности.

Принадлежность. Рефлексивность даёт $a \in [a]$: каждый элемент лежит в своём классе.

Равенство классов. Если $[a] = [b]$, то по предыдущему пункту элемент b лежит в классе $[a]$, значит, a и b эквивалентны.

Обратно, если $a \sim b$, то для любого x справедлива цепочка:

$$x \in [a] \rightarrow a \sim x \rightarrow b \sim x \rightarrow x \in [b]$$

Первый и последний переходы — определения класса, средний использует симметричность и транзитивность. Аналогично $x \in [b] \rightarrow x \in [a]$. Следовательно, $[a] = [b]$.

Непересечение. Если $[a] \cap [b] \neq \emptyset$, возьмём $c \in [a] \cap [b]$. Тогда $a \sim c, b \sim c$: оба связаны с общим элементом. Симметричность разворачивает вторую пару, транзитивность склеивает обе в $a \sim b$. По предыдущему пункту получаем $[a] = [b]$. Значит, классы либо не пересекаются, либо совпадают.

Значит, отношение эквивалентности разбивает множество на непересекающиеся блоки. На рис. 13 видно, как числа $\{1, \dots, 10\}$ распадаются на три класса: в каждом собраны числа с одним и тем же остатком от деления на 3.

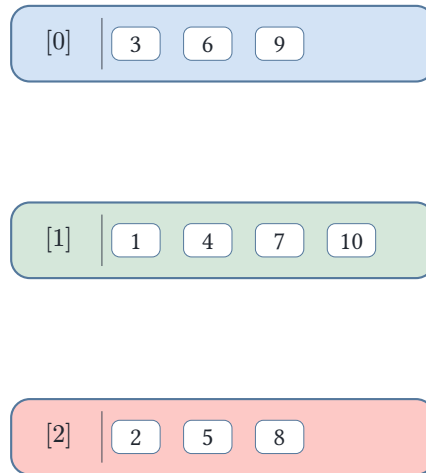


Рис. 13. Три класса эквивалентности по модулю 3.

Пример Равенство в Rust: PartialEq и Eq

Отношение эквивалентности — это рефлексивное, симметричное и транзитивное равенство. В Rust роль равенства играет трейт `PartialEq`: его метод `eq` — это оператор `==`. Документация Rust требует симметричности и транзитивности, но не рефлексивности, поэтому не всякий `PartialEq` является `Eq`.

Трейт `Eq` — маркер. Реализовать его — значит пообещать, что $a = a$ верно всегда, то есть что `==` задаёт настоящее отношение эквивалентности.

Контрпример дают числа с плавающей точкой. Тип `f64` реализует `PartialEq`, но не `Eq`, потому что `NaN != NaN`: рефлексивность нарушена, и `==` на `f64` — не отношение эквивалентности.

```
let nan = f64::NaN;
assert!(nan != nan); // рефлексивность нарушена: NaN не равен себе
```

Для множества равенство по элементам — настоящее отношение эквивалентности, поэтому `Eq` реализуем:

```
/// Множество как отсортированный список без повторений.
struct Set(Vec<u32>);

impl PartialEq for Set {
    fn eq(&self, other: &Self) -> bool { self.0 == other.0 }
}
impl Eq for Set {} // a == a верно для любого a
```

§ 5.3 Теорема о разбиении

ТЕОРЕМА 5.4 Отношения эквивалентности и разбиения

Множество всех классов эквивалентности $A/\sim = \{[a] \mid a \in A\}$ образует разбиение исходного множества. Обратно, каждое разбиение определяет отношение эквивалентности: $a \sim b$, если a и b принадлежат одной части разбиения.

Доказательство

Докажем эквивалентность в обе стороны.

Покажем сначала, что классы эквивалентности образуют разбиение. Классы попарно не пересекаются (по свойству выше) и покрывают всё множество (каждый элемент принадлежит своему классу в силу рефлексивности). Следовательно, они образуют разбиение.

Обратно: дано разбиение $\{A_i\}$ множества на части. Определим $a \sim b$, если a и b лежат в одной части разбиения. Это отношение рефлексивно: каждый элемент лежит в своей части. Симметрично: если a и b в одной части, то и b и a в одной. Транзитивно: если a с b в одной части и b с c в одной части, то все трое в одной.

Пример Разбиение по году рождения

Возьмём множество студентов и разобьём его на части по году рождения: студенты 2005 года рождения, студенты 2006 года и так далее. Определим отношение: два студента эквивалентны, если родились в одном году.

Это отношение автоматически рефлексивно: каждый родился в своём году. Симметрично: если a родился в одном году с b , то и b с a . Транзитивно: если a с b в одном году и b с c в одном году, то все трое — в одном. Классы эквивалентности — в точности части исходного разбиения.

Эта теорема устанавливает взаимно-однозначное соответствие между отношениями эквивалентности на A и разбиениями A . Каждое отношение эквивалентности задаёт разбиение. Каждое разбиение задаёт отношение эквивалентности.

Геолог смотрит на ту же территорию, что и гидролог, — а видит другое. Одна карта разрезает землю по составу пород: гранит отдельно, базальт отдельно. Другая — по бассейнам рек. Границы проходят в разных местах, хотя земля под ними одна. Каждая карта задаёт отношение эквивалентности: две точки эквивалентны, если попали в одну зону. Зоны не пересекаются и покрывают всю территорию — в точности как классы.

Теорема о разбиении говорит: провести границу и задать критерий «находиться в одной зоне» — не два последовательных шага. Это одна структура, увиденная с двух ракурсов.

§ 5.4 Примеры отношений эквивалентности

Пример Сравнимость по модулю m (*congruence modulo m*)

Зафиксируем целое $m \geq 1$. Определим отношение на $\mathbb{Z}$: $a \sim b \leftrightarrow a \equiv b \pmod{m}$. То есть m делит $(a - b)$.

Это отношение эквивалентности. Классы эквивалентности: $\mathbb{Z}_m = \{[0], [1], \dots, [m-1]\}$ — кольцо вычетов по модулю m .

Сложение и умножение *корректно определены* (*well-defined*) на классах эквивалентности — результат не зависит от выбора представителей. Действительно, пусть $a \equiv a' \pmod{m}$, $b \equiv b' \pmod{m}$. Тогда $(a' + b') - (a + b) = (a' - a) + (b' - b)$: каждое слагаемое делится на m , значит, и сумма делится. Для умножения: $a'b' - ab = a'(b' - b) + b(a' - a)$ — оба слагаемых делятся на m . Тем самым $\mathbb{Z}_m$ становится алгебраической структурой — основой модульной арифметики и криптографии.

Тот же вопрос — что считать одинаковым — можно задать не только числам, но и множествам.

Пример Равномощность множеств

На совокупности всех множеств определим: $X \sim Y$, если между X и Y существует биекция — взаимно однозначное соответствие (подробно — глава 6).

Это отношение эквивалентности. Класс эквивалентности — это **кардинальное число** («количество элементов» для бесконечных множеств). Так, все счётные множества ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$) принадлежат одному классу эквивалентности. Все множества мощности континуума ($\mathbb{R}$, $\mathcal{P}(\mathbb{N})$) — другому. Подробнее — в главе 7.

От множеств перейдём к объектам с внутренней структурой — графам.

Пример Изоморфизм графов

На множестве всех графов определим: $G \sim H$, если графы изоморфны — биекция между вершинами, сохраняющая рёбра (см. главу 6).

Изоморфизм графов — отношение эквивалентности. Класс эквивалентности — все графы с одинаковой структурой («непомеченный граф»). Вернёмся к этому в главе 9.

§ 5.5 Приложения отношений эквивалентности

Отношения эквивалентности — рабочий инструмент. Рассмотрим два примера из практики программиста.

Группировка записей по значению атрибута (например, «город проживания») разбивает набор данных на классы эквивалентности: две записи эквивалентны, если значения атрибута совпадают. Каждый класс — множество записей с одинаковым значением атрибута. Блоки разбиения попарно не пересекаются, а их объединение покрывает всю выборку.

Иерархическая кластеризация обобщает это: вместо одного атрибута используется *мера близости* (расстояние между точками), а вместо одного разбиения — последовательность всё более грубых отношений эквивалентности, параметризованных порогом близости. При пороге $\varepsilon = 0$ каждая точка — отдельный класс. С ростом ε близкие точки сливаются в кластеры. При достаточно большом ε все точки слиты в один кластер. Результат изображается дендрограммой: дерево, листья которого — исходные точки, а внутренние узлы — слияние кластеров на конкретном пороге.²⁶ Полное построение — от расстояния к дереву и обратно — разобрано в дополнительном разделе 5.9 ниже.

Самый известный пример иерархической кластеризации — эволюционное дерево видов: листья — ныне живущие организмы, а внутренние узлы — их общие предки. Как сравнивают виды, не имея ДНК вымершего предка, и что из этого сравнения получается, разобрано в разделе 5.9.3.

Хеш-функция $h : K \rightarrow \{0, \dots, m - 1\}$ задаёт отношение эквивалентности на пространстве ключей: Пишем $k_1 \sim k_2$, если $h(k_1) = h(k_2)$ — ключи эквивалентны, когда попадают в одну корзину. Это отношение рефлексивно: $h(k) = h(k)$ — значение совпадает с собой. Симметрично: равенство значений взаимно. Транзитивно: цепочка $h(k_1) = h(k_2) = h(k_3)$ означает совпадение всех трёх значений.

Каждая корзина (bucket) — класс эквивалентности: множество всех ключей с данным значением хеш-функции. Коллизия — ситуация, когда два различных ключа принадлежат одному классу. Коллизии неизбежны: если $|K| > m$, хотя бы одна корзина содержит более одного ключа. Разрешение коллизий (цепочки, открытая адресация) — это реализация работы с классом эквивалентности как с множеством. Ключи в одной корзине хранятся вместе, и поиск нужного требует дополнительного сравнения внутри класса.²⁷

§ 5.6 Минимизация ДКА через отношение неразличимости

Детерминированный конечный автомат (ДКА) — простейшая модель вычислений: конечный набор состояний и правило смены состояния при чтении символа. Автоматы подробно разбираются в главе 23, здесь нужны только обозначения.

Обозначения простые: алфавит — конечный набор символов (Σ), множество всех строк над ним — Σ^* , функция переходов δ действует на одном символе, а её расши-

²⁶A. K. Jain, M. N. Murty, P. J. Flynn. «Data Clustering: A Review». ACM Computing Surveys, 1999.

²⁷D. E. Knuth. «The Art of Computer Programming». Vol. 3: Sorting and Searching, Section 6.4: Hashing, 1973.

рение δ^* — на строке целиком. Принимающие состояния образуют множество F . Автомат принимает строку, если после её чтения оказывается в одном из них.

На состояниях ДКА определяется отношение неразличимости: $p \sim q$, если для любой входной строки автомат из состояния p принимает её тогда и только тогда, когда принимает из q . Формально: $p \sim q$ тогда и только тогда, когда

$$\forall w \in \Sigma^* : \delta^*(p, w) \in F \leftrightarrow \delta^*(q, w) \in F.$$

Это отношение эквивалентности. Склеивание эквивалентных состояний (замена класса одним состоянием) даёт *минимальный* ДКА — автомат с наименьшим возможным числом состояний, распознающий тот же язык.

Минимальность доказывается теоремой Майхилла–Нерода:²⁸ число состояний минимального ДКА равно числу классов эквивалентности отношения Майхилла–Нерода на строках.

Подробнее — в главе 23.

Проверка Отношения эквивалентности

1. Почему хеш-функция задаёт отношение эквивалентности на ключах, и что служит классами?
2. Почему в отношении $R = \{(1, 1), (2, 2), (3, 3), (1, 2), (2, 1), (2, 3), (3, 2)\}$ не хватает ровно пар $(1, 3)$ и $(3, 1)$ для эквивалентности?

§ 5.7 Частичные порядки

Частичный порядок — второй основной тип бинарных отношений, наряду с отношением эквивалентности.

ОПРЕДЕЛЕНИЕ 5.21 Частичный порядок

Бинарное отношение $\preceq$ на множестве называется **частичным порядком**, если оно рефлексивно, антисимметрично и транзитивно.

Подробнее обсуждение частичных порядков — в главе 8.

Пример Делимость и включение

Отношение делимости $a \mid b$ на $\mathbb{N}$ — частичный порядок: Рефлексивность: $a \mid a$. Антисимметричность: если $a \mid b$ и $b \mid a$, то $a = b$. Транзитивность: если $a \mid b$ и $b \mid c$, то $a \mid c$. При этом 2 и 3 несравнимы: ни 2 не делит 3, ни 3 не делит 2.

²⁸J. Myhill. «Finite Automata and the Representation of Events». WADD TR-57-624, 1957. A. Nerode. «Linear Automaton Transformations». Proceedings of the AMS, 1958.

Отношение включения $\subseteq$ на $\mathcal{P}(A)$ — тоже частичный порядок: Рефлексивность: $X \subseteq X$. Антисимметричность: если $X \subseteq Y$ и $Y \subseteq X$, то $X = Y$. Транзитивность: если $X \subseteq Y$ и $Y \subseteq Z$, то $X \subseteq Z$. Множества $\{1\}$ и $\{2\}$ несравнимы по включению.

В отличие от отношения эквивалентности, которое группирует элементы в классы «одинаковых», частичный порядок выстраивает иерархию — и допускает, что некоторые элементы могут быть несравнимы (ни $a \preceq b$, ни $b \preceq a$).

Графически частичный порядок изображают *диаграммой Хассе*: элементы — точки на плоскости, меньшее ниже большего, и рисуются только связи, не выводимые транзитивностью. На рис. 14 показана диаграмма Хассе включения на $\mathcal{P}(\{1, 2\})$: пустое множество расположено ниже всех, а $\{1\}$ и $\{2\}$ несравнимы.

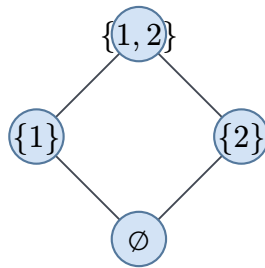


Рис. 14. Диаграмма Хассе включения на $\mathcal{P}(\{1, 2\})$.

Диаграмма Хассе — основной инструмент визуализации порядков, и мы будем активно им пользоваться в следующей главе.

Строгие и линейные порядки, экстремальные элементы, грани и топологическая сортировка — предмет следующей главы.

Если у любых двух элементов существуют наименьшая общая верхняя грань и наибольшая общая нижняя грань, порядок называется *решёткой*. Ричард Дедекинд описал её в 1897 году, изучая делимость целых чисел: НОД и НОК двойственны друг другу. Та же двойственность связывает $\cup$ с $\cap$ в алгебре множеств и $\vee$ с $\wedge$ в логике. Определение, дистрибутивность и характеристика Биркгофа разбираются в главе 8, а построение чисел у Дедекинда — в главе 22. Решётка — рабочий инструмент статического анализа: свойства программы моделируются элементами решётки, а информация из разных ветвей условного оператора объединяется *супремумом*.

Последний инструмент главы — комбинаторный приём, а не очередная структура. Принципу Дирихле не нужны ни матрицы, ни графы: он опирается на простой подсчёт и из него извлекает гарантии.

§ 5.8 Принцип Дирихле

Принцип Дирихле (pigeonhole principle) — одно из самых простых комбинаторных рассуждений, с которого начинаются многие нетривиальные доказательства.

ТЕОРЕМА 5.5 Принцип Дирихле

Если $n + 1$ объектов разместить в n ящиках, то хотя бы в одном ящике окажется не менее двух объектов. Общая форма: если m объектов размещены в n ящиках, то в некотором ящике находится $\geq \lceil \frac{m}{n} \rceil$ объектов.

Пример День рождения

В группе из 367 человек у двоих совпадает день рождения: возможных дат 366, а людей 367 — по принципу Дирихле совпадение гарантировано.

Пример Общая форма: месяцы рождения

В группе из 40 человек по крайней мере четверо родились в одном месяце: $\lceil \frac{40}{12} \rceil = 4$. Принцип Дирихле даёт нижнюю границу, ничего не зная о конкретных датах.

На языке функций: если $|A| > |B|$, то не существует инъекции $f : A \rightarrow B$.

Детальное изложение принципа Дирихле, его обобщений и приложений (включая теорему Эрдёша–Секереша) — в главе 18 «Комбинаторика».

§ 5.9 * Ультраметрика и иерархическая кластеризация

В разделе 5.5 мы строили дендрограмму: дерево, листья которого — точки, а внутренние узлы — слияния кластеров на некотором пороге ε . За этой картинкой стоит числовая структура: порог, на котором две точки впервые попадают в один класс, измеряет, насколько близкими они должны были оказаться.

Иерархия разбиений сама задаёт расстояние между точками, и это расстояние подчиняется закону строже обычного неравенства треугольника. Что это за закон и как по таблице попарных расстояний восстановить дерево?

5.9.1 Ультраметрика**ОПРЕДЕЛЕНИЕ 5.22** Ультраметрика

Функция $d : X \times X \rightarrow \mathbb{R}$ на конечном множестве X называется **ультраметрикой** (от англ. ultrametric), если для любых $x, y, z \in X$:

- $d(x, y) \geq 0$, причём $d(x, y) = 0 \leftrightarrow x = y$;
- $d(x, y) = d(y, x)$;
- $d(x, z) \leq \max(d(x, y), d(y, z))$ — **усиленное неравенство треугольника**.

Обычное неравенство треугольника связывает расстояния суммой: $d(x, z) \leq d(x, y) + d(y, z)$. Усиленное неравенство заменяет сумму максимумом: $d(x, z) \leq$

$\max(d(x, y), d(y, z))$. Для неотрицательных чисел максимум не превосходит суммы, поэтому усиленная форма влечёт обычную.

Смысл усиления — в том, как неравенство действует на отношение «на расстоянии не больше ε ». Из $d(x, y) \leq \varepsilon$ и $d(y, z) \leq \varepsilon$ обычное неравенство даёт лишь $d(x, z) \leq 2\varepsilon$, а усиленное — $d(x, z) \leq \varepsilon$. Транзитивность порога — в точности то свойство, которое делает отношение $x \equiv_\varepsilon y$ отношением эквивалентности. Ниже это станет теоремой.

ТЕОРЕМА 5.6 Равнобедренность ультаметрики

Симметричная функция $d : X \times X \rightarrow \mathbb{R}$ с $d(x, y) = 0 \leftrightarrow x = y$ является ультаметрикой тогда и только тогда, когда в каждой тройке x, y, z из трёх расстояний $d(x, y), d(y, z), d(x, z)$ два наибольших равны.

Доказательство

Докажем эквивалентность в обе стороны.

Покажем сначала, что ультаметрика даёт два равных наибольших расстояния. Перенумеруем точки так, чтобы $d(x, y) \geq d(x, z) \geq d(y, z)$. Усиленное неравенство треугольника даёт $d(x, y) \leq \max(d(x, z), d(y, z)) = d(x, z)$. Вместе с $d(x, y) \geq d(x, z)$ получаем $d(x, y) = d(x, z)$ — два наибольших расстояния равны.

Обратно: пусть в каждой тройке два наибольших расстояния равны. Для любых x, y, z расстояние $d(x, z)$ — либо наименьшее из трёх, либо равно наибольшему. В обоих случаях $d(x, z) \leq \max(d(x, y), d(y, z))$, то есть усиленное неравенство выполнено.

Геометрически это значит: в ультаметрике каждый треугольник равнобедренный, причём основание не длиннее равных сторон. На обычной плоскости три точки обычно дают три разных расстояния. Ультаметрика такое запрещает. Именно поэтому ультаметрические данные складываются в чистые вложенные кластеры.

ТЕОРЕМА 5.7 Ультаметрика и иерархия разбиений

На конечном множестве X ультаметрики находятся во взаимно однозначном соответствии с иерархиями — вложенными последовательностями разбиений, в которых каждое разбиение снабжено уровнем.

В одну сторону: ультаметрика d задаёт для каждого порога $\varepsilon \geq 0$ отношение

$$x \equiv_\varepsilon y \leftrightarrow d(x, y) \leq \varepsilon,$$

которое является отношением эквивалентности, и с ростом ε классы только сливаются.

В другую: вложенная последовательность разбиений с уровнями $0 = l_0 < l_1 < \dots < l_k$ задаёт ультраметрику $d(x, y) = l_i$, где i — первый индекс, на котором x и y попадают в один класс.

Доказательство

Докажем эквивалентность в обе стороны.

Сначала ультраметрика в иерархию. Отношение $x \equiv_\varepsilon y \leftrightarrow d(x, y) \leq \varepsilon$ рефлексивно ($d(x, x) = 0 \leq \varepsilon$) и симметрично ($d(x, y) = d(y, x)$). Транзитивность — в точности усиленное неравенство треугольника: если $d(x, y) \leq \varepsilon$ и $d(y, z) \leq \varepsilon$, то $d(x, z) \leq \max(d(x, y), d(y, z)) \leq \varepsilon$. При росте ε классы не расщепляются, а только сливаются, поэтому различные разбиения образуют вложенную цепочку.

Обратно: иерархия в ультраметрику. Зададим $d(x, y)$ уровнем первого разбиения, где x и y оказались в одном классе. Тогда $d(x, y) = 0 \leftrightarrow x = y$ (на нулевом уровне классы — одиночные элементы), и d симметрична по построению. Проверим усиленное неравенство: пусть $m = \max(d(x, y), d(y, z))$. На уровне m точки x и y уже в одном классе, и y с z тоже. Транзитивность отношения эквивалентности этого разбиения даёт, что x и z лежат в одном классе на уровне m , откуда $d(x, z) \leq m$.

Иерархию удобно рисовать корневым деревом: листья — точки, внутренние узлы — классы, а уровень каждого узла — его высота. Деревья сами по себе — предмет главы 9, здесь дерево нужно лишь как рисунок иерархии.

ОПРЕДЕЛЕНИЕ 5.23 Ультраметрическое дерево

Корневое дерево с листьями — элементами X — называется **ультраметрическим деревом**, если каждый внутренний узел снабжён высотой, высоты не убывают от корня к листьям, а все листья имеют высоту 0. Расстояние между двумя листьями — высота их ближайшего общего предка.

По теореме выше расстояние между листьями — ультраметрика, и всякая ультраметрика получается из некоторого дерева. Дендрограмма на рис. 15 и есть такое дерево.

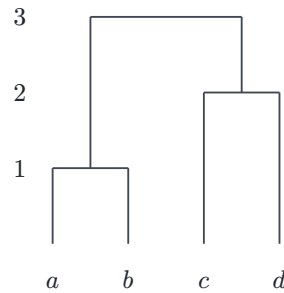


Рис. 15. Дендрограмма четырех точек: a и b сливаются на высоте 1, c и d — на высоте 2, все вместе — на высоте 3.

5.9.2 Иерархическая кластеризация

Обратное построение — теорема, но на практике мы получаем не иерархию, а таблицу попарных несходств, которая ультратметрикой быть не обязана, и дерево всё равно нужно. UPGMA строит его жадными слияниями — тем же приёмом, что алгоритм Крускала (глава 9). Оба опираются на одну и ту же структуру — матроид —, и она разбирается в главе 17.

АЛГОРИТМ UPGMA

1. Каждая точка — кластер из одного элемента на высоте 0.
2. Повторять $n - 1$ раз:
 - Найти два кластера A, B с наименьшим расстоянием $d(A, B)$.
 - Слить их в кластер $C = A \cup B$ с высотой $h(C) = \frac{d(A, B)}{2}$.
 - Для каждого другого кластера D :

$$d(C, D) = \frac{|A| \cdot d(A, D) + |B| \cdot d(B, D)}{|A| + |B|}.$$

3. Расстояние между одноэлементными кластерами — данная матрица несходств.

АЛГОРИТМ WPGMA

Как UPGMA, но расстояние до объединённого кластера — простое среднее, без весов $|A|, |B|$:

$$d(C, D) = \frac{d(A, D) + d(B, D)}{2}.$$

ЛОВУШКА Названия UPGMA и WPGMA обманчивы

UPGMA расшифровывается как *unweighted*, а делит на $|A| + |B|$ — то есть взвешивает по размеру кластера. WPGMA называется *weighted*, а даёт обоим кластерам равный вес независимо от размера. Разгадка в том, что относится к чему: UPGMA

считает каждый элемент ровно один раз, WPGMA — каждый кластер ровно один раз.

Пример UPGMA на четырёх точках

Дана матрица попарных расстояний:

	a	b	c	d
a	0	2	6	6
b	2	0	6	6
c	6	6	0	4
d	6	6	4	0

Наименьшее расстояние — $d(a, b) = 2$, поэтому сливаем a и b в кластер $C_1 = \{a, b\}$ на высоте 1. Расстояния до нового кластера — средние по его элементам, то есть $d(C_1, c) = \frac{6+6}{2} = 6$ и $d(C_1, d) = \frac{6+6}{2} = 6$.

	C_1	c	d
C_1	0	6	6
c	6	0	4
d	6	4	0

Теперь наименьшее — $d(c, d) = 4$, поэтому сливаем c и d в кластер $C_2 = \{c, d\}$ на высоте 2, и расстояние $d(C_2, C_1) = \frac{6+6}{2} = 6$.

	C_1	C_2
C_1	0	6
C_2	6	0

Осталось слить C_1 и C_2 на высоте 3.

Восстановленные расстояния совпадают с исходными: $d(a, c) = 2 \cdot 3 = 6$, $d(a, b) = 2 \cdot 1 = 2$, $d(c, d) = 2 \cdot 2 = 4$. Входная матрица уже была ультرامетрической, поэтому UPGMA вернул её без искажений, а дерево — в точности рис. 15.

УТВЕРЖДЕНИЕ 5.8 UPGMA строит ультраметрическое дерево

UPGMA всегда завершается дендрограммой: высоты слияний не убывают, и построенное дерево ультраметрическое.

Доказательство

Докажем, что высоты слияний не убывают.

На каждом шаге UPGMA выбирает пару кластеров с минимальным расстоянием t . Всякое новое расстояние $d(C, D)$ — взвешенное среднее существующих расстояний $d(A, D)$ и $d(B, D)$, а каждое из них не меньше t (иначе минималь-

ной была бы другая пара). Поэтому $d(C, D) \geq t$ для любого D , и следующий минимум не меньше текущего.

Высота слияния равна половине выбранного расстояния, так что высоты не убывают, а это в точности условие корректной дендрограммы — ультраметрического дерева.

Если входная матрица — точная ультраметрика, UPGMA восстанавливает её без искажений (как в примере). Если нет — строит ультраметрическое дерево, приближающее данные.

Замечание Критерии связи и остовное дерево

UPGMA и WPGMA задают расстояние между кластерами средним, но среднее — не единственный выбор. В *single-linkage* расстояние между кластерами — минимум расстояний между их элементами: $d(A, B) = \min_{a \in A, b \in B} d(a, b)$. Агломерация тогда сливает кластеры в порядке возрастания весов минимального остовного дерева: кластеры *single-linkage* — в точности компоненты связности, которые проходит алгоритм Крускала из главы 9. Та же жадная структура — один матроид, один алгоритм, разные задачи — объясняется в главе 17.

UPGMA несёт дополнительное допущение о равномерной скорости: все листья лежат на одной высоте от корня. Это разумно для данных, где все ветви эволюционируют с одинаковой скоростью, и искажает дерево там, где скорости разные.

5.9.3 Филогенетические деревья

Эволюционное дерево — дендрограмма, в которой листья — ныне живущие виды, а внутренние узлы — их общие предки. Построить такое дерево напрямую нельзя — предки вымерли, и их ДНК до нас не дошла. В распоряжении биолога — только геномы живущих видов, поэтому родство восстанавливают по попарным расстояниям между ними, как в иерархической кластеризации.

ОПРЕДЕЛЕНИЕ 5.24 Филогенетическое дерево

Ультраметрическое дерево, листья которого помечены *таксонами* (видами или группами видов), а высоты внутренних узлов равны времени расхождения соответствующих групп, называется **филогенетическим деревом**.

Расстояние между видами извлекают из сравнения их геномов. У всех сравниваемых видов берут один и тот же участок — консервативный маркер, например ген рибосомальной РНК, который есть у всех живых организмов.²⁹ Последовательности этого участка выравнивают и считают позиции, в которых они различаются.

²⁹C. R. Woese, G. E. Fox. «Phylogenetic Structure of the Prokaryotic Domain: The Primary Kingdoms». PNAS 74, 1977.

Доля различающихся позиций называется *p-расстоянием* (*p-distance*) и служит генетическим расстоянием. За очень долгое время в одной позиции успевает произойти несколько замен, и доля различий перестаёт расти линейно — тогда вводят поправку, например модель Джукса–Кантора.

В основе метода лежит гипотеза *молекулярных часов* (*molecular clock*)³⁰: замены накапливаются с примерно постоянной скоростью. Если за единицу времени на позицию приходится μ замен, то виды, разошедшиеся t единиц времени назад, в среднем различаются на $2\mu t$ замен на позицию — каждая линия накопила свои μt . Расстояние оказывается пропорционально времени расхождения и потому является ультраметрикой — расстояние от общего предка до любого листа одно и то же. Ровно это допущение заложено в UPGMA — все листья лежат на одной высоте от корня — поэтому UPGMA и стал классическим методом построения эволюционных деревьев по генетическим расстояниям.

Классический пример — дерево плацентарных млекопитающих на рис. 16. Человек и шимпанзе разошлись около 6 млн лет назад³¹, тогда как приматы отделились от остальных млекопитающих около 85 млн лет назад.

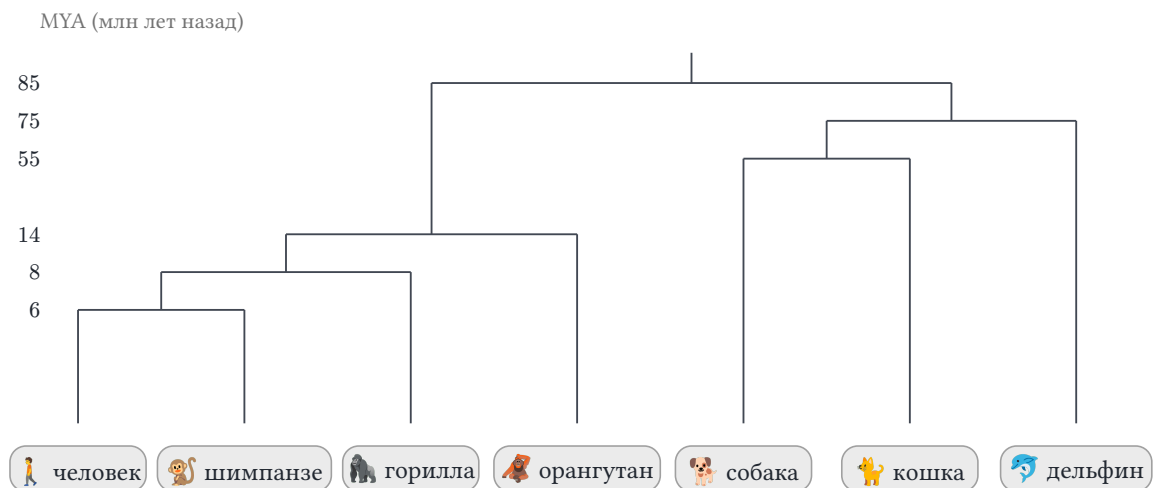


Рис. 16. Филогенетическое дерево млекопитающих.

Когда часы тикают неравномерно, ультраметрическое допущение ломается. UPGMA стягивает быстро эволюционирующие линии друг к другу, и *притяжение длинных ветвей* (*long branch attraction*) искажает дерево. Методы, свободные от часов, справляются лучше. *neighbour-joining*³² строит дерево по расстояниям без допущения о равномерной скорости. Методы максимального правдоподобия и байесовского вывода работают напрямую с последовательностями, а не с одной лишь матрицей расстояний.

³⁰É. Zuckerkandl, L. Pauling. «Molecular Disease, Evolution, and Genic Heterogeneity». In: Horizons in Biochemistry, 1962.

³¹V. M. Sarich, A. C. Wilson. «Immunological Time Scale for Hominid Evolution». Science 158, 1967.

³²N. Saitou, M. Nei. «The Neighbor-joining Method: A New Method for Reconstructing Phylogenetic Trees». Molecular Biology and Evolution 4, 1987.

Наконец, дерево — это модель, а не бесспорный факт. У бактерий гены переносятся горизонтально — между неродственными организмами, а не от предка к потомку — и история перестаёт быть деревом. Конвергентная эволюция и неполная сортировка линий тоже размывают сигнал. Но для большинства эукариот модель дерева работает настолько хорошо, что филогенетические деревья стали основным инструментом систематики.

Проверка Ультраметрика

1. Почему именно усиленное неравенство треугольника делает отношение $x \equiv_{\varepsilon} y$ транзитивным, а обычное — нет?
2. Как по дендрограмме прочитать расстояние между двумя листьями?
3. Чем UPGMA отличается от WPGMA?

§ 5.10 Итоги главы

Связь между объектами записывается парой, и бинарное отношение — это и есть список упорядоченных пар, ничего больше. Декартово произведение собирает все возможные пары, отношение выбирает из них нужные, а связь есть ровно тогда, когда пара принадлежит отношению. Три представления — список пар, матрица смежности, ориентированный граф — оказались тремя взглядами на одну структуру, и каждое свойство читается в каждом из них.

Свойства — рефлексивность, симметричность, антисимметричность, транзитивность — классифицируют связи. Замыкания достраивают отношение до нужного свойства, добавляя ровно те пары, которых не хватает. Композиция склеивает связи в цепочки, степени дают шаги, а транзитивное замыкание собирает все шаги сразу: из связи родителя вырастает предок, и замыкание оказывается просто достижимостью — приёмом, который связывает главу воедино.

Главное наблюдение — как из рефлексивности и транзитивности при разном выборе третьего свойства складываются два класса отношений. Отношение эквивалентности разбивает множество на классы неразличимых объектов. Вложенные разбиения — иерархии — задают и расстояние: ультраметрику, по которой строится дендрограмма. Частичный порядок выстраивает иерархии, где несравнимость означает независимость. Теорема о разбиении показала, что классы и разбиения — одна структура, увиденная с двух сторон, а диаграмма Хассе сжимает порядок до скелета из отношений покрытия, опуская транзитивные рёбра.

Словарь отношений работает в инженерии напрямую: реляционная база — буквально отношение, SQL-запрос — выражение реляционной алгебры. Коллизии хеш-функций — классы одного отношения эквивалентности. Минимизация конечного автомата — склеивание неразличимых состояний. Отношения остаются и мостом к следующей теме: функция — дисциплинированное отношение, в котором на каждый вход приходится ровно один выход.

Проверка Проверьте себя

1. Почему равенство одновременно симметрично и антисимметрично?
2. Что такое транзитивное замыкание и почему одного прохода по всем тройкам недостаточно?
3. Почему классы эквивалентности либо совпадают, либо не пересекаются?
4. Чем частичный порядок отличается от отношения эквивалентности?
5. Зачем нужно рефлексивно-транзитивное замыкание, если уже есть транзитивное?
6. Почему на диаграмме Хассе опускают транзитивные рёбра?

Что дальше?

От отношений общего вида мы переходим к их частному случаю — функциям. Функция — это бинарное отношение с дополнительным ограничением: каждому входу соответствует ровно один выход. Вся теория отношений — свойства, замыкания, композиция — непосредственно применяется к функциям, но с дополнительными гарантиями, которые и составляют предмет следующей главы. В главе 6 мы изучим отображения, их классификацию (инъекция, сюръекция, биекция), композицию и алгебру образов и прообразов — инструментарий, необходимый для всей последующей математики.

§ 5.11 Упражнения

Свойства бинарных отношений.

1. Для отношения $R = \{(1, 2), (2, 3), (3, 1), (1, 1)\}$ на множестве $\{1, 2, 3\}$ проверьте рефлексивность, симметричность, антисимметричность и транзитивность.
2. Постройте отношение на $\{1, 2, 3\}$, которое рефлексивно и симметрично, но не транзитивно. Приведите его матрицу смежности.
3. * Чем антисимметричность отличается от асимметричности? Приведите пример отношения, которое антисимметрично, но не асимметрично. Приведите пример отношения, которое асимметрично.

Замыкания отношений.

4. Постройте рефлексивное, симметричное и транзитивное замыкания отношения $R = \{(1, 2), (2, 3)\}$ на множестве $\{1, 2, 3, 4\}$.
5. Докажите: транзитивное отношение совпадает со своим замыканием: $R^+ = R$. Приведите пример, где $R^+ \neq R$.
6. * Пусть $R = \{(a, b), (b, c), (c, d), (d, b)\}$ на множестве $\{a, b, c, d\}$. Найдите R^+ , R^* . Чем они различаются?

Отношения эквивалентности.

7. Является ли отношение aRb , если $a + b$ чётно, отношением эквивалентности на $\mathbb{Z}$? Если да, опишите классы эквивалентности.
8. Докажите, что пересечение двух отношений эквивалентности на A — отношение эквивалентности. Верно ли это для объединения? Приведите контрпример.
9. * Сколько различных отношений эквивалентности существует на множестве из 3 элементов? Из 4 элементов? (Подсказка: числа Белла.)
10. ** Докажите, что $a \sim b$, когда $a - b \in \mathbb{Q}$, задаёт отношение эквивалентности на $\mathbb{R}$. Опишите мощность каждого класса эквивалентности. Сколько различных классов?

Принцип Дирихле.

11. Покажите, что среди любых 5 точек внутри единичного квадрата найдутся две на расстоянии не более $\frac{1}{\sqrt{2}}$.
12. * Докажите, что в любой компании из 6 человек найдутся либо трое попарно знакомых, либо трое попарно незнакомых (частный случай теоремы Рамсея). Подсказка: зафиксируйте одного человека и рассмотрите его связи с пятью остальными.

ПРОЕКТ Машина отношений: свойства, замыкания, порядки

Бинарное отношение на конечном множестве — это таблица, и по ней отвечают на все вопросы главы: рефлексивно ли, симметрично ли, транзитивно ли, каково замыкание, отношение ли это эквивалентности, частичный ли порядок. Соберите инструмент, который по такой таблице сообщает свойства и строит производные объекты.

① Отношение как матрица

Представьте отношение на $0..n$ булевой матрицей. Реализуйте проверки: рефлексивность (диагональ), симметричность, транзитивность, антисимметричность. Выведите таблицу в читаемом виде.

② Замыкания

Добавьте рефлексивное, симметричное и транзитивное замыкания. Транзитивное — алгоритмом Уоршелла ($O(n^3)$): на шаге k все пути через k становятся рёбрами. Покажите по шагам, как замыкание растёт (промежуточные матрицы).

③ Классы эквивалентности

Проверьте отношение на эквивалентность (рефлексивно + симметрично + транзитивно). Выведите классы (разбиение) через union-find по рёбрам. Сверьте с числами Белла: отношений эквивалентности на n элементах — 1, 2, 5, 15, 52, 203.

④ Частичные порядки

Проверьте частичный порядок (рефлексивно + антисимметрично + транзитивно). Подсчитайте линейные расширения поиском с отсечением: ставим

элемент, только когда все его предшественники уже поставлены. Вычислите ширину (наибольшую антицепь) для малых n . Сверьте: у цепочки из n элементов ровно одно линейное расширение, а у булевой решётки B_2 — два.

⑤ *Диаграмма Хассе*

По частичному порядку постройте транзитивную редукцию — минимальное отношение с тем же транзитивным замыканием. Нарисуйте диаграмму Хассе (SVG или текст): рёбра только к непосредственным преемникам, без петель и транзитивных дуг. Сверьте: у булевой решётки B_2 диаграмма — ромб, у цепочки из n элементов — прямая линия.

Итог: инструмент, который по таблице отношения сообщает все его свойства, строит рефлексивное/симметричное/транзитивное замыкания, выделяет классы эквивалентности и перечисляет линейные расширения частичного порядка.

VI

Функции

Функция, равная единице на рациональных числах и нулю на иррациональных, не записывается никакой формулой в привычном смысле — ни дробью, ни радикалом, ни рядом. Это не помеха: каждому входу она отвечает ровно одним выходом и потому честно является функцией — соответствием, а не записью.

Соответствие простое: каждому элементу области определения — ровно один элемент выхода. Два условия, и оба несут инженерный смысл. «Каждому» — у правила нет дыр: программа обязана ответить на любой допустимый вход. «Ровно один» — правило не раздваивается: два разных ответа на один вопрос обесценивают и проверку пароля, и декодирование символов, и поиск по хеш-таблице. Отношения из главы 5 связывали входы с выходами произвольно: одному — много, другому — ничего. Одно слово «ровно» превращает рыхлую сеть в механизм с предсказуемым поведением.

Однозначность — условие полезности, и инженерия на ней стоит. Сериализация и десериализация — пара взаимно обратных функций: объект превращается в представление и возвращается обратно. Кодирование данных (ASCII, Base64) держится на однозначности в обе стороны. Компилятор — функция из текстов в программы, и его крах на странном входе — дыра в области определения.

Наивная картина «функция как формула» разбивается и о вычислении. Функция, не определённая на части входов, — честная модель программы, которая может не завершиться. Функцию определяет то, что она делает с каждым входом, а не то, можно ли записать правило формулой.

Когда по выходу функции восстанавливается её вход — и какими свойствами функция должна для этого обладать?

ИСТОРИЯ От формулы к вычислимому отображению

Иоганн Петер Густав Лежён Дирихле в работе 1837 года дал первое современное определение функции как произвольного соответствия: каждому x сопоставляется ровно один y , без требования выразимости формулой. Это был разрыв с XVIII веком, где функция мыслилась исключительно как аналитическое выражение (Эйлер, Лагранж). Дирихле привёл знаменитый пример: функция, равная 1 на рациональных числах и 0 на иррациональных, — никакой «формулой» в традиционном смысле она не задаётся, но является функцией по новому опре-

делению. За примером закрепилось прозвище «монстр Дирихле» (Dirichlet's monster): он нигде не непрерывен и не интегрируем по Риману.

Готлоб Фреге в работе «Funktion und Begriff»³³ предложил философский анализ понятия функции, независимый от теории множеств. Функция, по Фреге, «ненасыщена» (ungesättigt) — она содержит пустое место, которое заполняется аргументом. Выражение $2 \cdot x + 1$ неполно, пока x не заменено конкретным числом. Выражение $2 \cdot 3 + 1$ полно и обозначает объект.

Эта философская метафора — функция как форма с дыркой — предвосхитила лямбда-абстракцию в программировании: $\lambda x. 2x + 1$ — это в точности «ненасыщенное» выражение Фреге, ожидающее подстановки аргумента. Фреге строил логический фундамент для всей математики, и его анализ функции был частью этого проекта.

Алонзо Чёрч в 1936 году формализовал понятие вычислимой функции через λ -исчисление — формальную систему, в которой вычисление сводится к подстановке и редукции термов. В том же году Алан Тьюринг предложил альтернативную модель — машину Тьюринга — и независимо доказал неразрешимость проблемы разрешения (Entscheidungsproblem) Гильберта. Вскоре была установлена эквивалентность λ -исчисления, машин Тьюринга и общерекурсивных функций Гёделя: все три формализма определяют один и тот же класс вычислимых функций.

Тезис Чёрча–Тьюринга утверждает, что этот класс исчерпывает интуитивное понятие вычислимости. Это эмпирическая гипотеза: все попытки построить более сильную модель вычислений — включая квантовые вычисления (по классу вычислимых функций) — до сих пор приводили лишь к эквивалентным формализмам. Понятие функции эволюционировало от «формулы» до «вычислимого отображения» почти ровно за сто лет.

ОБЗОР ГЛАВЫ

Функция — дисциплинированное отношение: каждому входу соответствует ровно один выход. Разберём её части — область определения и кодомен — и связанные с ними образ и прообраз. Разграничим тотальные и частичные функции и увидим, что вторые — честная модель вычислений, которые могут расходиться. Классификация по обратимости — инъекция, сюръекция, биекция — даст каждому типу свою обратную: левую, правую или двустороннюю. Композиция склеит функции в цепочки, а алгебра образов и прообразов покажет, что сохраняется при переходе туда и обратно. Затем возьмём в руки инструменты анализа: асимптотические обозначения, характеристическую функцию, пол и потолок, нотацию Айверсона и

³³G. Frege. «Funktion und Begriff». 1891.

лямбда-абстракцию. Они понадобятся не только в этой главе: от комбинаторики до теории сложности мы будем говорить языком функций.

После этой главы вы сможете:

1. описывать функцию как отношение и проверять, что каждому входу соответствует ровно один выход;
2. вычислять образ, прообраз и композицию и понимать, что они сохраняют, а что теряют;
3. классифицировать функции на инъекции, сюръекции и биекции и строить обратные функции там, где они существуют;
4. различать тотальные и частичные функции и моделировать ими программы, которые могут не завершиться;
5. сравнивать скорость роста асимптотическими обозначениями и пользоваться характеристической функцией, полом и потолком.

§ 6.1 Определения и базовые понятия

6.1.1 Функция как отношение

ОПРЕДЕЛЕНИЕ 6.1 Функция

Бинарное отношение $f \subseteq A \times B$ называется **функцией**, если выполнены два условия.

- Тотальность: $\forall a \in A, \exists b \in B : (a, b) \in f$.
- Однозначность: $(a, b_1), (a, b_2) \in f \rightarrow b_1 = b_2$.

Такая функция обозначается $f : A \rightarrow B$.

Пример Функция и не-функция

Пусть $A = \{1, 2\}, B = \{a, b\}$.

1. $f_1 = \{(1, a), (2, b)\}$ — функция: каждому аргументу из A сопоставлен ровно один элемент B .
2. $f_2 = \{(1, a), (1, b)\}$ — не функция: аргументу 1 сопоставлены два разных значения.
3. $f_3 = \{(1, a)\}$ — функция над $A = \{1\}$, но не над $A = \{1, 2\}$: аргумент 2 не имеет образа.

Функция задана — теперь нужно назвать её составные части. Множество допустимых входов, множество разрешённых выходов, множество реально достигаемых значений — у каждого из этих множеств есть точное имя.

ОПРЕДЕЛЕНИЕ 6.2 Область определения

Множество допустимых входов A функции $f : A \rightarrow B$ называется **областью определения** (англ. domain) и обозначается $\text{Dom}(f) = A$.

ОПРЕДЕЛЕНИЕ 6.3 Кодомен

Множество разрешённых выходов B называется **кодоменом** функции $f : A \rightarrow B$. Обозначается $\text{Cod}(f) = B$.

Область определения и кодомен — два множества, между которыми работает отображение. Рис. 17 показывает их в одной картине: слева A — область определения, справа B — кодомен, а стрелка f между ними — само отображение.

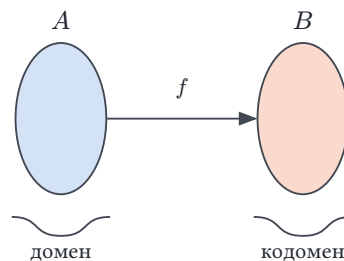


Рис. 17. Область определения и кодомен функции $f : A \rightarrow B$.

Функция может «обещать» кодомен целиком, а «производить» лишь его подмножество.

ОПРЕДЕЛЕНИЕ 6.4 Образ элемента

Образом элемента $x \in A$ называется его значение $f(x)$.

ОПРЕДЕЛЕНИЕ 6.5 Образ множества

Образом множества $X \subseteq A$ называется множество значений всех его элементов:

$$f(X) = \{f(x) \mid x \in X\}.$$

Запись $f(X)$ — то же имя f , что и в $f(x)$, но применённое ко множеству.

ОПРЕДЕЛЕНИЕ 6.6 Область значений

Образ всей области определения, $f(A)$, называется **областью значений** функции (англ. range).

Образ смотрит по стрелке: из области определения в кодомен. Прообраз смотрит против стрелки: он собирает все входы, отображающиеся в заданные выходы.

ОПРЕДЕЛЕНИЕ 6.7 Прообраз множества

Прообразом множества $Y \subseteq B$ называется множество всех элементов, отображающихся в него:

$$f^{-1}(Y) = \{a \in A \mid f(a) \in Y\}.$$

ОПРЕДЕЛЕНИЕ 6.8 Прообраз элемента

Прообразом элемента $b \in B$ называется множество всех элементов, отображающихся в него:

$$f^{-1}(b) = \{a \in A \mid f(a) = b\}.$$

Прообраз элемента не обязательно одноэлементный.

Пример Область определения, кодомен, образ и прообраз

Пусть $f : \{1, 2, 3\} \rightarrow \{a, b\}$, $f = \{(1, a), (2, b), (3, a)\}$.

- Область определения: $\{1, 2, 3\}$.
- Кодомен: $\{a, b\}$.
- Образ: $f(\{1, 2, 3\}) = \{a, b\}$ — образ совпал со всем кодоменом.
- Прообраз a : $f^{-1}(a) = \{1, 3\}$ (два элемента).
- Прообраз b : $f^{-1}(b) = \{2\}$ (один элемент).
- Образ подмножества: $f(\{1, 2\}) = \{a, b\}$ (те же значения, но только для аргументов 1 и 2).

6.1.2 Прообраз и обратная функция

Запись $f^{-1}(b) = \{a \mid f(a) = b\}$ обозначает **прообраз** — множество всех таких элементов. За тем же значком f^{-1} прячется другая сущность — **обратная функция** $f^{-1} : B \rightarrow A$, которая существует далеко не у каждой функции (раздел о биекциях ниже). Смешивать их нельзя: $f^{-1}(4)$ как прообраз — множество, как значение обратной функции — один элемент.

Контрпример: $f(x) = x^2$ на $\mathbb{R}$. $f^{-1}(4) = \{-2, 2\}$ — прообраз из двух элементов, и определён он всегда. Обратной функции у f нет: значения -2 и 2 склеились в один выход, и по выходу вход не восстановить. Запись $f^{-1}(4)$ в смысле обратной функции здесь некорректна.

Если же каждый элемент кодомена достигается ровно один раз, прообраз одноэлементного множества $\{b\}$ состоит из одного элемента $a = f^{-1}(b)$. В этом случае (и только в этом) обозначение f^{-1} для прообраза и для обратной функции согласовано.

6.1.3 Тотальные и частичные функции

Не каждое правило сопоставления определено для всех элементов области определения. Это различие — между «определено всегда» и «определено иногда» — важно для теории вычислений.

ОПРЕДЕЛЕНИЕ 6.9 Тотальная функция

Функция, определённая для каждого элемента $a \in A$, называется **тотальной** и обозначается $f : A \rightarrow B$. Область определения тотальной функции совпадает со всем A .

ОПРЕДЕЛЕНИЕ 6.10 Частичная функция

Функция, определённая лишь на некотором подмножестве A , называется **частичной** и обозначается $f : A \mapsto B$.

Примечание Область определения частичной функции

Для частичной функции $f : A \mapsto B$ множество A перечисляет потенциальные входы, а область определения — те входы, на которых функция реально вычисляется. При переходе к частичному случаю термин сужается: область определения может быть собственным подмножеством A .

Пример Частичная функция

Функция $f(x) = \frac{1}{x}$ частична на $\mathbb{R}$: она не определена в точке $x = 0$ (деление на ноль). Однако f тотальна на суженной области $\mathbb{R} \setminus \{0\}$. Выбор области определения меняет статус функции с частичной на тотальную.

Примечание Частичность в языках программирования

Частичную функцию $f : A \mapsto B$ языки программирования изображают тотальной функцией с расширенным кодом: к B добавляется специальный элемент «ничего». В Rust для этого служат типы `Option` и `Result`, в Haskell — `Maybe`, в Java — `Optional`. Система типов заставляет вызывающий код разобрать оба случая: значение пришло или ответа нет. Исключения дают ту же частичность без поддержки со стороны типа: пропущенная обработка оборачивается аварийным завершением, которого сигнатура функции не показывает.

6.1.4 Частичные функции, тотальность и вычислимость

Частичная функция моделирует вычисление, которое на некоторых входах расходится: бросает исключение, уходит в бесконечный цикл или аварийно завершается. Тотальная функция моделирует вычисление, которое всегда завершается и возвращает значение.

В главе 26 мы встретим два связанных, но различных вопроса:

- Проблема остановки: можно ли по описанию программы определить, завершится ли она на конкретном входе?
- Проблема тотальности: можно ли по описанию программы определить, завершится ли она на всех входах?

Ответ на оба вопроса — нет. Остановка сводится к тотальности: по паре (M, x) строится программа, тотальная ровно тогда, когда M останавливается на x .³⁴ Но это разные вопросы: остановка спрашивает про один конкретный вход, тотальность — про все сразу.

Доказательство тотальности программы — задача верификации. Системы вроде Coq³⁵ и Agda³⁶ требуют доказательства тотальности для каждой определённой в них функции: компилятор отвергает код, для которого не доказана завершимость. Требование тотальности — следствие логики: в соответствии Карри–Говарда незавершающаяся функция соответствовала бы некорректному доказательству (из ложной посылки можно вывести что угодно).



Частичные функции — честная модель вычислений, которые могут расходиться. Но классификация по обратимости — инъекции, сюръекции, биекции — относится к тотальным функциям: свойства определены для всех элементов области, и частичный случай в неё не вписывается.

6.1.5 График функции

ОПРЕДЕЛЕНИЕ 6.11 График функции

Графиком функции $f : A \rightarrow B$ называется множество упорядоченных пар:

$$\{(a, f(a)) \mid a \in A\},$$

то есть сама функция f , рассматриваемая как бинарное отношение.

Пример График функции

Для $f : \{1, 2, 3\} \rightarrow \mathbb{N}$, $f(x) = x^2$ график — это множество

$$\{(1, 1), (2, 4), (3, 9)\}.$$

Не каждое отношение является функцией. Отношение $R \subseteq A \times B$ — функция тогда и только тогда, когда каждый элемент области определения $a \in A$ появляется ровно в одной паре. Для всякого a найдётся b с $(a, b) \in R$, и такое b единственно.

³⁴A. Turing. «On Computable Numbers, with an Application to the Entscheidungsproblem». Proceedings of the London Mathematical Society, 1936–1937.

³⁵Coq proof assistant, INRIA, <https://coq.inria.fr>.

³⁶U. Norell. «Towards a Practical Programming Language Based on Dependent Type Theory». PhD thesis, Chalmers, 2007.

Уравнение $x^2 + y^2 = 1$ задаёт окружность и не задаёт функцию. В точке $x = 0$ допустимы оба значения $y = 1, y = -1$, и формула не выбирает между ними.

Формула — ещё не функция: чтобы стать функцией, она обязана для каждого входа оставить ровно один выход. «Монстр» Дирихле был функцией без формулы. Окружность — формула без функции.

Проверка Определения и базовые понятия

1. Почему $f(x) = \frac{1}{x}$ частична на $\mathbb{R}$, но тотальна на $\mathbb{R} \setminus \{0\}$?
2. Чем прямой образ $f(X)$ отличается от прообраза $f^{-1}(Y)$?

§ 6.2 Инъекция

ОПРЕДЕЛЕНИЕ 6.12 Инъекция

Функция f называется **инъективной**, если различные входы отображаются в различные выходы:

$$f(a_1) = f(a_2) \rightarrow a_1 = a_2.$$

Эквивалентно: $a_1 \neq a_2 \rightarrow f(a_1) \neq f(a_2)$.

ЛОВУШКА Функция $\neq$ инъекция

«Каждому входу ровно один выход» — условие однозначности, а не инъективности. Инъективность требует большего: $a_1 \neq a_2 \rightarrow f(a_1) \neq f(a_2)$.

Функция $f(x) = x^2$ корректна на $\mathbb{R}$, но не инъективна. Контрпример: $f(-1) = f(1) = 1$.

Для конечных множеств инъективность возможна лишь при $|A| \leq |B|$.

Пример Инъективная функция

$f(x) = 2x + 1$ инъективна: из равенства $f(a) = f(b)$ следует $2a + 1 = 2b + 1$, то есть $a = b$. $g(x) = x^2$ не инъективна: $g(2) = g(-2) = 4$ — два разных входа дают один выход. По значению 4 вход не восстановить: на входе могло оказаться любое из чисел $2, -2$.

Сужение области определения может вернуть инъективность. Функция $g(x) = x^2$ не инъективна на $\mathbb{R}$, но инъективна на множестве неотрицательных чисел. Из пары аргументов $a, -a$ с одним квадратом остаётся ровно один. Ограничение выбора — цена за однозначное восстановление входа по выходу.

Пример Конечная инъекция

$f : \{1, 2, 3\} \rightarrow \{a, b, c, d\}$, $f = \{(1, a), (2, b), (3, c)\}$ инъективна: все три выхода различны. Но кодомен покрыт не весь: при $|A| < |B|$ какой-то элемент остаётся без прообраза, здесь это d .

Инъективность f означает, что по значению $f(a)$ можно однозначно восстановить аргумент a . Разные входы дают разные выходы, и никакой выход не «принадлежит» двум разным входам одновременно. Это свойство эквивалентно существованию левой обратной функции: функции, которая откатывает действие f , возвращая исходный аргумент.

ТЕОРЕМА 6.1 Левая обратная

При $A \neq \emptyset$: f инъективна тогда и только тогда, когда существует $g : B \rightarrow A$, такая что

$$g \circ f = \text{id}_A.$$

Доказательство

Докажем эквивалентность в обе стороны.

Покажем сначала, что из инъективности f следует существование левой обратной. Построим функцию g : для каждого элемента образа существует ровно один прообраз в области. Для $b \in f(A)$ положим $g(b) = a$, где $f(a) = b$ (такой a единственен в силу инъективности). Для $b \notin f(A)$ положим $g(b) = a_0$ (произвольный фиксированный элемент области A). Тогда $g(f(a)) = a$ для всех $a \in A$. То есть $g \circ f = \text{id}_A$.

Обратно: пусть $g \circ f = \text{id}_A$. Предположим $f(a_1) = f(a_2)$. Тогда $a_1 = g(f(a_1)) = g(f(a_2)) = a_2$. Следовательно, f инъективна.

Пример Левая обратная для конечной функции

Пусть $A = \{1, 2, 3\}$, и функция $f : A \rightarrow \{a, b, c, d\}$ задана образами $f(1) = a$, $f(2) = b$, $f(3) = c$. Функция инъективна: все три образа различны. По доказательству теоремы строим левую обратную $g : \{a, b, c, d\} \rightarrow A$: на образе полагаем $g(a) = 1$, $g(b) = 2$, $g(c) = 3$, а для d , не лежащего в образе, берём произвольный элемент области — $g(d) = 1$.

Проверим, что $g(f(1)) = g(a) = 1$, аналогично $g(f(2)) = 2$, $g(f(3)) = 3$, то есть $g \circ f = \text{id}_A$. Функция g не единственна и не обязана быть инъективной: значение на d можно выбрать любым из A . Двусторонняя обратная функция у f отсутствует: элемент d не имеет прообраза.

Пример Коллизии хешей

Пусть хеш-функция h отображает 1000 студентов в 365 дней рождения. По принципу Дирихле хотя бы два студента получают один день — коллизия неизбежна, потому что $|A| > |B|$. Инъективность здесь невозможна.

Кодирование символов (ASCII, Base64) инъективно — потому и возможно однозначное декодирование. Криптографические хеш-функции, напротив, не инъективны по принципу Дирихле.

Инъекция гарантирует, что разные входы не склеиваются: каждый вход различим по своему выходу. Но она ничего не говорит о том, все ли потенциальные выходы достижимы. Функция $f(x) = e^x$ инъективна, но её образ — положительная полуось, а не вся вещественная прямая. Сюръекция — гарантия того, что каждый элемент кодомена достижим из какого-либо входа.

§ 6.3 Сюръекция

ОПРЕДЕЛЕНИЕ 6.13 Сюръекция

Функция f называется **сюръективной**, если каждый элемент кодомена B достигается:

$$\forall b \in B, \exists a \in A : f(a) = b.$$

Эквивалентно: $f(A) = B$.

Сюръективную функцию называют функцией *из A на B* . Предлог «на» здесь несёт математическое содержание: образ покрывает весь кодомен, а не вкладывается в него. В английских источниках тот же смысл передаёт *onto*, а для произвольной функции — *into*.

Для конечных множеств сюръективность возможна лишь при $|A| \geq |B|$. Сюръективность — свойство всей тройки: область определения, правило, кодомен.

Замечание Почему кодомен — часть функции

Одна и та же формула e^x задаёт разные функции при разных кодоменах. Как отображение $\mathbb{R} \rightarrow \mathbb{R}$ экспонента не сюръективна: ноль и отрицательные числа остаются без прообраза. Как отображение $\mathbb{R} \rightarrow \mathbb{R}^+$ она сюръективна и даже биективна: каждый положительный выход достигается ровно с одного входа. Отсюда следствие: вопрос «сюръективна ли данная функция» без указания кодомена не имеет ответа.

Пример Сюръективная функция

$f(x) = x^3$ сюръективна на $\mathbb{R}$: для любого y число $\sqrt[3]{y}$ существует и $f(\sqrt[3]{y}) = y$. Конечный случай: $f : \{1, 2, 3, 4\} \rightarrow \{a, b\}$, где $f = \{(1, a), (2, a), (3, b), (4, b)\}$. Сюръективность выполнена: оба элемента кодомена достигнуты. Инъективности при этом нет: $f(1) = f(2) = a$ — разные входы склеились в один выход.

Сюръективность f означает, что каждый элемент кодомена достижим. Для любого b существует хотя бы один a с $f(a) = b$. Это свойство эквивалентно существованию правой обратной функции — функции, которая по значению b подбирает какой-нибудь прообраз. Для конечных множеств такая правая обратная строится явно. Для бесконечных её существование опирается на аксиому выбора.

ТЕОРЕМА 6.2 Правая обратная

f сюръективна тогда и только тогда, когда существует $g : B \rightarrow A$, такая что

$$f \circ g = \text{id}_B.$$

Доказательство

Докажем эквивалентность в обе стороны.

Покажем сначала, что из сюръективности f следует существование правой обратной. Для каждого $b \in B$ выберем произвольный $a \in A$, такой что $f(a) = b$. Такой a существует по определению сюръективности. Положим $g(b) = a$. Тогда $f(g(b)) = f(a) = b$ для всех $b \in B$. То есть $f \circ g = \text{id}_B$.

Для бесконечных множеств выбор прообраза для каждого b требует аксиомы выбора. Для конечных множеств построение явное.

Обратно: пусть $f \circ g = \text{id}_B$. Для любого $b \in B$ имеем $f(g(b)) = b$. То есть $g(b)$ является прообразом b . Следовательно, f сюръективна.

Сюръективность — не только абстрактное свойство: она прямо выражает требование достижимости в прикладных задачах.

- Генератор перестановок должен уметь выдать любую из $n!$ перестановок — иначе некоторые конфигурации недостижимы.
- Десериализация JSON должна восстанавливать объект по любому корректному JSON-представлению.
- Пара сериализация–десериализация образует биекцию между объектами и их каноническими представлениями.
- Переход «состояние» $\rightarrow$ «следующее состояние» генератора псевдослучайных чисел (ГПСЧ) должен покрывать всё пространство состояний.

Каждое из этих требований — в точности сюръективность: каждый элемент кодомена должен быть достижим.

Сюръекция гарантирует покрытие всей кодомена, но допускает склеивание: разным входам может соответствовать один выход. Функция $f(x) = x^3 - x$ сюръективна на $\mathbb{R}$. Но $f(-1) = f(0) = f(1) = 0$ — три входа дают один и тот же выход. Инъекция и сюръекция независимы: возможны четыре комбинации. Инъекция без сюръекции оставляет элементы кодомена без прообраза. Сюръекция без инъекции создаёт коллизии. Биекция — это обе одновременно: каждый выход достижим, и каждому выходу соответствует ровно один вход. Только для биекции существует двусторонняя обратная функция. Рис. 18 иллюстрирует различие между инъекцией, сюръекцией и биекцией.

§ 6.4 Биекция

ОПРЕДЕЛЕНИЕ 6.14 Биекция

Функция f называется **биективной**, если она одновременно инъективна и сюръективна.

Биекция — единственный тип функции, для которого существует полноценная обратная функция. Инъекция имеет лишь левую обратную (определённую на образе). Сюръекция — правую (неоднозначную). Только биекция даёт двустороннюю обратную, которая сама является функцией в строгом смысле.

Равенства $f \circ f^{-1} = \text{id}_B$, $f^{-1} \circ f = \text{id}_A$ — это смысл слова «обратная». Обратная функция гасит действие f : применили f , затем f^{-1} — вернулись туда, откуда начали. Это как умножить число на 3, а потом разделить на 3.

ТЕОРЕМА 6.3 Обратная функция

f биективна тогда и только тогда, когда существует $f^{-1} : B \rightarrow A$, такая что

$$\begin{aligned} f \circ f^{-1} &= \text{id}_B \\ f^{-1} \circ f &= \text{id}_A. \end{aligned}$$

Для конечных множеств биекция возможна лишь при $|A| = |B|$.

Доказательство

Докажем эквивалентность в обе стороны.

Покажем сначала, что из биективности f следует существование двусторонней обратной. Для каждого $b \in B$ существует единственный $a \in A$, такой что $f(a) = b$. Существование — из сюръективности, единственность — из инъективности. Определим $f^{-1}(b) = a$. Тогда $f(f^{-1}(b)) = b$, $f^{-1}(f(a)) = a$ по построению.

Обратно: если такая f^{-1} существует, то из $f^{-1} \circ f = \text{id}_A$ следует инъективность f (теорема о левой обратной). Из $f \circ f^{-1} = \text{id}_B$ следует сюръективность f (теорема о правой обратной). Следовательно, f биективна.

Пример Кодирование Base64

Base64 переводит произвольную последовательность байтов в текст над алфавитом из 64 печатных символов. Входные байты нарезаются на группы по три, каждая группа из 24 битов — на четыре куса по шесть битов, и каждый кусок заменяется одним символом алфавита. Кодирование тотально: на вход годится любая последовательность байтов. Кодирование инъективно: разрезание и замена однозначны, поэтому по тексту кода байты собираются обратно без вариантов. Сюръективным кодирование не будет: кодами служат не все тексты над алфавитом, а лишь те, у которых длина кратна четырём, а хвост выровнен знаками равенства. Декодирование определено на образе: корректный код возвращает исходные байты, а строка вне образа отвергается как ошибка. Перед нами теорема о левой обратной в готовом виде: на образе декодер действует как построенная в доказательстве функция g , вне образа он доопределён сообщением об ошибке.

Доказать, что функция биективна — значит доказать два независимых утверждения: инъективность (разные входы дают разные выходы) и сюръективность (любой заявленный выход достижим). Разберём на конкретном примере.

Пример Инъективность и сюръективность: $f(x) = 2x + 1$

Рассмотрим функцию $f : \mathbb{Z} \rightarrow \mathbb{Z}, f(x) = 2x + 1$.

Инъективность: цепочка импликаций

$$\begin{aligned} f(a) = f(b) &\Rightarrow 2a + 1 = 2b + 1 \\ &\Rightarrow 2a = 2b \\ &\Rightarrow a = b. \end{aligned}$$

Сюръективность: для данного $y \in \mathbb{Z}$ решаем уравнение

$$2x + 1 = y \Rightarrow x = \frac{y - 1}{2}.$$

Если y нечётно, то $y = 2k + 1$ при некотором целом k . Тогда $x = k \in \mathbb{Z}$. Если y чётно, целого x не существует. Если объявить кодоменом множество нечётных чисел, функция становится биективной. На $\mathbb{Z}$ она инъективна, но не сюръективна (выходы нечётны).

$f : \mathbb{Z} \rightarrow \{2k + 1 \mid k \in \mathbb{Z}\}$ — биекция. Нахождение обратной: $f^{-1}(y) = \frac{y-1}{2}$.

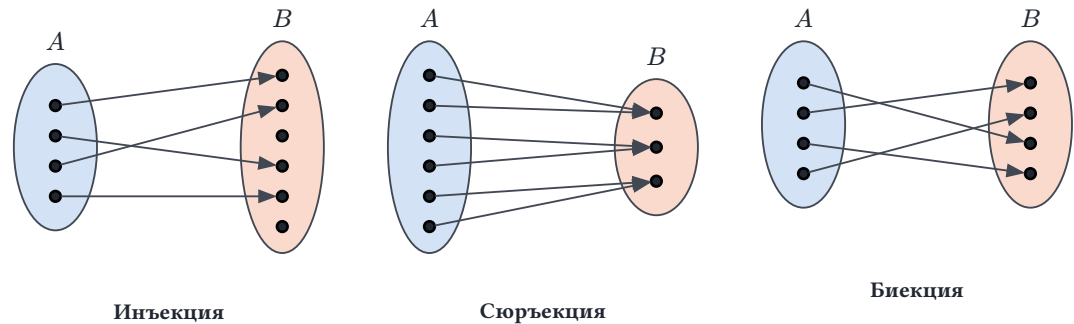


Рис. 18. Инъекция, сюръекция и биекция.

Биекция, сохраняющая операции, — это **изоморфизм**, с которым мы встретились в главе 4 (алгебры подмножеств и логических значений изоморфны). Изоморфизм означает, что две структуры устроены одинаково: можно переименовать элементы одной в элементы другой так, что все операции совпадут.

§ 6.5 Критерии мощности

УТВЕРЖДЕНИЕ 6.4 Классификация по мощности			
Тип	Условие	Мощность	Обратная
Инъекция	$f(a_1) = f(a_2) \rightarrow a_1 = a_2$	$ A \leq B $	Левая обратная $g \circ f = \text{id}_A$
Сюръекция	$\forall b \exists a : f(a) = b$	$ A \geq B $	Правая обратная $f \circ g = \text{id}_B$
Биекция	Инъективна + сюръективна	$ A = B $	Двусторонняя обратная f^{-1}



Для конечных множеств одинакового размера любая инъекция автоматически биективна. Для бесконечных — нет: $f(n) = 2n$ на $\mathbb{N}$ инъективна, но не сюръективна.

Уже здесь прячется сюрприз, который станет главным в следующей главе. Как отображение $\mathbb{N} \rightarrow \mathbb{N}$ формула $f(n) = 2n$ не сюръективна: нечётные числа остаются без прообраза. Как отображение $\mathbb{N} \rightarrow \{0, 2, 4, 6, \dots\}$ та же формула — биекция. Собственное подмножество бесконечного множества оказалось равномощным целому: для бесконечности «часть» и «целое» ведут себя не так, как для конечных множеств.

§ 6.6 Композиция функций

Инъекция, сюръекция, биекция — это классификация одной функции. Но вычисление редко состоит из одного шага. Мы применяем функции последовательно: выход одной становится входом другой.

Композиция — операция склеивания функций в цепочки. Её алгебраические свойства — ассоциативность, но не коммутативность — зеркально отражают свойства последовательного выполнения операторов в программе. Порядок имеет значение: сначала открыть файл, потом читать из него — не наоборот.

ОПРЕДЕЛЕНИЕ 6.15 Композиция

Функция, заданная равенством

$$(g \circ f)(a) = g(f(a))$$

для $f : A \rightarrow B, g : B \rightarrow C$, называется **композицией** g и f , обозначается $g \circ f$.

УТВЕРЖДЕНИЕ 6.5 Свойства композиции

Композиция ассоциативна: $h \circ (g \circ f) = (h \circ g) \circ f$. Она не коммутативна: $g \circ f \neq f \circ g$ в общем случае.

Набросок доказательства

Ассоциативность проверяется подстановкой: для любого a обе стороны равны $h(g(f(a)))$. Не-коммутативность показывает пример ниже.

Пример Композиция конкретных функций

Пусть $f : \mathbb{R} \rightarrow \mathbb{R}, f(x) = x + 1, g : \mathbb{R} \rightarrow \mathbb{R}, g(x) = x^2$. Тогда $(g \circ f)(x) = g(f(x)) = g(x + 1) = (x + 1)^2$. А $(f \circ g)(x) = f(g(x)) = f(x^2) = x^2 + 1$. Результаты разные: $(x + 1)^2 \neq x^2 + 1$ — композиция не коммутативна.

Что касается свойств f и g : f биективна (строго возрастает). Функция g не инъективна. Контрпример: $g(-2) = g(2) = 4$. g не сюръективна (не принимает отрицательных значений).

Тогда $g \circ f$ не инъективна и не сюръективна. Инъективность композиции требует инъективности f и инъективности g лишь на образе f . Но $g(x) = x^2$ не инъективна даже на образе $f = \mathbb{R}$.

Композиция сохраняет структурные свойства звеньев: цепочка инъективных функций инъективна, цепочка сюръективных функций сюръективна, цепочка биекций биективна.

ТЕОРЕМА 6.6 Композиция сохраняет тип функции

Если f, g инъективны, то композиция $g \circ f$ инъективна. Если f, g сюръективны, то композиция $g \circ f$ сюръективна. Если f, g биективны, то композиция $g \circ f$ биективна.

Доказательство

Докажем по очереди сохранение инъективности и сюръективности. Утверждение о биективности из них следует.

Сначала докажем сохранение инъективности. Инъективность f и g означает, что $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$ и аналогично для g . Тогда

$$\begin{aligned}(g \circ f)(x_1) &= (g \circ f)(x_2) \Rightarrow g(f(x_1)) = g(f(x_2)) \\ &\Rightarrow f(x_1) = f(x_2) \\ &\Rightarrow x_1 = x_2.\end{aligned}$$

Второй переход использует инъективность g , третий — инъективность f .

Осталось показать сюръективность. Любой элемент z кодомена достижим как $g(y)$. Элемент y достижим как $f(x)$. Значит, z достижим как $(g \circ f)(x)$.

Примечание

Цепочка сохраняет свойства каждого звена — три утверждения выше суть одно. Именно это позволяет строить сложные преобразования из простых, проверяя корректность каждого шага по отдельности. В этом состоит принцип модульной верификации: если каждый модуль корректен относительно своей спецификации, то и композиция модулей корректна.³⁷

Если композиция биективна, то и обратная к ней существует. Вычисляется она по простому правилу — но с поворотом порядка: размотка цепочки идёт в обратную сторону.

ТЕОРЕМА 6.7 Обратная композиции

Пусть $f : A \rightarrow B, g : B \rightarrow C$ биективны. Тогда

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1}.$$

Доказательство

Докажем, что обратная к композиции равна композиции обратных.

³⁷B. Liskov, J. Guttag. «Program Development in Java: Abstraction, Specification, and Object-Oriented Design». 2000, глава о композиции спецификаций.

Пусть f, g биективны. Для любого c : $(g \circ f)^{-1}(c) = a \leftrightarrow (g \circ f)(a) = c$. То есть $g(f(a)) = c$. Это равносильно $f(a) = g^{-1}(c)$. Значит $a = f^{-1}(g^{-1}(c)) = (f^{-1} \circ g^{-1})(c)$.

Порядок меняется, потому что разматывать цепочку приходится с конца: последнее действие отменяется первым, сначала g^{-1} , затем f^{-1} . Функциональные конвейеры $(x \mapsto f \mapsto g \mapsto h)$ — это композиция, записанная слева направо.

Образ может терять информацию (два разных элемента сливаются в один), прообраз — *никогда*. Различие между образом и прообразом, однажды усвоенное, делает все последующие тождества прямым следствием определений.

Замечание Две функции в одной

Канторова пара — биекция $J : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, нумерующая каждую пару натуральных чисел одним числом. Её диагональное устройство разбирается в главе 7, здесь достаточно факта биективности. Композиция с ней упаковывает две функции в одну: для $f, g : \mathbb{N} \rightarrow \mathbb{N}$ функция $h(n) = J(f(n), g(n))$ несёт оба значения в одном числе. Обратимость J гарантирует, что оба значения извлекаются из h однозначно. Тем же приёмом теория вычислимости кодирует пары и последовательности чисел одним числом: конфигурация машины Тьюринга (глава 25) — тоже такое кодирование.

Проверка Композиция функций

1. Почему $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$, и почему порядок при этом меняется?
2. Почему инъективность композиции $g \circ f$ влечёт инъективность f , но не обязательно g ?

§ 6.7 Алгебра образов и прообразов

Образ множеств и прообраз ведут себя по-разному: образ сохраняет объединение в точности, а для пересечения гарантирует лишь включение. Прообраз сохраняет и объединение, и пересечение, и разность. Механизм виден в доказательствах: каждая операция над множествами превращается в логическую связку над условием принадлежности.

УТВЕРЖДЕНИЕ 6.8 Образ объединения и пересечения

$$f(A \cup B) = f(A) \cup f(B)$$

$$f(A \cap B) \subseteq f(A) \cap f(B)$$

Равенство выполняется, если f инъективна.

Доказательство

Докажем первое равенство — образ объединения.

Образ объединения.

$$\begin{aligned} y \in f(A \cup B) &\leftrightarrow \exists x \in A \cup B : f(x) = y \\ &\leftrightarrow (\exists x \in A : f(x) = y) \vee (\exists x \in B : f(x) = y) \\ &\leftrightarrow y \in f(A) \cup f(B) \end{aligned}$$

Для пересечения прямое включение верно всегда, а обратное требует инъективности.

Образ пересечения.

$$\begin{aligned} y \in f(A \cap B) &\Rightarrow \exists x \in A \cap B : f(x) = y \\ &\Rightarrow y \in f(A) \wedge y \in f(B) \end{aligned}$$

Обратное верно, когда f инъективна:

$$\begin{aligned} y \in f(A) \cap f(B) &\Rightarrow \exists a \in A, b \in B : f(a) = f(b) = y \\ &\Rightarrow a = b \quad (\text{инъективность}) \\ &\Rightarrow a \in A \cap B \\ &\Rightarrow y \in f(A \cap B) \end{aligned}$$

УТВЕРЖДЕНИЕ 6.9 Прообраз сохраняет все операции

$$f^{-1}(C \cup D) = f^{-1}(C) \cup f^{-1}(D)$$

$$f^{-1}(C \cap D) = f^{-1}(C) \cap f^{-1}(D)$$

$$f^{-1}(C \setminus D) = f^{-1}(C) \setminus f^{-1}(D)$$

Прообраз коммутует со всеми булевыми операциями.

Доказательство

Проверим каждое из трёх равенств по отдельности — все доказательства одно-типны.

Пересечение.

$$\begin{aligned} x \in f^{-1}(C \cap D) &\leftrightarrow f(x) \in C \cap D \\ &\leftrightarrow f(x) \in C \wedge f(x) \in D \\ &\leftrightarrow x \in f^{-1}(C) \cap f^{-1}(D) \end{aligned}$$

Объединение.

$$\begin{aligned}
x \in f^{-1}(C \cup D) &\leftrightarrow f(x) \in C \cup D \\
&\leftrightarrow f(x) \in C \vee f(x) \in D \\
&\leftrightarrow x \in f^{-1}(C) \cup f^{-1}(D)
\end{aligned}$$

Разность.

$$\begin{aligned}
x \in f^{-1}(C \setminus D) &\leftrightarrow f(x) \in C \setminus D \\
&\leftrightarrow f(x) \in C \wedge f(x) \notin D \\
&\leftrightarrow x \in f^{-1}(C) \setminus f^{-1}(D)
\end{aligned}$$

Каждый шаг заменяет операцию над множествами её логическим аналогом в условии принадлежности: $\cap \rightarrow \wedge, \cup \rightarrow \vee, \setminus \rightarrow \wedge, \neg$.

Пример Образ пересечения уже пересечения образов

Функция $f(x) = x^2$ на $\mathbb{R}$ и два множества $A = \{-1, 0\}, B = \{0, 1\}$. Образы: $f(A) = \{0, 1\}, f(B) = \{0, 1\}$, а пересечение $A \cap B = \{0\}$ имеет образ $f(\{0\}) = \{0\}$. Сравнение: $f(A \cap B) = \{0\} \neq \{0, 1\} = f(A) \cap f(B)$. Включение из предложения здесь строгое: элемент 1 не имеет общего аргумента в $A \cap B$, он получается из $-1 \in A$ и из $1 \in B$, а эти аргументы живут в разных множествах.

Прообраз ведёт себя иначе: $f^{-1}(\{0\} \cap \{1\}) = f^{-1}(\emptyset) = \emptyset$, и пересечение прообразов $f^{-1}(\{0\}) \cap f^{-1}(\{1\}) = \{0\} \cap \{-1, 1\} = \emptyset$. Равенство прообразов выполняется в точности, как обещает предложение.

§ 6.8 Асимптотические обозначения

Поточечное сравнение — $f(x) = g(x)$ для конкретного x — не подходит для анализа алгоритмов. Там нужно сравнивать *скорость роста* функции при больших значениях аргумента. Алгоритм за n^2 операций рано или поздно станет медленнее алгоритма за $n \log n$ операций. Отношение $\frac{n^2}{n \log n} = \frac{n}{\log n}$ неограниченно растёт. Предел: $\lim_{n \rightarrow \infty} \frac{n}{\log n} = \infty$. При $n = 3$ константы ещё могут скрывать разницу. При $n = 10^6$ рост решает всё.

Асимптотические обозначения дают язык для такого сравнения. Все они определены для функций $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ и описывают поведение при $n \rightarrow \infty$.

ОПРЕДЕЛЕНИЕ 6.16 О-большое

Запись $f(n) \in \mathcal{O}(g(n))$ означает: первая функция растёт *не быстрее* второй (с точностью до константы). Определение:

$$\exists C > 0, n_0 : \forall n \geq n_0, f(n) \leq C \cdot g(n).$$

$\mathcal{O}(g(n))$ — это множество функций, а не одна функция. Запись $f(n) \in \mathcal{O}(g(n))$ читается: f принадлежит классу $\mathcal{O}(g(n))$. Традиционное написание $f(n) = \mathcal{O}(g(n))$ означает то же самое, но знак равенства здесь условен: по смыслу это принадлежность классу, а не равенство величин. Равенства $n = \mathcal{O}(n^2)$, $n^2 = \mathcal{O}(n^2)$ истинны. Но $n = n^2$ из них не следует. Поэтому строгая форма записи через $\in$ надёжнее, и мы будем пользоваться ею в определениях и примерах.

О-большое ($\mathcal{O}$) даёт *верхнюю* границу роста. Симметричную *нижнюю* границу задаёт омега-большое (Ω).

ОПРЕДЕЛЕНИЕ 6.17 Омега-большое

Запись $f(n) \in \Omega(g(n))$ означает: первая функция растёт *не медленнее* второй (с точностью до константы). Определение:

$$\exists c > 0, n_0 : \forall n \geq n_0, f(n) \geq c \cdot g(n).$$

Если функция ограничена и сверху, и снизу, её рост известен с точностью до константы — это точное описание.

ОПРЕДЕЛЕНИЕ 6.18 Тета-большое

Запись $f(n) \in \Theta(g(n))$ означает: первая функция растёт с *той же скоростью*, что и вторая (с точностью до констант). Определение: одновременно выполнены оба условия

$$\begin{aligned} f(n) &\in \mathcal{O}(g(n)) \\ f(n) &\in \Omega(g(n)). \end{aligned}$$

Большие $\mathcal{O}$, Ω довольствуются «какой-то» константой. Θ — их пересечение. Строчные o , ω требуют «любой» константы и потому различают строгие неравенства. $o(g)$ — строго медленнее. $\omega(g)$ — строго быстрее.

ОПРЕДЕЛЕНИЕ 6.19 о-малое

Запись $f(n) \in o(g(n))$ означает: первая функция растёт *строго медленнее* второй. Определение:

$$\forall c > 0, \exists n_0 : \forall n \geq n_0, f(n) \leq c \cdot g(n).$$

ОПРЕДЕЛЕНИЕ 6.20 омега-малое

Запись $f(n) \in \omega(g(n))$ означает: первая функция растёт *строго быстрее* второй. Определение:

$$\forall c > 0, \exists n_0 : \forall n \geq n_0, f(n) \geq c \cdot g(n).$$

Строчные и большие обозначения связаны: $f \in o(g) \leftrightarrow g \in \omega(f)$. Из строгих оценок следуют нестрогие:

$$f \in o(g) \Rightarrow f \in \mathcal{O}(g)$$

$$f \in \omega(g) \Rightarrow f \in \Omega(g)$$

Пример Омега-большое: нижняя граница

Запись $n^2 \in \Omega(n)$ читается: квадратичная функция растёт не медленнее линейной. Проверка по определению: при всех $n \geq 1$ выполнено $n^2 \geq 1 \cdot n$, поэтому подходят $c = 1, n_0 = 1$.

Строгая нижняя оценка передаёт больше информации: $n \log n \in \omega(n)$, потому что $\frac{n \log n}{n} = \log n \rightarrow \infty$ при $n \rightarrow \infty$. Из строгой оценки сразу следует нестрогая: $n \log n \in \Omega(n)$. В анализе алгоритмов нижняя граница — гарантия, что быстрее решить нельзя: сортировка сравнением требует $\Omega(n \log n)$ операций в худшем случае.

Подбор констант вручную каждый раз утомителен. Когда у отношения $\frac{f(n)}{g(n)}$ существует предел, класс функции читается прямо из него.

УТВЕРЖДЕНИЕ 6.10 Предельный критерий

Пусть $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. Предел $L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ существует (конечный или бесконечный). Тогда:

- При $L = 0$: $f(n) \in o(g(n))$.
- При $0 < L < \infty$: $f(n) \in \Theta(g(n))$.
- При $L = \infty$: $f(n) \in \omega(g(n))$.

Доказательство

Разберём три случая для предела L .

Случай 1: $L = 0$. Для любого $c > 0$ при всех достаточно больших n выполнено неравенство

$$\frac{f(n)}{g(n)} < c,$$

то есть $f(n) \leq c \cdot g(n)$. Это и есть $f(n) \in o(g(n))$.

Случай 2: $0 < L < \infty$. Положим $c = \frac{L}{2}, C = 2L$. При всех достаточно больших n выполнено двойное неравенство

$$c \leq \frac{f(n)}{g(n)} \leq C,$$

откуда $c \cdot g(n) \leq f(n) \leq C \cdot g(n)$. Это и есть $f(n) \in \Theta(g(n))$.

Случай 3: $L = \infty$. Для любого $c > 0$ при всех достаточно больших n выполнено неравенство

$$\frac{f(n)}{g(n)} \geq c,$$

откуда $f(n) \geq c \cdot g(n)$. Это $f(n) \in \omega(g(n))$.

Предельный критерий удобен, но предел существует не у каждой функции. Тогда оценку устанавливают по определению — подбором констант.

Пример Оценка без предела

Функция $f(n)$, заданная как

$$f(n) = \begin{cases} 2n & \text{если } n \text{ чётно} \\ 1 & \text{если } n \text{ нечётно} \end{cases},$$

колеблется: на чётных n она принимает значения порядка n , на нечётных — ровно 1. На чётных аргументах $\frac{f(n)}{n} = 2$. На нечётных — $\frac{f(n)}{n} = \frac{1}{n}$. Поэтому у отношения $\frac{f(n)}{n}$ предела нет.

При этом $f(n) \in \mathcal{O}(n)$. При всех $n \geq 1$ выполнено $f(n) \leq 2n$. Значит подходят $C = 2, n_0 = 1$.

Пример Иерархия роста через предел отношения

Предел отношения $\frac{f(n)}{g(n)}$ — рабочий инструмент: вычислил предел, получил ответ.

- $5n^2 + 3n + 10 \in \Theta(n^2)$: предел отношения равен

$$\lim_{n \rightarrow \infty} \frac{5n^2 + 3n + 10}{n^2} = 5 \in (0, \infty)$$

, критерий даёт Θ .

- $n \log n \in o(n^2)$: предел отношения

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^2} = \lim_{n \rightarrow \infty} \frac{\log n}{n} = 0$$

.

- $2^n \in \omega(n^{10})$: предел отношения равен

$$\lim_{n \rightarrow \infty} \frac{n^{10}}{2^n} = 0$$

, откуда $\frac{2^n}{n^{10}} \rightarrow \infty$.

- $\forall \varepsilon > 0 : \log n \in o(n^\varepsilon)$. Предел отношения: $\lim_{n \rightarrow \infty} \frac{\log n}{n^\varepsilon} = 0$.

Здесь использованы два стандартных факта: логарифм растёт медленнее любой положительной степени, экспонента — быстрее любого полинома. Отсюда выстраивается строгая иерархия скоростей роста:

$$\begin{aligned} 1 &\in o(\log n) \\ \log n &\in o(n) \\ n &\in o(n \log n) \\ n \log n &\in o(n^2) \\ n^2 &\in o(n^3) \\ n^3 &\in o(2^n) \\ 2^n &\in o(n!) \end{aligned}$$

Пример Проверка по определению

- $n^2 + n \in \mathcal{O}(n^2)$: при $n \geq 1$ верно неравенство

$$n^2 + n \leq 2n^2$$

, значит подходят $C = 2, n_0 = 1$.

- $n^2 \notin \mathcal{O}(n)$. Для любой константы $C > 0$ при $n > C$ выполнено неравенство

$$n^2 > C \cdot n$$

. Так что ограничивающей константы не существует.

- $n \in o(n^2)$. Для любого $c > 0$ возьмём порог

$$n_0 > \frac{1}{c}$$

. При $n \geq n_0$ будет $n \leq c \cdot n^2$.

Порядок кванторов решает всё: в $\mathcal{O}$ константа «какая-то» ($\exists C$), в o — «любая» ($\forall c > 0$).

Асимптотические обозначения — рабочий язык анализа алгоритмов:

- $\mathcal{O}(n)$: линейное время (просмотреть массив).
- $\mathcal{O}(n \log n)$: сортировка сравнением.
- $\mathcal{O}(n^2)$: перебор всех пар.
- $\mathcal{O}(2^n)$: экспоненциальное время. Практическая недоступность начинается уже при $n > 50$.

Мы будем использовать эту нотацию на протяжении всей книги, в последующих главах — от комбинаторики до теории сложности.

Проверка Асимптотические обозначения

1. Почему из $f \in o(g)$ следует $f \in \mathcal{O}(g)$, но не наоборот?
2. Какой класс даёт предельный критерий при $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$?
3. Почему для $n^2 \notin \mathcal{O}(n)$ не помогает никакая константа C ?

§ 6.9 Характеристическая функция**ОПРЕДЕЛЕНИЕ 6.21** Характеристическая функция

Функция $\chi_X : U \rightarrow \{0, 1\}$, заданная как

$$\chi_X(x) = \begin{cases} 1 & \text{если } x \in X \\ 0 & \text{если } x \notin X \end{cases}$$

называется **характеристической функцией** множества $X \subseteq U$.

Характеристическая функция переводит принадлежность множеству в число — 0 или 1. Она же — инструмент перевода между логикой и арифметикой: пересечение множеств становится умножением, объединение — сложением с вычитанием пересечения: $\chi_{A \cup B} = \chi_A + \chi_B - \chi_A \cdot \chi_B$. Мы воспользуемся этим в комбинаторике.

Пример Характеристическая функция простых чисел

Пусть $U = \{1, \dots, 10\}$, $P = \{2, 3, 5, 7\}$ — простые числа в нём. Тогда:

- $\chi_P(1) = 0$
- $\chi_P(2) = 1$
- $\chi_P(3) = 1$
- $\chi_P(4) = 0$
- $\chi_P(5) = 1$
- $\chi_P(6) = 0$
- $\chi_P(7) = 1$
- $\chi_P(8) = 0$
- $\chi_P(9) = 0$
- $\chi_P(10) = 0$

Сумма $\sum_{x=1}^{10} \chi_P(x) = 4$ даёт количество простых чисел в U . Характеристическая функция превращает вопрос принадлежности в арифметику: мощность конечного множества равна сумме значений его характеристической функции.

§ 6.10 Пол, потолок и нотация Айверсона

Функции пола и потолка переводят непрерывное в дискретное: каждая округляет вещественное число до целого, но по-своему.

ОПРЕДЕЛЕНИЕ 6.22 Пол

Наибольшее целое, не превышающее x , называется **полом** x и обозначается $\lfloor x \rfloor$.

$$x - 1 < \lfloor x \rfloor \leq x$$

ОПРЕДЕЛЕНИЕ 6.23 Потолок

Наименьшее целое, не меньшее x , называется **потолком** x и обозначается $\lceil x \rceil$.

$$x \leq \lceil x \rceil < x + 1$$

Пример Пол и потолок для конкретных чисел

$$\lfloor 3.7 \rfloor = 3, \lceil 3.7 \rceil = 4. \lfloor -3.7 \rfloor = -4, \lceil -3.7 \rceil = -3.$$

Бинарный поиск выполняет $\lfloor \log_2 n \rfloor + 1$ сравнений в худшем случае. Разбиение n элементов на блоки размера k требует $\lceil \frac{n}{k} \rceil$ блоков.

ОПРЕДЕЛЕНИЕ 6.24 Нотация Айверсона

Выражение $[P]$, равное 1, если P истинно, и 0 иначе, называется **нотацией Айверсона**.

Скобки Айверсона — характеристическая функция условия: $[x \in X]$ равно $\chi_X(x)$. Условие превращается в число и становится слагаемым суммы — приём, незаменимый в комбинаторных суммах.

Пример Подсчёт через нотацию Айверсона

- $\pi(n) = \sum_{k=1}^n [k \text{ простое}]$ — количество простых чисел, не превышающих n .
- $\sigma(n) = \sum_{d=1}^n [d \mid n] \cdot d$ — сумма делителей n .

Без нотации Айверсона пришлось бы писать: «сумма по всем d от 1 до n , таким что d делит n ». Нотация делает условие частью формулы, а не окружающего текста.

Пол и потолок двойственны: $\lfloor -x \rfloor = -\lceil x \rceil$, а потолок выражается через пол одной добавкой:

$$\lceil x \rceil = \lfloor x \rfloor + [x \notin \mathbb{Z}].$$

Для целого x добавка нулевая, для дробного равна единице.

Другой нотационный инструмент, лямбда-абстракция, решает противоположную задачу: записать функцию, не давая ей имени.

§ 6.11 Лямбда-нотация

ОПРЕДЕЛЕНИЕ 6.25 Лямбда-абстракция

Нотация $\lambda x \in A \cdot e(x)$ называется **лямбда-абстракцией** и обозначает функцию $f : A \rightarrow B$, отображающую x в $e(x)$.

Лямбда-нотация позволяет определить функцию «на месте», там же где она используется, не заводя для неё отдельное имя. Например, $\lambda x \in \mathbb{R} \cdot x^2 + 2x + 1$ — функция, которая возводит число в квадрат и прибавляет $2x + 1$. Подстановка $x = 3$ даёт $(\lambda x \in \mathbb{R} \cdot x^2 + 2x + 1)(3) = 9 + 6 + 1 = 16$.

Пример Отображение списка через лямбду

Отображение списка чисел через лямбду: $\lambda x \in \mathbb{N} \cdot 2x + 1$ — функция, удваивающая число и прибавляющая единицу. Применив её ко всем элементам списка, получим $[0, 1, 2, 3] \rightarrow [1, 3, 5, 7]$. В Python это записывается как `map(lambda x: 2*x + 1, [0, 1, 2, 3])`. В JavaScript: `[0, 1, 2, 3].map(x => 2*x + 1)`.

Именованная функция требует четырёх шагов:

1. придумать имя,
2. объявить сигнатуру,
3. написать тело,
4. вызвать по имени.

С лямбдой — один шаг: написать тело там, где функция нужна. Это не косметическое различие: когда функция используется ровно один раз (предикат фильтрации, компаратор сортировки), имя — лишняя сущность, загромождающая код.

В языках с первоклассными функциями (Python, JavaScript, Haskell, Rust) лямбды можно передавать как аргументы, возвращать как значения и сохранять в переменных. Замыкание (closure) — лямбда, захватывающая переменные из внешней области видимости — реализует частичное применение и каррирование. Эти понятия естественно возникают из концепции лямбда-абстракции и лежат в основе функционального программирования.

§ 6.12 * Каррирование и λ -исчисление

Функция двух аргументов $f : A \times B \rightarrow C$ принимает пару. Мы подаём оба аргумента одновременно и получаем результат. Но можно иначе: подать первый аргумент,

а вместо результата получить функцию, которая ждёт второй. Тип такой функции записывается как $f : A \rightarrow (B \rightarrow C)$.

Эта операция — превращение функции от пары в функцию, возвращающую функцию — называется *каррированием*. Названа в честь Хаскелла Карри, хотя впервые описана Моисеем Шейнфинкелем в 1924 году. В Haskell и ML все функции каррированы по умолчанию: применение f к x даёт функцию, ждущую y , — частичное применение встроено в синтаксис.

Каррирование — не трюк с нотацией. Это структурный принцип, на котором построено λ -исчисление Чёрча (1932). Вся система состоит из трёх конструкций: переменные, λ -абстракция (определение функции) и применение (подстановка аргумента). Функция двух аргументов $f(x, y) = x + y$ в лямбда-нотации записывается как $\lambda x \cdot \lambda y \cdot x + y$. Применение к тройке даёт $(\lambda x \cdot \lambda y \cdot x + y)(3) \rightarrow \lambda y \cdot 3 + y$ — функцию одного аргумента, готовую принять второй.

Любая вычислимая функция кодируется в λ -исчислении — без машин, без памяти, без присваивания: только определение и подстановка. Тьюринг-полнота λ -исчисления означает эквивалентность машинам Тьюринга: один и тот же класс вычислимых функций, два разных взгляда. Машина Тьюринга — вычисление через состояние и переходы. λ -исчисление — вычисление через определение и подстановку.

Для нашей главы каррирование проясняет три вещи.

- Частичное применение: фиксируем часть аргументов — получаем новую функцию. Это работает потому, что $A \rightarrow (B \rightarrow C)$ позволяет подавать аргументы по одному.
- Замыкание: функция, полученная частичным применением, «помнит» уже переданные аргументы.
- Ассоциативность композиции: $h \circ (g \circ f) = (h \circ g) \circ f$ — алгебраический факт, но каррирование показывает его вычислительную природу. И слева, и справа один и тот же конвейер: f, g, h .

В функциональном программировании каррирование — повседневный инструмент. Когда мы пишем `map (\x -> x + 1)`, мы применяем каррированную функцию `map` к одному аргументу и получаем новую функцию, готовую обработать список: это тип $A \rightarrow (B \rightarrow C)$ вместо типа $A \times B \rightarrow C$.

§ 6.13 Итоги главы

Функция — дисциплинированное отношение: каждому входу соответствует ровно один выход. Это ограничение и делает её полезной: правило, которое на один вопрос даёт два разных ответа, никуда не годится — будь то хеш-функция, кодирование символов или сериализация. Оба условия существенны: выход существует для каждого входа, и он единственный.

Классификация по обратимости — инъекция, сюръекция, биекция — показывает, что тип функции определяет, что можно восстановить из результата. Прообраз определён всегда, а обратная функция — только для биекций. Композиция склеивает функции в цепочки, наследуя свойства звеньев. Она ассоциативна, но не коммутативна. Частичная функция добавила честную модель вычисления, которое может не завершиться.

Сквозной приём, работающий дальше по книге: вопрос о множествах переводится на язык функций и решается арифметикой. Характеристическая функция, нотация Айверсона, пол и потолок кодируют множества и количества числами, асимптотические обозначения сравнивают скорость роста, а лямбда-абстракция и каррирование дают безымянные функции и частичное применение.

Функции обобщают отношения условием однозначности и служат фундаментом последующей математики — от порядка (глава 8) до вычислимости (глава 25). Биекции оказываются инструментом сравнения, и с их помощью мы готовы измерить бесконечность в главе 7.

Проверка Проверьте себя

1. Почему инъекция и сюръекция вместе дают биекцию и что добавляет каждая из них?
2. Что такое прообраз и чем он отличается от обратной функции? Почему прообраз определён всегда, а обратная функция — только для биекций?
3. Почему частичная функция — подходящая модель вычисления, которое может не завершиться?
4. Зачем нужна лямбда-абстракция, если функции можно давать имена?
5. Почему композиция функций ассоциативна, но не коммутативна?
6. Почему $\mathcal{O}(g(n))$ — это множество функций, а не одна функция? И что означает запись $f(n) = \mathcal{O}(g(n))$?

Что дальше?

От функций общего вида мы переходим к вопросу о размере бесконечных множеств: одинакового ли размера натуральные и вещественные числа? В главе 7 мы введём равномощность, счётные и несчётные множества и диагональный аргумент Кантора — биекции, изученные в этой главе, станут там основным инструментом сравнения. А затем обратимся к отношениям порядка: в главе 8 — частичные порядки, диаграммы Хассе, решётки и топологическая сортировка, моделирующие зависимости и иерархии.

§ 6.14 Упражнения

Функция как отношение.

1. Какие из отношений $R \subseteq \{1, 2, 3\} \times \{a, b\}$ являются функциями? $R_1 = \{(1, a), (2, b), (3, a)\}$, $R_2 = \{(1, a), (1, b), (2, a)\}$, $R_3 = \{(1, a), (2, a)\}$. Для каждого нефункционального отношения укажите, какое условие нарушено.
2. Найдите $f(A)$, $f^{-1}(B)$ для функции $f(x) = x^2$. Здесь $A = \{-2, -1, 0, 1, 2\}$, $B = \{0, 1, 4, 9\}$. Объясните, почему $f^{-1}(B)$ — множество, а не одно значение.
3. Приведите пример частичной функции $f : \mathbb{R} \mapsto \mathbb{R}$, которая не определена ровно в двух точках. Можно ли сузить область определения так, чтобы она стала тотальной?

Инъекция, сюръекция, биекция.

4. Какие из следующих функций $f : \mathbb{R} \rightarrow \mathbb{R}$ инъективны, сюръективны, биективны? $f(x) = x^3$, $f(x) = x^2$, $f(x) = 2x + 1$, $f(x) = e^x$.
5. Приведите пример функции $f : \mathbb{N} \rightarrow \mathbb{N}$, которая сюръективна, но не инъективна. Приведите пример функции $g : \mathbb{N} \rightarrow \mathbb{N}$, которая инъективна, но не сюръективна.
6. * Сколько существует инъективных функций $f : \{1, 2, 3\} \rightarrow \{1, 2, 3, 4, 5\}$? Сколько сюръективных? Выведите общие формулы для произвольных m, n .
7. ** Постройте биекцию $\mathbb{N} \rightarrow \mathbb{Z}$. Докажите, что построенное отображение действительно биективно.

Композиция функций.

8. Пусть $f(x) = 2x + 1$, $g(x) = x^2$ (над $\mathbb{R}$). Найдите $f \circ g$, $g \circ f$. Совпадают ли они? Какая из этих композиций инъективна?
9. Пусть $f : A \rightarrow B$, $g : B \rightarrow C$. Докажите: если $g \circ f$ инъективна, то f инъективна. Верно ли обратное?
10. * Докажите: если оба звена биективны, то $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$. Почему порядок в правой части меняется?

Образы и прообразы.

11. Докажите, что $f(A \cup B) = f(A) \cup f(B)$. Покажите на примере, что для пересечения равенство может не выполняться.
12. * Докажите: $f^{-1}(C \cap D) = f^{-1}(C) \cap f^{-1}(D)$. Почему прообраз сохраняет пересечение, а образ — не всегда?

Асимптотические обозначения.

13. Расположите функции в порядке возрастания скорости роста: $n \log n$, n^2 , 2^n , $\log n$, n , 1 . Для каждой пары укажите, верно ли, что левая функция есть $\mathcal{O}$ от правой.
14. * Докажите: $f(n) \in \Theta(g(n)) \leftrightarrow g(n) \in \Theta(f(n))$. Приведите пример двух функций, для которых $f(n) \in \mathcal{O}(g(n)) \setminus \Theta(g(n))$.

ПРОЕКТ Функциональные графы и поиск циклов

Реализуйте инструмент анализа итерации функции на конечном множестве. Функция $f : [0, n) \rightarrow [0, n)$, заданная массивом образов, превращает множество в ориентированный граф, в котором из каждой вершины выходит ровно одно ребро. Итерация любой точки неизбежно входит в цикл, поэтому траектория принимает форму буквы «ро»: хвост, затем цикл. Соберите его из композиции и итерации функций и примените к четырём задачам: декомпозиция на циклы и деревья, поиск цикла без хранения траектории, статистика случайных функций и порядок элемента по модулю.

① *Декомпозиция графа*

Разложите граф на компоненты: каждая компонента связности состоит ровно из одного цикла и входящих в него деревьев. Реализуйте двухпроходный алгоритм с раскраской вершин, который за $O(n)$ находит все циклы, для каждой вершины — длину хвоста μ и длину цикла λ её компоненты, а также высоты входящих деревьев. Продумайте, почему двух проходов достаточно, и проверьте инварианты декомпозиции на случайных функциях.

② *Поиск цикла без хранения траектории*

Реализуйте поиск цикла одной траектории тремя способами: наивный запоминает все посещённые состояния и требует памяти $O(\mu + \lambda)$, а алгоритмы Флойда и Брента обходятся памятью $O(1)$. Подсчитайте число вычислений функции в каждом алгоритме и сравните их. Объясните, почему память $O(1)$ критична, когда состояние велико, а шаг функции дорог.

③ *Случайные функции и теория*

Сгенерируйте случайную функцию на $[0, n)$ и измерьте средние длину хвоста μ , длину цикла λ и длину «ро» $\mu + \lambda$ до первого повторения. Сравните замеры с теорией:

- средние μ и λ асимптотически равны $\sqrt{\pi \frac{n}{8}}$;
- длина «ро» $\mu + \lambda \sim \sqrt{\pi \frac{n}{2}}$;
- число компонент $\sim \frac{\ln n}{2}$.

Прогоните ряд значений n и проследите сходимость замеров к теории. Почему ведущий член одинаков для хвоста и цикла, а для «ро» другой?

④ *Порядок элемента по модулю*

Примените поиск цикла к итерации хеш-функций и к порядку элемента по модулю. Порядок элемента g по модулю простого p — длина цикла траектории единицы под умножением на g : найдите его без перебора всех степеней, с памятью $O(1)$. Для аффинного отображения $f(x) = (ax + c) \bmod m$ и квадратичного $f(x) = x^2 \bmod p$ определите, когда траектория образует полный цикл, а когда хвост с петлёй. Проверьте найденный порядок сертификатом: $g^{\text{ord}} \equiv 1 \pmod p$. Убедитесь, что $g^{\frac{\text{ord}}{q}} \not\equiv 1 \pmod p$ для каждого простого делителя q числа ord . Как тот же приём поиска цикла применить к обнаружению коллизий хеша?

Итог: библиотека, которая разлагает функциональный граф на циклы и входящие деревья за линейное время, находит длину хвоста и цикла любой траектории с памятью $O(1)$, сверяет средние случайных функций с теорией $\sqrt{\pi \frac{n}{2}}$ и $\sqrt{\pi \frac{n}{8}}$ и вычисляет порядок элемента по модулю поиском цикла.

VII

Мощность и бесконечность

Сколько всего натуральных чисел, а сколько действительных? На первый взгляд, ответ одинаков: и там и там бесконечно много. С детства мы привыкли, что у размера есть число: пять, сто, миллион. У бесконечности числа нет, а размер есть. Георг Кантор показал, что интуиция ошибается: бесконечностей много, и они разного размера.

В предыдущей главе (6) мы определили функцию как способ сопоставления элементов одного множества элементам другого. Биекция — взаимно-однозначное соответствие — даёт инструмент для вопроса «сколько элементов во множестве», не требующий пересчитывания. Два множества имеют одинаковый размер, если между ними существует биекция. Для конечных множеств этот критерий совпадает с обычным подсчётом: яблоки из двух мешков можно разложить парами, по одному из каждого, и сказать, поровну ли яблок. Для бесконечных множеств он приводит к выводам, которые расходятся с повседневной интуицией.

Часть может равняться целому: натуральных чисел столько же, сколько чётных. Добавление не увеличивает размер: заполненный бесконечный отель принимает ещё бесконечный автобус гостей и остаётся заполненным. Рациональные числа, плотно заполняющие прямую, — того же размера, что и натуральные. А булеан натуральных чисел уже больше: его элементы нельзя занумеровать. Аксиомы конечного мира — часть меньше целого, добавление увеличивает размер — в бесконечности перестают действовать. Остаётся один инструмент, который работает везде, — биекция, и на нём держится всё дальнейшее измерение.

Галилей заметил, что квадратов «столько же», сколько натуральных чисел, и два с половиной столетия это наблюдение оставалось курьёзом. Кантор превратил курьёз в определение: два множества имеют одинаковый размер, если между ними существует биекция.

От вопроса «сколько» до вопроса «что можно вычислить» — один шаг. Программ, которые можно записать конечным текстом, — счётное число. Языков — подмножеств всех строк над конечным алфавитом — континуум. Значит, почти все языки не вычислимы ни одной программой: неразрешимость оказывается нормой, а вычислимость — редким исключением. К этому выводу приведёт глава 26. Полную арифметику кардиналов — с аксиомой выбора и континуум-гипотезой — построим в главе 22.

Конечный опыт — плохой советчик в бесконечном мире. Часть равняется целому, добавление ничего не меняет, размер перестаёт быть числом. Но у этих нарушений есть точный порядок: их можно измерить, сравнить, сложить и возвести в степень. Чем измерять размер, когда привычных чисел больше нет?

ОБЗОР ГЛАВЫ

Мы сравним размер множеств через биекцию — так определяется равномощность — и на отеле Гильберта увидим, как счётная бесконечность поглощает добавления. Затем научимся доказывать счётность $\mathbb{Z}$ и $\mathbb{Q}$. Убедимся, что плотно заполняющие прямую рациональные числа имеют мощность $\aleph_0$.

Диагональный аргумент Кантора покажет, что $\mathbb{R}$ несчётно. Он откроет лестницу кардиналов: $\aleph_0 < 2^{\aleph_0} < \dots$. Теорема Кантора–Бернштейна–Шрёдера позволит сравнивать мощности, не строя явных биекций. Кардинальная арифметика научит складывать, умножать и возводить бесконечности в степень: первые две операции размер не меняют, третья рождает новый кардинал. В конце диагональ свяжет счёт с вычислимостью: программ счётное множество, языков — континуум.

После этой главы вы сможете:

1. доказывать равномощность двух множеств, строя биекцию между ними;
2. проверять, счётно ли множество, и пользоваться счётностью $\mathbb{Z}$ и $\mathbb{Q}$;
3. воспроизводить диагональный аргумент Кантора и доказывать несчётность $\mathbb{R}$ и рост булеана;
4. сравнивать мощности с помощью теоремы Кантора–Бернштейна–Шрёдера;
5. объяснять, почему программ счётное число, а языков — континуум, и что из этого следует для вычислимости.

§ 7.1 Равномощность

Как сравнить размеры двух бесконечных множеств? Для конечных множеств мы просто пересчитываем элементы. Для бесконечных этот подход не работает, но идея взаимно-однозначного соответствия остаётся — два множества «одинакового размера», если можно установить биекцию между ними.

ОПРЕДЕЛЕНИЕ 7.1 Равномощность (*equinumerosity*)

Два множества называются **равномощными**, если существует биекция $f : A \rightarrow B$. Равномощность обозначается $|A| = |B|$.

Равномощность — прямое обобщение идеи «одинакового количества» на случай, когда пересчитать элементы нельзя. Для конечных множеств она совпадает с обычным подсчётом: два конечных множества равномощны тогда и только тогда, когда в них поровну элементов.

ОПРЕДЕЛЕНИЕ 7.2 Конечное множество

Множество называется **конечным**, если оно равномощно множеству $\{0, 1, \dots, n-1\}$ для некоторого $n \in \mathbb{N}$. При $n = 0$ это пустое множество.

ОПРЕДЕЛЕНИЕ 7.3 Бесконечное множество

Множество называется **бесконечным**, если оно не является конечным.

Пример Конечное и бесконечное по определению

Множество $\{x, y, z\}$ конечно: биекция с $\{0, 1, 2\}$ задаётся правилом $x \mapsto 0, y \mapsto 1, z \mapsto 2$, здесь $n = 3$. Множество $\mathbb{N}$ бесконечно: для любого n в $\mathbb{N}$ найдётся $n+1$ элемент $(0, 1, \dots, n)$, так что биекции с $\{0, 1, \dots, n-1\}$ быть не может.

Бесконечные множества ведут себя контринтуитивно: они могут быть равномощны своему собственному подмножеству. Для конечных множеств такое невозможно — часть всегда меньше целого. Для бесконечных это свойство становится характеристическим: Дедекинд именно так и определял бесконечность.³⁸

Пример Равномощность бесконечного множества своему подмножеству

Множества $\mathbb{N}$ и $E = \{0, 2, 4, 6, \dots\}$ равномощны:

- Биекция $f : \mathbb{N} \rightarrow E, f(n) = 2n$.
- Обратная биекция $g : E \rightarrow \mathbb{N}, g(m) = \frac{m}{2}$.

Хотя $E \subset \mathbb{N}$, оба множества имеют одинаковую мощность, которая обозначается $\aleph_0 = |\mathbb{N}|$.

ЛОВУШКА Равномощность $\neq$ равенство

Биекция между A и B означает одинаковую мощность, а не совпадение множеств. $\mathbb{N}$ и $\mathbb{Z}$ равномощны, но это разные множества. Биекция ничего не говорит о том, что элементы одинаковы: она лишь устанавливает взаимно-однозначное соответствие.

Замечание Почему биекция?

Почему мы определяем равномощность через биекцию, а не через включение? Отношение включения $A \subseteq B$ как критерий размера не симметрично: чётные числа были бы «меньше» натуральных, хотя интуитивно они одного размера. Биекция даёт симметричное, рефлексивное и транзитивное отношение — отно-

³⁸Р. Дедекинд. «Was sind und was sollen die Zahlen?». 1888. В ZF из дедекиндовского определения выводится стандартное: множество, равномощное собственному подмножеству, бесконечно. Обратная импликация требует аксиомы выбора (достаточно счётного выбора), поэтому эквивалентность двух определений — теорема ZFC. В одной ZF она не доказуема.

шение эквивалентности. Классы этой эквивалентности — кардинальные числа (формальное определение — в разделе о кардинальной арифметике ниже). Альтернатива — сравнение через подмножество — дала бы лишь частичный порядок, не позволяющий сказать «эти два множества имеют одинаковый размер». Выбор биекции как критерия — это выбор между порядком и классификацией в пользу классификации.

Итак, у нас есть способ сказать «два множества имеют одинаковый размер», не пересчитывая элементы. Естественный следующий вопрос: какие вообще бывают размеры у бесконечных множеств? Мы ожидаем, что натуральных чисел «меньше», чем действительных — хотя и те, и другие бесконечны. Но интуиция — плохой советчик в этой области. Множество рациональных чисел, которое плотно заполняет числовую прямую, имеет *тот же размер*, что и натуральные. А множество, которое кажется «всего лишь» булеаном натуральных, уже несчётно. Разграничение счётного и несчётного — первый шаг в картографировании бесконечности.

§ 7.2 ОТЕЛЬ ГИЛЬБЕРТА

В бесконечном отеле Гильберта бесконечное (счётное) число комнат. Комнаты пронумерованы натуральными числами $0, 1, 2, \dots$ — по одной на каждое натуральное число. Все комнаты заняты: в каждой комнате живёт ровно один постоялец. Отель заполнен полностью — свободных комнат нет. Будь у отеля последняя комната, пересчёт комнат когда-нибудь закончился бы — и их было бы конечное число.

Мы рассмотрим три ситуации — от простой к сложной. В каждой из них прибывают новые гости, и администратор расселяет их, *никого не выселяя и не подсекая двоих в одну комнату*. После каждого заселения отель снова заполнен полностью: каждое натуральное число остаётся номером ровно одной занятой комнаты.

7.2.1 Один новый гость

Приходит новый гость. Администратор просит каждого постояльца переехать в комнату со следующим номером: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \dots$. После сдвига комната 0 освобождается, и новый гость заселяется в неё.

Комнаты перенумеровали постояльцев по правилу $n \rightarrow n + 1$. Отображение $\mathbb{N} \rightarrow \mathbb{N} \setminus \{0\}, n \mapsto n + 1$ — биекция (все комнаты, кроме нулевой). Новый гость занял комнату 0, и все комнаты снова заняты. Тот же приём сработает для любого конечного числа гостей: достаточно сдвига $n \rightarrow n + k$. Он освобождает комнаты $0, 1, \dots, k - 1$.

Отель был заполнен ($\aleph_0$ постояльцев), добавился один гость — и отель снова заполнен. Мощность множества постояльцев не изменилась:

$$\aleph_0 + 1 = \aleph_0.$$

7.2.2 Автобус с бесконечным числом гостей

Прибывает автобус с бесконечным (счётным) числом новых гостей. Пронумеруем пассажиров автобуса натуральными числами: $b_0, b_1, b_2, \dots$

Администратор переселяет текущих постояльцев в чётные комнаты по правилу $n \rightarrow 2n$. После этого все чётные комнаты заняты, а все нечётные — $1, 3, 5, \dots$ — свободны. Нечётных комнат ровно столько же, сколько натуральных чисел: $|\mathbb{N}| = |\{1, 3, 5, \dots\}|$. Заселение пассажиров идёт по правилу $b_k \rightarrow 2k + 1$.

Отображение $f(n) = 2n$ задаёт биекцию на множество чётных комнат (старые постояльцы). Отображение $g(k) = 2k + 1$ задаёт биекцию на множество нечётных комнат (новые гости). Вместе они образуют биекцию между двумя копиями $\mathbb{N}$ и всеми комнатами.

Отель был заполнен, добавилось ещё $\aleph_0$ гостей — и отель снова заполнен:

$$\aleph_0 + \aleph_0 = \aleph_0.$$

7.2.3 Бесконечное число автобусов

Прибывает бесконечное (счётное) число автобусов, в каждом — бесконечное (счётное) число гостей. Самих автобусов столько же, сколько натуральных чисел. Мест в каждом автобусе — тоже столько же, сколько натуральных чисел. Обозначим пассажира через $a_{i,j}$. Здесь $i, j \in \mathbb{N}$: индексы — номера автобуса и места соответственно.

Сначала, как в предыдущем шаге, переселим текущих постояльцев в чётные комнаты по правилу $n \rightarrow 2n$. Все нечётные комнаты свободны — их столько же, сколько натуральных чисел. Осталось разместить в них новых гостей.

Новых гостей — по одному на каждую пару натуральных чисел (i, j) . Нумерацию пар по диагоналям задаёт отображение $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Его формулу и обоснование биективности приводим в примере «Номера диагоналей» ниже. Шаги обхода показаны на рис. 19 — это тот же метод, которым мы перечислим $\mathbb{Q}$. Первые шаги обхода: $(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), \dots$ Пары перечисляются по возрастанию суммы $i + j$, внутри диагонали — по первой координате. Заселение пассажиров идёт по правилу $a_{i,j} \rightarrow 2 \cdot h(i, j) + 1$ (нечётные комнаты).

Отель был заполнен, добавилось $\aleph_0 \cdot \aleph_0$ новых гостей — и отель снова заполнен. Следовательно,

$$\aleph_0 + \aleph_0 \cdot \aleph_0 = \aleph_0.$$

Сама диагональная нумерация $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ является биекцией, что напрямую доказывает

$$\aleph_0 \cdot \aleph_0 = \aleph_0.$$

7.2.4 Что мы увидели

Отель Гильберта показывает: добавление новых элементов (счётного числа) к бесконечному множеству не меняет его мощности.

- Один новый гость: $\aleph_0 + 1 = \aleph_0$.
- Бесконечное (счётное) число гостей: $\aleph_0 + \aleph_0 = \aleph_0$.
- Бесконечное число бесконечных автобусов: $\aleph_0 \cdot \aleph_0 = \aleph_0$.

С точки зрения биекций каждая ситуация — это построение взаимно-однозначного соответствия между номерами комнат ($\mathbb{N}$) и расширенным составом постояльцев. Отель был заполнен *до*, остался заполнен *после* — хотя постояльцев стало «больше». Ни одна из этих операций не вывела нас за пределы $\aleph_0$. Отель Гильберта примет любой счётный наплыв гостей — но отступит перед несчётным. Действительные числа не удастся расселить ни по какой схеме: это покажет диагональный аргумент ниже.

Проверка Равномощность и отель Гильберта

1. Почему множество чётных чисел равномощно $\mathbb{N}$, хотя и является его собственным подмножеством?
2. Почему сдвиг постояльцев $f(n) = n + 1$ размещает нового гостя в заполненном отеле Гильберта, но не сработал бы в конечном отеле?

§ 7.3 Счётные и несчётные множества

ОПРЕДЕЛЕНИЕ 7.4 Счётное множество

Множество называется **счётным**, если оно конечно или равномощно $\mathbb{N}$. Его элементы можно занумеровать натуральными числами.

ОПРЕДЕЛЕНИЕ 7.5 Несчётное множество

Множество называется **несчётным**, если оно бесконечно и не является счётным.

Название «счётное» обманчиво: бесконечное множество нельзя *пересчитать* до конца. Смысл в другом: его элементы можно *перечислить* — выписать в один бесконечный список.

ЛОВУШКА Счётно-бесконечное $\neq$ конечное

Счётно-бесконечное множество — то есть равномощное $\mathbb{N}$ — бесконечно: нумерация натуральными числами не делает его «почти конечным». По принятому определению конечные множества тоже счётны, так что «счётное» охватывает и

конечные, и счётно-бесконечные. $|\mathbb{Z}| = |\mathbb{Q}| = \aleph_0 < |\mathbb{R}|$ — бесконечность и счётность лежат на разных осях.

Счётность — это «наименьшая» бесконечность. Если множество счётно и бесконечно, его элементы можно выписать в виде бесконечной последовательности $a_1, a_2, \dots$

Пример Счётное множество

Множество нечётных натуральных чисел $O = \{1, 3, 5, \dots\}$ счётно.

Биекция $\mathbb{N} \rightarrow O, n \mapsto 2n + 1$.

Хотя нечётные числа образуют собственное подмножество натуральных ($O \subset \mathbb{N}$), оба множества имеют одинаковую мощность:

$$|O| = |\mathbb{N}| = \aleph_0$$

7.3.1 Счётность $\mathbb{Z}$ и $\mathbb{Q}$

Многие множества, которые выглядят «больше» $\mathbb{N}$, оказываются счётными.

ТЕОРЕМА 7.1 $\mathbb{Z}$ и $\mathbb{Q}$ счётны

Множества $\mathbb{Z}$ и $\mathbb{Q}$ (целых и рациональных чисел) счётны.

Доказательство

Докажем счётность $\mathbb{Z}$ и $\mathbb{Q}$ по отдельности, для каждого построив явное перечисление.

Начнём с целых чисел.

Целые числа: $\mathbb{Z}$

Биекция $f: \mathbb{N} \rightarrow \mathbb{Z}$ задаётся перечислением:

$$0, -1, 1, -2, 2, -3, 3, \dots$$

Явная формула:

$$f(n) = \begin{cases} \frac{n}{2} & \text{если } n \text{ чётно} \\ -\frac{n+1}{2} & \text{если } n \text{ нечётно} \end{cases}$$

Теперь рациональные числа.

Рациональные числа: $\mathbb{Q}$

Нельзя выписать дроби построчно: перечисляя одну строку за другой, мы никогда не дойдём до второй. Обход по диагоналям — «змейка» — добирается до каждой клетки. Перечисляем все несократимые дроби $\frac{p}{q}$ ($q > 0$) в порядке

возрастания высоты $|p| + q$. Внутри одной высоты — по возрастанию числителя. На высоте 1 стоит дробь $\frac{0}{1}$. На высоте 2 — $-\frac{1}{1}$ и $\frac{1}{1}$. На высоте 3 — $-\frac{2}{1}$, $-\frac{1}{2}$, $\frac{1}{2}$, $\frac{2}{1}$. На высоте 4 — $-\frac{3}{1}$, $-\frac{1}{3}$, $\frac{1}{3}$, $\frac{3}{1}$, причём дроби $-\frac{2}{2}$, $\frac{2}{2}$ сократимы и пропускаются. И так далее по всем натуральным высотам.

Это диагональное перечисление пар (Кантор, 1873): организуем пары (p, q) в бесконечную матрицу и обходим её по диагоналям. Каждое рациональное число появляется в перечислении ровно один раз.

Пример Зигзаг по целым числам

Перечисление $0, -1, 1, -2, 2, -3, 3, \dots$ обходит все целые числа, чередуя знаки. Первые значения биекции $f: \mathbb{N} \rightarrow \mathbb{Z}$:

$$f(0) = 0, f(1) = -1, f(2) = 1, f(3) = -2, f(4) = 2, f(5) = -3, f(6) = 3.$$

Чётные аргументы дают неотрицательные числа: $f(2k) = k$. Нечётные — отрицательные: $f(2k+1) = -(k+1)$. Обратная биекция $g: \mathbb{Z} \rightarrow \mathbb{N}$ собирает целые числа обратно:

$$g(z) = \begin{cases} 2z & \text{если } z \geq 0 \\ -2z - 1 & \text{если } z < 0 \end{cases}$$

Проверим: $g(5) = 10$, то есть целое число 5 стоит на позиции 10 перечисления (нумерация с нуля). А $g(-5) = 9$: число -5 — на позиции 9. Каждое целое число встречается в перечислении ровно один раз, поэтому $|\mathbb{Z}| = |\mathbb{N}| = \aleph_0$.

Диагональный обход пар — тот же приём, которым перечисляется $\mathbb{Q}$. На рис. 19 показано, как пары (i, j) выстраиваются по возрастанию суммы $i + j$, а цифры на диагоналях задают порядок нумерации.

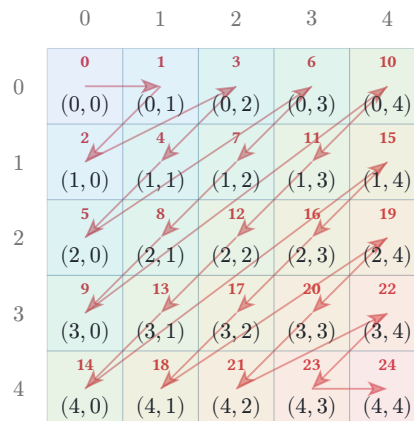


Рис. 19. Диагональное перечисление пар по возрастанию суммы $i + j$.

Пример Номера диагоналей

Диагональный обход на рис. 19 присваивает каждой паре (i, j) номер. Первые диагонали:

$$(0, 0) \mapsto 0, (0, 1) \mapsto 1, (1, 0) \mapsto 2, (0, 2) \mapsto 3, (1, 1) \mapsto 4, (2, 0) \mapsto 5, \dots$$

На диагонали $d = i + j$ стоят $d + 1$ пар: $(0, d), (1, d - 1), \dots, (d, 0)$. Перед ней лежат диагонали с суммами $0, 1, \dots, d - 1$. Всего в них $1 + 2 + \dots + d$ пар. Поэтому номер пары (i, j) задаётся формулой

$$h(i, j) = (i + j) \frac{i + j + 1}{2} + i.$$

Пара $(1, 2)$ получает номер $h(1, 2) = \frac{3 \cdot 4}{2} + 1 = 7$. Сопоставление $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ биективно: пары одной диагонали $d = i + j$ занимают отрезок номеров от $\frac{d(d+1)}{2}$ до $\frac{d(d+1)}{2} + d$ без повторов, соседние диагонали — соседние отрезки, а вместе отрезки покрывают все $\mathbb{N}$ без дыр. Это та самая биекция, которая расселяет гостей бесконечных автобусов по нечётным комнатам: $\aleph_0 \cdot \aleph_0 = \aleph_0$.

Множество $\mathbb{Q}$ *плотно* (*dense*): между любыми двумя различными рациональными числами существует ещё одно рациональное. Однако $|\mathbb{Q}| = |\mathbb{N}|$: рациональных чисел столько же, сколько натуральных. Диагональное перечисление на рис. 19 показывает, как организовать все пары в одну последовательность. Мощность абстрагируется от топологии и порядка — она измеряет только количество элементов, а не их организацию на числовой прямой.

Замечание Плотность не означает величину

Рациональные числа лежат на прямой так плотно, что между любыми двумя есть ещё одно. И всё же их можно закрыть интервалами сколь угодно малой суммарной длины: вокруг дробей кладут интервалы длин $\frac{\varepsilon}{2}, \frac{\varepsilon}{4}, \frac{\varepsilon}{8}, \dots$. Сумма длин всей ленты равна $\frac{\varepsilon}{2} + \frac{\varepsilon}{4} + \frac{\varepsilon}{8} + \dots = \varepsilon$ — сколь угодно мала, хотя покрывает каждое рациональное число. Плотность описывает расположение, мощность — количество: две независимые оси.

Пример Пробел в рациональных числах

Множество $B = \{q \in \mathbb{Q} \mid q^2 < 2\}$ ограничено сверху: подходит дробь $\frac{3}{2}$. Наибольшего элемента у множества нет: между любой дробью с квадратом меньше двух и границей $\sqrt{2}$ найдётся ещё одна такая дробь. Точной верхней грани в $\mathbb{Q}$ у множества тоже нет: она обязана равняться $\sqrt{2}$, а это число рациональным не является. Плотность от этого пробела не спасает: он стоит не между соседними элементами, а на месте отсутствующего предела. Пополнение $\mathbb{Q}$ до $\mathbb{R}$ устраняет такие пробелы: в $\mathbb{R}$ каждое ограниченное множество имеет точную верхнюю грань. Эта конструкция разбирается в главе 22.

7.3.2 Две бесконечности: история открытия

ИСТОРИЯ «Je le vois, mais je ne le crois pas»

До Кантора математики избегали актуальной бесконечности. Аристотель настаивал: бесконечное существует только потенциально — можно продолжать неограниченно, но нельзя завершить. Галилей заметил, что квадратов «столько же», сколько натуральных чисел (биекция $n \mapsto n^2$), но счёл это курьёзом, а не поводом пересмотреть понятие размера. Гаусс в 1831 году писал: «Я протестую против использования бесконечной величины как чего-то завершённого. В математике это никогда не разрешено».

Кантор проигнорировал запрет. В письме от 29 ноября 1873 года он спрашивает Рихарда Дедекинда: можно ли занумеровать действительные числа натуральными? Дедекинд отвечает, что не видит ни доказательства, ни опровержения. Через несколько недель Кантор доказывает: нельзя. Бесконечностей как минимум две: счётная и континуум ($\mathbb{N}$ и $\mathbb{R}$).

Следующая задача казалась Кантору столь же безнадёжной: доказать, что между отрезком и квадратом биекции *не существует*. Отрезок одномерен, квадрат двумерен. Кажется, что в квадрате «больше» точек. С 1874 по 1877 год Кантор искал доказательство невозможности — и не находил.

Кантор нашёл обратное: биекция отрезка на квадрат существует, и в письме Дедекинду он признался: «Je le vois, mais je ne le crois pas» — «Я вижу это, но не верю». Оказалось, что отрезок, квадрат, куб и любое конечномерное пространство равномощны: для мощности размерность не имеет значения.

7.3.3 Несчётность $\mathbb{R}$

Кантор поставил вопрос: все ли бесконечные множества счётны? До него математики не предполагали, что бесконечности могут быть разного размера. Ответ — нет, не все: доказательство даёт *диагональный аргумент*.

ТЕОРЕМА 7.2 Диагональный аргумент Кантора

$\mathbb{R}$ несчётно.

Доказательство

Докажем от противного.

Покажем, что интервал $(0, 1)$ несчётный. Предположим противное: числа интервала $(0, 1)$ перечислимы. Пусть $r_1, r_2, r_3, \dots$ — их перечисление в десятичной записи:

$$\begin{aligned} r_1 &= 0.d_{11}d_{12}d_{13}\dots \\ r_2 &= 0.d_{21}d_{22}d_{23}\dots \\ r_3 &= 0.d_{31}d_{32}d_{33}\dots \\ &\vdots \end{aligned}$$

Построим число $r = 0.e_1e_2e_3\dots$, цифры которого определяются по диагонали:

$$e_i = \begin{cases} 4 & \text{если } d_{ii} \neq 4 \\ 5 & \text{если } d_{ii} = 4 \end{cases}$$

Цифры 4 и 5 вместо 0 и 9 выбраны намеренно. У числа с хвостом из нулей или девяток есть альтернативная запись, $0.4999\dots = 0.5000\dots$, поэтому «отличается в i -й цифре» не всегда означало бы «число другое». У r , составленного из одних четвёрок и пятёрок, второй записи нет, и отличие в i -й цифре действительно означает отличие чисел.

Тогда $r \neq r_i$ для каждого i : в первой цифре, во второй, и так далее. Следовательно, r не совпадает ни с одним членом списка — противоречие.

Интервал $(0, 1)$ несчётен. На всю прямую результат переносится биекцией $x \mapsto \frac{1}{1+e^{-x}}$, отображающей $\mathbb{R}$ на $(0, 1)$: перечислимость $\mathbb{R}$ означала бы перечислимость интервала.

Примечание Почему 4 и 5

Пара цифр в диагональном правиле не уникальна: вместо 4 и 5 подойдут любые две различные цифры от 1 до 8, единственное требование — уйти от 0 и 9. Для битовых строк $\{0, 1\}^\infty$ такого запаса нет: обе цифры участвуют в двусмысленности $0.0111\dots = 0.1000\dots$. Поэтому диагональ для строк проводят инверсией бита: строка — не число, и двусмысленности записи у неё нет. Двусмысленность появляется лишь при отождествлении строк с числами, и там её снимает соглашение о единственной записи.

Пример Диагональ на конкретном списке

Возьмём пять чисел интервала $(0, 1)$:

$$\begin{aligned} r_1 &= 0.51842\dots \\ r_2 &= 0.27369\dots \\ r_3 &= 0.90411\dots \\ r_4 &= 0.65733\dots \\ r_5 &= 0.14285\dots \end{aligned}$$

Диагональные цифры — 5, 7, 4, 3, 5. Цифры числа r строятся по тому же правилу, что в доказательстве:

$$e_i = \begin{cases} 4 & \text{если } d_{ii} \neq 4 \\ 5 & \text{если } d_{ii} = 4 \end{cases}$$

Получаем

$$r = 0.44544...$$

Число r отличается от r_1 в первой цифре: $e_1 = 4 \neq 5 = d_{11}$. От r_2 — во второй: $e_2 = 4 \neq 7 = d_{22}$. Аналогично $r \neq r_3, r \neq r_4, r \neq r_5$. Список из пяти чисел неполон: число r в него не попало, хотя список выглядел правдоподобно. Диагональная конструкция работает для списка любой длины, в том числе бесконечного.

Диагональ хорошо видна на рис. 20: подсвеченные разряды дают число r , которое отличается от каждого r_i .

r_1	5	1	8	4	2	6	9
r_2	2	7	3	6	9	1	4
r_3	9	0	4	1	1	7	3
r_4	6	5	7	3	3	8	0
r_5	1	4	2	8	5	7	1

$r = 0.$ $\neq$ $\neq$ $\neq$ $\neq$ $\neq$... $\neg \in \{r_1, r_2, \dots\}$
4 4 5 4 4

Рис. 20. Диагональный аргумент Кантора.

Пример Несчётное множество

Несчётна не только вся прямая, но и каждый её маленький кусочек: интервал $(0, 1)$, только что разобранный диагональю, равномошен $\mathbb{R}$. Размер бесконечного множества не связан с геометрической протяжённостью — бесконечная прямая и конечный отрезок вмещают одинаковое число точек.

ЛОВУШКА Наивное перечисление не перечисляет

Соблазнительная, но неверная попытка перечислить интервал $(0, 1)$: сначала все числа с одной цифрой после запятой, затем с двумя, с тремя, и так далее. Кажется, что каждое число рано или поздно появится в списке. Дыра в том, что перечисление захватывает только числа с *конечным* десятичным разложением — а это лишь счётная часть рациональных чисел. Число $0.1010010001\dots$ имеет бесконечное разложение и никогда не будет выписано целиком ни на одном шаге. Единицы стоят на позициях 1, 3, 6, 10, Конечные разложения не покрывают интервал $(0, 1)$ — именно поэтому нужен диагональный аргумент.

Вернёмся ко второй истории из заметки выше — о биекции между отрезком и квадратом. Вот формальное утверждение и конструкция, которую Кантор предъявил Дедекинду: её схема приведена на рис. 21.

ТЕОРЕМА 7.3 **Равномощность отрезка и квадрата**

Единичные отрезок и квадрат равномощны: $L = [0, 1]$ и $S = [0, 1]^2$.

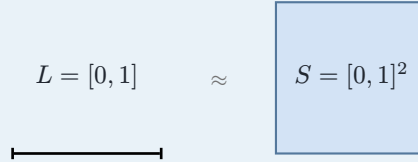


Рис. 21. Схема биекции отрезка на квадрат.

Равномощность сохраняется при переходе к любой конечной размерности: $|\mathbb{R}| = |\mathbb{R}^2| = |\mathbb{R}^n|$ для любого конечного n .

Доказательство

Докажем через теорему Кантора–Бернштейна–Шрёдера: построим инъекции в обе стороны.

Инъекция $L \rightarrow S, x \mapsto (x, 0)$ тривиальна: отрезок уходит в нижнюю сторону квадрата.

Инъекция $S \rightarrow L$ — чередование десятичных цифр, применённое ко внутренности квадрата. Граница не мешает: $|(0, 1)| = |(0, 1)|$ (инъекции в обе стороны: включение и сдвиг концов внутрь, теорема Кантора–Бернштейна–Шрёдера — ниже в этой главе). Поэтому $|[0, 1]^2| = |(0, 1)^2|$. Достаточно отождествить $(0, 1)^2$ с частью интервала.

Точке внутренности квадрата соответствует пара координат (x, y) , где

$$x = 0.x_1x_2x_3\dots$$

$$y = 0.y_1y_2y_3\dots,$$

обе координаты строго между нулём и единицей. Построим число

$$z = 0.x_1y_1x_2y_2x_3y_3\dots,$$

чередую цифры x и y . Это число лежит в интервале $(0, 1)$. Обратно, $z \rightarrow (x, y)$, где первая координата — нечётные позиции, а вторая — чётные.

Чтобы отображение было инъективным, нужно обойти неоднозначность десятичных разложений: $0.0999\dots = 0.1000\dots$ — одно и то же число. Для этого достаточно зафиксировать соглашение: всегда использовать разложение, не заканчивающееся бесконечной последовательностью девяток. Координаты лежат строго между нулём и единицей, поэтому граничные значения вроде $x = 1$ в отображении не участвуют. Для исключённых граничных значений запись

$0.x_1x_2\dots$ не нужна, и разложение единственно на всей области. Тогда чередование цифр даёт инъекцию $S \rightarrow L$.

По теореме Кантора–Бернштейна–Шрёдера (формулировка и доказательство — ниже в этой главе), $|L| = |S|$. Индукцией по размерности получаем $|\mathbb{R}| = |\mathbb{R}^n|$ для любого конечного n .

Диагональный аргумент появляется в нескольких теоремах, образующих структурный «скелет» современной математики. Он же лежит в основе доказательства неразрешимости проблемы остановки (глава 26) и первой теоремы Гёделя о неполноте. В обоих случаях используется та же диагональная конструкция — построение объекта, отличающегося от каждого элемента предполагаемого перечисления. Форма диагонализации в каждом доказательстве своя.

§ 7.4 Теорема Кантора

Вот та же конструкция в общем виде — для любого множества, не только для чисел.

ТЕОРЕМА 7.4 Теорема Кантора

Для любого множества A

$$|A| < |\mathcal{P}(A)|.$$

Булеан множества всегда строго больше самого множества.

Доказательство

Докажем два утверждения: инъекция в булеан существует, а сюръекции — нет. Вместе они дают строгое неравенство.

Инъекция $A \rightarrow \mathcal{P}(A)$, $a \mapsto \{a\}$ тривиальна.

Покажем, что не существует сюръекции.

Пусть $f : A \rightarrow \mathcal{P}(A)$ — произвольная функция. Определим

$$D = \{a \in A \mid a \notin f(a)\}$$

— «диагональное» множество элементов, не принадлежащих своему образу.

Предположим, что $\exists d \in A : D = f(d)$. По определению множества D получаем $d \in D \Leftrightarrow d \notin f(d)$. Но $f(d) = D$, откуда $d \in D \Leftrightarrow d \notin D$ — противоречие.

Следовательно, D не лежит в образе отображения, и f не сюръективна.

Булеан любого множества имеет строго большую мощность, чем само множество.

Пример Диагональное множество на конечном примере

Пусть $A = \{1, 2, 3\}$, а функция $f : A \rightarrow \mathcal{P}(A)$ задана образами $f(1) = \{2, 3\}$, $f(2) = \{1, 3\}$, $f(3) = \{1, 2\}$. Диагональное множество $D = \{a \in A \mid a \notin f(a)\}$: ни один элемент не принадлежит своему образу, поэтому $D = \{1, 2, 3\} = A$. В образе f лежат только двухэлементные множества: $f(A) = \{\{2, 3\}, \{1, 3\}, \{1, 2\}\}$. Множества A среди них нет, значит, D не покрыт — f не сюръективна.

Тот же приём работает для любого A : доказательство теоремы строит D по произвольной функции, не глядя на её конкретные значения.

Если позволить A быть «множеством всех множеств», диагональное множество D становится множеством всех множеств, не содержащих себя: это парадокс Рассела, записанный той же диагональю. Допустим на минуту универсум V , собирающий все множества: тогда $\mathcal{P}(V)$ было бы подмножеством V и не могло бы быть строго больше, как требует теорема Кантора. Универсального множества поэтому не существует, а полный разбор парадокса сделан в главе о множествах (4).

Этот результат порождает бесконечную иерархию мощностей:

$$|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| < |\mathcal{P}(\mathcal{P}(\mathbb{N}))| < \dots$$

Теорема Кантора утверждает, что булеан строго больше исходного множества, но не говорит *насколько* больше. Для конечных множеств ответ даёт комбинаторика.

ТЕОРЕМА 7.5 Мощность булеана конечного множества

Для конечного множества A : если $|A| = n$, то

$$|\mathcal{P}(A)| = 2^n.$$

Доказательство

Докажем подсчётом независимых выборов.

Каждое подмножество $S \subseteq A$ однозначно задаётся выбором. Для каждого элемента мы либо включаем его в S , либо нет. n независимых бинарных выборов дают $2 \times 2 \times \dots \times 2 = 2^n$ различных подмножеств.

Другой взгляд на то же доказательство — подмножеству взаимно-однозначно соответствует характеристическая функция $\chi_S : A \rightarrow \{0, 1\}$:

$$\chi_S(a) = \begin{cases} 1 & \text{если } a \in S \\ 0 & \text{если } a \notin S \end{cases}$$

Множество *всех* функций из n -элементного множества в двухэлементное имеет мощность 2^n по комбинаторному правилу произведения. Поэтому $|\mathcal{P}(A)| = |\{0, 1\}^A| = 2^{|A|}$.

Конечный случай этого соответствия — диаграмма Хассе на рис. 22.

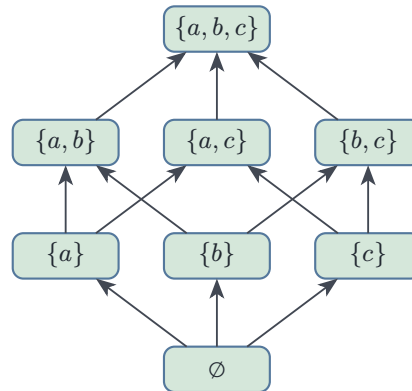


Рис. 22. Диаграмма Хассе для $\mathcal{P}(\{a, b, c\})$, упорядоченного по включению.

Для бесконечного A буквально перемножить двойки нельзя, но конструкция остаётся той же. Множество всех подмножеств — это в точности множество всех функций в $\{0, 1\}$. Поэтому *кардинальная степень* определяется как $2^\kappa = |\{0, 1\}^\kappa|$. Это мощность множества всех функций из множества размера κ в двухэлементное множество. При таком определении равенство $|\mathcal{P}(A)| = 2^{|A|}$ выполняется для любого множества — и конечного, и бесконечного.

Проверка Счётность и теорема Кантора

1. Почему инъекция $a \mapsto \{a\}$ доказывает только неравенство?
2. Что добавляет диагональное множество D ?
3. Почему из теоремы Кантора следует, что бесконечных мощностей бесконечно много?

§ 7.5 Теорема Кантора–Бернштейна–Шрёдера

У нас есть способ сказать «множества одинакового размера» (биекция) и «одно множество не больше другого» (инъекция). Естественный вопрос: если $|A| \leq |B| \leq |A|$, то одинаковы ли множества по размеру? Для конечных множеств ответ — тривиальное «да». Для бесконечных это требует отдельного доказательства.

ОПРЕДЕЛЕНИЕ 7.6 Нестрогое сравнение мощностей

Мощность множества A не превосходит мощности множества B , обозначается $|A| \leq |B|$, если существует инъекция $A \rightarrow B$.

ОПРЕДЕЛЕНИЕ 7.7 Строгое сравнение мощностей

Мощность множества A строго меньше мощности множества B , обозначается $|A| < |B|$, если инъекция $A \rightarrow B$ существует, а биекции нет.

Пример Сравнение мощностей

Тождественная инъекция даёт $|\mathbb{N}| \leq |\mathbb{R}|$. Диагональный аргумент Кантора показывает, что биекции нет, то есть $|\mathbb{N}| < |\mathbb{R}|$.

На конечных множествах $|A| \leq |B|$ — обычное сравнение числа элементов. Инъекция существует ровно тогда, когда $|A| \leq |B|$: элементов в A не больше, чем в B .

Сравнение мощностей транзитивно: композиция инъекций $A \rightarrow B \rightarrow C$ снова инъекция. Антисимметричность — отдельное утверждение: если инъекции идут в обе стороны, биекция существует. Именно это и доказывает теорема Кантора–Бернштейна–Шрёдера.

ТЕОРЕМА 7.6 Кантор–Бернштейн–Шрёдер

Если $|A| \leq |B| \leq |A|$, то

$$|A| = |B|.$$

Доказательство

Докажем, построив биекцию h из двух инъекций.

Идея — разбить элементы $A \cup B$ на цепочки (предыстории) и задать h по-разному в зависимости от категории цепочки. Пусть $f : A \rightarrow B, g : B \rightarrow A$ — инъекции. Требуется построить биекцию $h : A \rightarrow B$.

Рассмотрим орбиты элементов под действием попеременного применения f и g . Для $a \in A$ проследим его «предысторию». Если $a \notin g(B)$, цепочка обрывается. Иначе $\exists b_1 \in B : g(b_1) = a$, и дальше — тот же выбор: обрыв или шаг назад, и так далее.

Точно так же устроена предыстория элемента $b \in B$. Шаги назад: $f^{-1}, g^{-1}, f^{-1}, \dots$ — пока возможно. Одно и то же слово «цепочка» удобно использовать для элементов обоих множеств: цепочка элемента — его предыстория, продолженная в обе стороны.

Каждый элемент $A \cup B$ попадает в одну из трёх категорий: обрыв в A (в предыстории есть элемент $A \setminus g(B)$), обрыв в B (в предыстории есть элемент $B \setminus f(A)$) или бесконечная цепочка (обрыва нет ни в одну сторону).

Первые две категории не пересекаются: иначе в одной цепочке нашлись бы элементы $A \setminus g(B)$ и $B \setminus f(A)$, но отрезок между ними невозможен, потому что на одном конце шаг назад требует $a \in g(B)$, а на другом — $b \in f(A)$.

Определим $h : A \rightarrow B$ так: при обрыве в A или бесконечной цепочке положим $h(a) = f(a)$, а при обрыве в B — $h(a) = g^{-1}(a)$. Обратное отображение g^{-1} определено на образе g , и здесь $g^{-1}(a)$ действительно определён: в цепочке есть

элемент $B \setminus f(A)$ (обрыв в B), и, двигаясь от него вперёд, мы доходим до a , причём каждый шаг вперёд из B имеет вид $g(b')$, откуда $a \in g(B)$.

Категория элемента не меняется вдоль цепочки. Шаг вперёд добавляет к предыстории ровно сам элемент. Инъективность g гарантирует единственность прообраза. Обрыв в той же стороне сохраняется, аналогично для шага g .

Проверим, что h — биекция.

Инъективность: f инъективна. Функция g^{-1} тоже. Образы f и g^{-1} не пересекаются. Если бы $f(a_1) = g^{-1}(a_2)$, элементы были бы в одной цепочке. Категория вдоль цепочки не меняется, тогда как они из разных категорий — противоречие.

Сюръективность: для каждого $b \in B$ рассмотрим его цепочку. Если она обрывается в A или бесконечна, то $b \in f(A)$: иначе предыстория b не продолжилась бы назад (шаг f^{-1} невозможен) и цепочка оборвалась бы в B . Тогда $b = f(a)$ для некоторого a из той же категории, и $h(a) = b$. Если же цепочка b обрывается в B , то $a = g(b)$ имеет цепочку той же категории, и $h(a) = g^{-1}(g(b)) = b$.

Теорема Кантора–Бернштейна–Шрёдера — инструмент, которым пользуются постоянно. Чтобы доказать равномощность двух множеств, не нужно строить явную биекцию. Достаточно построить инъекции в обе стороны.

Например, $|(0, 1)| = |[0, 1]|$:

- Инъекция $(0, 1) \rightarrow [0, 1]$ тривиальна (включение).
- Обратная инъекция $[0, 1] \rightarrow (0, 1)$, $x \mapsto \frac{x}{2} + \frac{1}{4}$: сдвиг концов внутрь.

ИСТОРИЯ Три имени — одна теорема

Георг Кантор сформулировал утверждение в 1883 году, но не смог его доказать. Феликс Бернштейн, девятнадцатилетний студент Кантора, доказал теорему в 1897 году, и Кантор немедленно включил доказательство в свой курс. Независимо от Бернштейна, теорему доказал Фридрих Шрёдер (1896), но его доказательство содержало пробел — он предполагал, что классы эквивалентности сравнимы без доказательства. Дедекиннд также нашёл доказательство в 1887 году (не опубликовано при жизни, обнаружено в архивах).

Теорема носит три имени, хотя каждое из них имеет разный статус: Кантор — постановщик, Бернштейн — автор первого корректного доказательства, Шрёдер — автор независимой, но ошибочной попытки. Название «теорема Кантора–Бернштейна» распространено в англоязычной литературе. В немецкой традиции добавляют Шрёдера.

§ 7.6 Кардинальные числа и арифметика

Мы говорили о равномощности и счётности, не дав формального имени самим «размерам» бесконечных множеств — время это исправить.

ОПРЕДЕЛЕНИЕ 7.8 Алеф-нуль

Мощность множества натуральных чисел обозначается $\aleph_0$ (алеф-нуль).

ОПРЕДЕЛЕНИЕ 7.9 Континуум

Мощность множества действительных чисел называется **континуумом**, обозначается $2^{\aleph_0}$.

ОПРЕДЕЛЕНИЕ 7.10 Кардинальное число

Кардинальным числом называется класс эквивалентности всех множеств, равномощных данному.

Равенство $|\mathbb{R}| = |\mathcal{P}(\mathbb{N})| = 2^{\aleph_0}$ — не часть определения, а утверждение о мощностях. Соответствие «подмножество $\mathbb{N}$ — битовая строка — двоичная дробь», на котором оно держится, строится ниже, в примере о подмножествах, строках и числах.

§ 7.7 Две иерархии: алефы и беты

Мы видели две лестницы бесконечностей, но не различали их. Обе начинаются с $\aleph_0 = |\mathbb{N}|$ и обе растут неограниченно, но устроены по-разному. *Бет-иерархия* строится булеаном, *алеф-иерархия* — последовательным шагом «следующий кардинал».

ОПРЕДЕЛЕНИЕ 7.11 Бет-числа

Бет-числа $\beth_0, \beth_1, \beth_2, \dots$ задаются рекуррентно через булеан:

$$\beth_0 = \aleph_0, \quad \beth_{n+1} = 2^{\beth_n}, \quad \beth_\lambda = \sup_{\alpha < \lambda} \beth_\alpha.$$

ОПРЕДЕЛЕНИЕ 7.12 Алеф-числа

Алеф-числа $\aleph_0, \aleph_1, \aleph_2, \dots$ — все бесконечные кардиналы, перечисленные по порядку возрастания:

- $\aleph_0$ — наименьший бесконечный кардинал;
- $\aleph_{\alpha+1}$ — наименьший кардинал, строго больший $\aleph_\alpha$;
- на предельном шаге $\aleph_\lambda = \sup_{\alpha < \lambda} \aleph_\alpha$.

На рис. 23 обе лестницы показаны рядом: верхняя ветвь — алефы, нижняя — беты.

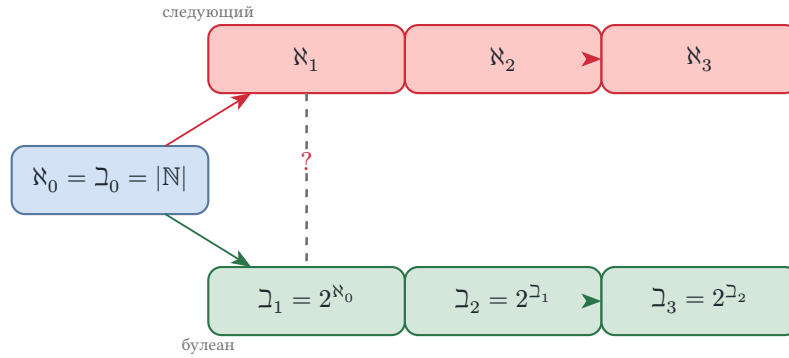


Рис. 23. Две иерархии: алефы и беты.

Различие — в том, как определяется следующий уровень. Беты дают *конструктивную* лестницу: на каждом этапе мы знаем точное множество — булеан предыдущего, и его мощность задана явно. Алефы дают *последовательную* лестницу: $\aleph_1$ — это просто «наименьший кардинал больше $\aleph_0$ », но как он устроен, мы не знаем. Алефы перечисляют все кардиналы по порядку (это обеспечивается упорядочиванием, доступным при аксиоме выбора), беты — только те, что получаются булеаном.

ПРЕДУПРЕЖДЕНИЕ

С двух иерархий начинается континуум-гипотеза. На первом шаге: $\beth_1 = 2^{\aleph_0} = |\mathbb{R}|$. А $\aleph_1$ — наименьший кардинал строго больше $\aleph_0$. Совпадение $\aleph_1 = \beth_1$ — в точности предмет континуум-гипотезы, формулируемой ниже.

ОПРЕДЕЛЕНИЕ 7.13 Континуум-гипотеза (CH)

Континуум-гипотеза утверждает, что не существует кардинала, строго промежуточного между $\aleph_0$ и континуумом:

$$\aleph_1 = \beth_1 = 2^{\aleph_0} = |\mathbb{R}|.$$

Эквивалентно: всякое бесконечное подмножество $\mathbb{R}$ либо счётно, либо равномощно $\mathbb{R}$.

ОПРЕДЕЛЕНИЕ 7.14 Обобщённая континуум-гипотеза (GCH)

Обобщённая континуум-гипотеза — то же для каждой ступени: возведение в степень всегда делает следующий шаг алеф-иерархии,

$$2^{\aleph_\alpha} = \aleph_{\alpha+1}$$

для каждого ординального индекса α . Эквивалентно: для любого бесконечного кардинала κ выполняется $2^\kappa = \kappa^+$, где κ^+ — наименьший кардинал, строго больший κ . На всех уровнях это значит, что алеф- и бет-иерархии совпадают: $\aleph_\alpha = \beth_\alpha$ для каждого α .



Бет — это просто булеан: как только вы умеете считать булеан, вы умеете считать беты. Алеф — это «следующий»: как только вы умеете упорядочивать кардиналы, вы умеете считать алефы. Две иерархии совпадают тогда и только тогда, когда всякий кардинал — булеан предыдущего уровня, то есть когда в упорядочении кардиналов нет «пропусков». CH — это утверждение, что первый пропуск не существует уже на первом шаге. GCH — что не существует нигде.

ПРЕДУПРЕЖДЕНИЕ

Ни CH , ни GCH нельзя ни доказать, ни опровергнуть в ZFC : Коэн показал независимость CH от ZFC . Истинность CH и GCH — вопрос выбора аксиоматики, а не вывода. Это первый великий результат о том, что аксиомы множеств сами не фиксируют размер континуума.

Пример Кардиналы конкретных множеств

- $|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{Q}| = \aleph_0$ — все эти множества счётны.
- $|\mathbb{R}| = |\mathcal{P}(\mathbb{N})| = 2^{\aleph_0}$ — континуум.
- $|\mathbb{N} \times \mathbb{N}| = \aleph_0$ (декартово произведение двух счётных множеств счётно).
- $|\mathcal{P}(\mathbb{R})| = 2^{2^{\aleph_0}}$ — мощность булеана действительных чисел. Это уже третий этаж иерархии.

Пример Подмножества, строки и числа

Множество простых чисел $P = \{2, 3, 5, 7, 11, \dots\}$ — подмножество $\mathbb{N}$. Его характеристическая функция $\chi_P : \mathbb{N} \rightarrow \{0, 1\}$ равна 1 на простых числах и 0 на остальных. Позиция n последовательности отвечает числу n (нумерация с единицы):

$$\chi_P = 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, \dots$$

Первые члены — $\chi_P(1) = 0, \chi_P(2) = 1$: единица не простая, двойка — простая. Припишем к строке слева «0,» и прочитаем её как двоичную дробь:

$$0.01101010001\dots_2$$

Подмножество $\mathbb{N}$ превратилось в число отрезка $[0, 1]$. Обратный ход: каждая двоичная дробь задаёт подмножество $\mathbb{N}$ — единица на позиции n означает, что n лежит в множестве. Двоичная запись обрабатывается соглашением: $0.1000\dots = 0.0111\dots$ — одно число, и для каждого числа фиксируется ровно одно разложение. Так строится соответствие $|\{0, 1\}^{\mathbb{N}}| = |[0, 1]| = |\mathbb{R}|$: подмножеств натуральных чисел ровно континуум.

Арифметика бесконечных кардиналов устроена иначе, чем арифметика натуральных чисел: сложение и умножение перестают увеличивать мощность, тогда как

возведение в степень — единственная операция, сохраняющая способность порождать новые кардиналы.

Напомним, что значит складывать и умножать кардиналы. Если $|A| = \kappa$ и $|B| = \lambda$, то по определению: $\kappa + \lambda = |A \sqcup B|$ (дизъюнктное объединение), $\kappa \cdot \lambda = |A \times B|$, $\kappa^\lambda = |A^B|$ — мощность множества всех функций из B в A . Результат не зависит от того, какие множества A и B данной мощности мы выбрали.

Возьмём два бесконечных коридора с комнатами, пронумерованными $0, 1, 2, \dots$. Постояльцы двух коридоров образуют множества A и B соответственно. Чтобы разместить всех в одном коридоре (то есть доказать $\aleph_0 + \aleph_0 = \aleph_0$), достаточно расселить постояльцев первого коридора в нечётные комнаты, а второго — в чётные. Места хватит всем, и никого не придётся выселять. Бесконечность *поглощает* конечные манипуляции: добавление, удвоение, возведение в квадрат не меняют счётной мощности.

- $\aleph_0 + \aleph_0 = \aleph_0$: объединение двух счётных множеств счётно.
- Произведение счётных множеств счётно: $\aleph_0 \cdot \aleph_0 = \aleph_0$. Декартово произведение $\mathbb{N} \times \mathbb{N}$ перечисляется по сумме индексов. Тот же диагональный аргумент, что и для $\mathbb{Q}$.
- $2^{\aleph_0} = |\mathbb{R}|$: булеан натуральных чисел имеет мощность континуума — соответствие «подмножество — битовая строка — двоичная дробь» разобрано в примере выше.
- При АС кардинальное сложение и умножение для любых бесконечных кардиналов сводятся к взятию максимума: $\kappa + \lambda = \kappa \cdot \lambda = \max(\kappa, \lambda)$.

Примечание Цена правила максимума

За равенство $\kappa \cdot \kappa = \kappa$ для произвольного бесконечного кардинала платят аксиомой выбора. Это теорема Хессенберга (1906): её доказательство вполне упорядочивает $A \times A$ и по индукции вдоль этого порядка проверяет, что каждый начальный сегмент мал. Без выбора встречаются бесконечные множества, не равномощные никакому собственному подмножеству (дедекиндово-конечные), и для них $\kappa + 1 > \kappa$: поглощение из отеля Гильберта перестаёт быть общим законом. Для $\aleph_0$ аксиома выбора не нужна: сдвиги и диагональное перечисление строят все нужные биекции явно.

Пример Сколько функций из $\mathbb{N}$ в $\mathbb{N}$

Возведение в степень отвечает на вопрос, сколько функций из B в A . Функций из $\mathbb{N}$ в $\mathbb{N}$ — ровно $\aleph_0^{\aleph_0}$. Оценим эту мощность с двух сторон:

$$2^{\aleph_0} \leq \aleph_0^{\aleph_0} \leq (2^{\aleph_0})^{\aleph_0} = 2^{\aleph_0 \cdot \aleph_0} = 2^{\aleph_0}.$$

Первое неравенство: функций в $\{0, 1\}$ не больше, чем функций в $\mathbb{N}$, ведь $\{0, 1\} \subseteq \mathbb{N}$. Второе: функций из $\mathbb{N}$ в $\mathbb{N}$ не больше, чем функций из $\mathbb{N}$ в $2^{\aleph_0}$, а $(2^{\aleph_0})^{\aleph_0} = 2^{\aleph_0 \cdot \aleph_0} = 2^{\aleph_0}$, где произведение $\aleph_0 \cdot \aleph_0 = \aleph_0$ уже доказано диагональным пере-

числением пар. По теореме Кантора–Бернштейна–Шрёдера $\aleph_0^{\aleph_0} = 2^{\aleph_0}$: функций из $\mathbb{N}$ в $\mathbb{N}$ — континуум, столько же, сколько действительных чисел.



В кардинальной арифметике (при AC) сложение и умножение бесконечных кардиналов сводятся к максимуму. Эти операции не создают новых кардиналов. Возведение в степень — единственная операция, создающая новый кардинал ($2^\kappa > \kappa$). И оно же — единственное, чьё конкретное значение ZFC не фиксирует (теорема Истона). Странно, что производство новых кардиналов не регулируется аксиоматикой, а операции, ничего не производящие, — регулируются...

Диагональный аргумент — прямой путь к теории вычислимости. Каждая машина Тьюринга — модель вычисления, которую определяет глава 25 — кодируется конечной строкой символов: её таблицей переходов. Конечных строк над конечным алфавитом счётное число — $\aleph_0$. Следовательно, существует лишь счётное число машин Тьюринга, и каждая распознаёт ровно один язык. Но языков — подмножеств Σ^* , множества всех строк над алфавитом Σ , знакомого по автоматам из главы 23 — континуум. Итак: разрешимых и даже распознаваемых языков $\aleph_0$, а всех языков — континуум.

Почти все языки не только не разрешимы, но и не распознаваемы — неразрешимость становится *нормой*, а вычислимость — *редким исключением*. Этот факт мы докажем в главе 26.

§ 7.8 * Недостижимые кардиналы

Лестница кардиналов не обрывается: по теореме Кантора $2^\kappa > \kappa$ для любого кардинала, поэтому

$$\aleph_0 < 2^{\aleph_0} < 2^{2^{\aleph_0}} < \dots$$

Всю лестницу строят две операции: булеан перескакивает с κ на 2^κ , а переход к следующему кардиналу даёт κ^+ . Существует ли кардинал, до которого обе операции, применённые снизу, добраться не смогут?

Несчётный кардинал κ называется *недостижимым*, если выполняются два условия. κ — сильный предел: $2^\lambda < \kappa$ для всякого $\lambda < \kappa$, то есть булеан любого меньшего кардинала не дотягивается до κ . κ регулярен: его нельзя представить как объединение менее чем κ множеств мощности меньше κ . Для контраста, $\aleph_\omega$ нерегулярен: он представим объединением счётной цепочки $\aleph_0, \aleph_1, \aleph_2, \dots$ — меньше $\aleph_\omega$ множеств, каждое меньше $\aleph_\omega$. Ни степени, ни объединения не выводят с этажей ниже κ на сам κ — отсюда и имя.

Само $\aleph_0$ формально удовлетворяет обоим условиям: конечные степени конечных чисел конечны, а конечное объединение конечных множеств конечно. Поэтому условие несчётности существенно: недостижимый кардинал — это несчётный двойник $\aleph_0$, первая точка, где бесконечный мир воспроизводит собственную неприступность.

Существование недостижимого кардинала в ZFC доказать нельзя. Начальный сегмент универсума высоты κ — множество V_κ — для недостижимого κ сам служит моделью ZFC, так что из аксиомы «недостижимый кардинал существует» следует непротиворечивость ZFC. По второй теореме Гёделя о неполноте ZFC не доказывает собственную непротиворечивость, значит, не доказывает и существование недостижимого кардинала.

Недостижимый кардинал — первая ступень *иерархии больших кардиналов*: аксиом о всё более крупных кардиналах (кардинал Мало, слабокомпактный, измеримый и далее), каждая из которых строго сильнее предыдущей по доказательной силе. Ни одна из этих аксиом не выводится из ZFC, а их относительная непротиворечивость проверяется только через ещё более сильные аксиомы той же иерархии. Как и с континуум-гипотезой, вопрос уходит в выбор аксиоматики. Аксиоматический статус больших кардиналов разбирается в главе 22.

§ 7.9 Итоги главы

Определение размера через биекцию превращает вопрос о том, сколько элементов в множестве, в точную задачу сравнения множеств. Два множества равномощны, если между ними есть биекция. Для конечных множеств это совпадает с обычным подсчётом, а для бесконечных — ведёт к выводам, расходящимся с интуицией. Часть может равняться целому: натуральных чисел столько же, сколько чётных, а заполненный отель Гильберта принимает ещё целый автобус гостей и остаётся заполненным.

Мы убедились, что плотно заполняющие прямую рациональные числа счётны — их можно занумеровать диагональным перечислением пар — а действительные уже нет. Диагональный аргумент Кантора доказывает несчётность $\mathbb{R}$ и открывает лестницу кардиналов $\aleph_0 < 2^{\aleph_0} < \dots$. Теорема Кантора–Бернштейна–Шрёдера позволяет сравнивать мощности, не строя явных биекций. Булеан $\mathcal{P}(A)$ всегда строго больше исходного множества.

Кардинальная арифметика свела сложение и умножение мощностей к максимуму — $\aleph_0 + \aleph_0 = \aleph_0$ — и только возведение в степень рождает следующий кардинал. Полная картина — алеф- и бет-иерархии, континуум-гипотеза и недостижимые кардиналы — складывается в главе 22.

Единый приём — диагональ — работает и за пределами меры: приложенный к вычислимости, он показывает, что множество всех языков мощнее множества всех программ. Программ существует лишь счётное число, а языков — континуум, и

это различие мощностей станет отправной точкой главы 26. Сравнение отвечает на вопрос, сколько элементов в множестве. Вопрос о том, как они устроены, мы переводим в отношения порядка.

Проверка Проверьте себя

1. Почему $\mathbb{Q}$ счётно, если счётны целые числа? Как диагональное перечисление пар помогает занумеровать рациональные числа?
2. В чём суть диагонального аргумента Кантора и почему построенное число не может лежать в перечислении?
3. Почему равномощность определяется через биекцию, а не через включение?
4. Что гарантирует теорема Кантора–Бернштейна–Шрёдера и почему она избавляет от построения явных биекций?
5. Почему множество всех программ счётно, а множество всех языков — нет?
6. Почему $\aleph_0 + \aleph_0 = \aleph_0$, но $2^{\aleph_0} > \aleph_0$?
7. В чём различие алеф- и бет-иерархий и что утверждает континуум-гипотеза? Ответьте и на обобщённую континуум-гипотезу.

Что дальше?

Мы научились измерять размер бесконечных множеств. Теперь мы обратимся к структуре — к отношениям порядка. В главе 8 мы изучим частичные порядки, диаграммы Хассе и решётки — алгебраические структуры, в которых порядок и операции переплетаются.

§ 7.10 Упражнения

Равномощность.

1. Докажите, что множество чётных натуральных чисел равномощно $\mathbb{N}$. Приведите явную биекцию и обратную к ней.
2. Покажите, что произведение $A \times B$ счётных множеств счётно. Используйте диагональное перечисление пар, показанное на рис. 19.
3. * Постройте биекцию $\mathbb{N} \rightarrow \mathbb{Z}$. Докажите, что построенное отображение действительно биективно.

Счётные множества.

4. Докажите, что множество всех конечных подмножеств $\mathbb{N}$ счётно. Подсказка: каждое конечное подмножество можно закодировать конечной битовой строкой.
5. Покажите, что множество всех многочленов с целыми коэффициентами счётно. Используйте этот факт, чтобы доказать существование трансцендентных чисел (чисел, не являющихся корнем никакого многочлена с целыми коэффициентами).

Несчётные множества и диагональный аргумент.

6. Покажите, что множество всех бесконечных битовых строк $\{0, 1\}^\infty$ несчётно. Используйте диагональный аргумент.
7. Докажите, что $|\mathbb{N}^{\mathbb{N}}|$ — множество всех функций из натуральных в натуральные — несчётно. Сравните с доказательством несчётности $\{0, 1\}^\infty$.
8. * Докажите, что $|\mathbb{R}| = |\mathcal{P}(\mathbb{N})|$. Постройте инъекции в обе стороны. Аккуратно обработайте неоднозначность двоичного представления (например, $0.0111\dots = 0.1000\dots$).

Теорема Кантора.

9. Докажите теорему Кантора: $|A| < |\mathcal{P}(A)|$ для любого множества A . Почему из неё следует, что существует бесконечно много различных бесконечных мощностей?
10. * Докажите, что $|\mathbb{R}^2| = |\mathbb{R}|$. Подсказка: постройте инъекцию квадрата $(0, 1)^2$ в интервал чередованием десятичных цифр и примените теорему Кантора–Бернштейна–Шрёдера.

Теорема Кантора–Бернштейна–Шрёдера.

11. Докажите через теорему Кантора–Бернштейна–Шрёдера, что $|(0, 1)| = |[0, 1]|$. Постройте явные инъекции в обе стороны.

Кардинальная арифметика и вычислимость.

12. Объясните, почему $\aleph_0 + \aleph_0 = \aleph_0$ и $\aleph_0 \cdot \aleph_0 = \aleph_0$. Приведите соответствующие биекции.
13. * Множество всех машин Тьюринга счётно (каждая кодируется конечной строкой). Множество всех языков имеет мощность $2^{\aleph_0}$. Языки — это подмножества Σ^* . Что из этого следует о соотношении разрешимых и неразрешимых языков?
14. * Найдите ошибку в «доказательстве» счётности интервала $(0, 1)$. Перечислим сначала числа с одной цифрой после запятой, затем с двумя, затем с тремя, и так далее. Каждое число из $(0, 1)$ имеет десятичную запись, значит, оно рано или поздно появится в перечислении. Какие числа никогда не появятся, и почему диагональный аргумент показывает, что $(0, 1)$ несчётно?

Две иерархии.

15. Выпишите первые четыре бет-числа и для каждого назовите конкретное множество соответствующей мощности. Например, что имеет мощность $\beth_2$?
16. Докажите, что $\aleph_0 \leq \beth_1$ и $\aleph_1 \leq \beth_1$. Почему обратное неравенство $\beth_1 \leq \aleph_1$ — в точности континуум-гипотеза?
17. * Покажите, что для любого n верно $\aleph_n \leq \beth_n$. Воспользуйтесь тем, что $\aleph_n$ — наименьший кардинал, больший всех $\aleph_k$ при $k < n$, а $\beth_{n+1} = 2^{\beth_n}$.

18. * Сформулируйте, что утверждает обобщённая континуум-гипотеза, и объясните, почему $\aleph_\omega = \sup_{n < \omega} \aleph_n$, а $\beth_\omega = \sup_{n < \omega} \beth_n$, и что из этого следует при GCH.

ПРОЕКТ **Диагонализатор: число вне списка**

Диагональный аргумент — конструкция, и её можно поручить программе: на вход подаётся перечисление десятичных разложений, на выходе программа строит число, которого в перечислении нет, и проверяет это построение. Соберите такой диагонализатор и посмотрите его глазами на выбор цифр из теоремы: чем безопасная пара отличается от опасной.

① **Разложения и каноническая запись**

Представьте числа интервала $(0, 1)$ последовательностями десятичных цифр после запятой. Реализуйте извлечение цифры номер i длинным делением дроби $\frac{p}{q}$: каждый шаг перевычисляет остаток, плавающая точка не участвует. Проверьте на дробях вроде $\frac{1}{2}$ и $\frac{1}{7}$, что цифры сходятся с известными разложениями, и зафиксируйте каноническую запись: длинное деление никогда не выдаёт хвоста девяток.

② **Диагональ**

Реализуйте построение r по правилу теоремы: цифра 4, если диагональная цифра не 4, иначе 5. Для конечного списка продолжите r четвёрками после исчерпания списка. Выведите первые разряды r для списков из главы и проверьте, что r отличается от каждого члена списка в своём разряде.

③ **Опасные цифры**

Переключите правило на пару 0 и 9 и подберите список, на котором построенное число совпадает с одним из членов списка по значению, хотя записи различаются. Научите программу распознавать период в цифровой последовательности и переводить $0.0999\dots$ в каноническое значение — без этого шага коллизия не видна. Сформулируйте, какое свойство пары 4 и 5 исключает такую коллизию.

④ **Диагональ на булеане**

Обобщите конструкцию на таблицу $f: A \rightarrow \mathcal{P}(A)$ для конечного A : программа вычисляет $D = \{a \in A \mid a \notin f(a)\}$ и проверяет, что D не лежит в образе f . Проведите полный перебор всех таблиц на множествах из двух и трёх элементов и убедитесь, что D не попадает в образ ни разу.

Итог: диагонализатор, который по любому поданному перечислению строит число вне списка, показывает обе ловушки опасной пары цифр — уход за границу интервала и коллизию под другой записью — и перебором подтверждает теорему Кантора на всех конечных таблицах.

VIII

Отношения порядка

Сборка крупного проекта сводится к вопросу о порядке: сотни модулей связаны зависимостями, и компилировать их можно только в совместимой последовательности. Одни модули обязаны быть готовы раньше других, другие не связаны вовсе и собираются параллельно. Та же картина в менеджере пакетов: версию библиотеки разрешено обновлять только до совместимой, а совместимость не всегда выстраивает версии в одну цепочку. Документ в системе контроля версий имеет предков в истории изменений, и у двух веток разработки общего предка может не быть. За всеми этими ситуациями стоит одна математическая структура — отношение порядка.

Сравнение чисел — знакомый пример: для любых двух натуральных чисел определено, какое из них не больше другого. Отношения эквивалентности из главы 5 отвечали на вопрос «одинаковы ли объекты» и собирали их в классы. Порядок отвечает на другой вопрос: что из чего следует и что раньше чего. Мощности из предыдущей главы измеряла множества целиком, а порядку нужно сравнение внутри множества. Для классификации хватает симметрии, для расписаний и зависимостей нужна направленность.

Частичный порядок допускает несравнимые пары: модули без общих зависимостей не сравнимы, и ни один из них не обязан идти раньше другого. Несравнимость при этом несёт точную информацию — она означает независимость. Независимые задачи планировщик запускает одновременно, а линейный порядок такой свободы не оставляет: в нём любые два объекта сравнимы и выстраиваются в одну очередь. Иерархия типов в языке программирования устроена так же: два класса-потомка одного родителя сами могут не быть связаны подтипированием.

У порядка есть и алгебраическая сторона. Если у каждой пары элементов существуют точная верхняя и точная нижняя грани, порядок вырастает в решётку — структуру с двумя операциями, устроенными наподобие объединения и пересечения множеств. Булевы алгебры, известные из логики высказываний, — частный случай решёток, а на числах решётку образуют натуральные числа с операциями наибольшего общего делителя и наименьшего общего кратного. В статическом анализе программ возможные значения переменной аппроксимируются элементом решётки, и слияние двух ветвей условного оператора выполняет супремум.

Как работать с порядком, в котором не всякая пара элементов сравнима, и какие задачи от сборки проекта до анализа программ это решает?

ИСТОРИЯ Гаусс и рождение структурной теории чисел

Карл Фридрих Гаусс опубликовал «Disquisitiones Arithmeticae»³⁹ в возрасте 24 лет. Эта книга заложила фундамент современной теории чисел — и, неявно, теории отношений порядка.

Гаусс ввёл сравнения по модулю: $a \equiv b \pmod{m} \leftrightarrow m \mid (a - b)$. Сегодня мы опознаём в этой конструкции отношение эквивалентности (глава 5) — но сам Гаусс мыслил в терминах арифметики, а не теории отношений. Сами понятия «отношение эквивалентности» и «класс эквивалентности» появятся лишь столетие спустя.

В той же работе Гаусс изучал делимость целых чисел — структуру, которую мы сегодня называем частичным порядком: $a \mid b$ рефлексивно, антисимметрично (на натуральных числах) и транзитивно. Он заметил, что не все числа сравнимы по делимости: 2 и 3 не делят друг друга — свойство, отличающее частичный порядок от линейного. Гаусс не использовал термина «частичный порядок». Он говорил о «решете Эратосфена», «квадратичных формах» и «теории сравнений». Но математическая структура, которую он исследовал, — это в точности наш частичный порядок делимости, а множество делителей числа с операциями НОД и НОК — прообраз решётки.

Это характерная черта математики XIX века: структурные понятия вызревают внутри конкретных теорий — чисел, уравнений, функций — прежде чем быть опознанными как самостоятельные абстракции. Гаусс не знал, что изучает отношение порядка. Но без «Disquisitiones Arithmeticae» теория порядков лишилась бы одного из своих опорных примеров.

ОБЗОР ГЛАВЫ

В этой главе мы разберём три аксиоматики — частичный, строгий и линейный порядок — и увидим, что несравнимость — инструмент: она означает независимость. Диаграммы Хассе сделают структуру видимой, а на экстремальных элементах, гранях, супремуме и инфимуме мы научимся находить «первые», «последние» и точные границы. Там, где у каждой пары есть супремум и инфимум, порядок вырастает в решётку. Топологическая сортировка развернёт частичный порядок в линейную последовательность, а теорема Кнастера–Тарского в конце главы гарантирует неподвижные точки монотонных функций. Весь аппарат применим к анализу программ — от диаграмм Хассе до абстрактной интерпретации.

После этой главы вы сможете:

1. определять частичный, строгий и линейный порядок и проверять их аксиомы;

³⁹C. F. Gauß. «Disquisitiones Arithmeticae». 1801.

2. строить диаграммы Хассе и находить экстремальные элементы, супремумы и инфимумы;
3. распознавать решётки и булевы алгебры среди частичных порядков;
4. применять топологическую сортировку для выстраивания зависимостей;
5. формулировать теорему Кнастера–Тарского и видеть её в абстрактной интерпретации программ.

§ 8.1 Частичный порядок

Отношение порядка — это формализация идеи «меньше или равно». На числах любые два элемента сравнимы. *Частичный порядок* допускает, что некоторые элементы несравнимы — ни один из них не меньше другого. Это ровно то обобщение, которое позволяет моделировать зависимости, иерархии и предпочтения.

ОПРЕДЕЛЕНИЕ 8.1 Частичный порядок (*partial order*)

Бинарное отношение $\preceq$ на множестве A называется **частичным порядком**, если оно:

- **Рефлексивно**: для всех $a \in A$

$$a \preceq a.$$

- **Антисимметрично**: если $a \preceq b, b \preceq a$, то

$$a = b.$$

- **Транзитивно**: если $a \preceq b, b \preceq c$, то

$$a \preceq c.$$

ОПРЕДЕЛЕНИЕ 8.2 ЧУМ (*poset*)

Пара $(A, \preceq)$, где $\preceq$ — частичный порядок на A , называется **ЧУМ** (частично упорядоченное множество).

Пример Проверка аксиом частичного порядка

Пусть $A = \{1, 2, 3, 6\}$, $a \preceq b \leftrightarrow a \mid b$. Проверим аксиомы:

- Рефлексивность: каждое число делит себя: $1 \mid 1, 2 \mid 2, 3 \mid 3, 6 \mid 6$.
- Антисимметричность: $a \mid b \wedge b \mid a \rightarrow a = b$ (для натуральных чисел это верно).
- Транзитивность: $a \mid b \wedge b \mid c \rightarrow a \mid c$ (свойство делимости).

Элементы 2 и 3 несравнимы: 2 не делит 3 и 3 не делит 2. Поэтому порядок частичный, а не линейный.

Определённый выше частичный порядок является нестрогим: он допускает равенство элемента с самим собой, как $\leq$ на числах. Для многих конструкций удобнее

строгая версия — аналог $<$, исключающий равенство. Переход между ними механический, и каждому нестрогому порядку соответствует ровно один строгий.

ОПРЕДЕЛЕНИЕ 8.3 Строгий порядок

Бинарное отношение $\prec$ на множестве A называется **строгим порядком**, если оно:

- **Иррефлексивно:**

$$\neg(a \prec a).$$

- **Асимметрично:** если $a \prec b$, то

$$\neg(b \prec a).$$

- **Транзитивно:** если $a \prec b, b \prec c$, то

$$a \prec c.$$

ЛОВУШКА Антисимметрия $\neq$ асимметрия

Антисимметричность и асимметричность — свойства разных порядков. Антисимметричность: $a \preceq b, b \preceq a \rightarrow a = b$. Асимметричность: $a \prec b, b \prec a$ несовместны. Нестрогий порядок рефлексивен, строгий — иррефлексивен: различие ровно как между $\leq, <$ на числах. Смещение двух свойств в доказательстве даёт неверный вывод.

УТВЕРЖДЕНИЕ 8.1 Переход между строгим и нестрогим порядком

Имея нестрогий порядок, определим строгий: $a \prec b \leftrightarrow a \preceq b \wedge a \neq b$. Имея строгий порядок, определим нестрогий: $a \preceq b \leftrightarrow a \prec b \vee a = b$. Тогда первая конструкция даёт строгий порядок, а вторая — нестрогий.

Доказательство

Докажем проверкой всех шести аксиом, разбирая случаи определений.

Иррефлексивность строгого порядка. Утверждение $a \prec a$ означало бы $a \preceq a$ и $a \neq a$ одновременно, что невозможно.

Асимметричность строгого порядка. Пусть $a \prec b$ и $b \prec a$. Тогда $a \preceq b, b \preceq a$ и $a \neq b$. Антисимметричность нестроного порядка даёт $a = b$ — противоречие с $a \neq b$.

Транзитивность строгого порядка. Из $a \prec b$ и $b \prec c$ следует $a \preceq b$ и $b \preceq c$, откуда по транзитивности нестроного порядка $a \preceq c$. Осталось исключить $a = c$: при $a = c$ из $a \prec b$ и $b \prec a$ следует противоречие с асимметричностью. Значит, $a \neq c$, и $a \prec c$.

Рефлексивность нестроного порядка. Из $a = a$ определение даёт $a \preceq a$.

Антисимметричность нестрогого порядка. Пусть $a \preceq b$ и $b \preceq a$. Если $a \neq b$, то $a \prec b$ и $b \prec a$, что противоречит асимметричности строгого порядка. Значит, $a = b$.

Транзитивность нестрогого порядка. Пусть $a \preceq b$ и $b \preceq c$. При $a \neq b$ и $b \neq c$ обе пары строгие, по транзитивности строгого порядка $a \prec c$, а значит и $a \preceq c$. При $a = b$ первое неравенство даёт $a \preceq c$. При $b = c$ второе неравенство даёт $a \preceq c$.

Пример Строгий и нестрогий порядок

На множестве $\{1, 2, 3\}$ определим нестрогий порядок как отношение делимости: $a \preceq b \leftrightarrow a \mid b$. Тогда $\preceq = \{(1, 1), (1, 2), (1, 3), (2, 2), (3, 3)\}$. Соответствующий строгий порядок $\prec$ получается исключением равенств: $a \prec b \leftrightarrow a \mid b \wedge a \neq b$, поэтому $\prec = \{(1, 2), (1, 3)\}$ — те же пары, но без рефлексивных пар.

На числах нестрогий порядок $a \leq b$ содержит пары $(3, 3), (3, 5)$, а строгий $a < b$ — только $(3, 5)$.

ОПРЕДЕЛЕНИЕ 8.4 Линейный порядок (*total order*)

Частичный порядок называется **линейным**, если для любых элементов a, b выполнено $a \preceq b$ или $b \preceq a$.

ОПРЕДЕЛЕНИЕ 8.5 Цепь

Подмножество S ЧУМ называется **цепью**, если любые два его элемента сравнимы.

Цепь из трёх элементов удобно рисовать вертикально — три элемента по возрасту, как на рис. 24.



Рис. 24. Цепь из трёх элементов.

ОПРЕДЕЛЕНИЕ 8.6 Антицепь

Подмножество S ЧУМ называется **антицепью**, если любые два различных его элемента несравнимы.

В ЧУМ делителей числа 12 цепью служит набор 1, 2, 4, 12, а антицепью — пара 2, 3. В множестве подмножеств $\{1, 2\}$, упорядоченном включением, антицепь образуют $\{1\}$ и $\{2\}$. Раздел 8.3 связывает наибольшие антицепи с разбиением на цепи.

Пример Линейные и частичные порядки

- $(\mathbb{N}, \leq), (\mathbb{R}, \leq)$ — линейные порядки: любые два числа сравнимы.
- Отношение делимости на $\mathbb{N}^+$ — частичный порядок: 2 и 3 несравнимы (ни одно из них не делит другое).
- $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$ — частичный порядок: $\{1\}$ и $\{2\}$ несравнимы (ни одно из них не является подмножеством другого).

Замечание Почему именно антисимметричность?

Рефлексивность и транзитивность вместе образуют *предпорядок* — отношение, в котором два различных элемента могут быть эквивалентны в смысле порядка. Тогда $a \preceq b, b \preceq a, a \neq b$. Предпорядок задаёт направление, но не различает элементы, замкнутые в цикл длины два.

Добавление симметричности превращает предпорядок в отношение эквивалентности — структуру, которая классифицирует: разбивает множество на классы неразличимых. Добавление антисимметричности — в частичный порядок: структуру, которая ранжирует, выстраивает элементы по уровням. Одно свойство определяет, будет ли отношение *группировать* по сходству или *упорядочивать* по старшинству. Рефлексивность и транзитивность — общий фундамент обеих теорий. Выбор между симметрией и антисимметрией — точка ветвления.

На графе различие видно невооружённым глазом. Симметрия тянет за собой обратные стрелки, транзитивность — новые связи: вся компонента связности схлопывается в один комок. Антисимметрия между разными элементами запрещает обратные стрелки, и картинка вытягивается в иерархию ярусов.

Не все линейные порядки устроены одинаково.

Примечание Плотность рациональных чисел

Порядок $(\mathbb{Q}, \leq)$ *плотен*: между любыми двумя различными рациональными числами найдётся третье, ведь $a < \frac{a+b}{2} < b$ по свойствам сложения и деления. Формально: $\forall a, b \in \mathbb{Q}, a < b \rightarrow \exists c \in \mathbb{Q} : a < c < b$. Порядки $(\mathbb{N}, \leq)$ и $(\mathbb{Z}, \leq)$, в отличие от этого, *дискретны*: между n и $n + 1$ нет других целых чисел.

Например, $\frac{1}{3} < \frac{\frac{1}{3} + \frac{1}{2}}{2} = \frac{5}{12} < \frac{1}{2}$. Между любыми двумя рациональными числами можно вставить их среднее арифметическое и повторить это построение без конца.

Пример Порядок в Rust: PartialOrd и Ord

Частичный порядок — это способ сравнивать элементы множества: мы определили, когда один не превосходит другого, но не потребовали, чтобы любые два были сравнимы. В Rust частичный порядок выражается трейтом PartialOrd: его метод partial_cmp возвращает Option<Ordering>, и значение None как раз означает несравнимость. Трейт Ord добавляет тотальность — метод cmp возвращает Ordering всегда, поэтому Ord задаёт линейный порядок.

Упорядочим семейство множеств по включению: $a \preceq b \leftrightarrow a \subseteq b$. Это частичный порядок: {1} и {2} несравнимы, потому что ни одно из них не является подмножеством другого. Значит, включение выражается через PartialOrd, а не через Ord.

```
use std::cmp::Ordering;

/// Множество как отсортированный список без повторений.
struct Set(Vec<u32>);

impl PartialEq for Set {
    fn eq(&self, other: &Self) -> bool { self.0 == other.0 }
}
impl Eq for Set {}

impl PartialOrd for Set {
    /// a <= b означает, что a подмножество b.
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        let sub = |x: &Set, y: &Set|
            x.0.iter().all(|e| y.0.binary_search(e).is_ok());

        match (sub(self, other), sub(other, self)) {
            (true, true) => Some(Ordering::Equal),    // a = b
            (true, false) => Some(Ordering::Less),    // a входит в b
            (false, true) => Some(Ordering::Greater), // b входит в a
            (false, false) => None,                  // несравнимы
        }
    }
}
```

None в partial_cmp — это математическая несравнимость: $a \preceq b, b \preceq a$ оба ложны. Ord мы намеренно не реализуем: включение не тотально, поэтому произвольная досортировка несравнимых множеств задала бы уже другой порядок, не совместимый с включением.

§ 8.2 Диаграммы Хассе

Диаграмма Хассе изображает частичный порядок, рисуя элементы как точки на плоскости, а само отношение — как вертикальное расположение и рёбра.

ОПРЕДЕЛЕНИЕ 8.7 Диаграмма Хассе

Диаграмма Хассе конечного ЧУМ $(A, \preceq)$ строится так:

1. Убрать все петли.
2. Убрать все рёбра, следующие из транзитивности, оставив только отношение покрытия: $b \prec a$ без промежуточных элементов.
3. Расположить элементы по вертикали: при $b < a$ элемент b ниже элемента a .
4. Нарисовать ненаправленные линии между элементами, связанными отношением покрытия.

Диаграмма Хассе — это «скелет» частичного порядка: она показывает в точности те пары, которые нельзя вывести из транзитивности. Каждое сокращение безопасно: то, что убрано, восстанавливается по определению ЧУМ. Петли опущены, но подразумеваются рефлексивностью. Транзитивные рёбра опущены: если до элемента можно дойти по рёбрам, связь следует из транзитивности. Стрелки не нужны, ведь все они указывали бы в одну сторону.

Пример Диаграмма делителей

Рассмотрим ЧУМ делителей числа 12: $D_{12} = \{1, 2, 3, 4, 6, 12\}$, упорядоченное отношением $a \mid b$ (делимости).

Отношения покрытия в этом множестве:

- 1 покрывается элементами 2 и 3 (простые делители).
- 2 покрывается элементами 4 и 6.
- 3 покрывается элементом 6.
- 4 и 6 покрываются элементом 12.

Ребро 1–4 не рисуется, хотя $1 \mid 4$ верно: оно следует из транзитивности через 2. Диаграмма Хассе этого ЧУМ — на рис. 25.

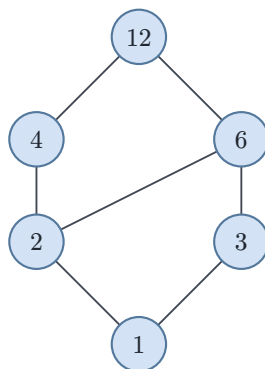
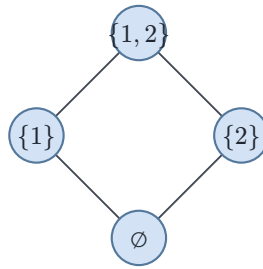
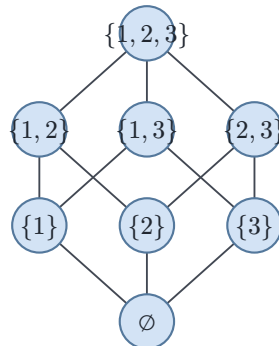


Рис. 25. Диаграмма Хассе делителей числа 12.

Семейства подмножеств двух- и трёхэлементного множества, упорядоченные по включению, дают булевы решётки B_2 и B_3 — их диаграммы Хассе показаны на рис. 26 и 27. Термин «решётка» определим в разделе 8.6.

Рис. 26. Булева решётка B_2 .Рис. 27. Булева решётка B_3 .**Пример Булева решётка размерности 3**

Для $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$ диаграмма Хассе представляет собой трёхмерный куб: 8 элементов (все подмножества трёхэлементного множества), 12 рёбер покрытия (добавление ровно одного элемента). Пустое множество $\emptyset$ находится внизу, $\{1, 2, 3\}$ — вверху. Семейство подмножеств с включением — простейший пример булевой решётки B_3 .

Замечание Пищевые сети как диаграммы Хассе

Экологи рисуют пищевые сети по тому же правилу. Стрелка $A \rightarrow B$ означает, что A ест B . Но изображают только прямые пищевые связи: волк ест зайца, заяц ест траву. Связь «волк ест траву» — через зайца — не рисуют: она следует из транзитивности. Отношение покрытия здесь — «кто кого ест непосредственно».

Трофический уровень организма — его высота в такой диаграмме. Трава стоит внизу, волк — наверху. Заяц и сурок на среднем уровне несравнимы: ни один не ест другого, но оба едят траву. Это частичный порядок, а не линейный. Несравнимость здесь означает *независимость*, а не отсутствие связи.

§ 8.3 * Ширина порядка и теорема Дилуорса

Цепь собирает сравнимые элементы, антицепь — попарно несравнимые. Для конечного ЧУМ обе структуры измеряются числом и связаны точным равенством.

ОПРЕДЕЛЕНИЕ 8.8 **Ширина**

Шириной конечного ЧУМ называется размер его наибольшей антицепи.

В ЧУМ делителей числа 12 ширина равна 2: антицепи $\{2, 3\}$ и $\{4, 6\}$ содержат по два элемента, а любые три элемента этого множества включают сравнимую пару. В булевой решётке $\mathcal{P}(\{1, 2, 3\})$ шире антицепи из трёх одноэлементных множеств нет, поэтому ширина равна 3.

В терминах зависимостей ширина отвечает на вопрос, сколько задач может понадобиться выполнять параллельно. Разбиение ЧУМ на цепи — это распределение задач по последовательным потокам: внутри потока задачи идут по порядку, разные потоки друг от друга не зависят. Понадобится ширина и в проекте главы 5: там она вычисляется по таблице отношения для малых порядков.

ТЕОРЕМА 8.2 **Теорема Дилуорса⁴⁰**

В конечном ЧУМ наименьшее число цепей, на которые разбивается множество, равно ширине.

Набросок доказательства

Докажем неравенство в одну сторону полностью, а обратное сведём к паросочетаниям.

Пусть множество разбито на k цепей, а S — антицепь. Два элемента одной цепи сравнимы, а элементы S попарно несравнимы, поэтому каждая цепь содержит не более одного элемента из S . Значит, $|S| \leq k$: размер любой антицепи не превосходит размера любого разбиения на цепи.

Обратное утверждение — что хватит разбиения ровно на ширину — опирается на паросочетания в двудольных графах, разобранные в главе 9. Для каждого элемента x ЧУМ P заведём левую копию x^L и правую копию x^R , а рёбра проведём из x^L в y^R для всех пар $x \prec y$. Максимальное паросочетание склеивает элементы в цепи: у каждого элемента не более одного ребра-входа и одного ребра-выхода, и вдоль склеенных рёбер элементы строго возрастают. Если паросочетание содержит m рёбер, получается разбиение на $|P| - m$ цепей: каждое склеивающее ребро уменьшает число кусков на единицу. По теореме Кёнига у графа есть вершинное покрытие из m вершин. Элементы, обе копии которых остались вне покрытия, образуют антицепь: ребро между сравнимой парой таких элементов осталось бы непокрытым. Каждая вершина покрытия принадлежит максимум одному элементу, поэтому обе копии вне покрытия остаются не менее чем у $|P| - m$ элементов. Значит, ширина не меньше $|P| - m$, а разбиение из $|P| - m$ цепей существует. Вместе с неравенством первой части это даёт равенство.

⁴⁰R. P. Dilworth. «A Decomposition Theorem for Partially Ordered Sets». Annals of Mathematics, 1950.

Двойственное утверждение известно как теорема Мирского⁴¹: минимальное число антицепей, на которые разбивается конечное ЧУМ, равно длине его наибольшей цепи. Доказывается оно проще: в i -ю антицепь попадают элементы, у которых наибольшая кончающаяся в них цепь имеет длину i .

§ 8.4 Экстремальные элементы

В частичном порядке одни элементы занимают привилегированное положение: выше (или ниже) них ничего нет, либо они сравнимы со всеми. Эти понятия обобщают «первый» и «последний» на случай, когда порядок не линейен. Всюду ниже $\prec$ обозначает строгий порядок, индуцированный нестрогим порядком $\preceq$.

ОПРЕДЕЛЕНИЕ 8.9 Минимальный элемент

Элемент m называется **минимальным**, если нет элемента строго меньше:

$$\begin{aligned} &\nexists a \in A. a \prec m \\ &\equiv \\ &\forall a \in A. a \not\prec m \end{aligned}$$

ОПРЕДЕЛЕНИЕ 8.10 Максимальный элемент

Элемент m называется **максимальным**, если нет элемента строго больше:

$$\begin{aligned} &\nexists a \in A. m \prec a \\ &\equiv \\ &\forall a \in A. m \not\prec a \end{aligned}$$

Первая пара свойств локальна: минимальный и максимальный элементы определяются отсутствием элемента строго меньше или, соответственно, строго больше, и могут оставаться несравнимыми с остальными элементами. Вторая пара глобальна: наименьший и наибольший элементы сравнимы с каждым элементом множества.

ОПРЕДЕЛЕНИЕ 8.11 Наименьший элемент

Элемент m называется **наименьшим**, если он меньше (нестрого) всех:

$$\begin{aligned} &\forall a \in A. m \preceq a \\ &\equiv \\ &\nexists a \in A. m \not\preceq a \end{aligned}$$

ОПРЕДЕЛЕНИЕ 8.12 Наибольший элемент

Элемент m называется **наибольшим**, если он больше (нестрого) всех:

⁴¹L. Mirsky. «A dual of Dilworth's decomposition theorem». American Mathematical Monthly, 1971.

$$\begin{aligned} \forall a \in A. a \preceq m \\ \equiv \\ \nexists a \in A. a \not\preceq m \end{aligned}$$

Эти четыре понятия легко спутать. Наименьший элемент — всегда минимальный, но не наоборот: минимальных может быть несколько (если они несравнимы в нижней части ЧУМ), а наименьший — элемент, сравнимый со всеми и меньший всех. Аналогично связаны наибольший и максимальные элементы.

Минимальные и максимальные элементы лежат на самом нижнем и самом верхнем уровне диаграммы. В диаграмме с несколькими связными компонентами у каждой из них есть свои минимальные и максимальные элементы. В конечном ЧУМ они всегда существуют: иначе из любого элемента можно было бы бесконечно спускаться к строго меньшему, а в конечном множестве бесконечная цепочка не помещается. Наименьший и наибольший такой гарантии не имеют. Если наименьший или наибольший существует — он единственен.

Пример Экстремальные элементы в ЧУМ делителей

Рассмотрим ЧУМ $(\{2, 3, 4, 6, 12\}, |)$ с отношением делимости:

- Минимальные элементы: 2 и 3 (оба являются простыми числами, ни одно из них не делит другое, то есть несравнимы).
- Максимальный элемент: 12 (делится на все остальные).
- Наименьший элемент: не существует (2 и 3 несравнимы).
- Наибольший элемент: 12 (делится на все остальные).

Следующая таблица обобщает различия между экстремальными элементами:

Понятие	Условие	Единственен?	Существует в конечном ЧУМ?
Минимальный	$\neg(\exists a : a \prec m)$	Нет — может быть несколько	Всегда
Максимальный	$\neg(\exists a : m \prec a)$	Нет — может быть несколько	Всегда
Наименьший	$\forall a \in A. m \preceq a$	Да	Не гарантирован
Наибольший	$\forall a \in A. a \preceq m$	Да	Не гарантирован

Таблица 2. Сравнение экстремальных элементов частичного порядка.

Проверка Экстремальные элементы

1. Может ли у конечного ЧУМ быть несколько максимальных элементов, но при этом существовать наибольший?
2. Почему максимальность определяется через строгий порядок — «нет элемента строго больше» — а не через нестрогий?

§ 8.5 Грани, супремум, инфимум

Экстремальные элементы характеризуют всё ЧУМ. Когда нас интересует подмножество S ЧУМ, мы говорим о гранях — элементах, ограничивающих его сверху или снизу.

Пусть $(A, \preceq)$ — ЧУМ, а $S \subseteq A$ — его подмножество.

ОПРЕДЕЛЕНИЕ 8.13 Верхняя грань

Элемент $u \in A$ называется **верхней гранью** множества S , если $\forall s \in S : s \preceq u$.

ОПРЕДЕЛЕНИЕ 8.14 Нижняя грань

Элемент $l \in A$ называется **нижней гранью** множества S , если $\forall s \in S : l \preceq s$.

Граней может быть много. Интерес представляют «наилучшие» грани — наименьшая среди верхних и наибольшая среди нижних.

ОПРЕДЕЛЕНИЕ 8.15 Супремум

Наименьшая верхняя грань множества $S \subseteq A$ называется **супремумом**, обозначается $\sup S$.

ОПРЕДЕЛЕНИЕ 8.16 Инфимум

Наибольшая нижняя грань множества $S \subseteq A$ называется **инфимумом**, обозначается $\inf S$.

Супремум называют также точной верхней гранью, инфимум — точной нижней. Для пары элементов a, b вместо $\sup\{a, b\}$ и $\inf\{a, b\}$ пишут $a \vee b$ и $a \wedge b$.

Пример Наименьшего элемента нет, инфимум есть

У конечного множества действительных чисел наименьший элемент всегда существует: перебрали все элементы — нашли самый малый. У бесконечного множества его может не быть. Множество чисел $\{\frac{1}{n} : n \geq 1\}$ не имеет наименьшего элемента. Все элементы множества больше нуля и сколь угодно близко к нему подходят, но ноль в множество не входит. Однако инфимум у этого множества есть — это ноль. Инфимум обобщает наименьший элемент тем, что не обязан принадлежать множеству.

Пример Грани, супремум и инфимум в ЧУМ делителей

Рассмотрим подмножество $S = \{2, 3\}$ делителей числа 12.

- Верхние грани S : $\{6, 12\}$ (оба делятся и на 2, и на 3).

- Супремум S : 6 — наименьшая из верхних граней (6 делит 12).
- Нижние грани S : $\{1\}$ (1 делит и 2, и 3).
- Инфимум S : 1 — наибольшая из нижних граней.

У подмножества $\{4, 6\}$ верхняя грань — $\{12\}$, супремум — 12. Нижняя грань — $\{1, 2\}$, инфимум — 2 (наибольшая из нижних).

ЛОВУШКА Супремум $\neq$ максимум

Супремум — наименьшая верхняя грань, он может не принадлежать множеству. В $\mathbb{R}$ выполнено $\sup\{q \in \mathbb{Q} \mid q^2 < 2\} = \sqrt{2}$, причём $\sqrt{2} \notin \mathbb{Q}$. Максимум должен принадлежать множеству — у этого множества максимума нет.

Супремум и инфимум единственны, когда существуют. Но в произвольном ЧУМ они могут отсутствовать. Возьмём ЧУМ ровно из трёх элементов $\{a, b, c\}$, где $a < b$ и $a < c$, а b и c несравнимы. У пары $\{b, c\}$ нет верхней грани: элемент, лежащий над обоими, в множестве просто не встречается. Поэтому $b \vee c$ не существует.

Частичный порядок даёт структуру сравнения: для любой пары элементов ровно одно из $a \leq b$, $b \leq a$ или несравнимость. Для моделирования зависимостей этого достаточно — несравнимость позволяет говорить о параллелизме в планировании задач и о конкурентности событий в распределённых системах. Но для статического анализа программ и верификации этого уже недостаточно.

Когда две ветви условного оператора сходятся, необходимо объединить информацию из обеих ветвей. Пара несравнимых элементов может не иметь ни «наименьшего общего потомка», ни «наибольшего общего предка» — супремум и инфимум могут отсутствовать. Без гарантированного супремума (join) алгоритм не может корректно слить частичные результаты — он вынужден либо терять точность, либо полагаться на эвристики. То же верно для абстрактной интерпретации, верификации, оптимизации запросов — везде, где информация течёт по графу программы и сливается в точках соединения.

Замечание Ближайший общий предок

Корневое дерево становится ЧУМ, если упорядочить вершины от корня вниз: потомок меньше предка. Корень — наибольший элемент, листья — минимальные. У любой пары вершин в таком порядке есть супремум — их ближайший общий предок. Общие предки у пары всегда есть (хотя бы корень), и среди них всегда есть наименьший.

Инфимум у пары вершин существует не всегда — поэтому дерево не дотягивает до решётки. Супремум здесь — рабочий инструмент алгоритмов. Расстояние между вершинами выражается через него: $\text{dist}(a, b) = \text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(a \vee b)$.

Решётка добавляет ровно то, чего не хватало для алгоритмических применений: существование бинарных супремума и инфимума для каждой пары элементов. Это минимальное усиление структуры. Не требуется линейность — несравнимость по-прежнему допускается. Не требуется полнота — не нужно определять $\sup$ и $\inf$ для бесконечных подмножеств. Достаточно двух бинарных операций, чтобы частичный порядок превратился в алгебраическую структуру, пригодную для вычислений.

Мы подошли к понятию решётки. Решётки возникают везде, где есть иерархия и операции объединения/пересечения: булевы алгебры, типы в языках программирования, статический анализ, топология.

§ 8.6 Решётки

ОПРЕДЕЛЕНИЕ 8.17 Решётка

ЧУМ называется **решёткой**, если у каждой пары элементов существуют $a \vee b, a \wedge b$.

Пример ЧУМ, не являющееся решёткой

ЧУМ $(\{2, 3, 4, 6, 12\}, |)$ с отношением делимости не является решёткой. У пары $\{2, 3\}$ нет нижней грани в этом множестве: элемент 1 (который делит и 2, и 3) отсутствует. Поэтому инфимум не существует — и структура не дотягивает до решётки.

Добавление 1 в множество превращает его в D_{12} — решётку делителей числа 12, с которой мы уже работали.

Пример Примеры решёток

- $(\mathcal{P}(A), \subseteq)$: супремум — $X \cup Y$, инфимум — $X \cap Y$. Это булева решётка.
- $(\mathbb{N}^+, |)$: супремум — НОК(a, b), инфимум — НОД(a, b).
- Линейный порядок: супремум и инфимум пары — $\max(a, b), \min(a, b)$.

Пример Решётка делителей числа 60

Рассмотрим множество делителей числа 60: $D_{60} = \{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\}$ с отношением $a \mid b$ (делимости).

Это решётка. Возьмём пару $\{6, 10\}$: элементы несравнимы (6 не делит 10, 10 не делит 6). Их НОД($6, 10$) = 2 — инфимум (наибольшая нижняя грань). Их НОК($6, 10$) = 30 — супремум (наименьшая верхняя грань).

Для пары $\{4, 5\}$: ни одно не делит другое. Их НОД($4, 5$) = 1, НОК($4, 5$) = 20, причём НОД — наименьший элемент решётки. Для пары $\{2, 4\}$: 2 делит 4. По-

этому $\text{НОД}(2, 4) = 2$, $\text{НОК}(2, 4) = 4$ — один из элементов пары является гранью другого.

Элементы 2 и 3 несравнимы (2 не делит 3, 3 не делит 2), но НОД и НОК определены — это 1 и 6 соответственно. Именно это делает структуру решёткой, а не линейным порядком. В линейном порядке несравнимых элементов не существует. В решётке они *допустимы*, и на каждой паре определены операции $\vee$, $\wedge$. Конкретный пример: на диаграмме Хассе делителей 12, показанной на рис. 25, элементы 4 и 6 несравнимы, но их инфимум равен 2, а супремум — 12.

ИСТОРИЯ От делителей чисел до статического анализа программ

Рихард Дедекинд в работе «Über Zerlegungen von Zahlen durch ihre grössten gemeinsamen Teiler»⁴² изучал структуру делителей чисел и ввёл понятия, ставшие основой теории решёток: операции пересечения и объединения, дистрибутивные законы. Он показал, что множество натуральных чисел с операциями НОД и НОК образует структуру, аналогичную булевой алгебре, но без дополнений.

Гаррет Биркгоф в монографии «Lattice Theory»⁴³ систематизировал разрозненные результаты и превратил теорию решёток в самостоятельную математическую дисциплину. Книга выдержала три издания (1940, 1948, 1967) и стала стандартным справочником. Биркгоф показал, что решётки возникают в алгебре (подгруппы, подкольца), топологии (замкнутые множества), логике (булевы алгебры) и проективной геометрии.

Патрик и Радия Кузо в 1977 году опубликовали «Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs», где предложили использовать решётки для статического анализа программ. Каждое свойство программы (знак переменной, диапазон значений) моделируется элементом решётки. Объединение ветвей условного оператора — супремум в решётке. Абстрактная интерпретация стала фундаментом промышленных анализаторов: Astrée доказал отсутствие ошибок времени выполнения в ПО Airbus A380. Infer от Meta находит ошибки в миллионах строк кода ежедневно.

Свойства операций решётки напоминают свойства арифметических операций, но с важными отличиями. Ассоциативность, коммутативность и идемпотентность выполняются во всех решётках, а дистрибутивность — нет.

ОПРЕДЕЛЕНИЕ 8.18 Дистрибутивная решётка

Решётка называется **дистрибутивной**, если

⁴²R. Dedekind. «Über Zerlegungen von Zahlen durch ihre grössten gemeinsamen Teiler». 1897.

⁴³G. Birkhoff. «Lattice Theory». 1940.

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c).$$

(И двойственное тождество.)

ОПРЕДЕЛЕНИЕ 8.19 Булева алгебра

Дистрибутивная решётка с наименьшим элементом (0), наибольшим (1) и операцией дополнения $\bar{a}$ называется **булевой алгеброй**. Дополнение удовлетворяет $a \wedge \bar{a} = 0, a \vee \bar{a} = 1$.

Примечание

Логика высказываний работает с операциями $\wedge, \vee, \neg$. Алгебра множеств — с $\cap, \cup, \bar{X}$. Вместе с булевой алгеброй это проявления одной структуры: дистрибутивной решётки с дополнениями. Это объединение подробно изучается в главе 10.

Пример Булева алгебра из четырёх элементов

Решётка $\mathcal{P}(\{1, 2\})$, показанная на рис. 26, — булева алгебра. Наименьший элемент — $\emptyset$, наибольший — $\{1, 2\}$. Дополнение множества X — его разность с $\{1, 2\}$: $\overline{\{1\}} = \{2\}, \overline{\{2\}} = \{1\}$.

Проверим аксиому дополнения на паре $\{1\}, \{2\}$: $\{1\} \cup \overline{\{1\}} = \{1\} \cup \{2\} = \{1, 2\}$, и $\{1\} \cap \overline{\{1\}} = \emptyset$. Те же операции — пересечение, объединение и дополнение — на подмножествах любого множества дают булеву алгебру.

Пример Дистрибутивные и недистрибутивные решётки

Рассмотрим две дистрибутивные решётки. Булева решётка $(\mathcal{P}(\{1, 2, 3\}), \subseteq)$ дистрибутивна. Для любых подмножеств A, B, C верно $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. В решётке делителей $(D_{12}, |)$ обе части равенства на элементах 2, 3, 6 дают 6:

$$\begin{aligned} 2 \vee (3 \wedge 6) &= \text{НОК}(2, \text{НОД}(3, 6)) = \text{НОК}(2, 3) = 6, \\ (2 \vee 3) \wedge (2 \vee 6) &= \text{НОД}(\text{НОК}(2, 3), \text{НОК}(2, 6)) = \text{НОД}(6, 6) = 6. \end{aligned}$$

Одна проверка не доказывает дистрибутивность. Решётка делителей любого числа дистрибутивна по другой причине: НОД и НОК действуют на показателях простых множителей поточечно — НОК берёт максимум показателя по каждому простому, а НОД — минимум. Дистрибутивность сводится к тождеству $\min(x, \max(y, z)) = \max(\min(x, y), \min(x, z))$.

Недистрибутивный пример — решётка N_5 (пентагон). Её элементы: $\{0, a, b, c, 1\}$. Порядок: $0 < a < b < 1, 0 < c < 1$, а c несравним с a и b . Проверим нарушение закона на тройке a, b, c , взяв двойственную форму. Для недистрибутивности достаточно нарушения любой из двух, а в решётках они равносильны. Общая

нижняя грань b и c — только 0: элемент a с c несравним. Поэтому $a \vee (b \wedge c) = a \vee 0 = a$. Верхняя грань a и c — только 1, поэтому $(a \vee b) \wedge (a \vee c) = b \wedge 1 = b$. Левая часть дала a , правая — b , и поскольку $a \neq b$, дистрибутивный закон нарушен.

В главе об отношениях (5) характеристика Биркгофа была лишь анонсирована. Здесь разберём понятия, на которых она стоит. Наименьший элемент решётки пишут как 0 или $\perp$. Наибольший — как 1 или $\top$. Это лишь разные записи одного понятия. Формулировка теоремы использует две малые решётки M_3, N_5 , изображённые на рисунках 28 и 29. M_3 и N_5 — две наименьшие недистрибутивные решётки, причём N_5 — ещё и наименьшая немодулярная (модулярность рассмотрим ниже). «Изоморфная» подрешётка означает «устроенная так же»: те же соотношения между элементами, с точностью до переименования.

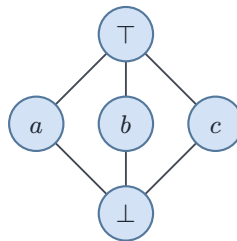


Рис. 28. Решётка M_3 (ромб).

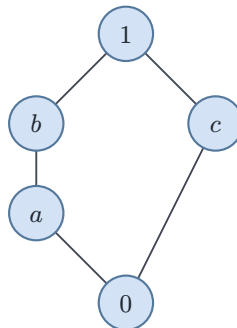


Рис. 29. Решётка N_5 (пентагон).

ОПРЕДЕЛЕНИЕ 8.20 Подрешётка

Подмножество элементов решётки называется **подрешёткой**, если оно замкнуто относительно операций $\vee, \wedge$, то есть результаты обеих операций над элементами подмножества снова лежат в подмножестве.

Пример Подрешётки

В булевой решётке $\mathcal{P}(\{1, 2, 3\})$, показанной на рис. 27, цепочка $S_1 = \{\emptyset, \{1\}, \{1, 2\}, \{1, 2, 3\}\}$ — подрешётка: супремум и инфимум любых двух её элементов снова лежат в цепочке, например $\{1\} \vee \{1, 2\} = \{1, 2\}$ и $\{1\} \wedge \{1, 2\} = \{1\}$.

Множество $S_2 = \{\emptyset, \{1\}, \{2\}\}$ подрешёткой не является: $\{1\} \vee \{2\} = \{1, 2\}$, а этого элемента в S_2 нет. Одна недостающая операция разрушает замкнутость.

ТЕОРЕМА 8.3 Характеризация дистрибутивных решёток

Решётка дистрибутивна тогда и только тогда, когда она не содержит подрешётки, изоморфной M_3, N_5 .

Утверждение «тогда и только тогда» называют *характеризацией* — оно задаёт необходимое и достаточное условие. Направление «только тогда» даёт необходимость, обратное — достаточность.

Набросок доказательства

Докажем направление «только тогда» переносом тождества.

Пусть решётка дистрибутивна, а S — её подрешётка. Тождество $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ выполняется для всех элементов решётки, в том числе для элементов S , а операции $\vee$ и $\wedge$ над элементами S не выводят из S . В M_3 и N_5 тождество нарушается — для N_5 проверка в примере выше, для M_3 в примере о модулярных решётках ниже. Решётка с такой подрешёткой дистрибутивной быть не может.

Обратное направление — что всякая недистрибутивная решётка содержит подрешётку, изоморфную M_3 или N_5 — нетривиально: из тройки, нарушающей дистрибутивный закон, нужно построить одну из двух фигур. Доказательство приведено в классических монографиях по теории решёток.⁴⁴

Пример Применение: IFDS в статическом анализе

Задачи анализа потока данных формулируются на решётках фактов о программе. Элемент решётки — набор фактов, например «переменная x — константа» или «указатель p может быть нулевым». Порядок — включение. Супремум — объединение наборов фактов, приходящих из разных ветвей. Функция перехода f описывает эффект отдельного оператора: набор фактов на входе она переводит в набор фактов на выходе.

Монотонная функция удовлетворяет условию $x \preceq y \rightarrow f(x) \preceq f(y)$. Монотонность даёт только одну сторону неравенства: $f(x) \vee f(y) \preceq f(x \vee y)$. Слить входы до применения f — значит потерять точность. Может оказаться, что $f(x \vee y)$ грубее, чем $f(x) \vee f(y)$. Слияние стирает информацию о происхождении фактов. Поэтому в общем случае анализ нельзя разложить на независимые подзадачи по отдельным фактам.

⁴⁴G. Grätzer. «General Lattice Theory». 1978.

Если функция перехода *дистрибутивна*, то $f(x \vee y) = f(x) \vee f(y)$ для любой пары x, y . Оба порядка операций дают одинаковый результат. На булеане дистрибутивность означает $f(S) = \bigcup_{e \in S} f(\{e\})$. Каждый факт можно прогнать через f по отдельности, а результаты объединить. Анализ распадается на независимые подзадачи. Дистрибутивность здесь — требование к функциям, а не к решётке: решётка фактов (булеан) дистрибутивна и так, но этого недостаточно.

На этом основан класс задач IFDS (Interprocedural, Finite, Distributive, Subset) — межпроцедурный анализ потока данных с конечным множеством фактов и дистрибутивными функциями перехода. Буква Subset напоминает, что функции действуют на подмножествах этого конечного множества. Межпроцедурность — что анализ должен переживать вызовы функций и возвраты из них. Задачи этого класса решаются за полиномиальное время: анализ сводится к достижимости в специально построенном графе — той же технике, что и в главе о графах (9).

Среди приложений — достигающие определения (reaching definitions) и taint-анализ: проверка, что непроверенные данные не попадают в критичные места программы.⁴⁵ Дистрибутивность нужна именно этому классу задач. Общая абстрактная интерпретация работает и на недистрибутивных решётках, и к ней мы вернёмся в главе 31.

Дистрибутивность — сильное свойство, однако есть и естественные недистрибутивные примеры, например решётка подпространств векторного пространства: для плоскости это нуль, вся плоскость и прямые через начало. Над полем из двух элементов прямых ровно три, и такая решётка совпадает с M_3 из примера ниже, удовлетворяя более слабому закону — модулярности.

ОПРЕДЕЛЕНИЕ 8.21 Модулярная решётка

Решётка называется **модулярной**, если из $a \preceq c$ следует

$$a \vee (b \wedge c) = (a \vee b) \wedge c.$$

Всякая дистрибутивная решётка модулярна, а обратное неверно — контрпримером служит решётка M_3 из примера ниже.

УТВЕРЖДЕНИЕ 8.4 Дистрибутивность влечёт модулярность

Пусть L — дистрибутивная решётка, $a, b, c \in L$ и $a \preceq c$. Тогда $a \vee (b \wedge c) = (a \vee b) \wedge c$.

⁴⁵T. Reps, S. Horwitz, S. Sagiv. «Precise Interprocedural Dataflow Analysis via Graph Reachability». POPL, 1995.

Доказательство

Докажем равенство взаимным включением, обозначив $x = (a \vee b) \wedge c$ и $y = a \vee (b \wedge c)$.

Сначала покажем $y \preceq x$. Элемент a не превосходит x : $a \preceq a \vee b$ всегда, а $a \preceq c$ по условию. Элемент $b \wedge c$ тоже не превосходит x : $b \wedge c \preceq b \preceq a \vee b$ и $b \wedge c \preceq c$. Значит, x — общая верхняя грань пары $a, b \wedge c$, поэтому наименьшая верхняя грань y не превосходит x .

Теперь покажем $x \preceq y$. Элемент x не превосходит c и не превосходит $a \vee b$, поэтому по дистрибутивности $x = x \wedge (a \vee b) = (x \wedge a) \vee (x \wedge b)$. Здесь $x \wedge a \preceq a$, а также $x \wedge b \preceq b$ и $x \wedge b \preceq x \preceq c$, откуда $x \wedge b \preceq b \wedge c$. Обе грани лежат под $a \vee (b \wedge c)$, значит и их супремум x не превосходит y .

Включения $y \preceq x$ и $x \preceq y$ по антисимметричности дают $x = y$.

Пример Модулярная, но не дистрибутивная решётка

Решётка M_3 — ромб, показанный на рис. 28, состоит из пяти элементов. Три из них — a, b, c — попарно несравнимы. Каждый лежит между $\perp$ и $\top$. Решётка M_3 модулярна. Для любой тройки выполняется $x \preceq z \rightarrow x \vee (y \wedge z) = (x \vee y) \wedge z$. Но M_3 не дистрибутивна. $a \wedge (b \vee c) = a \wedge \top = a$. $(a \wedge b) \vee (a \wedge c) = \perp \vee \perp = \perp$ — дистрибутивный закон нарушен.

Та же идея — запрещённая подрешётка — характеризует и модулярные решётки.

УТВЕРЖДЕНИЕ 8.5 Характеризация модулярных решёток

Решётка модулярна тогда и только тогда, когда она не содержит подрешётки, изоморфной N_5 .

Набросок доказательства

Докажем направление «только тогда» прямым подсчётом в N_5 .

Из $a \preceq b$ следует $a \vee (c \wedge b) = a \vee 0 = a$: у c и b общая нижняя грань только 0. С другой стороны, $(a \vee c) \wedge b = 1 \wedge b = b$, ведь $a \vee c = 1$ — единственная общая верхняя грань a и c . Получилось $a \neq b$ — модулярный закон нарушен. M_3 модулярна, поэтому в модулярной характеристике она не участвует.

Обратное направление — всякая немодулярная решётка содержит N_5 — доказывалось в тех же источниках, что и дистрибутивная характеристика.

Модулярные решётки возникают в линейной алгебре: решётка линейных подпространств модулярна, и это используется при анализе кодов (глава 13).

Проверка Решётки

1. Почему M_3 модулярна, хотя дистрибутивный закон в ней нарушен?
2. Достаточно ли проверить дистрибутивный закон на одной тройке элементов, чтобы объявить решётку дистрибутивной?
3. Почему корневое дерево с порядком «потомок меньше предка» не является решёткой?

Частичный порядок задаёт ограничения: одно должно быть раньше другого. Превратить эти ограничения в конкретную последовательность действий — задача топологической сортировки. Несравнимость означает независимость: между a, b нет зависимостей, поэтому их можно выполнять в любом порядке или параллельно. Результат — линейный порядок, совместимый с исходным частичным порядком.

§ 8.7 Топологическая сортировка

Частичный порядок моделирует зависимости: одно действие должно быть выполнено раньше другого. Чтобы составить расписание, нужно «развернуть» частичный порядок в линейный — топологическую сортировку.

ОПРЕДЕЛЕНИЕ 8.22 Топологическая сортировка

Линейный порядок на конечном ЧУМ $(A, \preceq)$ называется **топологической сортировкой** (или линейным расширением), если $a \preceq b \rightarrow a \leq b$.

Пример Топологическая сортировка учебного плана

Курсы учебного плана с зависимостями: «Основы программирования» — обязательный пререквизит для «Алгоритмов» и «Структур данных». «Структуры данных» — пререквизит для «Базы данных». Тогда $ОП \preceq Алг$, $ОП \preceq СД$, $СД \preceq БД$. Топологическая сортировка даёт допустимый порядок прохождения: ОП, Алг, СД, БД (или ОП, СД, Алг, БД, или ОП, СД, БД, Алг — «Алгоритмы» и «Базы данных» несравнимы, их можно проходить в любом порядке после пререквизитов).

Определение топологической сортировки ставит естественный вопрос: всякий ли частичный порядок допускает линеаризацию. Существуют ли ЧУМ, которые принципиально нельзя выстроить в линейную последовательность без нарушения зависимостей? Следующая теорема отвечает утвердительно — топологическая сортировка существует всегда.

ТЕОРЕМА 8.6 Существование топологической сортировки

Любое конечное ЧУМ имеет хотя бы одну топологическую сортировку.

Набросок доказательства

Докажем построением: предъявим алгоритм, который строит топологическую сортировку.

В конечном ЧУМ всегда существует хотя бы один минимальный элемент. Алгоритм выбирает любой минимальный элемент, выводит его, удаляет из ЧУМ и рекурсивно повторяет процедуру для оставшейся части. Порядок вывода даёт топологическую сортировку.

Корректность. Когда мы выводим элемент, все элементы, которые должны предшествовать ему, уже удалены, ведь он был минимальным в оставшемся множестве.

Примечание

Каждая топологическая сортировка конечного ЧУМ даёт линейный порядок, совместимый с исходным: если $a \preceq b$ в исходном порядке, то a предшествует b и в линейном. Несравнимые элементы получают произвольное упорядочение, и разные сортировки — разные способы разрешить эту свободу. Именно поэтому частичный порядок допускает параллелизм: несравнимые задачи можно выполнять одновременно. Этой же сортировкой системы сборки и планировщики задач выстраивают компиляцию и расписания: модуль-зависимость собирается раньше, несравнимые задачи идут параллельно.⁴⁶

Пример Топологическая сортировка в коде

Алгоритм Кана — та же жадная схема, что и в наброске доказательства выше: очередь минимальных элементов, извлечение, обновление счётчиков входящих рёбер. Функция возвращает порядок сборки или None, если граф содержит цикл: у циклического графа топологической сортировки нет.

```
use std::collections::VecDeque;

/// Топологическая сортировка (алгоритм Кана):
/// порядок сборки или None при цикле.
fn topo_sort(
    n: usize,
    edges: &[(usize, usize)],
) -> Option<Vec<usize>> {
    let mut indeg = vec![0; n];
    let mut adj = vec![vec![]; n];
    for &(u, v) in edges { adj[u].push(v); indeg[v] += 1; }

    let mut q: VecDeque<usize> =
        (0..n).filter(|&v| indeg[v] == 0).collect();
    let mut order = vec![];
```

⁴⁶A. B. Kahn. «Topological Sorting of Large Networks». Communications of the ACM, 1962.


```

while let Some(u) = q.pop_front() {
    order.push(u);
    for &v in &adj[u] {
        indeg[v] -= 1;

        if indeg[v] == 0 { q.push_back(v); }
    }
}
(order.len() == n).then_some(order) // None, если есть цикл
}

```

Топологическая сортировка — лишь одно из применений порядков в программировании. Ниже — ещё несколько: подтипирование в языках с типами, семантическое версионирование, векторные часы и лексикографический порядок.

§ 8.8 Лексикографический порядок

Порядок можно не только изучать, но и строить. Иногда нужен лишь новый способ упорядочить уже знакомые объекты — например, пары элементов или строки символов. Ответ даёт лексикографический порядок: когда каждый компонент пары взят из своего частичного порядка, сравниваем первые компоненты, а при равенстве — вторые. Ровно так работают словари: сначала по первой букве, потом по второй, и так далее.

8.8.1 На парах

Пусть $(A, \preceq_A)$, $(B, \preceq_B)$ — два ЧУМ. Имея порядки на компонентах, мы строим порядок на декартовом произведении $A \times B$.

ОПРЕДЕЛЕНИЕ 8.23 Лексикографический порядок на парах

Строгий **лексикографический порядок** $\prec_{\text{lex}}$ на $A \times B$ определяется так:

$$(a_1, b_1) \prec_{\text{lex}} (a_2, b_2) \leftrightarrow \left(a_1 \prec_A a_2 \right) \vee \left(a_1 = a_2 \wedge b_1 \prec_B b_2 \right),$$

где $\prec_A, \preceq_A, \prec_B, \preceq_B$ — строгие и нестрогие порядки компонент.

Если исходные порядки частичные, то результирующий порядок тоже частичный, а если исходные порядки линейные, то и результирующий линейный.

Пример Лексикографический порядок на парах

Пусть $A = B = \{0, 1\}$ с обычным порядком $0 < 1$. Тогда $A \times B = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. Лексикографический порядок выстраивает их так:

$$(0, 0) \prec_{\text{lex}} (0, 1) \prec_{\text{lex}} (1, 0) \prec_{\text{lex}} (1, 1).$$

Сравним $(0, 1)$, $(1, 0)$: первые компоненты различаются. Так как $0 < 1$, кортеж $(0, 1)$ меньше, хотя его вторая компонента больше. Первая компонента имеет приоритет.

8.8.2 Обобщение на кортежи одинаковой длины

Следующий шаг — k компонентов. Кортеж длины k (или k -кортеж) — упорядоченный набор $(a_1, a_2, \dots, a_k)$ элементов ЧУМ. Множество всех k -кортежей обозначается A^k .

ОПРЕДЕЛЕНИЕ 8.24 Лексикографический порядок на k -кортежах

Для двух k -кортежей из A^k положим

$$(a_1, \dots, a_k) \prec_{\text{lex}} (b_1, \dots, b_k)$$

если в первой позиции, где $a_i \neq b_i$, выполнено $a_i \prec b_i$.

Алгоритмически сравниваем пару a_1, b_1 , при равенстве переходим к следующей паре, и так далее. Порядок определяет первая же позиция, в которой кортежи различаются.

Пример Лексикографический порядок на тройках

Пусть $A = \{0, 1\}$ с порядком $0 < 1$. Кортежи длины 3 (битовые строки из трёх битов) упорядочиваются так:

$$(0, 0, 0) \prec_{\text{lex}} (0, 0, 1) \prec_{\text{lex}} (0, 1, 0) \prec_{\text{lex}} (0, 1, 1) \prec_{\text{lex}} (1, 0, 0) \prec_{\text{lex}} (1, 0, 1) \prec_{\text{lex}} (1, 1, 0) \prec_{\text{lex}} (1, 1, 1).$$

Первый бит — старший: все кортежи с 0 на первой позиции идут раньше всех кортежей с 1.

8.8.3 Строки разной длины: два подхода

До сих пор мы сравнивали кортежи одинаковой длины. Со строками разной длины — например, а и аа — правило «сравниваем первые символы» не даёт ответа: символы кончаются раньше, чем находится различие, а одна строка является префиксом другой. Возможны два расширения.

Если одна строка — префикс другой, более короткая считается меньшей. Именно так устроены бумажные словари: «кот» идёт раньше «котик», потому что «кот» — префикс «котика».

ОПРЕДЕЛЕНИЕ 8.25 Словарный порядок

Пусть Σ — алфавит с линейным порядком на символах. Для строк $s, t \in \Sigma^*$ **словарный порядок** $\prec_{\text{dict}}$ определяется так:

1. Если одна строка — собственный префикс другой, то $s \prec_{\text{dict}} t$, то есть более короткая строка меньше.
2. Иначе порядок определяют символы на первой позиции, где строки различаются.

Пример Словарный порядок

Над алфавитом $\{a, b\}$ с порядком $a < b$ словарный порядок даёт:

$$a \prec_{\text{dict}} aa \prec_{\text{dict}} aaa \prec_{\text{dict}} aab \prec_{\text{dict}} ab \prec_{\text{dict}} b \prec_{\text{dict}} ba \prec_{\text{dict}} bb.$$

$b \prec_{\text{dict}} ba$, потому что b — префикс ba . $aa \prec_{\text{dict}} ab$, потому что на второй позиции $a < b$.

Словарный порядок линеен над строками конечной длины, если порядок на алфавите линеен. Но не всякое множество строк имеет в нём наименьший элемент. Множество $\{b, ab, aab, aaab, \dots\}$ не имеет наименьшего элемента: каждая строка с дополнительным a в начале оказывается меньше.

$$b \succ ab \succ aab \succ aaab \succ \dots$$

Бесконечная убывающая цепь не имеет минимума: в словарном порядке такие цепи существуют.

Альтернативный подход — сравнивать строки в первую очередь по длине. Сначала идут все строки длины 0, затем длины 1, затем длины 2, и так далее, а внутри каждой длины строки упорядочены обычным лексикографическим порядком.

ОПРЕДЕЛЕНИЕ 8.26 Shortlex

Для строк $s, t \in \Sigma^*$ порядок **shortlex** (или length-lexicographic) определяется так:

$$s \prec_{\text{slex}} t \leftrightarrow (|s| < |t|) \vee (|s| = |t| \wedge s \prec_{\text{lex}} t),$$

где $|s|$ — длина строки.

Пример Shortlex

Над алфавитом $\{a, b\}$ с порядком $a < b$:

$$\varepsilon \prec_{\text{slex}} a \prec_{\text{slex}} b \prec_{\text{slex}} aa \prec_{\text{slex}} ab \prec_{\text{slex}} ba \prec_{\text{slex}} bb \prec_{\text{slex}} aaa \prec_{\text{slex}} \dots$$

В отличие от словарного порядка, b идёт раньше aa — потому что длина 1 меньше длины 2.

УТВЕРЖДЕНИЕ 8.7 Вполнеупорядоченность shortlex

Пусть Σ — конечный алфавит с линейным порядком на символах. Любое непустое множество строк $S \subseteq \Sigma^*$ имеет наименьший элемент в порядке shortlex.

Доказательство

Докажем построением: выберем в S строку минимальной длины, а среди строк этой длины — лексикографически первую.

Длины строк из S образуют непустое множество натуральных чисел, а у любого непустого множества натуральных чисел есть наименьший элемент. Обозначим минимальную длину n . Строки из S длины n образуют конечное непустое множество: таких строк не больше, чем $|\Sigma|^n$. На строках равной длины shortlex совпадает с лексикографическим порядком, который линеен, ведь порядок на алфавите линеен. Всякое конечное непустое линейно упорядоченное множество имеет наименьший элемент: спускаясь к строго меньшему, получаем убывающую последовательность различных элементов, которая в конечном множестве обрывается. Пусть u — наименьшая строка длины n в S .

Возьмём произвольную строку $s \in S$. Если $|s| > n$, то $u \prec_{\text{sllex}} s$ по длине. Если $|s| = n$, то s участвовала в выборе u , поэтому $s = u$ или $u \prec_{\text{lex}} s$. В обоих случаях никакая строка из S не меньше u , поэтому u — наименьший элемент.

Это свойство делает shortlex полезным в алгоритмах полного перебора: перебор в порядке shortlex гарантирует, что каждая строка конечной длины будет достигнута за конечное число шагов.

Примечание Словарный или Shortlex?

Словарный порядок естественен для интерфейсов, где человек ожидает порядка бумажного словаря: файловые менеджеры, списки имён, глоссарии. Shortlex — инструмент алгоритмический: генерация всех строк, перебор конфигураций, построение контрпримеров минимальной длины. Выбор порядка определяется задачей: работаете с человеком — словарный. Пишете алгоритм — shortlex.

8.8.4 Колексикографический порядок

Лексикографический порядок сравнивает компоненты слева направо — первый компонент самый важный. Колексикографический порядок (colex) переносит приоритет на последний компонент: это лексикографический порядок на перевёрнутых кортежах, когда мы разворачиваем каждый кортеж справа налево и сравниваем лексикографически.

ОПРЕДЕЛЕНИЕ 8.27 Колексикографический порядок

Для двух k -кортежей из A^k положим

$$(a_1, \dots, a_k) \prec_{\text{colex}} (b_1, \dots, b_k) \leftrightarrow (a_k, \dots, a_1) \prec_{\text{lex}} (b_k, \dots, b_1).$$

Порядок определяет первый же компонент (считая справа), в котором кортежи различаются.

Чтобы понять разницу между lex и colex , сравним две конкретные тройки пошагово.

Пример Lex vs Colex: пошаговое сравнение

Возьмём кортежи $(0, 0, 1), (0, 1, 0) \in \{0, 1\}^3$ с порядком $0 < 1$.

Lex (слева направо):

- Позиция 1: $0 = 0$ — равны, идём дальше.
- Позиция 2: $0 < 1$ — найдено различие. Результат: $(0, 0, 1) \prec_{\text{lex}} (0, 1, 0)$.

Colex (справа налево):

- Позиция 3 (последняя): $1 > 0$ — найдено различие, и $0 < 1$, поэтому меньшим считается кортеж с 0 на этой позиции. Результат: $(0, 1, 0) \prec_{\text{colex}} (0, 0, 1)$.

Порядки противоположны: в lex раньше $(0, 0, 1)$, в colex — $(0, 1, 0)$. В lex различие обнаружилось на второй позиции, в colex — на третьей, поэтому итоговый порядок развернулся.

Теперь выстроим все тройки из $\{0, 1\}^3$ по порядку — сначала в lex , затем в colex .

Пример Все тройки из $\{0, 1\}^3$ в lex и colex

Порядок lex (первые компоненты — старшие):

$$(0, 0, 0) \prec_{\text{lex}} (0, 0, 1) \prec_{\text{lex}} (0, 1, 0) \prec_{\text{lex}} (0, 1, 1) \prec_{\text{lex}} (1, 0, 0) \prec_{\text{lex}} (1, 0, 1) \prec_{\text{lex}} (1, 1, 0) \prec_{\text{lex}} (1, 1, 1).$$

Порядок colex (последние компоненты — старшие):

$$(0, 0, 0) \prec_{\text{colex}} (1, 0, 0) \prec_{\text{colex}} (0, 1, 0) \prec_{\text{colex}} (1, 1, 0) \prec_{\text{colex}} (0, 0, 1) \prec_{\text{colex}} (1, 0, 1) \prec_{\text{colex}} (0, 1, 1) \prec_{\text{colex}} (1, 1, 1).$$

Cox группирует кортежи по последней компоненте: первая половина — все с 0 на конце, вторая — все с 1. Внутри группы с последней 0 кортежи упорядочены по средней компоненте (второй справа), а при равенстве средней — по первой (третьей справа). Структура ровно та же, что у lex , но развёрнутая: старший разряд — правый вместо левого.

Колексикографический порядок — рабочий инструмент комбинаторики. Подмножества множества $\{1, \dots, n\}$ часто кодируют характеристическими векторами длины n . Бит i равен 1, если элемент i входит в подмножество. Cox -порядок на таких векторах группирует подмножества по *наибольшему* элементу — потому что

последняя (самая правая) единица и есть максимум. Это удобно для индукции по наибольшему элементу и для алгоритмов, перебирающих семейства множеств.

Проверка Лексикографические порядки

1. Почему при сравнении $(0, 1)$ и $(1, 0)$ лексикографически меньше $(0, 1)$, хотя его вторая компонента больше?
2. Приведите пару строк, порядок которых в словарном порядке противоположен порядку shortlex.
3. Каким общим свойством обладают все кортежи первой половины colex-порядка на $\{0, 1\}^3$?

§ 8.9 Приложения отношений порядка

Отношения порядка проявляются в типах и версиях, в причинности событий и порядке вызовов методов.

8.9.1 Иерархии типов в ООП

Отношение «является подтипом» ($<:$) — это частичный порядок на типах.

- Рефлексивно: $T <: T$.
- Транзитивно: если $A <: B$ и $B <: C$, то $A <: C$.
- Антисимметрично: если $A <: B$ и $B <: A$, то A и B — один тип.

Если $Dog <: Animal$ и $Cat <: Animal$, то Dog и Cat несравнимы — ни один из них не является подтипом другого.

Множественное наследование вводит объединения типов: $Dog \text{ or } Cat$ — тип, надтипом которого являются Dog и Cat . Наименьший общий надтип $Dog \text{ or } Cat$ (если он существует) — супремум в решётке типов. Аналогично, пересечение типов $Dog \& Cat$ (если существует) — инфимум.

Большинство объектно-ориентированных языков не гарантируют существования наименьшего общего надтипа для произвольной пары типов. Решёточная модель — идеализация, описывающая структурные свойства системы типов, а не детали реализации конкретного компилятора.⁴⁷

8.9.2 Векторные часы и причинность

В распределённой системе невозможно определить глобальное «сейчас» без дорогостоящей синхронизации часов. Лесли Лэмпорт⁴⁸ предложил обойтись без глобального времени: достаточно знать, произошло ли одно событие раньше другого.

⁴⁷L. Cardelli. «A Semantics of Multiple Inheritance». Information and Computation, 1988. B. Pierce. «Types and Programming Languages». 2002, главы 15–16.

⁴⁸L. Lamport. «Time, Clocks, and the Ordering of Events in a Distributed System». Communications of the ACM, 1978.

Отношение «произошло-до» ($\rightarrow$) определяется тремя правилами:

1. Если a и b — события в одном процессе и a произошло раньше b , то $a \rightarrow b$.
2. Если a — отправка сообщения, а b — его получение, то $a \rightarrow b$.
3. Транзитивность: $a \rightarrow b$ и $b \rightarrow c$ влекут $a \rightarrow c$.

Это отношение — строгий частичный порядок. События с $a/(\rightarrow)b, b/(\rightarrow)a$ называются *конкурентными*: они происходят в разных процессах и не связаны причинно.

Векторные часы реализуют этот частичный порядок.

1. Каждый из N процессов хранит массив $VC[1..N]$ целых чисел (изначально нули).
2. При локальном событии процесс i инкрементирует свой счётчик: $VC[i] := VC[i] + 1$.
3. При отправке сообщения процесс вкладывает в него свой текущий вектор VC .
4. При получении сообщения процесс i обновляет вектор поэлементным максимумом: $VC[j] := \max(VC[j], VC_{msg[j]})$ для всех j . Затем инкрементирует $VC[i]$ (получение — локальное событие).

Сравнение векторных часов: $a \rightarrow b \leftrightarrow \forall j : VC(a)[j] \leq VC(b)[j]$, и хотя бы одно неравенство строгое. Если векторы несравнимы (ни один не превосходит другой покомпонентно) — события конкурентны.

8.9.3 Версионирование и совместимость

Семантическое версионирование (SemVer) сочетает два разных порядка на одном множестве версий. Лексикографический порядок ($1.2.0 < 1.2.1 < 1.3.0 < 2.0.0$) — линейный: любые две версии сравнимы по старшинству. Отношение обратной совместимости — предпорядок: после отождествления взаимно совместимых версий он становится частичным порядком. Версии 1.2.0, 2.0.0 несравнимы: мажорная версия изменилась, обратная совместимость не гарантирована. А 1.2.0, 1.2.1 сравнимы: вторая обратно совместима с первой.

Выбор совместимого набора зависимостей — это поиск максимальной конфигурации в ЧУМ версий с отношением совместимости: для каждого пакета нужно выбрать наибольшую версию, совместимую с уже выбранными версиями других пакетов. Именно эту задачу решают пакетные менеджеры (npm, Cargo, pip) при разрешении зависимостей: строится граф зависимостей с ограничениями на версии, и SAT/ASP-решатель ищет удовлетворяющую конфигурацию.⁴⁹

8.9.4 Наследование и линеаризация

В языках с множественным наследованием (Python, C++, Scala) иерархия классов образует ЧУМ. Класс A предшествует классу B , если A наследует от B (прямо или транзитивно). Наследник стоит раньше своего предка. Несравнимые классы — те, что находятся в разных ветках иерархии и не связаны наследованием.

⁴⁹T. Preston-Werner. «Semantic Versioning 2.0.0». <https://semver.org>, 2013.

Порядок разрешения методов (MRO — Method Resolution Order) — это линейзация данного ЧУМ: линейный порядок на классах, совместимый с отношением наследования (наследники предшествуют родителям: класс ищется раньше своих предков). Когда метод не найден в текущем классе, поиск идёт по линейному порядку MRO.

Алгоритм СЗ (используемый в Python)⁵⁰ вычисляет топологическую сортировку, удовлетворяющую двум дополнительным ограничениям:

1. порядок родителей в объявлении класса (слева направо) должен быть сохранён в MRO.
2. наследники должны появляться в MRO раньше родителей.

Если эти ограничения невыполнимы (например, из-за конфликта при «ромбовидном» наследовании с несогласованным порядком), СЗ отвергает определение класса — в Python это вызывает `TypeError`.⁵¹

§ 8.10 Неподвижные точки в решётках

Вернёмся к решёткам и посмотрим на них с новой стороны. Функция $f : L \rightarrow L$ называется *монотонной*, если $x \preceq y \rightarrow f(x) \preceq f(y)$. *Неподвижная точка* (fixed point) функции f — элемент x с $f(x) = x$.

Монотонные функции на решётках отличаются от функций на числах одним свойством: при слабых дополнительных условиях неподвижная точка гарантированно существует. Эти условия даёт понятие полной решётки.

ОПРЕДЕЛЕНИЕ 8.28 Полная решётка

Решётка называется **полной**, если *любое* подмножество (не только конечное) имеет супремум и инфимум.

Булева решётка $\mathcal{P}(A)$ полна: супремум произвольного семейства подмножеств — их объединение, инфимум — пересечение. Решётка $\mathbb{N}$ с обычным порядком не полна: у неё нет супремума, потому что нет наибольшего натурального числа.

ТЕОРЕМА 8.8 Кнастера–Тарского о неподвижной точке⁵²

Пусть $(L, \preceq)$ — полная решётка. Всякая монотонная функция $f : L \rightarrow L$ имеет наименьшую и наибольшую неподвижные точки. Их обозначают $\text{lfp}(f)$, $\text{gfp}(f)$.

⁵⁰Описание СЗ в документации Python: <https://www.python.org/download/releases/2.3/mro/>.

⁵¹K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, P. T. Withington. «A Monotonic Superclass Linearization for Dylan». OOPSLA, 1996.

⁵²B. Knaster. «Un théorème sur les fonctions d'ensembles». Annales de la Société Polonaise de Mathématique, 1928. A. Tarski. «A lattice-theoretical fixpoint theorem and its applications». Pacific Journal of Mathematics, 1955.

Набросок доказательства

Докажем построением: предъявим наименьшую неподвижную точку как инфимум множества пост-неподвижных точек, а наибольшую получим двойственностью.

Рассмотрим множество $P = \{x \in L \mid f(x) \preceq x\}$ («пост-неподвижные точки» — элементы, которые функция не увеличивает). Покажем сначала, что его инфимум $a = \inf P$ — наименьшая неподвижная точка.

$\forall x \in P : a \preceq x$, ведь инфимум меньше всех элементов. По монотонности $f(a) \preceq f(x) \preceq x$: последнее верно, потому что x принадлежит P . Значит, $f(a)$ — нижняя грань P : она не превосходит каждого элемента. Инфимум — наибольшая из нижних граней, поэтому $f(a) \preceq a$.

С другой стороны, $f(a) \preceq a \rightarrow f(f(a)) \preceq f(a)$ по монотонности. Значит, $f(a) \in P$. Тогда $a \preceq f(a)$, ведь инфимум меньше всех элементов P .

Из $f(a) \preceq a, a \preceq f(a)$ следует $f(a) = a$ по антисимметричности: a — неподвижная точка. Если b — неподвижная точка, то $f(b) = b$, значит $f(b) \preceq b$ и $b \in P$. Тогда $a \preceq b$: a действительно наименьшая.

Для наибольшей неподвижной точки конструкция двойственна: инфимум заменяем на супремум, а множество P — на $\{x \mid x \preceq f(x)\}$.

Пример Итерация к неподвижной точке

На решётке $\mathcal{P}(\{1, 2, 3\})$ рассмотрим монотонную функцию $f(X) = X \cup \{1\}$. Итерация от низа даёт цепочку:

$$\emptyset \rightarrow f(\emptyset) = \{1\} \rightarrow f(\{1\}) = \{1\} \rightarrow \dots$$

Уже на втором шаге множество перестало меняться: $f(\{1\}) = \{1\}$, и $\{1\}$ — неподвижная точка. Она наименьшая: любая неподвижная точка обязана содержать 1, иначе f добавила бы его, а $\{1\}$ — наименьшее из таких множеств.

Та же картина в анализе программ: решётка — множества фактов, функция — эффект цикла, и итерация от низа сходится к наименьшей неподвижной точке.

Анализ программы с циклами и рекурсией — это в точности поиск неподвижной точки. Состояние программы на каждом шаге — элемент абстрактной решётки. Тело цикла — монотонная функция на этой решётке. Результат анализа после цикла — наименьшая неподвижная точка этой функции.

Без теоремы Кнастера–Тарского абстрактная интерпретация висела бы в воздухе: мы применяли бы итеративный алгоритм, не зная, почему он сходится и к чему. С ней мы получаем гарантию: монотонная функция на полной решётке всегда имеет наименьшую неподвижную точку. В конечной решётке итеративный процесс $\perp \preceq f(\perp) \preceq f(f(\perp)) \preceq \dots$ достигает её за конечное число шагов. В бесконечной полной

решётке итерация может не остановиться за конечное число шагов. Наименьшая неподвижная точка строится трансфинитной итерацией от $\perp$:

$$\begin{aligned} f^0(\perp) &= \perp, \\ f^{\alpha+1}(\perp) &= f(f^\alpha(\perp)), \\ f^\lambda(\perp) &= \sup_{\alpha < \lambda} f^\alpha(\perp) \end{aligned}$$

где λ — предельный ординал.

Ординалы здесь не случайны: для достижения неподвижной точки может потребоваться больше чем ω итераций. Этот результат связывает теорию порядков с трансфинитной теорией. Полное введение в ординалы, предельные ординалы и трансфинитную индукцию даётся в главе 29.

Примечание Неподвижные точки и языки программирования

Неподвижные точки возникают в computer science в нескольких обличьях:

- *Абстрактная интерпретация*: состояние программы приближается элементом решётки. Тело цикла — функция на решётке. Анализ — вычисление lfp.
- *Денотационная семантика*: значение рекурсивной функции — наименьшая неподвижная точка функционала, построенного по её определению (Скотт, 1970-е).
- *Теория типов*: рекурсивные типы (например, `data List a = Nil | Cons a (List a)`) определяются как неподвижные точки функторов на категории типов.
- *Model checking*: вычисление достижимых состояний — наименьшая неподвижная точка оператора перехода. Вычисление инвариантов — наибольшая неподвижная точка.

§ 8.11 Первый взгляд на абстрактную интерпретацию

О программе можно рассуждать, не запуская её. Рассмотрим программу с делением: $x := \text{read}(), y := \frac{100}{x}$. Хочется гарантировать, что деление никогда не выполнится с $x = 0$. Запуск этого не доказывает: программа читает вход, и перебрать все возможные значения нельзя. Рассуждение о программе без запуска называется статическим анализом.

Идея в том, чтобы заменить множество возможных значений элементом решётки и пересчитывать его по операторам программы. Самый простой домен — знаки: «отрицательное», «ноль», «положительное», упорядоченные по включению представимых множеств. Слияние ветвей условного оператора — это супремум. Циклы — неподвижные точки. Если перед делением знак x — отрицательный или положительный, деление безопасно. Если может быть ноль — возможно падение. Ошибиться анализ может лишь в одну сторону: вердикт «безопасно» даётся только с гарантией, а «возможно падение» бывает ложной тревогой.

Знаки — лишь первый шаг. Для других вопросов нужны другие домены: интервалы против выхода за границы массива, классы вычетов, произведения доменов. Как устроены домены, что означает «корректный» анализ и как заставить циклы сходиться — в главе об абстрактной интерпретации (31).

§ 8.12 Итоги главы

Частичный порядок описывает предшествование, но его главный ресурс — несравнимость: элементы, не связанные порядком, независимы, и именно эта независимость допускает параллелизм. Антицепи переводят несравнимость на язык чисел: ширина порядка — размер наибольшей антицепи, и теорема Дилуорса уравнивает её с наименьшим числом цепей, на которые разбивается множество. Три аксиоматики — частичный, строгий и линейный порядок — дают разную силу: линейный порядок запрещает несравнимость вовсе, и потому частичный порядок ценнее там, где есть параллельные ветви.

Диаграмма Хассе сжимает порядок до скелета из отношений покрытия, опуская транзитивные рёбра, и делает структуру видимой. На ней мы научились находить экстремальные элементы и грани: минимальный и максимальный отличаются от наименьшего и наибольшего, а супремум и инфимум — точные границы, которых может и не быть.

Там, где у каждой пары элементов есть супремум и инфимум, порядок вырастает в решётку — алгебраическую структуру, пригодную для вычислений — и булевы алгебры оказываются её частным случаем. Топологическая сортировка разворачивает скелет обратно в линейную последовательность, а теорема Кнастера–Тарского гарантирует неподвижные точки монотонных функций.

Порядки связывают отношения (глава 5) с графами (глава 9). Решётки с монотонными функциями подводят к неподвижным точкам — к абстрактной интерпретации (глава 31), где состояние программы описывается элементом решётки, а анализ сводится к поиску неподвижной точки. Тот же аппарат работает в планировщиках, компиляторах и распределённых системах.

Проверка Проверьте себя

1. Почему частичный порядок допускает параллелизм, а линейный — нет?
2. Чем супремум отличается от наибольшего элемента и может ли супремум не существовать?
3. Чем минимальный элемент отличается от наименьшего?
4. Что гарантирует теорема Кнастера–Тарского?
5. Почему диаграмма Хассе не рисует все рёбра отношения?
6. Чем словарный порядок отличается от `shortlex` и у какого из них всякое непустое множество строк имеет наименьший элемент?

Что дальше?

От упорядоченных структур мы переходим к универсальному языку связей — графам. Диаграмма Хассе, которую мы только что рисовали, — это уже граф: вершины и рёбра, только специального вида. В главе 9 мы изучим общую теорию: деревья, двудольные и планарные графы, эйлеровы и гамильтоновы циклы, алгоритмы BFS, DFS и Дейкстры, раскраску — инструментарий для моделирования сетей, зависимостей и маршрутов.

§ 8.13 Упражнения

Частичные порядки и аксиомы.

1. Проверьте, что отношение делимости является частичным порядком на натуральных числах: выполните рефлексивность, антисимметричность и транзитивность.
2. Является ли отношение делимости частичным порядком на целых числах? Указание: проверьте антисимметричность на паре $2, -2$.
3. Приведите пример бинарного отношения, которое рефлексивно и транзитивно, но не антисимметрично.
4. * Докажите, что отношение включения $\subseteq$ на множестве всех подмножеств является частичным порядком. Является ли оно линейным?
5. Найдите ширину ЧУМ делителей числа 12: предъявите антицепь наибольшего размера и разбиение множества на то же число цепей.

Диаграммы Хассе и экстремальные элементы.

6. Нарисуйте диаграмму Хассе для ЧУМ $(\{1, 2, 3, 4, 6, 12\}, |)$. Найдите минимальные, максимальные, наибольший и наименьший элементы.
7. Приведите пример частичного порядка, в котором есть ровно два минимальных элемента и нет наименьшего.
8. Докажите, что в частичном порядке наименьший элемент, если он существует, единственен.
9. * Исследуйте: сколько существует частичных порядков на множестве из n элементов? Для $n = 1, 2, 3, 4$ вычислите точные значения.

Грани, супремум, инфимум и решётки.

10. Найдите все верхние и нижние грани, супремум и инфимум для пары 4 и 6 в ЧУМ $(\{1, 2, 3, 4, 6, 12\}, |)$.
11. Является ли множество всех подмножеств $\{1, 2, 3\}$ с отношением включения дистрибутивной решёткой? Проверьте аксиомы.

12. Докажите, что $(\mathbb{N}^+, |)$ является решёткой. Что является супремумом и инфимумом двух чисел?
13. * Докажите законы поглощения в любой решётке: $a \vee (a \wedge b) = a$, $a \wedge (a \vee b) = a$.
14. * Докажите теорему Кнастера–Тарского для конечных решёток: любое монотонное отображение $f : L \rightarrow L$ имеет неподвижную точку.

Лексикографический порядок.

15. Сравните строки apple и apricot в словарном порядке и в shortlex.
16. * Приведите пример бесконечного множества строк без наименьшего элемента в словарном порядке.

Топологическая сортировка.

17. Предложите топологическую сортировку для следующего частичного порядка (зависимости курсов):
 Алгебра $\rightarrow$ Дискретная математика.
 Алгебра $\rightarrow$ Линейная алгебра.
 Программирование $\rightarrow$ Алгоритмы.
 Дискретная математика $\rightarrow$ Алгоритмы.
 Дискретная математика $\rightarrow$ Логика.
18. Докажите, что в конечном ЧУМ топологическая сортировка существует всегда (подсказка: индукция или жадный алгоритм).
19. * Придумайте частичный порядок, для которого существует ровно одна топологическая сортировка, и докажите это.
20. * Найдите ошибку в «доказательстве» того, что в любом непустом частичном порядке есть максимальный элемент. Построим цепь: на каждом шаге, если есть элемент строго больше текущего, добавим его. Если процесс не завершается, цепь бесконечна, и её супремум a^* является максимальным элементом. Почему супремум бесконечной цепи может не существовать в P , и почему даже существующий супремум не обязан быть максимальным?

ПРОЕКТ Статический анализ знаков для поиска деления на ноль

Постройте статический анализатор, который отвечает на вопрос, может ли переменная x быть нулём перед делением, не выполняя программу. Соберите его из решётки знаков, монотонных функций и теоремы Кнастера–Тарского о наименьшей неподвижной точке.

① Решётка знаков

Реализуйте множество знаков («отрицательное», «ноль», «положительное»), упорядоченное по включению представляемых множеств, с операци-

ями $\vee$ и $\wedge$. Продумайте, почему получается решётка и что является её низом и верхом.

② *Абстрактная семантика*

Задайте знак суммы, произведения и деления. Продумайте, какой знак даёт деление на «может быть ноль». Отображение такого деления в низ решётки немонотонно: итерация к наименьшей неподвижной точке перестаёт сходиться. Продумайте альтернативу: верх решётки с отдельным предупреждением об ошибке.

③ *Программа как функция*

Представьте программу (присваивания, if, while) монотонной функцией на решётке: слияние ветвей условного оператора — супремум, тело цикла — итерация функции. Объясните, почему построенная функция монотонна. Пометьте каждую точку программы флагом достижимости: состояние с низом решётки означает недостижимую ветку, и её тело при анализе пропускается.

④ *Неподвижная точка*

Вычислите наименьшую неподвижную точку итерацией от низа: $\perp \leq f(\perp) \leq f(f(\perp)) \leq \dots$. Покажите, что на конечной решётке итерация останавливается, и выведите знак x в каждой точке программы.

⑤ *Проверка программ*

Соберите корпус программ и ответьте для каждой, может ли x быть нулём перед делением. Вердикты бывают трёх видов: знак делителя 0 — деление на ноль неизбежно, реальная ошибка. Знак $\top$ — делитель может быть нулём, но анализатор мог просто не суметь исключить это — решётка знаков теряет информацию (ложное срабатывание). Знак $+$ или знак $-$ — деление доказано безопасным.

Включите в корпус программу с guard'ом $x \neq 0$: знак $\{-, +\}$ не представим в решётке и округляется до $\top$ — ложное срабатывание, а $x > 0$ доказывается даже внутри цикла. Продумайте, как устранить ложные срабатывания, уточнив решётку.

Итог: статический анализатор, который находит или исключает деление на ноль в небольших программах, используя решётку знаков и теорему о неподвижной точке.

IX

Графы

Схема метро не похожа на географическую карту. Станции сдвинуты, линии выпрямлены, расстояния искажены. Но пассажир находит путь без труда: важно только то, кто с кем соединён. Рисунок, где значение имеет только связность, а не геометрия, — это граф.

Отношения между объектами можно описать словами. Можно нарисовать таблицу смежности — она скажет, кто с кем связан. Этого достаточно, чтобы передать информацию. Но этого недостаточно, чтобы *увидеть* структуру. Таблица перечисляет связи, но не показывает их форму. Рисунок графа даёт то, что таблица скрывает: геометрию связей, циклы, симметрии. А форма — то, как именно устроена связность целого — часто и есть *главное*.

Граф решает эту проблему одним шагом: оставляет только вершины и рёбра. Вершина — точка без размера и положения, ребро — связь без формы и длины. Двух типов объектов достаточно, чтобы сформулировать теорему о четырёх красках: она продержалась нерешённой больше века и была доказана компьютерным перебором.

В предыдущей главе (8) мы изучали порядки — структуры, выстраивающие элементы в иерархии: один больше, другой меньше, третий несравним. Диаграмма Хассе — это уже граф: вершины, соединённые рёбрами, только специального вида. Граф общего вида снимает ограничение, которое порядок накладывает на связи: ребро соединяет любую пару вершин, без транзитивности и без иерархии. Порядок отвечал на вопрос «что раньше?», эквивалентность — на вопрос «одинаково ли?». Граф отвечает на более общий вопрос: «кто с кем связан?». Иерархия оказывается частным случаем связей.

Из точек и линий строятся интернет (маршрутизаторы и кабели), социальные сети (люди и связи), карты дорог, зависимости программ, электрические цепи. Граф зависимостей в системе сборки подсказывает, какие модули компилировать раньше. Социальный граф показывает, кто в сети влиятелен и какие группы образуют сообщества. Веб-граф, где страницы соединены ссылками, стоит за поисковыми системами.

Теория графов даёт меры и инструменты: степень вершины, диаметр, компоненты связности, пути и циклы, деревья, поиск в ширину и в глубину.

Две задачи об обходе различаются одним словом: «пройти каждое ребро ровно один раз» и «посетить каждую вершину ровно один раз». Первая решается за линейное

время, вторая — нет (при стандартном предположении $P \neq NP$). Разный результат при почти одинаковой формулировке — одна из нитей, ведущих к теории сложности (глава 30).

ИСТОРИЯ Кёнигсбергские мосты и рождение теории графов

В 1736 году Леонард Эйлер представил решение задачи о кёнигсбергских мостах: можно ли пройти по всем семи мостам города, не проходя ни по одному дважды? Эйлер показал, что это невозможно, и в процессе рассуждения абстрагировался от географии: участки суши стали вершинами, мосты — рёбрами. Статья «*Solutio problematis ad geometriam situs pertinentis*» считается первой работой по теории графов, хотя сам термин «граф» появится лишь полтора века спустя.

Сэр Уильям Роуэн Гамильтон в 1857 году изобрёл «Икосианскую игру» — головоломку, породившую понятие гамильтонова цикла (история — ниже, в исторической справке). Артур Кэли в 1889 году опубликовал работу о перечислении деревьев — формула Кэли n^{n-2} для числа помеченных деревьев стала одним из первых нетривиальных результатов теории графов. К химии дерева он применил ещё раньше, в 1850–1870-е годы: деревья моделировали структурные формулы органических соединений.

Густав Кирхгоф в 1845 году, разрабатывая законы электрических цепей, независимо пришёл к графовым понятиям: циклы в графе цепи соответствуют законам Кирхгофа. Денеш Кёниг в 1936 году опубликовал «*Theorie der endlichen und unendlichen Graphen*» — первую монографию, где теория графов была представлена как самостоятельная математическая дисциплина с собственной системой понятий и результатов. За двести лет головоломка о мостах превратилась в универсальный язык дискретных структур.

Карта дорог, сеть труб, схема электрической цепи, граф вызовов программы — всё это один и тот же объект, и инструменты у него одни. Булева алгебра следующей главы тоже окажется графом: гиперкуб, в котором рёбра соединяют битовые строки, различающиеся ровно одним битом (глава 10). Рисунок графа можно перерисовать сотней способов, и структура — пары вершин и рёбер — не изменится. Именно поэтому граф работает как язык: он переносит форму сети с одной карты на другую, не искажая структуры. Что можно узнать о сети, глядя только на пары вершин и рёбер?

ОБЗОР ГЛАВЫ

В этой главе мы определим граф и его разновидности, а затем решим, как хранить его в памяти: выбор между матрицей и списками смежности определяет скорость всех дальнейших алгоритмов. Степень, диаметр и компоненты связности — параметры, по которым измеряется устройство графа, а пути и циклы описывают дви-

жение по нему. Поиск в ширину и в глубину превращают граф в структуру данных, топологическая сортировка выстраивает зависимости в порядок сборки, деревья дают минимальный каркас, а эйлеровы и гамильтоновы циклы — две задачи об обходе с разными классами сложности. Затем к рёбрам приходит вес: Дейкстра и Беллман–Форд найдут кратчайшие пути, потоки и паросочетания — двойственность «max = min». Двудольные, планарные и раскрашиваемые графы опишут паросочетания, печатные платы и конфликты. В конце главы графы встречаются с приложениями — маршрутизация, компиляторы, PageRank — а факультативный раздел о случайных графах покажет, как вероятность управляет рождением связности. Model checking — верификацию систем обходом графа состояний — мы разберём в главе 33.

После этой главы вы сможете:

1. записывать граф матрицей и списками смежности и выбирать представление по задаче;
2. измерять граф: степень, диаметр, компоненты связности;
3. различать пути, циклы, деревья и остовные каркасы;
4. применять поиск в ширину и в глубину для обхода и проверки связности;
5. различать эйлеровы и гамильтоновы пути и объяснять, почему сложность задач о них разная.

§ 9.1 Основные определения

Любой граф начинается с двух списков: перечня вершин и перечня рёбер. Всё остальное — степени, связность, циклы, раскраски — вырастает из этих двух списков и правил их взаимодействия. В этом разделе мы определим граф, его разновидности и способы хранения.

9.1.1 Графы и их разновидности

Граф — это множество точек (вершин) и соединяющих их линий (рёбер). Это определение настолько общее, что графы оказываются применимы повсюду: от моделирования социальных сетей до представления автоматов и пространств состояний программ.

ОПРЕДЕЛЕНИЕ 9.1 Граф

Графом называется пара

$$G = (V, E),$$

где V — множество вершин, а E — множество рёбер.

ОПРЕДЕЛЕНИЕ 9.2 Неориентированный граф

Граф называется **неориентированным**, если его рёбра — неупорядоченные пары $\{u, v\}$ (связь симметрична).

ОПРЕДЕЛЕНИЕ 9.3 Ориентированный граф

Граф называется **ориентированным** (орграфом), если его рёбра (дуги) — упорядоченные пары (u, v) (связь направлена).

Простейший граф, который можно построить по этому определению, — треугольник: три вершины, три ребра, каждая вершина связана с двумя другими.

Пример Граф-треугольник

$V = \{1, 2, 3\}$, $E = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$ — неориентированный граф-треугольник. Полный граф K_n имеет все $\frac{n(n-1)}{2}$ возможных рёбер между n вершинами.

- K_3 — треугольник.
- K_4 — тетраэдр.
- Полный граф на пяти вершинах K_5 показан на рис. 51.

Примечание Почему именно (V, E) ?

Почему определение графа именно такое — множество вершин и множество рёбер, и больше ничего? Можно было бы добавить веса, ориентацию, атрибуты вершин — и все эти расширения действительно полезны на практике. Но минимальное определение фиксирует единственную существенную абстракцию: бинарное отношение на конечном множестве. Всё остальное — веса, ориентация, разметка — надстраивается поверх этой базы. Альтернативный путь — определять граф сразу как матрицу смежности, но это привязывает к конкретному представлению. Пара (V, E) остаётся инвариантом, не зависящим от реализации.

Но именно эта «надстройка» — веса — и создаёт большинство практических задач. Сеть дорог, где время проезда зависит от перегона, граф зависимостей, где у каждого шага своя стоимость, — всё это графы с числами на рёбрах.

ОПРЕДЕЛЕНИЕ 9.4 Взвешенный граф

Взвешенным графом называется пара (G, w) , где $G = (V, E)$ — граф, а $w : E \rightarrow \mathbb{R}$ — весовая функция, сопоставляющая каждому ребру число. Вес ребра $e = \{u, v\}$ обозначают $w(u, v)$ или $w(e)$.

Веса могут означать что угодно: время, стоимость, длину, пропускную способность. Смысл задаёт задача, а не граф. Неотрицательные веса (время, длина, стоимость) — частый случай. Ниже увидим, что алгоритмы по-разному ведут себя с отрицательными весами.

Абстракция (V, E) применима не только к искусственным системам. Природа использовала ту же структуру задолго до появления математики.

Под землёй лес — это граф.

Деревья не растут изолированно. Между корнями — микоризная сеть: грибные гифы соединяют берёзу с дубом, сосну с елью. По этим каналам идут вода, минералы, химические сигналы. Материнское дерево узнаёт свой саженец и направляет ему больше углерода — через общие грибные нити. В этом графе вершины — деревья, а рёбра — грибные гифы: одна и та же структура описывает лес, сеть дорог и схему электрической цепи.

Базовая пара (V, E) — это минимальный инвариант, но на практике одного определения недостаточно. Разные задачи требуют разных вариантов графа. Петли и кратные рёбра не нужны при моделировании социальных сетей: человек не дружит сам с собой, и между двумя людьми либо есть дружба, либо нет. Но они необходимы в других областях:

1. электрические цепи: между двумя узлами может идти несколько проводников.
2. автоматы: петля означает переход в то же состояние.

Аналогично, ориентация рёбер критична для моделирования потоков данных и безразлична для моделирования симметричных отношений. Выбор типа графа — это первый проектировочный шаг: он определяет, какие вопросы можно задать и какие алгоритмы применимы.

В зависимости от того, разрешены ли петли (рёбра из вершины в себя) и кратные рёбра (несколько рёбер между одной парой вершин), выделяют несколько типов графов:

ОПРЕДЕЛЕНИЕ 9.5 Простой граф

Граф называется **простым**, если в нём нет петель и кратных рёбер, то есть каждая пара вершин соединена не более чем одним ребром.

ОПРЕДЕЛЕНИЕ 9.6 Мультиграф

Мультиграфом называется граф, в котором разрешены петли и кратные рёбра.

Мультиграф полезен для моделирования, например, нескольких дорог между городами. Петли и кратные рёбра — вопрос о том, какие рёбра разрешены. Следующее различие касается устройства вершин: все ли вершины имеют одинаковое число соседей.

ОПРЕДЕЛЕНИЕ 9.7 k -регулярный граф

Граф называется **k -регулярным**, если степень каждой вершины равна k , то есть все вершины имеют одно и то же число соседей.

Пример Типы графов

- Простой граф — граф Петерсена на рис. 30: нет петель и кратных рёбер.

- Мультиграф — граф кёнигсбергских мостов на рис. 39: он может иметь несколько рёбер между одной парой вершин, что соответствует нескольким мостам между участками суши.
- k -регулярный граф: граф Петерсена 3-регулярен, K_4 (полный граф на 4 вершинах) 3-регулярен, C_n (цикл) 2-регулярен.

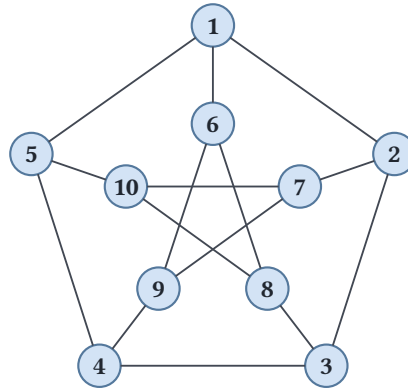


Рис. 30. Граф Петерсена.

Граф Петерсена на рисунке 30 3-регулярен: каждая вершина имеет ровно три соседа. Степень вершины — одна из базовых характеристик графа.

ОПРЕДЕЛЕНИЕ 9.8 Степень вершины

Степенью вершины v называется количество инцидентных ей рёбер. Обозначается она $\deg(v)$.

ОПРЕДЕЛЕНИЕ 9.9 Полустепень исхода и захода

В ориентированном графе различают **полустепень исхода** (*out-degree*) $\deg^+(v)$ — количество исходящих дуг, и **полустепень захода** (*in-degree*) $\deg^-(v)$ — количество входящих дуг.

Пример Степени в графе Петерсена

- В треугольнике K_3 из каждой вершины выходят два ребра — степень каждой равна 2.
- Граф Петерсена на рис. 30 3-регулярен: каждая вершина имеет степень ровно 3.
- В графе-звезде из 5 вершин: центр имеет степень 4, каждый лист — степень 1.
- В ориентированном графе на рис. 32. $\deg^+(1) = \deg^-(1) = 2$ (две дуги исходят к 2 и 4, две входят от 3 и 6).

Лемма о рукопожатиях — простейший пример метода двойного счёта. Интуитивно: каждое ребро добавляет по единице к степени каждой из двух своих конечных вершин.

ТЕОРЕМА 9.1 Лемма о рукопожатиях

$\sum_{v \in V} \deg(v) = 2|E|$. Следствие: число вершин нечётной степени всегда чётно.

Доказательство

Докажем методом двойного счёта: посчитаем рёбра двумя способами.

Каждое ребро инцидентно ровно двум вершинам (своим концам). При суммировании степеней всех вершин каждое ребро учитывается дважды — по разу для каждой из двух вершин, которые оно соединяет. Следовательно, сумма степеней равна удвоенному числу рёбер.

Для следствия: сумма степеней всех вершин чётна (равна $2|E|$). Разобьём вершины на два множества: с чётной степенью и с нечётной. Сумма степеней вершин с чётной степенью чётна. Значит, сумма степеней вершин с нечётной степенью тоже должна быть чётной. Но сумма нечётных чисел может быть чётной только если количество слагаемых чётно.

Двойной счёт — один из основных инструментов теории графов: многие тождества со степенями доказываются фразой «посчитаем рёбра двумя способами».

ЛОВУШКА Последовательность степеней: необходимая чётность — не критерий

Из леммы о рукопожатиях следует, что сумма степеней любой реализуемой последовательности чётна. Обратное неверно: последовательность $(3, 3, 3, 1)$ имеет чётную сумму, но графа с такой последовательностью не существует. В самом деле, вершина степени 1 соединена ровно с одной вершиной, а каждая из трёх остальных должна быть смежна со всеми тремя — в том числе и с этой. Полный критерий реализуемости даёт теорема Эрдёша–Галлаи, а жадный алгоритм Хавела–Хакими ещё и строит граф по последовательности (раздел 9.2). И последовательность степеней не определяет граф однозначно: два неизоморфных графа могут иметь одну и ту же последовательность степеней.

Последняя оговорка поднимает вопрос, который мы пока обходили: когда два графа — это «один и тот же» граф? Рисунки различаются: вершины названы разными буквами, рёбра проведены иначе, — а структура может совпадать.

ОПРЕДЕЛЕНИЕ 9.10 Изоморфизм графов

Графы $G = (V, E)$ и $H = (W, F)$ называются **изоморфными**, обозначается $G \sim H$, если существует биекция $\varphi : V \rightarrow W$, сохраняющая рёбра: $\{u, v\} \in E$ тогда и только тогда, когда $\{\varphi(u), \varphi(v)\} \in F$. Для орграфов биекция обязана сохранять направления дуг: $(u, v) \in E$ тогда и только тогда, когда $(\varphi(u), \varphi(v)) \in F$.

Изоморфность — отношение эквивалентности на множестве графов: тождественная биекция, обратная и композиция биекций дают рефлексивность, симметричность и транзитивность. Класс эквивалентности собирает все копии одной структуры — графы, различающиеся только именами вершин. В главе 5 это отношение вводилось как пример эквивалентности на графах. Теперь графы и биекции определены, и классы — «непомеченные» графы — получили точный смысл.

Пример Различение графов

Цепочка из четырёх вершин и звезда с центром и тремя листьями: в обоих графах по 4 вершины и 3 ребра, оба — деревья, но последовательности степеней различны — $(2, 2, 1, 1)$ против $(3, 1, 1, 1)$. Биекция обязана переводить вершины в вершины той же степени, поэтому её не существует: графы не изоморфны. Здесь вопрос закрыл один счётный инвариант.

Цикл из шести вершин и два отдельных треугольника: и здесь все простые инварианты совпадают — 6 вершин, 6 рёбер, все степени равны 2, — но первый граф связан, а второй нет. Изоморфизм переводит связный граф в связный, поэтому и эти графы не изоморфны. Совпадение инвариантов само по себе изоморфизм не доказывает: чтобы его предьявить, нужно указать биекцию, а кандидатов — $n!$ вариантов для n вершин.

Теперь, когда граф определён математически, возникает практический вопрос: как представить его в памяти компьютера? Математическое определение — пара множеств (V, E) — не диктует конкретную структуру данных. А выбор структуры данных критически влияет на скорость алгоритмов: проверка смежности двух вершин, перебор соседей, добавление и удаление рёбер — все эти операции имеют разную стоимость в разных представлениях. Основных вариантов три, и выбор между ними сводится к одному вопросу — насколько граф плотный.

9.1.2 Представления графов в памяти

Три основных способа — компромисс между памятью и скоростью типичных операций.

- **Список рёбер:** хранить все рёбра как пары вершин.

Память: $O(|E|)$. Проверка смежности двух вершин: $O(|E|)$ (нужно просмотреть все рёбра). Подходит для алгоритмов, перебирающих все рёбра (Крускал).

- **Матрица смежности:** матрица $n \times n$, где $A_{ij} = 1$, если вершины i, j соединены ребром.

Память: $O(n^2)$. Проверка смежности: $O(1)$. Свойство: $(A^k)_{ij}$ равно числу маршрутов длины k из i в j .

- **Список смежности:** для каждой вершины — список её соседей.

Память: $O(n + |E|)$. Проверка смежности: $O(\deg(v))$. Наилучший вариант для разреженных графов ($|E| \ll n^2$), стандартный выбор для большинства приложений.

Пример Один граф — три представления

Рассмотрим граф с вершинами $\{1, 2, 3, 4\}$ и рёбрами $\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 3\}$ (квадрат с диагональю). Один и тот же граф в трёх форматах:

- **Список рёбер:** $(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)$ — пять записей, $O(|E|)$ памяти.
 - **Матрица смежности:** матрица 4×4 , где единицы стоят в позициях $(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (3, 1), (3, 2), (3, 4), (4, 1), (4, 3)$ — шестнадцать ячеек, $O(n^2)$ памяти.
 - **Список смежности:**
 - 1 : [2, 3, 4]
 - 2 : [1, 3]
 - 3 : [1, 2, 4]
 - 4 : [1, 3]
- четыре списка, суммарно десять записей, $O(n + |E|)$ памяти.

§ 9.2 * Реализуемые последовательности степеней: Хавел–Хакими

Проектировщику сети часто известны не рёбра, а потребности: сколько линий связи должно подойти к каждому узлу. Социолог с анонимизированной выжимкой опроса знает лишь число знакомств каждого респондента, но не сами знакомства. В обоих случаях рёбер в руках нет — есть список чисел, и вопрос один: существует ли граф с такими степенями вершин? Перебирать графы безнадёжно, но ответ даёт жадная арифметика: последовательность либо за несколько вычитаний превращается в граф, либо разоблачается.

ОПРЕДЕЛЕНИЕ 9.11 Графовая последовательность

Последовательность неотрицательных целых чисел называется **графовой**, если она является последовательностью степеней вершин некоторого простого графа.

Не всякий список честен: из леммы о рукопожатиях следует, что сумма графовой последовательности чётна, и ни одно число не превосходит $n - 1$, но оба условия недостаточны — контрпример $(3, 3, 3, 1)$ разобран в ловушке выше. Полный ответ сложился в три шага: критерий Эрдёша–Галлаи⁵³ в 1960 году и жадные алгоритмы

⁵³P. Erdős, T. Gallai. «Gráfok előirt foksámú pontokkal». Matematikai Lapok, 1960.

⁵⁴V. Havel. «Poznámka o existenci konečných grafů». Časopis pro pěstování matematiky, 1955.

построения — Хавел⁵⁴ в 1955-м и Хакими⁵⁵ в 1962-м. Идея алгоритма — индукция: вопрос о n числах сводится к вопросу о $n - 1$ числах.

ТЕОРЕМА 9.2 Редукция Хавела–Хакими

Пусть $d_1 \geq d_2 \geq \dots \geq d_n$ и $d_1 \leq n - 1$. Последовательность $d_1, \dots, d_n$ графова тогда и только тогда, когда графова последовательность

$$d' = (d_2 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n),$$

полученная вычитанием единицы из d_1 чисел, следующих за d_1 .

Доказательство

Докажем эквивалентность в обе стороны.

Пусть сначала графова d' , и пусть H — граф с такой последовательностью степеней. Добавим к H новую вершину v и соединим её с вершинами, реализующими первые d_1 элементов d' , — теми, у которых степень стала на единицу меньше нужной. Новая вершина получает степень d_1 , степени её соседей возвращаются к $d_2, \dots, d_{d_1+1}$, остальные вершины не затронуты. Получился простой граф с последовательностью $d_1, \dots, d_n$.

Обратно: пусть графова d , и пусть G — граф со степенями $d_1, \dots, d_n$. Если $d_1 = 0$, достаточно удалить изолированную вершину, и останется реализация d' . Пусть $d_1 \geq 1$, и пусть v — вершина степени d_1 . Назовём *обязательными* те d_1 вершин, которые реализуют степени $d_2, \dots, d_{d_1+1}$. При равных степенях обязательными можно объявить любые из равных, и этой свободой мы воспользуемся. Покажем, что G можно перестраивать, не меняя степеней, пока v не окажется смежной со всеми обязательными.

Пока таких соседей меньше d_1 , возьмём обязательную вершину w , не смежную с v . Обязательных вершин ровно d_1 — столько же, сколько соседей у v , — значит, среди соседей v есть необязательная вершина x . По упорядоченности $d_2 \geq \dots \geq d_n$ степень обязательной вершины не меньше степени необязательной, поэтому возможны два случая.

Случай 1: $\deg(w) > \deg(x)$. Соседей у w строго больше, чем у x , поэтому все они не помещаются среди соседей x : найдётся вершина y , смежная с w , но не с x . Заменим рёбра $\{v, x\}$ и $\{w, y\}$ на $\{v, w\}$ и $\{x, y\}$. Оба удалённых ребра существовали, обе добавляемые пары рёбрами не были, и каждая из четырёх вершин отдала одно ребро и получила другое: все степени сохранились, а обязательных соседей у v стало на одного больше.

⁵⁵S. L. Hakimi. «On realizability of a set of integers as degrees of the vertices of a linear graph». Journal of the Society for Industrial and Applied Mathematics, 1962.

Случай 2: $\deg(w) = \deg(x)$. Достаточно поменять w и x именами. Степени равны, поэтому каждое из чисел $d_2, \dots, d_n$ остаётся при своём месте, а смежная с v вершина теперь несёт имя w из обязательного набора — обязательных соседей снова стало на одного больше.

Каждый шаг добавляет v одного обязательного соседа, и шагов не больше d_1 : процесс конечен. В полученной реализации v смежна ровно с обязательными вершинами, и её удаление даёт граф со степенями d' .

Алгоритм Хавела–Хакими повторяет редукцию: упорядочиваем числа по убыванию, отсекаем первое и вычитаем единицу из нужного количества следующих. Признак успеха — одни нули, признак отказа — отрицательное число, ведь степень не бывает отрицательной. Если максимальное число достигло длины списка, построение невозможно даже начать: у вершины не хватит потенциальных соседей. При равных степенях порядок выбора не влияет на ответ: после редукции получится один и тот же набор чисел. Рёбра запоминаем на ходу: отсечённая вершина соединяется с теми, у кого вычли единицу.

Протрассируем алгоритм на списке $(4, 4, 3, 2, 2, 1)$: числа уже упорядочены, вершины обозначим $v_1, \dots, v_6$.

- v_1 (степень 4): соединяем с v_2, v_3, v_4, v_5 , список сокращается до $(3, 2, 1, 1, 1)$.
- v_2 (степень 3): соединяем с v_3, v_4, v_5 , список сокращается до $(1, 1, 0, 0)$.
- v_3 (степень 1): соединяем с v_6 , список сокращается до $(0, 0, 0)$.

Нули означают успех: последовательность графова, а соединения из трассировки дают граф G_1 с рёбрами

$$\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_2, v_5\}, \{v_3, v_6\}.$$

Проверка по рукопожатиям сходится: у v_1 и v_2 по четыре соседа, у v_3 — три, у v_4 и v_5 — по два, у v_6 — один. Для списка $(3, 3, 3, 1)$ из ловушки выше алгоритм работает детектором: первая редукция даёт $(2, 2, 0)$, вторая — $(1, -1)$, и на отрицательном числе вычисление обрывается: никакой граф таких степеней не имеет.

Пример Одна последовательность — два графа

Последовательность $(4, 4, 3, 2, 2, 1)$ реализует не только G_1 из трассировки. Возьмём граф G_2 с рёбрами

$$\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_2, v_6\}, \{v_3, v_5\}.$$

Прямой подсчёт даёт тот же список степеней: $4, 4, 3, 2, 2, 1$. Но графы не изоморфны: изоморфизм переводит единственную вершину степени 1 в единственную вершину степени 1, а её соседа — в соседа, и степени соседей обязаны совпасть. В G_1 сосед v_6 имеет степень 3, а в G_2 — степень 4.

Алгоритм отвечает на вопрос построением, но иногда построение излишне — достаточно быстрой проверки. Её даёт критерий Эрдёша–Галлаи: набор неравенств вместо построения.

ТЕОРЕМА 9.3 Критерий Эрдёша–Галлаи

Последовательность $d_1 \geq d_2 \geq \dots \geq d_n$ неотрицательных целых чисел графова тогда и только тогда, когда её сумма чётна и для каждого k от 1 до n выполняется

$$\sum_{i=1}^k d_i \leq k(k-1) + \sum_{i=k+1}^n \min(d_i, k).$$

Слева в неравенстве — суммарная потребность k самых связанных вершин, справа — максимум, что они могут получить: $k(k-1)$ от рёбер внутри этой группы и не более k от каждой из остальных вершин. Доказательство критерия мы приводить не будем: алгоритм Хавела–Хакими уже решает за нас и проверку, и построение. Ловушка из раздела о рукопожатиях закрыта с обеих сторон: чётность суммы оказалась лишь первым необходимым условием, а полный ответ дают неравенства Эрдёша–Галлаи и жадная редукция, которая заодно строит граф.

§ 9.3 Пути, циклы, связность

Определив структуру графа, мы переходим к его динамическим свойствам: как можно «ходить» по графу и насколько граф «целостен».

ОПРЕДЕЛЕНИЕ 9.12 Маршрут

Маршрутом (walk) называется последовательность вершин $v_0, v_1, \dots, v_k$, где $\{v_{i-1}, v_i\} \in E$ для каждого i . Длина маршрута равна k (количество пройденных рёбер).

ОПРЕДЕЛЕНИЕ 9.13 Путь

Маршрут называется **путём** (path), если все его вершины различны (посещаем каждую не более одного раза).

ОПРЕДЕЛЕНИЕ 9.14 Цикл

Маршрут называется **циклом** (cycle), если он замкнут ($v_0 = v_k, k \geq 3$) и все промежуточные вершины различны.

Пример Маршруты, пути, циклы

В графе-треугольнике (вершины 1, 2, 3, полный набор рёбер):

- 1, 2, 1, 3 — маршрут длины 3, но не путь: вершина 1 повторяется.

- 1, 2, 3 — путь (все вершины различны).
- 1, 2, 3, 1 — цикл длины 3.
- 1, 2, 3, 2, 1 — замкнутый маршрут, но не цикл (вершина 2 повторяется в середине).

Разница между тремя понятиями — в дисциплине: маршрут позволяет повторять вершины, путь — нет, цикл — замкнутый путь. Дальше вопрос меняется: вместо «как пройти» спрашиваем, «можно ли пройти вообще». Следующая группа понятий измеряет целостность графа.

ОПРЕДЕЛЕНИЕ 9.15 Связность

Граф называется **связным**, если между любой парой его вершин существует путь.

ОПРЕДЕЛЕНИЕ 9.16 Компонента связности

Компонентой связности называется максимальный по включению связный подграф.

Связность и компоненты описывают граф целиком: сколько в нём кусков и насколько они крупны. У целостности есть обратная сторона — устойчивость к повреждениям. Граф может держаться на единственном ребре или на единственной вершине, и удаление этого элемента раскалывает его.

Пример Компоненты связности

Граф из двух треугольников $1 - 2 - 3 - 1$ и $4 - 5 - 6 - 4$ без общих вершин имеет две компоненты связности. Пути из вершины 1 в вершину 4 не существует, поэтому граф несвязен. Добавим ребро $\{3, 4\}$ — компоненты склеятся, и граф станет связным. Теперь $d(1, 4) = 2$: путь $1 - 3 - 4$ проходит через новое ребро. Одно ребро превратило два куска в один граф — именно такие переходы измеряет число компонент.

ОПРЕДЕЛЕНИЕ 9.17 Мост

Ребро называется **мостом**, если его удаление увеличивает число компонент связности.

ОПРЕДЕЛЕНИЕ 9.18 Точка сочленения

Вершина называется **точкой сочленения**, если её удаление увеличивает число компонент связности.

Связность отвечает на вопрос «соединены ли вершины», но не на вопрос «насколько далеко они друг от друга». Ответ даёт длина пути, и прежде всего — длина

кратчайшего пути. С этой мерой появляются два числа, описывающих граф целиком: диаметр и эксцентриситет.

ОПРЕДЕЛЕНИЕ 9.19 Расстояние

Расстоянием $d(u, v)$ называется длина кратчайшего пути между этими вершинами.

ОПРЕДЕЛЕНИЕ 9.20 Диаметр

Диаметром графа называется максимальное расстояние между его вершинами: $\max_{u,v} d(u, v)$.

ОПРЕДЕЛЕНИЕ 9.21 Эксцентриситет

Эксцентриситетом вершины называется максимальное расстояние от неё до любой другой вершины.

Пример Расстояния и диаметр

В графе-цепочке из пяти вершин ($1 - 2 - 3 - 4 - 5$): $d(1, 5) = 4$, $d(1, 3) = 2$ (кратчайший путь: $1, 2, 3, 4, 5$). Диаметр равен 4 — максимальное расстояние между вершинами. Эксцентриситет вершины 3 равен 2 (до 1 и 5), вершины 1 — 4 (до 5).

В графе-звезде (центр 1, листья $2, 3, 4, 5$): $d(2, 3) = 2$ (через центр), диаметр 2, эксцентриситет центра — 1, листьев — 2.

Пример Мост и точка сочленения

В графе-цепочке $1 - 2 - 3 - 4 - 5$ ребро $\{2, 3\}$ — мост: удалите его, и граф распадётся на две компоненты: $\{1, 2\}$, $\{3, 4, 5\}$. Вершина 3 в графе-восьмёрке (два треугольника, соединённые общей вершиной) — точка сочленения: удаление этой вершины разбивает граф на два отдельных треугольника.

Мост — ребро, удаление которого разрывает граф, точка сочленения — вершина с тем же свойством. Оба понятия иллюстрирует рис. 31.

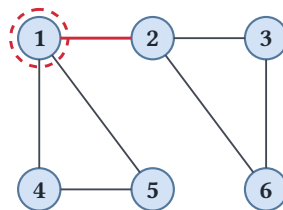


Рис. 31. Мост и точка сочленения.

Орграфы добавляют к этой картине сильную связность и её компоненты: формальные определения и алгоритм Косарайю вводятся ниже, в разделе о топологической сортировке и сильной связности.

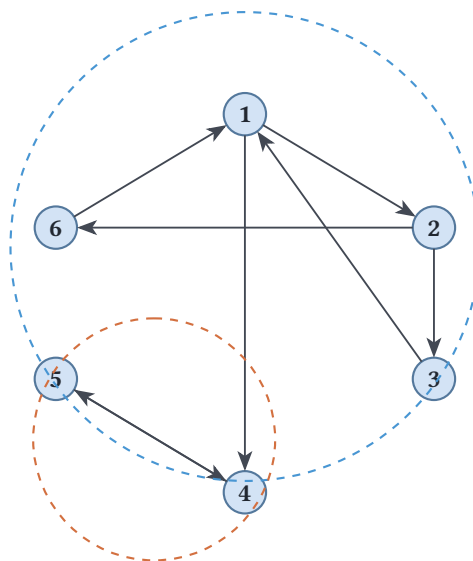


Рис. 32. Ориентированный граф.

Связность, компоненты, расстояния — всё это вычисляется одним инструментом: обходом графа. К обходам мы и переходим.

§ 9.4 Обходы графа

Графовые алгоритмы — рабочие инструменты computer science. Мы разберём их как конструкции: конкретный граф, пошаговая трассировка, объяснение корректности. Обходы — первый инструмент: BFS и DFS превращают граф в структуру данных, на которой работают остальные алгоритмы.

9.4.1 Поиск в ширину

BFS (Breadth-First Search) обходит граф «по слоям»: сначала вершины на расстоянии 1 от источника, затем на расстоянии 2, и так далее. Он находит кратчайшие пути в невзвешенном графе — графе без весов на рёбрах или, что эквивалентно, с единичными весами.

Алгоритм использует очередь: на каждом шаге извлекаем вершину из головы очереди и добавляем в хвост всех её непосещённых соседей, пометая их как посещённые. Порядок извлечения гарантирует обход по слоям: сначала все вершины слоя k , затем все вершины слоя $k + 1$.

АЛГОРИТМ Поиск в ширину

1. Инициализировать очередь Q с начальной вершиной s . Пометить стартовую вершину как посещённую.

2. Пока Q не пуста:

- Извлечь вершину u из головы очереди Q .
- Для каждого соседа v вершины u : если v не посещена, пометить её как посещённую, установить $\text{parent}(v) = u$, добавить v в хвост Q .

Родительские ссылки, записанные во время обхода, образуют *BFS-дерево* — дерево кратчайших путей от s до всех достижимых вершин.

Пример Трассировка BFS

Рассмотрим граф на рис. 33:

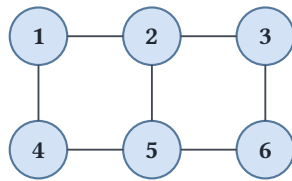


Рис. 33. Граф для трассировки BFS.

Старт из вершины 1. Очередь: [1].

- Извлекаем 1.

Соседи: {2, 4}. Оба не посещены. $Q = [2, 4]$, $\text{parent}(2) = 1$, $\text{parent}(4) = 1$.

- Извлекаем 2. Соседи: {1, 3, 5}. 1 посещена, 3 и 5 — нет. $Q = [4, 3, 5]$, $\text{parent}(3) = 2$, $\text{parent}(5) = 2$.
- Извлекаем 4. Соседи: {1, 5}. 1 посещена, 5 уже в очереди — пропускаем.
- Извлекаем 3. Соседи: {2, 6}. 2 посещена, 6 — нет. $Q = [5, 6]$, $\text{parent}(6) = 3$.
- Извлекаем 5. Соседи: {2, 4, 6}. 2 и 4 посещены, 6 уже в очереди.
- Извлекаем 6. Все соседи посещены. Q пуста — конец.

Кратчайшие расстояния:

- $d(1, 1) = 0$
- $d(1, 2) = d(1, 4) = 1$
- $d(1, 3) = d(1, 5) = 2$
- $d(1, 6) = 3$

Родительские ссылки дают кратчайшие пути: $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$.

Пример Реализация BFS на Rust

Тот же обход в коде: очередь явная, расстояния — числа, а непосещённые вершины помечены `usize::MAX`.

```
use std::collections::VecDeque;

/// Обход в ширину: порядок обхода, расстояния
/// и родители в дереве обхода.
```

```

fn bfs(adj: &[Vec<usize>], start: usize) -> (Vec<usize>, Vec<usize>,
Vec<Option<usize>> > {
    let n = adj.len();
    let mut dist = vec![usize::MAX; n];
    let mut parent = vec![None; n];
    let mut order = vec![];
    let mut q = VecDeque::from([start]);
    dist[start] = 0;

    while let Some(u) = q.pop_front() {
        order.push(u);

        for &v in &adj[u] {
            if dist[v] == usize::MAX {
                dist[v] = dist[u] + 1;
                parent[v] = Some(u);
                q.push_back(v);
            }
        }
    }

    (order, dist, parent)
}

```

Этот же граф в виде запускаемого примера — крейт code/graphs: пример bfs_dfs печатает порядок обхода, расстояния и родительские ссылки. Там же собраны остальные алгоритмы главы: кратчайшие пути, остовные деревья, эйлеровы циклы, раскраска.

BFS работает за $O(V + E)$: каждая вершина попадает в очередь ровно один раз, каждое ребро просматривается не более двух раз (по разу с каждой стороны). BFS-дерево на рис. 34 строится послойно: от стартовой вершины к соседям первого уровня, затем второго, и так далее.

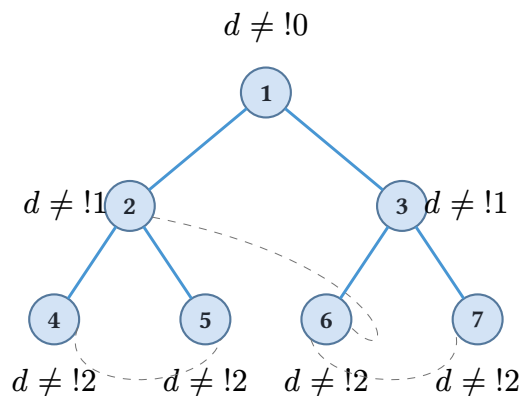


Рис. 34. BFS-дерево.

9.4.2 Поиск в глубину

DFS (Depth-First Search) идёт «вглубь»: рекурсивно исследует первого непосещённого соседа, затем его непосещённого соседа, и так далее, пока не упрётся в вершину без непосещённых соседей — и тогда возвращается (backtrack) к предыдущей развилке.

В отличие от BFS, который явно хранит очередь, DFS естественно выражается рекурсией. Стек вызовов играет роль структуры данных: текущий путь от стартовой вершины хранится в стеке рекурсии.

АЛГОРИТМ Поиск в глубину

1. Для каждой непосещённой вершины u : вызвать $\text{DFS}(u)$.
2. $\text{DFS}(u)$:
 - Пометить u как посещённую. Записать $\text{pre}(u)$ (время входа).
 - Для каждого соседа v вершины u : если v не посещена, установить $\text{parent}(v) = u$ и вызвать $\text{DFS}(v)$.
 - Записать $\text{post}(u)$ (время выхода).

Времена pre , post — это номера шагов, на которых вершина была впервые посещена и полностью обработана. Они дают структурную информацию о графе. Рёбра относительно дерева DFS делятся на четыре типа:

- рёбра дерева — ведут к непосещённому соседу и входят в DFS-дерево,
- обратные — ведут к предку,
- прямые — ведут к потомку по дереву, но не являются рёбрами дерева,
- перекрёстные — идут между разными ветвями.
- Для дерева/прямого ребра: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$.
- Для обратного ребра: $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$ (ребро ведёт к предку).
- Для перекрёстного ребра: $\text{post}(v) < \text{pre}(u)$ (ребро ведёт в уже полностью обработанную ветвь).

Четыре типа рёбер — классификация для ориентированных графов. В неориентированном графе прямых и перекрёстных рёбер не бывает: любое ребро между уже посещёнными вершинами ведёт к предку.

Если перебирать соседей по возрастанию номеров, DFS на графе из примера BFS, показанном на рис. 33, стартуя из вершины 1, даёт порядок 1, 2, 3, 6, 5, 4, а обратные рёбра — $\{1, 4\}, \{2, 5\}$. При другом порядке перебора и порядок обхода, и обратные рёбра изменятся.

DFS лежит в основе алгоритмов:

- Топологической сортировки (обратный postorder — это топологический порядок вершин ориентированного графа без циклов).

- Поиска компонент сильной связности (двухпроходный алгоритм Косарайю).
- Обнаружения циклов (обратное ребро в DFS-дереве означает цикл).
- Поиска мостов (рёбра, удаление которых нарушает связность).

Сложность DFS — $O(V + E)$: каждая вершина посещается один раз, каждое ребро просматривается дважды (по разу с каждого конца).

Примечание Как найти все мосты за один обход

За обещанием поиска мостов стоит тот же DFS с его временами pre . Вдоль обхода для каждой вершины v считается *low-link*: минимальное время pre , достижимое из поддерева v по пути не более чем с одним обратным ребром. Деревянное ребро $u \rightarrow v$ — мост тогда и только тогда, когда $\text{low}(v) > \text{pre}(u)$: поддерево v не имеет обходного пути выше ребра, и его удаление отрезает поддерево. Алгоритм Тарьяна (1972) находит все мосты и точки сочленения за один DFS за $O(V + E)$ ⁵⁶. Полный разбор оставим учебникам по графовым алгоритмам, а сам инвариант low ещё вернётся: он работает в однопроходном алгоритме сильной связности, к которому мы придём ниже.

Проверка Обходы графа

1. Какая структура данных задаёт порядок обхода в BFS и в DFS и какой порядок обхода каждая из них даёт?
2. Как по временам pre и post узнать обратное ребро и почему обратное ребро означает цикл?

§ 9.5 Топологическая сортировка и сильная связность

Сборка большого проекта — каскад зависимостей: сначала компилируется низкоуровневый модуль, затем тот, что на него опирается, и так до конечного приложения. Если отношения зависимостей образуют цикл, собрать проект невозможно: каждый модуль ждёт другой. Так возникает первый класс задач этого раздела — упорядочить вершины ориентированного графа так, чтобы все стрелки вели вперёд, когда такой порядок существует.

Второй класс — декомпозиция. Любой орграф, даже с циклами, распадается на связные блоки, внутри которых можно добраться из любой вершины в любую. Сжав эти блоки, мы получаем ациклическую картину отношений между ними: то, что было циклическим внутри, становится точкой, а связи между точками — снова ориентированное дерево зависимостей. Обе задачи решает один обход — тот самый DFS с временами pre , post , который мы уже построили.

⁵⁶R. E. Tarjan. «Depth-First Search and Linear Graph Algorithms». SIAM Journal on Computing, 1972.

9.5.1 Топологическая сортировка

Ориентированный граф без циклов называется **DAG** (Directed Acyclic Graph). Примером служит граф зависимостей сборки: стрелка из a в b означает «модуль a должен быть собран раньше модуля b ».

ОПРЕДЕЛЕНИЕ 9.22 Топологический порядок

Топологический порядок DAG — нумерация вершин $1, \dots, n$ такая, что для каждого ребра $u \rightarrow v$ номер u меньше номера v . В линейной записи это упорядоченный список, в котором каждая стрелка направлена слева направо.

Топологический порядок существует не для всякого орграфа: цикл $a \rightarrow b \rightarrow c \rightarrow a$ не позволяет расположить a, b, c так, чтобы каждая стрелка шла вперёд — они заиклены. Именно поэтому топологически упорядочить можно только DAG, и именно поэтому цикл в зависимостях — фатальная ошибка проекта.

АЛГОРИТМ Топологическая сортировка (алгоритм Кана)

1. Вычислить входящую степень каждой вершины.
2. Поместить в очередь все вершины с нулевой входящей степенью — истоки.
3. Пока очередь не пуста:
 - Извлечь исток u и вывести его.
 - Для каждого соседа v вершины u : уменьшить входящую степень v на единицу. Если входящая степень v стала нулём, добавить v в очередь.
4. Если выведено не n вершин, в графе остался цикл.

Вершина с нулевой входящей степенью — исток: в неё не входит ни одна стрелка, значит, ничто её не ждёт и она может быть собрана первой. Удалив исток, мы уменьшаем входящие степени его соседей — и среди них может появиться новый исток. Так очередь истоков «проталкивает» порядок слева направо.

Пример Топологическая сортировка графа сборки

Модули: $\text{core} \rightarrow \text{utils}$, $\text{utils} \rightarrow \text{app}$, $\text{utils} \rightarrow \text{tests}$. Входящие степени: $\text{core} = 0$, $\text{utils} = 1$, $\text{app} = 1$, $\text{tests} = 1$.

Исток ровно один — core .

- Выводим core . Удаляем стрелки $\text{core} \rightarrow \text{utils}$: входящая степень utils падает до 0.
- Выводим utils . Удаляем стрелки $\text{utils} \rightarrow \text{app}$ и $\text{utils} \rightarrow \text{tests}$: входящие степени app и tests падают до 0.
- Выводим app и tests в любом порядке.

Порядок $\text{core}, \text{utils}, \text{app}, \text{tests}$ — топологический: каждая зависимость собрана раньше своего потребителя. Порядок $\text{core}, \text{utils}, \text{tests}, \text{app}$ тоже подходит. Оба

вывода корректны, и в этом свобода задачи: любые вершины, между которыми нет пути по стрелкам, можно переставить.

Топологический порядок можно получить и одним проходом DFS без явного подсчёта степеней. Всякая вершина DAG выходит из обхода после всех, кого она из себя достижима, поэтому обратный порядок `post` — топологический.

Доказательство Корректность обратного `postorder`

Докажем, что для каждого ребра $u \rightarrow v$ вершина v выходит из обхода позже u . Тогда порядок по убыванию `post` — топологический.

Пусть $u \rightarrow v$ — ребро DAG, и обход DFS достиг u . Поскольку граф ациклический, v не может быть уже завершённой вершиной-предком u : обратное ребро к предку образовало бы цикл. Значит, когда u завершает обход, v либо ещё не посещена, либо находится в стеке и будет обработана позже. В обоих случаях $\text{post}(v) > \text{post}(u)$. Упорядочив вершины по убыванию `post`, для каждого ребра $u \rightarrow v$ получим u раньше v — топологический порядок.

Сложность алгоритма Кана — $O(V + E)$: каждая вершина один раз попадает в очередь и один раз обрабатывается, каждое ребро просматривается один раз. Обратный `postorder` тоже $O(V + E)$.

9.5.2 Сильная связность

DAG — редкий случай: циклов нет. В общем орграфе циклы есть, и именно они мешают говорить о «порядке» между любыми двумя вершинами. Но циклы бывают «локальными»: распад на группы, внутри которых всё взаимодостижимо, даёт ту же ациклическую картину на уровне групп.

ОПРЕДЕЛЕНИЕ 9.23 Сильная связность

Орграф называется **сильно связным**, если из каждой вершины достижима каждая другая (ориентированными путями в обе стороны).

ОПРЕДЕЛЕНИЕ 9.24 Компонента сильной связности

Компонента сильной связности (SCC) — максимальное подмножество вершин, которое сильно связно. Максимальность означает: добавление любой другой вершины разрушает сильную связность.

На рис. 32 вершины 1, 2, 3, 6 образуют одну компоненту: из 1 можно добраться до 6, из 6 — до 1 (через 2), и внутри тройки каждый из каждой. Вершина 4 с вершиной 5 — вторая компонента: между ними пути в обе стороны по циклу $4 \rightarrow 5 \rightarrow 4$. А из 1, 2, 3, 6 в 4 или 5 пути нет ни в одну сторону, поэтому эти две компоненты не сливаются в одну.

АЛГОРИТМ Компоненты сильной связности (алгоритм Косарайю)

1. Выполнить DFS на исходном графе, записывая порядок выхода post каждой вершины.
2. Построить обращённый граф G^R (развернуть каждую дугу).
3. Выполнить DFS на G^R в порядке убывания post исходного обхода.
4. Каждое дерево этого второго обхода — отдельная компонента сильной связности.

Почему нужно обращать граф и идти в обратном порядке выхода? Направление стрелок меняется на противоположное: если из u достигим v , то в обращённом графе из v достигим u . Порядок убывания post выбирает «корни» в обратном топологическом порядке конденсации: когда мы начинаем новый раунд обхода в G^R , мы замыкаем ровно одну новую компоненту.

Пример Косарайю на рис. 32

Первый проход (порядок соседей по возрастанию номеров) даёт, например, порядок выхода 2, 5, 4, 1, 6, 3 (зависит от порядка перебора). Обратный порядок: 3, 6, 1, 4, 5, 2.

Второй проход на G^R в этом порядке:

- Из 3: достижимы 3, 6, 1, 2 (в G^R дуги развёрнуты) — первая компонента $\{1, 2, 3, 6\}$.
- Из 4: достижимы 4, 5 в G^R — вторая компонента $\{4, 5\}$.
- Вершина 2 уже посещена.

Получили компоненты $\{1, 2, 3, 6\}$ и $\{4, 5\}$ — как в подписи к рис. 32.

Доказательство Корректность Косарайю

Докажем, что дерево второго обхода из u — в точности компонента C : покажем, что оно покрывает C и не выходит за C .

Рассмотрим вершину u , которая первой в порядке убывания post попадает во второй обход, и пусть C — её компонента. Покажем сначала, что обход покрывает весь C . Всякая вершина $v \in C$ достижима из u в G , значит, в G^R из v достигим u . Поэтому, когда второй обход стартует из u , он обойдёт весь C : каждую вершину компоненты можно достичь, идя по развёрнутым дугам обратными путями исходного графа.

Покажем теперь, что вне C обход не выйдет. Если бы из u в G^R была достижима вершина $w \notin C$, то в G из w была бы достижима u , а раз $u \rightarrow w$ в G^R — из u достижима w в G , и w была бы в C (сильная связность с u). Значит, дерево второго обхода — ровно C .

Сложность — $O(V + E)$: два прохода DFS по V вершинам и E дугам. Альтернатива — однопроходный алгоритм Тарьяна, который держит стек и находит компоненты во время одного DFS за ту же асимптотику.

Разложив оргграф на компоненты и сжав каждую в одну вершину, получаем **конденсацию**: оргграф, вершины которого — компоненты, а дуги наследуются из исходного. Конденсация всегда является DAG — если бы в ней был цикл, все компоненты цикла слились бы в одну. Значит, на конденсации определена топологическая сортировка, и это возвращает нас к первому классу задач: циклы внутри компонент не мешают выстроить ациклический порядок между ними.

§ 9.6 Деревья

Дерево — простейший нетривиальный класс графов: у графа без рёбер нет структуры, у дерева — минимальная структура, поддерживающая связность. Для этого класса известно пять эквивалентных определений. Каждое высвечивает свой аспект: комбинаторный (число рёбер), топологический (связность без циклов), алгоритмический (единственность пути). Множественность эквивалентных определений — признак хорошо определённой структуры: каждое удобно в своей ситуации. В комбинаторике, теории алгоритмов и базах данных деревья возникают повсеместно потому, что моделируют иерархию.

ОПРЕДЕЛЕНИЕ 9.25 Дерево

Граф называется **деревом**, если он связный и ациклический.

ОПРЕДЕЛЕНИЕ 9.26 Лес

Граф называется **лесом**, если он является дизъюнктивным объединением деревьев.

Пример Деревья и не-деревья

- Граф-цепочка из n вершин $(1 - 2 - \dots - n)$ — дерево: связен, имеет $n - 1$ рёбер, циклов нет.
 - Граф-цикл C_n не является деревом: содержит цикл.
 - Звезда (центр и k листьев) — дерево.
 - Лес из двух цепочек $1 - 2 - 3, 4 - 5$ имеет $3 + 2 = 5$ вершин и $2 + 1 = 3$ ребра, то есть $5 - 3 = 2$ компоненты связности.

Дерево на рис. 35 обладает минимальным числом рёбер для связности: добавление любого ребра создаёт цикл, удаление — разрывает граф.

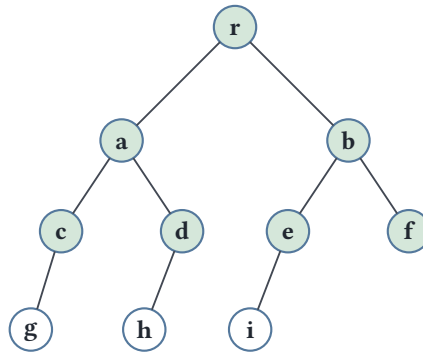


Рис. 35. Дерево.

ТЕОРЕМА 9.4 Эквивалентные характеристики дерева

Для графа с n вершинами следующие утверждения эквивалентны:

1. Граф связный и ациклический (является деревом).
2. Граф связный и имеет $n - 1$ рёбер.
3. Граф ациклический и имеет $n - 1$ рёбер.
4. Между любой парой вершин существует единственный простой путь.
5. Граф связный, и каждое ребро является мостом.

Набросок доказательства

Докажем эквивалентность пяти утверждений, показав цепочку $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (4) \Rightarrow (5) \Rightarrow (1)$.

$(1) \Rightarrow (2)$: граф связный и ациклический $\Rightarrow$ граф связный и имеет $n - 1$ рёбер. Проведём индукцию по n . Дерево имеет лист (вершину степени 1): конец максимального простого пути имеет степень 1, иначе путь удлинился бы или появился бы цикл. Удалив лист и инцидентное ему ребро, получим дерево на $n - 1$ вершинах, которое по индукции имеет $(n - 1) - 1 = n - 2$ рёбер. Добавляем удалённое ребро — получаем $n - 1$.

$(2) \Rightarrow (3)$: граф связный и имеет $n - 1$ рёбер $\Rightarrow$ граф ациклический и имеет $n - 1$ рёбер.

Если бы существовал цикл, удаление ребра из цикла оставило бы граф связным с $n - 2$ рёбрами. Но связный граф на n вершинах имеет не менее $n - 1$ рёбер: он содержит остовное дерево (определение — ниже в этой главе), а у дерева на n вершинах ровно $n - 1$ рёбер, — противоречие. Значит, циклов нет.

$(3) \Rightarrow (4)$: граф ациклический и имеет $n - 1$ рёбер $\Rightarrow$ между любой парой вершин существует единственный простой путь.

Если граф несвязен и состоит из c компонент, каждая компонента — дерево, и общее число рёбер $n - c$. При $|E| = n - 1$ получаем $c = 1$, то есть граф связен. Если бы между двумя вершинами существовали два разных простых пути, их симметрическая разность содержала бы цикл — противоречие с ациклическостью.

(4) $\Rightarrow$ (5): между любой парой вершин существует единственный простой путь $\Rightarrow$ граф связный, и каждое ребро является мостом. Удаление любого ребра $\{u, v\}$ уничтожает единственный путь между этими вершинами — граф теряет связность. Следовательно, каждое ребро — мост.

(5) $\Rightarrow$ (1): граф связный, и каждое ребро является мостом $\Rightarrow$ граф связный и ациклический (дерево). Если бы существовал цикл, никакое ребро цикла не было бы мостом (удаление ребра цикла оставляет альтернативный путь). Значит, циклов нет — граф ациклический. Вместе со связностью это даёт дерево.

Структура ветвления повторяется на всех масштабах. Дельта реки, бронхи лёгкого, нейрон с дендритами, граф вызовов рекурсивной функции — всё это деревья. Пять эквивалентных определений не случайны: они отражают разные способы увидеть одну структуру. «Связный и ациклический» — топология, « n вершин, $n - 1$ рёбер» — комбинаторика, «единственный путь между любой парой» — алгоритмический взгляд, «каждое ребро — мост» — структурная уязвимость. Выбор определения зависит от задачи.

СЛЕДСТВИЕ 9.5 Листья

Любое дерево с $n \geq 2$ имеет хотя бы два листа (вершины степени 1).

Доказательство

Докажем от противного и разбором случаев.

Пусть T — дерево с $n \geq 2$ вершинами. По характеристике дерева и лемме о рукопожатиях сумма степеней равна $2|E| = 2(n - 1) = 2n - 2$.

Предположим, что листьев меньше двух. Тогда либо их нет вовсе, либо ровно один. **Случай 1:** листьев нет. Тогда каждая вершина имеет степень ≥ 2 , и сумма степеней $\geq 2n > 2n - 2$ — противоречие. **Случай 2:** лист ровно один. Тогда остальные $n - 1$ вершин имеют степень ≥ 2 , и сумма степеней $\geq 1 + 2(n - 1) = 2n - 1 > 2n - 2$ — противоречие. Следовательно, дерево имеет хотя бы два листа.

Листья — крайние точки дерева. Их существование используется в алгоритмах обхода и доказательствах по индукции.

Дерево может быть подграфом более крупного графа — тогда оно служит скелетом, сохраняющим связность.

ОПРЕДЕЛЕНИЕ 9.27 Остовное дерево (*spanning tree*)

Остовным деревом графа называется его подграф, включающий все вершины и являющийся деревом.

Любой связный граф имеет остовное дерево: достаточно обойти граф (например, поиском в ширину) и оставить только те рёбра, которые ведут к ещё не посещённым вершинам — они не создадут циклов и покроют все вершины.

Пример Остовные деревья

K_3 (треугольник) — полный граф на трёх вершинах. Удаление любого из трёх рёбер даёт остовное дерево (цепочку из трёх вершин).

C_4 (цикл из четырёх вершин): удаление любого из четырёх рёбер даёт остовное дерево. Рисунок 36 показывает остовное дерево для графа из пяти вершин.

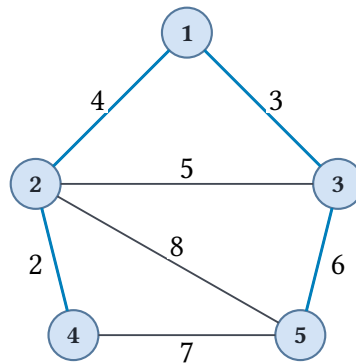


Рис. 36. Остовное дерево.

Остовное дерево сохраняет связность графа минимальным числом рёбер. Сколько вообще существует деревьев на заданном наборе помеченных вершин?

Примечание Помеченные вершины

Формула Кэли считает деревья с помеченными вершинами: метки — имена вершин из фиксированного множества $\{1, 2, \dots, n\}$. Любое конечное дерево можно пронумеровать: присвоить его вершинам метки $1, 2, \dots, n$ — и это всегда возможно. Разные нумерации одного и того же дерева дают разные помеченные деревья, поэтому речь идёт о подсчёте на фиксированном множестве меток.

ТЕОРЕМА 9.6 Формула Кэли⁵⁷

Количество помеченных деревьев на n вершинах равно

$$n^{n-2}.$$

Набросок доказательства

Докажем построением биекции между деревьями и кодами Прюфера.

⁵⁷ A. Cayley. «A theorem on trees». Quarterly Journal of Pure and Applied Mathematics, 1889.

Построим биекцию между помеченными деревьями на n вершинах и последовательностями длины $n - 2$ из чисел $\{1, 2, \dots, n\}$ (кодами Прюфера). Таких последовательностей ровно n^{n-2} .

Кодирование (из дерева в код): на каждом шаге находим лист с наименьшей меткой, записываем метку его соседа, удаляем лист. Повторяем $n - 2$ раза.

Декодирование (из кода в дерево): восстанавливаем последовательность удалённых листьев, затем строим дерево, соединяя удалённые листья с соответствующими элементами кода.

Биективность: каждой последовательности соответствует ровно одно дерево и наоборот. Следовательно, число помеченных деревьев равно числу последовательностей.

УТВЕРЖДЕНИЕ 9.7 У листа ровно один сосед

Лист — вершина степени 1 — соединён ровно с одной другой вершиной.

Доказательство

Докажем прямо из определения.

По определению степень вершины — число инцидентных ей рёбер. У листа степень 1, значит, инцидентно ровно одно ребро, а оно соединяет лист ровно с одной вершиной — его единственным соседом.

На этой единственности стоит кодирование Прюфера: удаляя лист, мы записываем номер его единственного соседа, и выбор однозначен.

Пример Кодирование кода Прюфера

Снимем код с дерева на рис. 37 с рёбрами $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{3, 5\}$. На каждом шаге удаляем лист с наименьшей меткой и записываем метку его соседа.

1. Листья — 1, 4, 5. Удаляем наименьший лист — вершину 1 — и записываем его соседа 2: код [2].
2. Теперь листья — 2, 4, 5. Удаляем 2, записываем 3: код [2, 3].
3. Листья — 4, 5. Удаляем 4, записываем 3: код [2, 3, 3].
4. Остаются две вершины 3, 5 — их в код не записываем.

Код дерева — [2, 3, 3].

Число 3 в коде повторяется — ничто не запрещает разным удалённым листьям иметь общего соседа. Метка 5 в код не попадает вовсе: это один из двух последних листьев, которые не кодируются. Цифры кода — номера родителей удаляемых листьев в порядке удаления, а не упорядоченный список.

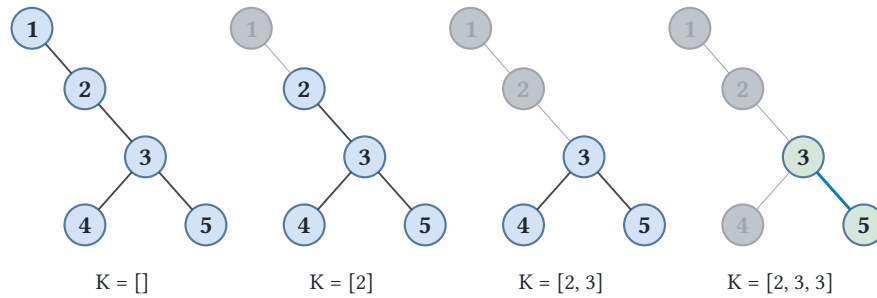


Рис. 37. Кодирование кода Прюфера.

Пример Декодирование кода Прюфера

Декодируем код $[3, 3, 1]$ при $n = 5$, вершины $\{1, 2, 3, 4, 5\}$, как на рис. 38. Степень вершины в искомом дереве равна числу её вхождений в код плюс один: метка 1 входит один раз, 3 — дважды, а 2, 4 и 5 — ни разу. Значит, начальные листья — 2, 4, 5.

1. Берём лист с наименьшей меткой — 2 — и соединяем его с первой цифрой кода: ребро $\{2, 3\}$. Убираем лист и первую цифру. Код становится $[3, 1]$.
2. Следующий лист — 4: ребро $\{4, 3\}$, код $[1]$.
3. Лист — 3: ребро $\{3, 1\}$, код пуст.
4. Остаются две вершины, не исчерпавшие свою степень, — 1, 5: соединяем их ребром $\{1, 5\}$.

Итоговое дерево — рёбра $\{2, 3\}$, $\{4, 3\}$, $\{3, 1\}$, $\{1, 5\}$. Пять вершин, четыре ребра, связно и без циклов — действительно дерево.

В коде $[3, 3, 1]$ метки 2, 4, 5 не встречаются вовсе — это изначальные листья, а метка 3 повторяется дважды: у листьев 2 и 4 один общий сосед.

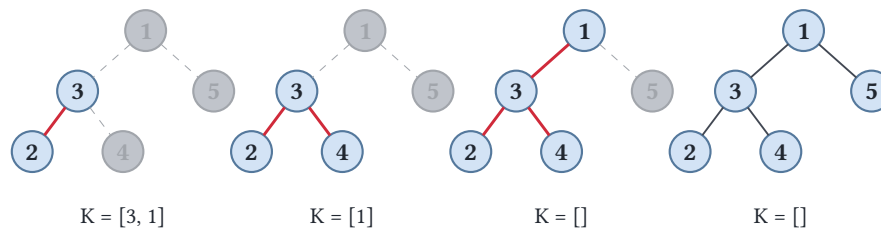


Рис. 38. Декодирование кода Прюфера.

Формула Кэли отвечает на вопрос «сколько», но не на вопрос «какое». Если рёбрам приписаны веса, разные остовные деревья имеют разный суммарный вес — возникает оптимизационная задача: найти остов минимального суммарного веса.

§ 9.7 Минимальные остовные деревья

Сеть дорог строится так, чтобы связать все города минимальной суммарной стоимостью. Каждый город — вершина, каждая возможная дорога — ребро с весом-ценой. Строительство всех дорог по всем рёбрам избыточно: достаточно выбрать подмно-

жество рёбер, которое связывает все вершины и имеет наименьший суммарный вес. Такое подмножество всегда является деревом — если бы в нём был цикл, лишнее ребро можно было бы удалить, не потеряв связность, и уменьшив вес. Так возникает задача о минимальном остовном дереве (MST).

ОПРЕДЕЛЕНИЕ 9.28 **Задача о минимальном остовном дереве**

Во взвешенном связном графе найти остовное дерево T минимального суммарного веса:

$$\sum_{e \in T} w(e) \rightarrow \min.$$

Остовное дерево включает все вершины и является деревом. Среди всех таких деревьев выбирается самое дешёвое.

Важно не путать MST с кратчайшими путями. Остовное дерево минимизирует **сумму** весов его рёбер, а не расстояния от источника до вершин. Дёшево связать все города — это совсем не то же самое, что построить из одного города короткие дороги до каждого.

Замечание **Дерево, а не лес**

Почему оптимальное решение — дерево, а не произвольный подграф? Любой связный подграф можно упростить до дерева, удаляя рёбра из циклов: вес при этом только уменьшается. Значит, искомый объект — именно дерево: ровно $V - 1$ рёбер и ни одного цикла.

Рассмотрим граф на рис. 36. Вершины 1..5, рёбра с весами: $w(1, 2) = 4$, $w(1, 3) = 3$, $w(2, 3) = 5$, $w(2, 4) = 2$, $w(2, 5) = 8$, $w(3, 5) = 6$, $w(4, 5) = 7$. Минимальное остовное дерево: 1 — 3 (3), 1 — 2 (4), 2 — 4 (2), 3 — 5 (6). Суммарный вес: $3 + 4 + 2 + 6 = 15$. Любой другой набор из четырёх рёбер, связывающий все вершины, дороже.

Обе задачи — о минимальном остовном дереве — решаются жадным методом, но жадность применяется по-разному. Крускал перебирает рёбра глобально, Прим строит дерево локальным ростом. Разберём оба подхода и убедимся, что каждый из них возвращает дерево минимального веса.

9.7.1 Алгоритм Крускала

Крускал действует на рёбрах, а не на вершинах: он упорядочивает все рёбра по весу и жадно добавляет легчайшее, которое не образует цикл. Чтобы быстро проверять, не образует ли ребро цикл, Крускал поддерживает компоненты связности текущего леса системой непересекающихся множеств (union-find).

АЛГОРИТМ **Минимальное остовное дерево (алгоритм Крускала)**

1. Отсортировать все рёбра по возрастанию веса.

2. Создать V компонент связности, по одной на вершину.
3. Для каждого ребра (u, v) в порядке возрастания веса:
 - Если $\text{find}(u) \neq \text{find}(v)$ — вершины в разных компонентах, значит, ребро не создаёт цикл. Добавить (u, v) в остов и выполнить $\text{union}(u, v)$.
 - Иначе ребро соединяет вершины одной компоненты и пропускается.
4. Остановиться, когда добавлено $V - 1$ рёбер.

Система непересекающихся множеств поддерживает компоненты: $\text{find}(v)$ возвращает представителя компоненты вершины v , $\text{union}(u, v)$ сливает компоненты u и v . Рёбра внутри одной компоненты отбрасываются как ведущие к циклу.

Пример Крускал на графе рис. 36

Упорядочим рёбра по весу: $2 - 4$ (2), $1 - 3$ (3), $1 - 2$ (4), $2 - 3$ (5), $3 - 5$ (6), $4 - 5$ (7), $2 - 5$ (8).

Проходим в этом порядке:

- $2 - 4$ (2): вершины в разных компонентах — берём.
- $1 - 3$ (3): разных — берём.
- $1 - 2$ (4): разных — берём.
- $2 - 3$ (5): вершины уже в одной компоненте (через 1) — пропускаем.
- $3 - 5$ (6): разных — берём.

Добавлено $4 = V - 1$ рёбер — останавливаемся.

Остов: $2 - 4$ (2), $1 - 3$ (3), $1 - 2$ (4), $3 - 5$ (6). Суммарный вес $2 + 3 + 4 + 6 = 15$ — искомый минимум.

Почему Крускал корректен? Жадность здесь не очевидна: мы на каждом шаге выбираем локально легчайшее ребро, а нужно глобально минимальное дерево. Доказательство — обменный аргумент.

Доказательство

Докажем обменным аргументом: покажем, что любое ребро, выбранное Крускалом, входит в некоторое минимальное остовное дерево.

Рассмотрим момент, когда Крускал выбирает ребро $e = (u, v)$ минимального веса среди всех, не создающих цикл. Пусть T — оптимальное остовное дерево. Если $e \in T$, мы ничего не меняем. Иначе e соединяет две компоненты текущего леса. В дереве T эти две компоненты соединены какой-то цепочкой, и в ней есть ребро f , концы которого лежат в разных компонентах.

Ребро f не может быть легче e : если бы было, Крускал рассмотрел бы f раньше e . В тот момент концы f лежали бы в разных компонентах (компоненты леса со временем только сливаются) — Крускал добавил бы f , и e не было бы первым соединяющим ребром этих компонент. Значит, $w(f) \geq w(e)$.

Заменим в T ребро f на e : связность сохранится (компоненты, которые связывал f , связаны и через e), а вес не вырастет ($w(e) \leq w(f)$). Получим оптимальное дерево, содержащее e . Повторяя обмен, придём к дереву, построенному Крускалом, — оно оптимально.

Сложность Крускала — $O(E \log E)$: сортировка рёбер доминирует, а каждая операция union-find почти константна (обратная функция Аккермана). Для разреженных графов ($E = O(V)$) это близко к $O(V \log V)$.

9.7.2 Алгоритм Прима

Прим противоположен Крускалу по духу: он растит дерево из одной стартовой вершины. На каждом шаге к уже построенному дереву добавляется легчайшее ребро, соединяющее его с внешней вершиной. По структуре это очень похоже на алгоритм Дейкстры: вместо расстояний до источника поддерживаются минимальные веса рёбер от каждой вершины до строящегося дерева.

АЛГОРИТМ Минимальное остовное дерево (алгоритм Прима)

1. Выбрать стартовую вершину s . Дерево состоит только из s .
2. $\text{key}(v)$ — минимальный вес ребра от вершины v к дереву. Инициализировать $\text{key}(v) = \infty$ для всех v , кроме s : $\text{key}(s) = 0$.
3. Пока дерево не покрывает все вершины:
 - Извлечь вершину u с минимальным key (min-heap).
 - Для каждого соседа v вершины u с ребром веса $w(u, v)$: если $w(u, v) < \text{key}(v)$, то $\text{key}(v) = w(u, v)$ и $\text{parent}(v) = u$.
4. Вернуть рёбра ($\text{parent}(v), v$) для всех $v \neq s$.

Каждая вершина по одному разу извлекается из кучи и добавляется в дерево. Рёбра дерева — те, по которым она вошла.

Пример Прим на графе рис. 36

Старт из вершины 1.

- Дерево: $\{1\}$. Рёбра к соседям: $1 - 3$ (3), $1 - 2$ (4). Минимум — $1 - 3$ (3): добавляем вершину 3.
- Дерево: $\{1, 3\}$. Новые кандидаты от 3: $3 - 5$ (6), $3 - 2$ (5). Текущие ключи: $\text{key}(2) = 4$ ($1 - 2$), $\text{key}(5) = 6$ ($3 - 5$). Минимум — $\text{key}(2) = 4$: добавляем вершину 2 по ребру $1 - 2$.
- Дерево: $\{1, 3, 2\}$. Новые кандидаты от 2: $2 - 5$ (8), $2 - 4$ (2). $\text{key}(5) = \min(6, 8) = 6$ через 3. $\text{key}(4) = 2$ через 2. Минимум — $\text{key}(4) = 2$ ($2 - 4$): добавляем вершину 4.
- Дерево: $\{1, 3, 2, 4\}$. Новые кандидаты от 4: $4 - 5$ (7). $\text{key}(5) = \min(6, 7) = 6$ через 3. Минимум — $\text{key}(5) = 6$ ($3 - 5$): добавляем вершину 5.

Остов: $1 - 3$ (3), $1 - 2$ (4), $2 - 4$ (2), $3 - 5$ (6). Суммарный вес $3 + 4 + 2 + 6 = 15$ — тот же, что у Крускала, и это тот же минимум.

Корректность Прима — тот же обменный аргумент, что и у Крускала: на каждом шаге добавляемое ребро — легчайшее на разрезе (между деревом и внешними вершинами), а всякое минимальное дерево пересекает этот разрез некоторым ребром не меньшего веса. Оба алгоритма жадно поддерживают один инвариант: построенные рёбра входят в некоторое минимальное остовное дерево.

ПРЕДУПРЕЖДЕНИЕ Оба алгоритма — жадные, но по-разному

Крускал жадный по **рёбрам** (глобальный порядок), Прим — по **разрезам** (локальный рост). В отличие от жадных алгоритмов вроде выбора монет, здесь жадность работает потому, что MST удовлетворяет свойству **оптимальной подструктуры**: минимальное дерево для графа содержит минимальные деревья для его частей. Это не общее свойство жадных решений — например, жадный выбор монет не всегда оптимален.

Сравнение сложностей: Крускал — $O(E \log E)$, лучший для разреженных графов ($E = O(V)$). Прим с бинарной кучей — $O(E \log V)$, а вариант без кучи — $O(V^2)$, лучший для плотных графов ($E = \Theta(V^2)$): каждая итерация находит ближайшую внешнюю вершину линейным просмотром за $O(V)$, а перебор всех рёбер сортировкой избыточен.

Проверка Минимальные остовные деревья

1. Чем минимальное остовное дерево отличается от дерева кратчайших путей из одного источника?
2. Какой аргумент гарантирует корректность жадного выбора Крускала и почему тот же аргумент работает у Прима?
3. Когда выгоднее Крускал, а когда — Прим?

§ 9.8 Эйлеровы графы

История теории графов началась с вопроса о том, можно ли обойти кёнигсбергские мосты, пройдя по каждому ровно один раз. Задача о мостах родилась из воскресной прогулки: жители Кёнигсберга спорили, можно ли за одну прогулку пройти каждый из семи мостов ровно один раз, — и никто не мог найти такой маршрут. Каждый раз, когда прогулка входит на участок суши и уходит с него, она расходует два моста: один на вход, один на выход. Участок с чётным числом мостов можно мысленно вычеркнуть из рассмотрения: входя и выходя, прогулка никогда на нём не застрянет. Участок с нечётным числом мостов — особый случай: прогулка обязана на нём начаться или закончиться. В Кёнигсберге нечётных участков было четыре,

а допустимо только два — прогулка невозможна. Определим нужные понятия, а затем сформулируем критерий.

ОПРЕДЕЛЕНИЕ 9.29 Эйлеров цикл

Цикл называется **эйлеровым**, если он проходит каждое ребро графа ровно один раз и возвращается в начало (здесь «цикл» — замкнутый маршрут, повторение вершин допускается).

ОПРЕДЕЛЕНИЕ 9.30 Эйлеров граф

Граф называется **эйлеровым**, если он содержит эйлеров цикл.

ОПРЕДЕЛЕНИЕ 9.31 Эйлерова цепь

Маршрут называется **эйлеровой цепью** (*Eulerian trail*), если он проходит каждое ребро графа ровно один раз, а его начало и конец различны.

Пример Эйлеровы графы

- K_5 эйлеров: каждая вершина имеет степень 4 (чётно).
- K_4 не эйлеров и эйлеровой цепи не имеет: все четыре вершины имеют нечётную степень 3, а цепь допускает не более двух таких вершин.
- Граф на рисунке 39 не эйлеров: четыре вершины имеют нечётную степень, что соответствует задаче о кёнигсбергских мостах.

ТЕОРЕМА 9.8 Критерий Эйлера⁵⁸

Связный неориентированный граф является эйлеровым тогда и только тогда, когда все степени чётны. Имеет эйлерову цепь тогда и только тогда, когда ровно две вершины имеют нечётную степень.

Набросок доказательства

Докажем эквивалентность в обе стороны.

($\Rightarrow$) Покажем необходимость. При каждом проходе через вершину эйлеров цикл использует два инцидентных ей ребра — одно для входа, одно для выхода. Поскольку цикл замкнут и проходит каждое ребро ровно один раз, степень каждой вершины чётна.

($\Leftarrow$) Докажем достаточность индуктивным построением алгоритмом Хирхольцера. Начинаем с произвольной вершины и идём по непосещённым рёбрам,

⁵⁸L. Euler. «Solutio problematis ad geometriam situs pertinentis» (Решение задачи, относящейся к геометрии положения). Commentarii academiae scientiarum Petropolitanae, 1741. Первая работа по теории графов — задача о кёнигсбергских мостах.

пока не зайдём в тупик. Поскольку все степени чётны, тупик возможен только в стартовой вершине. Полученный цикл C заметаем: если остались непосещённые рёбра, находим в C вершину с непосещёнными рёбрами, запускаем из неё новый обход и вставляем полученный цикл в C в этой вершине. Повторяем, пока не исчерпаем все рёбра.

Для эйлеровой цепи: если ровно две вершины нечётной степени, соединим их фиктивным ребром. В полученном графе все степени чётны — строим эйлеров цикл и удаляем фиктивное ребро, получая эйлерову цепь.

Эйлер доказал невозможность обхода кёнигсбергских мостов, но необходимое и достаточное условие существования эйлерова цикла сформулировал не он. Это сделал Карл Хирхольцер в 1873 году — спустя почти полтора века после статьи Эйлера. Хирхольцер также дал первый алгоритм построения эйлерова цикла, который, в современной терминологии, работает за $O(E)$ и лежит в основе всех практических реализаций. Постановка задачи и её полное решение могут разделяться веками.

Критерий Эйлера — рабочий инструмент, который можно проверить на современной карте. В 1944 году бомбардировки разрушили два из семи мостов Кёнигсберга — и прогулка, невозможная двести лет, внезапно стала возможной. Оставшиеся пять мостов соединяют участки суши так, что ровно два из них имеют нечётное число мостов: маршрут существует, начинать его нужно на одном из этих участков. Критерий Эйлера сводит вопрос, над которым ломали голову горожане, к одной проверке чётности.



В критерии Эйлера локальное и глобальное условие не взаимозаменяемы. Чётности всех степеней недостаточно без связности: два непересекающихся эйлеровых цикла в разных компонентах не дают единого обхода. Оба условия существенны.

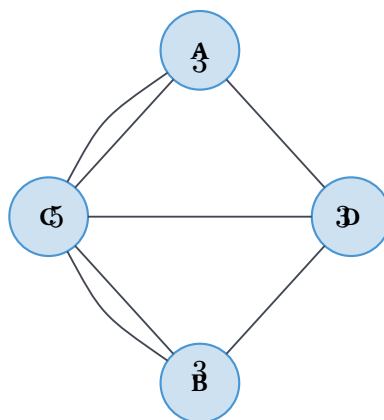


Рис. 39. Кёнигсбергские мосты.

Эйлеров цикл — замкнутый маршрут, проходящий каждое ребро ровно один раз. На рис. 40 показан граф-восьмёрка: цикл $B \rightarrow A \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow B$ обходит каждое ребро ровно один раз.

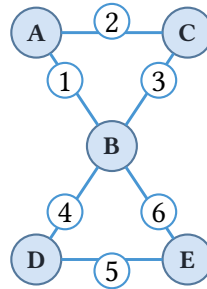


Рис. 40. Эйлеров граф-восьмёрка.

§ 9.9 * Произвольно вычерчиваемые графы

Критерий Эйлера отвечает на вопрос, *можно ли* вычертить граф одним росчерком. Следующий вопрос — можно ли вычертить его, не задумываясь о выборе пути. Графы, в которых карандаш не застревает ни при каком выборе, называются произвольно вычерчиваемыми.

ОПРЕДЕЛЕНИЕ 9.32 Произвольно вычерчиваемый граф

Граф называется **произвольно вычерчиваемым** из вершины v , если любой максимальный маршрут, начинающийся в v , проходит все рёбра графа.

Здесь маршрут понимается без повторения рёбер (trail): карандаш не проходит одно ребро дважды. Начав в v и каждый раз выбирая произвольное непройденное ребро, нельзя застрять, не исчерпав все рёбра.

ТЕОРЕМА 9.9 Критерий произвольной вычерчиваемости

Связный эйлеров граф произвольно вычерчиваем из вершины v тогда и только тогда, когда всякий цикл графа проходит через v .

Набросок доказательства

Докажем эквивалентность в обе стороны.

($\Leftarrow$) Покажем, что если всякий цикл проходит через v , то граф произвольно вычерчиваем из v . Пусть всякий цикл проходит через v , и пусть T — максимальный маршрут, начинающийся в v . Докажем, что T исчерпывает все рёбра.

Если T заканчивается в $w \neq v$, то через w пройдено нечётное число рёбер (каждый проход тратит два: вход и выход). А степень w чётна, значит осталось непройденное ребро, и T можно продолжить — противоречие с максимальной степенью. Значит, T заканчивается в v .

Если T не исчерпал всех рёбер, непройденные рёбра образуют граф с чётными степенями, а в таком графе есть цикл C . По условию C проходит через v , то есть в v осталось непройденное ребро, и T можно продолжить. Противоречие с максимальнойностью T : любой максимальный маршрут из v использует все рёбра.

($\Rightarrow$) Обратно: покажем, что если существует цикл C , не проходящий через v , то граф не произвольно вычерчиваем из v . Уберём рёбра такого цикла: оставшийся граф снова эйлеров, ведь у вершин C вычитается по два ребра и чётность степеней сохраняется. Максимальный маршрут из v в нём возвращается в v и исчерпывает все его рёбра, не тронув C . В исходном графе такой маршрут максимален: у v не осталось непройденных рёбер, потому что C не проходит через v . Но рёбра C не использованы — маршрут не эйлеров.

Пример Цветок

Возьмём несколько циклов с единственной общей вершиной v — такой граф называют **цветком**. Из v цветок произвольно вычерчиваем: каким бы лепестком мы ни начали, любой максимальный маршрут исчерпает все рёбра. Из другой вершины лепестка — нет: можно обойти свой лепесток целиком и застрять в нём, не добравшись до остальных.

§ 9.10 Гамильтоновы графы

Рис. 40 показывает эйлеров граф-восьмёрку: степени всех вершин чётны, и существует цикл, проходящий каждое ребро ровно один раз. Эйлеров цикл и его вариации разрешимы за полиномиальное время. Гамильтонов цикл — внешне похожая задача (посетить каждую вершину вместо каждого ребра), но её сложностной статус иной.

ОПРЕДЕЛЕНИЕ 9.33 Гамильтонов цикл

Цикл называется **гамильтоновым**, если он посещает каждую вершину графа ровно один раз и возвращается в начало.

ОПРЕДЕЛЕНИЕ 9.34 Гамильтонов граф

Граф называется **гамильтоновым**, если он содержит гамильтонов цикл.

Необходимого и достаточного условия, проверяемого за полиномиальное время, не известно — задача NP-полна. Существуют достаточные условия (теоремы Дирака, Оре), но они не являются необходимыми.

ЛОВУШКА Эйлеров и гамильтонов циклы — не одно и то же

Эйлеров цикл проходит каждое ребро ровно один раз, гамильтонов — каждую вершину ровно один раз. Эйлеровость имеет простой критерий — связность и чётность всех степеней — а для гамильтоновости такого критерия нет. Одно не следует из другого: граф-восьмёрка на рис. 40 эйлеров, но не гамильтонов, тогда как K_4 гамильтонов, но не эйлеров.

Пример Гамильтоновы и негамильтоновы графы

- Граф Петерсена на рис. 30 не гамильтонов: несмотря на 3-регулярность и отсутствие мостов, в нём нет цикла, посещающего все 10 вершин ровно один раз.
- K_n всегда гамильтонов при $n \geq 3$.
- Граф-звезда из 4 и более вершин не гамильтонов: центр вынужден повторяться при попытке обойти все листья.

ИСТОРИЯ Икосианская игра

В 1857 году сэр Уильям Роуэн Гамильтон изобрёл «Икосианскую игру»: деревянный додекаэдр, на вершинах которого были нанесены названия городов. Задача — обойти все вершины по рёбрам, посетив каждую ровно один раз и вернувшись в начало. Гамильтон продал идею лондонскому производителю игр за 25 фунтов. Игра коммерчески провалилась — викторианская публика не оценила математическую головоломку, — но породила понятие гамильтонова цикла, прочно вошедшее в теорию графов и теорию сложности. Деревянный додекаэдр за 25 фунтов оказался одной из самых выгодных инвестиций в математику XIX века.

Замечание Почему эйлеровость проще гамильтоновости

Условие эйлеровости — локальное: чётность степени вершины проверяется по её соседям. Можно проверить степени всех вершин по отдельности и объявить граф эйлеровым. Решение про одну вершину ничего не говорит про остальные.

Условие гамильтоновости — глобальное. Включив одно ребро в цикл, вы исключаете два других, инцидентных его концам. Выбор в одной части графа немедленно ограничивает весь остальной граф. Поэтому нет локального теста: любое решение ведёт к перебору.

ТЕОРЕМА 9.10 Дирак, 1952

Если в графе с $n \geq 3$ вершинами $\deg(v) \geq \frac{n}{2}$ для каждой вершины v , то G — гамильтонов.

Доказательство

Докажем от противного: рассмотрим максимальный простой путь и покажем, что его длина равна n .

Пусть $P = v_1, \dots, v_k$ — максимальный простой путь в G . По максимальнойности все соседи v_1, v_k лежат на P : иначе путь можно было бы удлинить.

По условию Дирака $\deg(v_1) \geq \frac{n}{2}$, $\deg(v_k) \geq \frac{n}{2}$. Определим множества индексов: $S = \{i \mid 1 \leq i < k, v_1 \text{ смежна с } v_{i+1}\}$, $T = \{i \mid 1 \leq i < k, v_k \text{ смежна с } v_i\}$. Тогда $|S| \geq \frac{n}{2}$, $|T| \geq \frac{n}{2}$, и оба множества — подмножества $\{1, \dots, k-1\}$.

По принципу включения-исключения:

$$|S \cap T| = |S| + |T| - |S \cup T| \geq \frac{n}{2} + \frac{n}{2} - (k-1) = n - (k-1)$$

. Поскольку $k \leq n$, имеем $|S \cap T| \geq 1$. Значит, существует индекс i такой, что v_1 смежна с v_{i+1} , а v_k — с v_i .

Построим цикл $C = v_1, \dots, v_i, v_k, v_{k-1}, \dots, v_{i+1}, v_1$ длины k . Рёбра цикла — это рёбра пути P плюс два новых: $v_i v_k, v_1 v_{i+1}$ (так как $i \in T, i \in S$ соответственно).

Если $k = n$, то C — гамильтонов цикл.

Если $k < n$, то из условия Дирака следует связность графа (сумма степеней любой пары несмежных вершин $\geq n$), поэтому существует вершина w вне C , смежная с некоторой вершиной C .

Разорвав цикл C у этой вершины и вставив w в разрыв, получим простой путь из $k+1$ вершин — противоречие с максимальнойностью P .

Следовательно, $k = n$, и G гамильтонов.

Теорема Дирака требует, чтобы *каждая* вершина имела высокую степень. Через восемь лет Ойстин Оре заметил, что условие можно ослабить: достаточно контролировать суммы степеней *несмежных* пар.

ТЕОРЕМА 9.11 Оре, 1960

Если в графе с $n \geq 3$ вершинами для каждой пары несмежных вершин u, v выполнено $\deg(u) + \deg(v) \geq n$, то G — гамильтонов.

Доказательство

Докажем тем же приёмом, что и Дирака: максимальный путь, множества S и T , включение-исключение — различие только в том, откуда берётся оценка степеней.

Пусть $P = v_1, \dots, v_k$ — максимальный простой путь в G . По максимальнойности все соседи v_1 и v_k лежат на P . Если v_1, v_k смежны, путь сразу замыкается в

цикл длины k : у Дирака отдельный случай не был нужен, так как оценка $\geq \frac{n}{2}$ выполнялась для каждой степени независимо от смежности. Если же v_1 и v_k несмежны, работает условие Оре: $\deg(v_1) + \deg(v_k) \geq n$. Для множеств $S = \{i \mid v_1 \text{ смежна с } v_{i+1}\}$ и $T = \{i \mid v_k \text{ смежна с } v_i\}$ это условие даёт точные значения $|S| = \deg(v_1)$ и $|T| = \deg(v_k)$ — не оценки снизу, как у Дирака, и связывает именно их сумму.

Далее — счёт Дирака: $|S \cap T| = |S| + |T| - |S \cup T| \geq n - (k - 1) \geq 1$, поэтому существует индекс i с $v_1 \sim v_{i+1}$ и $v_k \sim v_i$, и из P с двумя добавленными рёбрами собирается цикл C длины k . При $k = n$ цикл C гамильтонов. При $k < n$ условие Оре обеспечивает связность G : у каждой пары несмежных вершин сумма степеней не меньше n , поэтому вне C найдётся вершина, смежная с C , и её вставка в цикл даёт путь длиннее P — вопреки максимальнойности. Значит, $k = n$, и G гамильтонов.

Условие Оре слабее условия Дирака: если каждая вершина имеет степень $\geq \frac{n}{2}$, то сумма степеней любой пары (смежной или нет) не меньше n , так что теорема Дирака — прямое следствие теоремы Оре. Формулировка Дирака при этом часто удобнее для проверки: достаточно посчитать степени вершин, не вникая в то, какие пары смежны.

Пример Достаточные условия не необходимы

Цикл C_5 — гамильтонов: сам цикл и есть гамильтонов цикл. При этом ни одно из условий к нему не применимо. Условие Дирака требует $\deg(v) \geq \frac{5}{2}$ для каждой вершины, а все степени равны 2. Условие Оре требует $\deg(u) + \deg(v) \geq 5$ для каждой несмежной пары, а сумма двух двоек даёт 4.

Оба условия — достаточные, а не необходимые: гамильтоновость может держаться на структуре, которую степени не видят.

§ 9.11 Кратчайшие пути

Во взвешенном графе рёбра имеют числа — время, стоимость, длину. Как пройти от одной вершины до другой с наименьшей суммарной стоимостью, а не наименьшим числом шагов?

ОПРЕДЕЛЕНИЕ 9.35 Вес пути

Весом пути $P = v_0, v_1, \dots, v_k$ называется сумма весов его рёбер:

$$w(P) = \sum_{i=1}^k w(v_{i-1}, v_i).$$

ОПРЕДЕЛЕНИЕ 9.36 Кратчайший путь

Кратчайшим путём из s в t называется путь, имеющий минимальный вес среди всех путей из s в t .

ОПРЕДЕЛЕНИЕ 9.37 Взвешенное расстояние

Взвешенное расстояние $d(s, t)$ — вес кратчайшего пути из s в t . Если пути из s в t не существует, $d(s, t) = \infty$.

В невзвешенном графе расстояние $d(u, v)$ из раздела о связности — длина в рёбрах: частный случай весового расстояния при $w(e) = 1$ для всех рёбер. Здесь мы обобщаем эту меру на произвольные веса.

Пример Почему нельзя просто перебрать

Навигатор выбирает маршрут, решая задачу о кратчайшем пути на графе дорог, где вес ребра — время проезда. Всех путей на конечном графе конечно, но на карте города их так много, что перебор не успевает до следующего перекрёстка. Навигатор обязан выдать ответ за доли секунды, пока машина не проехала нужный поворот — поэтому нужен алгоритм, а не перебор.

Пример Почему BFS не справляется с весами

BFS находит кратчайшие пути в невзвешенном графе: он минимизирует число рёбер. Во взвешенном графе расстояние — сумма весов, а не их количество, и BFS ломается.

Рассмотрим граф из четырёх вершин: A, B, C, D . Рёбра: $A - B, B - C, C - D, A - D$ с весами 2, 2, 2, 10. BFS для пары A, D выдаст прямой путь $A \rightarrow D$ длины 1 (одно ребро), поскольку минимизирует количество рёбер. Но вес этого пути — 10.

Настоящий кратчайший путь — $A \rightarrow B \rightarrow C \rightarrow D$: три ребра, суммарный вес $2 + 2 + 2 = 6$. Он короче по весу, хотя и длиннее по числу рёбер. BFS такой путь пропускает: он считает рёбра, а не веса.

Итак, нужен алгоритм, который учитывает веса. Для неотрицательных весов это алгоритм Дейкстры. Для графов с отрицательными весами (и без отрицательных циклов) — Беллман–Форд. Оба возвращают кратчайшие пути от заданной вершины s до всех остальных, то есть вычисляют $d(s, v)$ и восстанавливают сами пути по предшественникам.

9.11.1 Алгоритм Дейкстры

Алгоритм Дейкстры обобщает BFS на графы с неотрицательными весами. Вместо очереди используется очередь с приоритетом (min-heap): на каждом шаге извлека-

ется вершина с минимальным известным расстоянием, и расстояния до её соседей обновляются (релаксируются).

АЛГОРИТМ Кратчайшие пути из одного источника (алгоритм Дейкстры)

1. Инициализировать расстояния: $\text{dist}(s) = 0$, $\text{dist}(v) = \infty$ для $v \neq s$. Добавить $(0, s)$ в min-heap.
2. Пока куча не пуста:
 - Извлечь (d, u) : вершина u с минимальным расстоянием d .
 - Если $d > \text{dist}(u)$ — пропустить (устаревшая запись).
 - Для каждого соседа v вершины u с весом ребра $w(u, v)$: если $\text{dist}(u) + w(u, v) < \text{dist}(v)$, то $\text{dist}(v) = \text{dist}(u) + w(u, v)$, $\text{parent}(v) = u$, добавить $(\text{dist}(v), v)$ в кучу.

Почему Дейкстра корректен? Когда вершина u извлекается из кучи, $\text{dist}(u)$ уже является длиной кратчайшего пути до u . Все рёбра имеют неотрицательный вес, поэтому любой другой путь до u проходит через вершину, всё ещё находящуюся в куче с расстоянием $\geq \text{dist}(u)$. Добавление неотрицательного веса не может уменьшить это расстояние. Это индукция по порядку извлечения вершин из кучи.

Именно неотрицательность весов критична. Если есть ребро отрицательного веса, Дейкстра может выдать неверный ответ: вершина, извлечённая раньше, могла бы быть достигнута короче через отрицательное ребро, проходящее через вершину, извлечённую позже.

На рис. 41 показан контрпример: путь через отрицательное ребро $A \rightarrow B$ оказывается короче прямого.

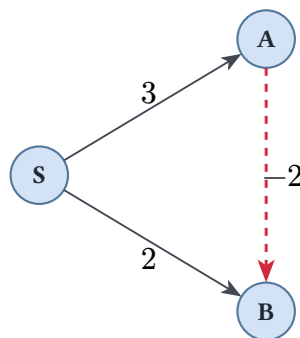


Рис. 41. Контрпример для Дейкстры: отрицательное ребро $A \rightarrow B$.

Для графов с отрицательными весами (но без отрицательных циклов) служит алгоритм Беллмана–Форда.

Пример Трассировка Дейкстры

Граф: вершины $\{A, B, C, D\}$. Рёбра: $A - B$, $A - C$, $B - C$, $B - D$, $C - D$ с весами соответственно 2, 5, 1, 4, 1.

Старт из A .

- Куча: $[(0, A)]$.

Извлекаем A . Релаксируем: $\text{dist}(B) = 2, \text{dist}(C) = 5$.

- Куча: $[(2, B), (5, C)]$.

Извлекаем B . Релаксируем: $\text{dist}(C) = \min(5, 2 + 1) = 3, \text{dist}(D) = \min(\infty, 2 + 4) = 6$.

- Куча: $[(3, C), (5, C_{\text{old}}), (6, D)]$.

Извлекаем C (3). Релаксируем: $\text{dist}(D) = \min(6, 3 + 1) = 4$.

- Куча: $[(4, D), (5, C_{\text{old}}), (6, D_{\text{old}})]$.

Извлекаем D (4). Затем извлекаем C_{old} (5): $5 > 3$ — пропускаем устаревшую запись C . Затем извлекаем D_{old} (6): $6 > 4$ — пропускаем устаревшую запись D .

Кратчайшие расстояния от A : до B, C, D — соответственно 2, 3, 4. Кратчайший путь до D : $A \rightarrow B \rightarrow C \rightarrow D$ (суммарный вес $2 + 1 + 1 = 4$).

Сложность Дейкстры с бинарной кучей — $O((V + E) \log V)$: каждая вершина извлекается из кучи один раз ($O(V \log V)$), каждое ребро может вызвать операцию обновления ключа в куче ($O(E \log V)$).

9.11.2 Алгоритм Беллмана–Форда

Беллман–Форд ($O(VE)$) решает задачу кратчайших путей с отрицательными весами (но без отрицательных циклов).

1. Он выполняет $V - 1$ раундов релаксации *всех* рёбер. После k раундов кратчайшие пути, состоящие не более чем из k рёбер, найдены корректно.
2. Дополнительный V -й раунд обнаруживает отрицательный цикл: если после него релаксация всё ещё меняет расстояния, достижимый отрицательный цикл существует.
3. Цикл, недостижимый из старта, алгоритму не мешает: он мог бы лишь улучшить расстояния до вершин, которых и так не достичь.

Пример Трассировка Беллмана–Форда

Вернёмся к графу с рис. 41: $S \rightarrow A$ весом 3, $S \rightarrow B$ весом 2, $A \rightarrow B$ весом -2 . Начальные расстояния: $\text{dist}(S) = 0, \text{dist}(A) = \text{dist}(B) = \infty$.

Первый раунд релаксации всех рёбер:

1. $S \rightarrow A$: $\text{dist}(A) = \min(\infty, 0 + 3) = 3$.
2. $S \rightarrow B$: $\text{dist}(B) = \min(\infty, 0 + 2) = 2$.
3. $A \rightarrow B$: $\text{dist}(B) = \min(2, 3 - 2) = 1$.

Второй раунд ничего не меняет — значит, отрицательного цикла нет, и ответ найден: $\text{dist}(B) = 1$ по пути $S \rightarrow A \rightarrow B$. Дейкстра на этом же графе извлекла бы B с расстоянием 2 раньше, чем дошла бы до A , и не узнала бы про отрицательное ребро.

9.11.3 Алгоритм Флойда–Уоршелла

Дейкстра и Беллман–Форд отвечают на вопрос «как пройти из s во все стороны». Иногда нужны расстояния между всеми парами сразу: таблица перегонов между городами, матрица задержек между маршрутизаторами. Наивный план — прогнать Дейкстру из каждой вершины. На плотном графе это $O(V^3 \log V)$ и куча в придачу. Флойд–Уоршелл решает ту же задачу за $O(V^3)$, не требуя ни кучи, ни иных структур данных — одна динамика.

Идея — вести перебор путей по их промежуточным вершинам. Пронумеруем вершины $1, \dots, n$ и обозначим через D_{ij}^k вес кратчайшего пути из i в j , все промежуточные вершины которого лежат среди первых k . Кратчайший путь либо заходит в вершину k , либо нет, поэтому

$$D_{ij}^k = \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1}).$$

Хранить все слои не нужно: значения слоя k получаются из слоя $k-1$ заменой на месте, и вся динамика сворачивается в тройной вложенный цикл — внешний по промежуточной вершине, внутренние по парам.

АЛГОРИТМ Все пары кратчайших путей (алгоритм Флойда–Уоршелла)

1. Заполнить D_{ij} : 0 при $i = j$, вес ребра $w(i, j)$, если ребро есть, и ∞ иначе.
2. Для каждой вершины k от 1 до n :
 - Для каждой пары i, j : если $D_{ik} + D_{kj} < D_{ij}$, заменить D_{ij} на $D_{ik} + D_{kj}$.
3. После всех итераций D_{ij} — взвешенное расстояние $d(i, j)$.

Корректность держится на инварианте внешнего цикла: после итерации k матрица D хранит D^k . Обновление на месте корректно, пока в графе нет отрицательных циклов: тогда кратчайшие пути просты, путь до k не проходит через k как промежуточную вершину, и значения D_{ik}, D_{kj} на слое k не меняются. Отрицательный цикл алгоритм обнаруживает по диагонали: после завершения $D_{ii} < 0$ тогда и только тогда, когда через i проходит отрицательный цикл.

Тот же тройной цикл считает и достижимость без весов: достаточно заменить $\min$ и $+$ на $\vee$ и $\wedge$ — получится алгоритм Уоршелла для транзитивного замыкания из главы 5. С матрицей смежности ситуация зеркальная: степень матрицы $(A^k)_{ij}$ помнит число маршрутов длины ровно k , а Флойд–Уоршелл отвечает на вопрос о минимуме веса среди маршрутов произвольной длины — и тоже одним проходом по промежуточным вершинам.

Проверка Кратчайшие пути

1. Почему извлечённая из кучи вершина больше не меняет своё расстояние и какое предположение о весах за этим стоит?
2. Что обнаруживает дополнительный V -й раунд Беллмана–Форда и почему $V-1$ раундов достаточно?

§ 9.12 Обходы с оптимизацией: почтальон и коммивояжёр

Эйлеров и гамильтонов циклы отвечают на вопрос «можно ли обойти всю сеть», но инженеру важна и цена обхода. Обе задачи этого раздела оптимизационные: почтальон обязан пройти каждое ребро (инспекция улиц и трубопроводов), коммивояжёр — посетить каждую вершину (объезд городов). При внешнем сходстве постановок сложностной статус разный: первая задача полиномиальна, вторая NP-трудна.

9.12.1 Задача китайского почтальона

Дан связный взвешенный граф. Требуется найти кратчайший замкнутый маршрут, проходящий каждое ребро хотя бы один раз. Если все степени чётны, задача сводится к эйлерову циклу и решается за $O(E)$.

Если есть вершины нечётной степени, их чётное число (по лемме о рукопожатиях). Паросочетание — множество рёбер без общих вершин (формальное определение — ниже, в разделе о двудольных графах). Совершенное на подграфе нечётных вершин — покрывающее их все. Алгоритм состоит из трёх шагов:

1. Найти паросочетание минимального веса на подграфе вершин нечётной степени (задача о взвешенном совершенном паросочетании, алгоритм

Эдмондса⁵⁹).

1. Добавить рёбра паросочетания как дублируемые — после этого все степени становятся чётными.
2. В полученном мультиграфе построить эйлеров цикл.

Итоговая сложность — $O(V^3)$.

Та же задача возникает при планировании маршрутов уборочной техники и инспекции трубопроводов: нужно покрыть все сегменты сети, минимизируя повторные проходы.⁶⁰

9.12.2 Задача коммивояжёра

Задача коммивояжёра (TSP, от англ. traveling salesman problem) просит найти кратчайший маршрут, обходящий все города и возвращающийся в начало, — кратчайший гамильтонов цикл во взвешенном графе. В общей постановке она NP-трудна, но это не остановило разработку практических методов. Современный рекорд точного решения — 85 900 городов (Concorde, 2006), достигнутый методом branch-and-cut. Алгоритм работает так:

1. Задача формулируется как целочисленная линейная программа.
2. Решается её линейная релаксация (переменные могут быть дробными).

⁵⁹J. Edmonds, E. L. Johnson. «Matching, Euler tours and the Chinese postman». Mathematical Programming, 1973.

⁶⁰M.-K. Kwan. «Graphic programming using odd or even points». Chinese Mathematics, 1962.

3. Нецелочисленные решения отсекаются добавлением отрезающих плоскостей — линейных неравенств, которым удовлетворяют все гамильтоновы циклы, но которые нарушаются текущим дробным решением.⁶¹

Для метрического TSP (расстояния удовлетворяют неравенству треугольника) алгоритм Кристофидеса (1976) даёт 1.5-приближение за полиномиальное время. Алгоритм работает в четыре шага:

1. Строится минимальное остовное дерево (MST).
2. На вершинах нечётной степени в MST находится совершенное паросочетание минимального веса.
3. Объединение MST и паросочетания даёт эйлеров мультиграф.
4. Из эйлерова цикла строится гамильтонов цикл shortcut'ами (пропуская повторно посещённые вершины).⁶²

§ 9.13 Двудольные графы

Во многих задачах объекты делятся на две группы, и связи возможны только между группами: работы и исполнители, студенты и курсы. Такие структуры выделяют в отдельный класс графов.

ОПРЕДЕЛЕНИЕ 9.38 Двудольный граф (*bipartite graph*)

Граф называется **двудольным**, обозначается

$$G = (X, Y, E),$$

если его вершины можно разбить на два непересекающихся множества X, Y , так что каждое ребро соединяет вершину из X с вершиной из Y .

Пример Двудольные графы

C_4 (цикл из 4 вершин) двудольен: $X = \{1, 3\}, Y = \{2, 4\}$, каждое ребро идёт между долями. $K_{3,3}$ — полный двудольный граф с $|X| = |Y| = 3$, показанный на рис. 52. C_3 (треугольник) не двудольен: при попытке раскрасить две вершины в разные цвета третья соседствует с обеими. Деревья всегда двудольны: BFS-раскраска по чётности уровня даёт разбиение.

ТЕОРЕМА 9.12 Критерий двудольности

Граф является двудольным тогда и только тогда, когда в нём нет циклов нечётной длины.

⁶¹G. Dantzig, R. Fulkerson, S. Johnson. «Solution of a large-scale traveling-salesman problem». J. Operations Research Society of America, 1954.

⁶²N. Christofides. «Worst-case analysis of a new heuristic for the travelling salesman problem». Report 388, Carnegie-Mellon University, 1976.

Доказательство

Докажем эквивалентность в обе стороны.

($\Rightarrow$) Покажем необходимость. Пусть граф двудолен с долями X, Y . Рассмотрим произвольный цикл $v_1, v_2, \dots, v_k, v_1$. Вершины цикла чередуются между долями: $v_1 \in X, v_2 \in Y, v_3 \in X$ и так далее. Чтобы вернуться в X на последнем шаге, число шагов должно быть чётным. Следовательно, k чётно.

($\Leftarrow$) Докажем достаточность. Пусть в графе нет циклов нечётной длины. BFS — обход по слоям: стартовая вершина, затем её соседи, затем соседи соседей и так далее (подробно — в разделе об обходах графа выше). В каждой компоненте связности запустим BFS из произвольной вершины и назовём уровень каждой вершины — расстояние от стартовой. Раскрасим вершины чётных уровней в один цвет, нечётных — в другой. Если бы ребро соединяло вершины одного цвета, то вместе с путями от стартовой вершины оно образовало бы цикл нечётной длины, что противоречит исходному предположению. Значит, каждое ребро соединяет вершины разных цветов — граф двудолен.

Проверка двудольности и построение разбиения выполняются за $O(V + E)$ одним обходом.

ЛОВУШКА Двудольность: проверять не только смежные пары

Проверив, что каждая смежная пара окрашена в разные цвета, заключают о двудольности всего графа. Но нужен непротиворечивый раскрас всего графа: цвет вершины един, и решение для одной пары стесняет всех остальных. Критерий двудольности — отсутствие циклов нечётной длины, и проверяется он одним обходом всего графа. Треугольник — контрпример: каждая пара его вершин смежна и требует разных цветов, а двух цветов на три вершины не хватает.

На рис. 42 вершины разбиты на две доли, и каждое ребро соединяет вершины из разных долей.

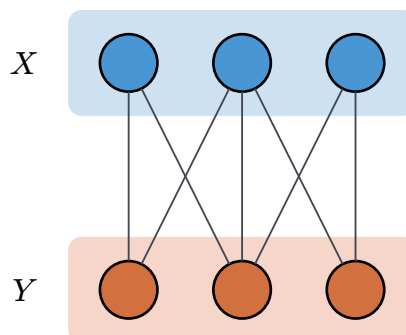


Рис. 42. Двудольный граф.

Двудольная структура естественно подводит к задаче о паросочетании: как установить взаимно-однозначное соответствие между элементами двух долей?

ОПРЕДЕЛЕНИЕ 9.39 Паросочетание (*matching*)

Паросочетанием называется множество рёбер, не имеющих общих вершин.

ОПРЕДЕЛЕНИЕ 9.40 Совершенное паросочетание

Паросочетание называется **совершенным**, если оно покрывает все вершины графа.

Пример Паросочетания

В двудольном графе $K_{3,2}$ (три вершины слева, две справа, все возможные рёбра): максимальное паросочетание состоит из двух рёбер — по одному на каждую вершину правой доли. Совершенного паросочетания нет: нечётное число вершин. В $K_{3,3}$ совершенное паросочетание существует: можно выбрать три ребра, попарно не имеющие общих вершин, покрывающие все шесть вершин.

Рис. 43 показывает максимальное паросочетание в двудольном графе (выделено жирным).

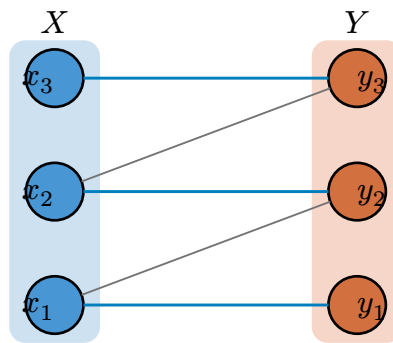


Рис. 43. Двудольный граф с максимальным паросочетанием (выделено жирным).

Существование паросочетания, покрывающего долю X , не гарантировано: если у нескольких вершин из X общие соседи, рёбер может не хватить. Теорема Холла даёт точное условие.

ТЕОРЕМА 9.13 Теорема Холла⁶³

В двудольном графе $(X \cup Y, E)$ паросочетание, покрывающее X , существует тогда и только тогда, когда

$$\forall S \subseteq X : |N(S)| \geq |S|,$$

где $N(S)$ — множество вершин из Y , смежных хотя бы с одной вершиной из S .

⁶³P. Hall. «On representatives of subsets». Journal of the London Mathematical Society, 1935.

Доказательство

Докажем эквивалентность в обе стороны.

Необходимость. Пусть M — паросочетание, покрывающее X . Рассмотрим произвольное $S \subseteq X$. Рёбра паросочетания, инцидентные вершинам S , не имеют общих концов, поэтому идут в попарно различные вершины Y . Следовательно, соседей у S не меньше, чем вершин в S : $|N(S)| \geq |S|$.

Достаточность. Докажем от противного. Пусть условие Холла выполнено, но максимальное паросочетание M покрывает не все вершины X : $|M| < |X|$.

Введём два понятия. **Чередующийся путь** — путь, рёбра которого попеременно не принадлежат и принадлежат M . **Увеличивающий путь** — чередующийся путь с обоими непокрытыми концами. Инверсия рёбер на увеличивающем пути (не-паросочетанные становятся паросочетанными) увеличивает паросочетание на одно ребро. Поэтому в максимальном паросочетании увеличивающих путей нет.

Возьмём непокрытую вершину $z_0 \in X$ и обозначим через $Z \subseteq X$ множество вершин, достижимых из z_0 чередующимися путями.

Покажем два свойства.

Свойство 1. Каждая вершина $y \in N(Z)$ покрыта паросочетанием M . Если бы y была не покрыта, то по определению $N(Z)$ нашлась бы вершина $z \in Z$ с ребром zy , которое не входит в M (y не покрыта). Продлив до z чередующийся путь из z_0 , получили бы чередующийся путь из z_0 в y с обоими непокрытыми концами — увеличивающий путь, противоречащий максимальнойности M .

Свойство 2. Партнёр каждой вершины $y \in N(Z)$ по паросочетанию лежит в $Z \setminus \{z_0\}$. Возьмём $y \in N(Z)$ и вершину $z \in Z$ с ребром zy . Продолжим чередующийся путь из z_0 в z ребром zy . Последнее ребро zy — не из M , поэтому следующий шаг чередующегося пути, если y покрыта, обязан быть ребром паросочетания. По свойству 1 вершина y покрыта, значит её партнёр y' связан с y ребром из M , и путь можно продолжить до y' . Следовательно, y' достижим из z_0 чередующимся путём: $y' \in Z$. Этот партнёр не может быть z_0 : вершина z_0 не покрыта паросочетанием, значит у неё партнёра нет. Итак, каждой $y \in N(Z)$ отвечает партнёр в $Z \setminus \{z_0\}$.

Разные y имеют разных партнёров — M есть паросочетание, — поэтому отображение $y \rightarrow \text{partner}(y)$ инъективно.

Из свойств 1 и 2 следует: $N(Z)$ инъективно отображается в $Z \setminus \{z_0\}$, значит $|N(Z)| \leq |Z| - 1$. Но условие Холла требует $|N(Z)| \geq |Z|$. Противоречие.

Следовательно, паросочетание M покрывает X .

Замечание Необходимое условие, оказывающееся достаточным

Условие Холла — пример паттерна, проходящего через всю главу: напрашивающееся необходимое условие оказывается и достаточным. Как в эйлеровости (чётность степеней), как в двудольности (отсутствие нечётных циклов), необходимость видна сразу, а достаточность нетривиальна. Теорема Холла — крайний случай этого паттерна: достаточность его условия опирается на всю теорию паросочетаний.

§ 9.14 Сетевые потоки

Задача о максимальном потоке моделирует транспортные сети, трубопроводы и информационные каналы. Материал движется от источника к стоку. Каждое соединение имеет ограничение на пропускную способность (*capacity*). Требуется пропустить максимально возможный объём от s к t , не нарушая ограничений.

Представим водопровод: труба от насосной станции s к городу t идёт через два узла — a, b . Труба $s \rightarrow a$ пропускает до 5 литров в секунду, $s \rightarrow b$ — до 3, $a \rightarrow t$ — до 3, $b \rightarrow t$ — до 4, и есть перемычка $a \rightarrow b$ на 2 литра. Сколько воды можно доставить из s в t за секунду?

Ответ — 7 литров в секунду. Трубы $a \rightarrow t, b \rightarrow t$ вместе пропускают не более 7 литров (сумма пропускных способностей $3 + 4$), и этот предел достигим. Это и есть задача о максимальном потоке.

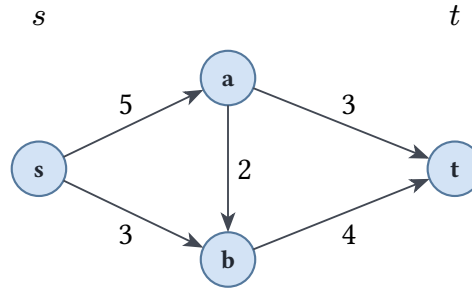
ОПРЕДЕЛЕНИЕ 9.41 Сеть

Сетью (flow network) называется ориентированный граф $G = (V, E)$ с выделенными вершинами:

- $s \in V$ — **источник** (source),
- $t \in V, t \neq s$ — **сток** (sink),

и функцией пропускных способностей $c : E \rightarrow \mathbb{R}^+$, сопоставляющей каждому ребру неотрицательное число — максимальный поток через это ребро.

Сеть из водопроводного примера — на рис. 44: источник s , сток t , промежуточные вершины a, b , а числа на рёбрах — пропускные способности $c(u, v)$. Сеть задаёт топологию и ограничения. Теперь определим, что значит «пропустить поток» по этой сети.

Рис. 44. Сеть с источником s и стоком t .**ОПРЕДЕЛЕНИЕ 9.42 Поток**

Потоком (flow) в сети называется функция $f : E \rightarrow \mathbb{R}$, удовлетворяющая двум условиям:

- **Ограничение пропускной способности:** $0 \leq f(u, v) \leq c(u, v)$ для каждого ребра $(u, v) \in E$.
- **Закон сохранения потока:** для каждой вершины $v \in V \setminus \{s, t\}$ суммарный входящий поток равен суммарному исходящему:

$$\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$$

Источник только производит поток, сток только поглощает. Остальные вершины — транзитные: сколько вошло, столько и вышло. На рис. 45 показан допустимый поток: на каждом ребре подписан $\frac{f(u, v)}{c(u, v)}$.

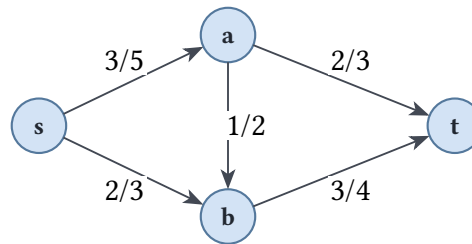


Рис. 45. Допустимый поток.

ОПРЕДЕЛЕНИЕ 9.43 Величина потока

Величиной потока $|f|$ называется чистый поток, выходящий из источника:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{u \in V} f(u, s)$$

Пример Допустимый поток

Рассмотрим сеть на рис. 44. Возьмём поток на рис. 45:

- $f(s, a) = 3, f(s, b) = 2,$
- $f(a, t) = 2, f(a, b) = 1, f(b, t) = 3.$

Проверим закон сохранения:

- В a : входящий 3, исходящие $2 + 1 = 3.$
- В b : входящие $2 + 1 = 3,$ исходящий 3.

Значит, поток допустим. Величина: $|f| = f(s, a) + f(s, b) = 3 + 2 = 5.$

Этот поток не максимален: максимум равен 7. Увеличим его: по пути $s \rightarrow a \rightarrow t$ добавим 1 единицу, по пути $s \rightarrow b \rightarrow t$ — ещё 1. Теперь $f(s, a) = 4, f(s, b) = 3, f(a, t) = 3, f(b, t) = 4,$ перемычка $f(a, b) = 1$ не меняется. Величина потока — $4 + 3 = 7.$

Больше провести нельзя: весь поток в сток t идёт через рёбра $a \rightarrow t, b \rightarrow t,$ а их суммарная пропускная способность $3 + 4 = 7$ — верхняя граница любого потока.

Алгоритм Форда–Фалкерсона (1956) ищет увеличивающий путь — путь $s \rightarrow t$ в остаточной сети, вдоль которого поток можно дополнительно пропустить.

ОПРЕДЕЛЕНИЕ 9.44 Остаточная сеть

Для потока f в сети G остаточная сеть G_f содержит те же вершины, а каждое ребро $(u, v) \in E$ порождает:

- **прямое ребро** (u, v) с остаточной пропускной способностью $c_f(u, v) = c(u, v) - f(u, v) > 0$ — сколько ещё можно послать;
- **обратное ребро** (v, u) с остаточной пропускной способностью $c_f(v, u) = f(u, v) > 0$ — сколько потока можно «отменить».

Обратное ребро — это возможность отказаться от части уже посланного потока: пустить его назад, освободив место. Математически это вторая стрелка с той же пропускной способностью, равной текущему потоку $f(u, v)$. Остаточная сеть для потока с рис. 45 — на рис. 46: сплошные стрелки — прямые рёбра с остаточной пропускной способностью $c_f(u, v) = c(u, v) - f(u, v)$, пунктирные — обратные рёбра с $c_f(v, u) = f(u, v)$.

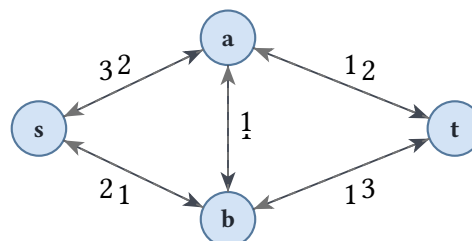


Рис. 46. Остаточная сеть.

ОПРЕДЕЛЕНИЕ 9.45 Увеличивающий путь (*augmenting path*)

Увеличивающий путь — путь $s \rightarrow t$ в остаточной сети G_f . Вдоль него можно пропустить дополнительный поток, равный минимальной остаточной пропускной способности на пути.

Пока такой путь существует, поток можно увеличить.

АЛГОРИТМ Максимальный поток (алгоритм Форда–Фалкерсона)

1. Начать с нулевого потока: $f(u, v) = 0$ для всех рёбер.
2. Пока существует увеличивающий путь $P : s \rightarrow t$ в остаточной сети:
 - Найти минимальную остаточную пропускную способность $\Delta = \min_{e \in P} c_f(e)$.
 - Пропустить Δ единиц потока вдоль P : для каждого ребра (u, v) на пути $f(u, v) + = \Delta$, для обратного ребра $f(v, u) - = \Delta$.
3. Когда увеличивающих путей нет — текущий поток максимален.

Время работы зависит от способа поиска увеличивающего пути. С BFS (алгоритм Эдмондса–Карпа) — $O(VE^2)$. С более сложными структурами (динамические деревья) — $O(VE \log V)$.

Второй вопрос: как ограничить поток сверху? Разрежем сеть на две части так, чтобы источник и сток оказались по разные стороны, — и посмотрим, сколько потока может пройти через разрез.

ОПРЕДЕЛЕНИЕ 9.46 Разрез

Разрезом (A, B) называется разбиение вершин на два множества A, B , где $A \cup B = V$, $A \cap B = \emptyset$, $s \in A$ и $t \in B$.

Разрез можно понимать двумя способами. Первый — пара множеств вершин: левая часть A содержит источник, правая B — сток. Второй — множество рёбер, пересекающих границу. Для разреза (A, B) выделяют два направленных множества:

$$\delta^+(A) := \{(u, v) \in E \mid u \in A, v \in B\},$$

$$\delta^-(A) := \{(u, v) \in E \mid u \in B, v \in A\}.$$

Первое — рёбра из A в B (пересекают границу по направлению потока), второе — рёбра из B в A (встречные, против потока).

ОПРЕДЕЛЕНИЕ 9.47 Пропускная способность разреза

Пропускная способность разреза (A, B) — сумма пропускных способностей рёбер из A в B :

$$c(A, B) = \sum_{(u,v) \in \delta^+(A)} c(u, v) = \sum_{u \in A, v \in B} c(u, v)$$

.

Примечание Считаются только рёбра из A в B

Рёбра из B в A (встречные) в пропускную способность не входят. Разрезаем сеть, чтобы «отрезать» t от s : важно, сколько воды может пройти из A в B , а не обратно.

Разрез $A = \{s, a, b\}$, $B = \{t\}$ показан на рис. 47: рёбра из A в B выделены жирным.

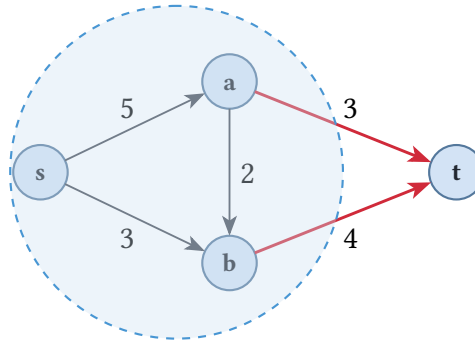


Рис. 47. Разрез $A = \{s, a, b\}$, $B = \{t\}$.

Пропускная способность разреза — это ответ на вопрос «сколько труб нужно перерезать, чтобы отключить сток». Но поток, уже идущий в сети, тоже можно измерить через разрез.

ОПРЕДЕЛЕНИЕ 9.48 Поток через разрез

Потоком через разрез (A, B) называется

$$f(A, B) = \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e)$$

. Это поток из A в B минус поток из B в A .

ТЕОРЕМА 9.14 Поток через любой разрез равен величине потока

Для любого допустимого потока f и любого разреза (A, B) :

$$f(A, B) = |f|.$$

Доказательство

Докажем двойным счётом: посчитаем одну и ту же сумму двумя способами.

Сложим закон сохранения по всем вершинам $v \in A$. Внутренние рёбра (оба конца в A) учитываются дважды с разными знаками и сокращаются. Остаются только рёбра границы: вклад $+f(e)$ для $e \in \delta^+(A)$ и $-f(e)$ для $e \in \delta^-(A)$. Итого $f(A, B)$.

С другой стороны, та же сумма равна потоку, вышедшему из s : для s это вклад $\sum_v f(s, v) - \sum_u f(u, s) = |f|$, а для остальных $v \in A \setminus \{s\}$ закон сохранения даёт нуль. Значит, $f(A, B) = |f|$.

СЛЕДСТВИЕ 9.15 Верхняя граница потока

Для любого потока f и любого разреза (A, B) :

$$|f| = f(A, B) \leq c(A, B).$$

Доказательство: $f(e) \leq c(e)$ по ограничению пропускной способности, поэтому каждое слагаемое в $f(A, B)$ не превосходит соответствующего в $c(A, B)$, а встречные рёбра лишь уменьшают разность. Значит, **величина любого потока ограничена пропускной способностью любого разреза сверху**. Каждый разрез даёт верхнюю оценку на максимальный поток — и чем меньше разрез, тем точнее оценка. На рис. 48 показан поток через разрез $A = \{s, a\}$, $B = \{b, t\}$.

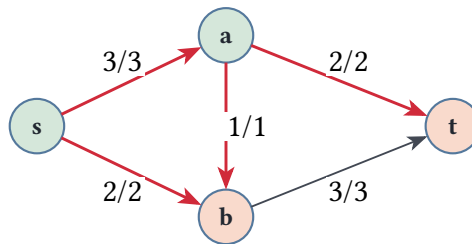


Рис. 48. Поток через разрез $A = \{s, a\}$, $B = \{b, t\}$.

Пример Считаем разрезы

Рассмотрим сеть с рис. 44 ($s \rightarrow a$ 5, $s \rightarrow b$ 3, $a \rightarrow t$ 3, $b \rightarrow t$ 4, $a \rightarrow b$ 2). Переберём несколько разрезом.

- $A = \{s\}$, $B = \{a, b, t\}$: рёбра из A — $s \rightarrow a$ (5), $s \rightarrow b$ (3). Пропускная способность $5 + 3 = 8$.
- $A = \{s, a\}$, $B = \{b, t\}$: рёбра из A — $s \rightarrow b$ (3), $a \rightarrow b$ (2), $a \rightarrow t$ (3). Пропускная способность $3 + 2 + 3 = 8$.
- $A = \{s, a, b\}$, $B = \{t\}$: рёбра из A — $a \rightarrow t$ (3), $b \rightarrow t$ (4). Пропускная способность $3 + 4 = 7$.

Минимальная из трёх — 7 (разрез $A = \{s, a, b\}$, $B = \{t\}$). Это ровно величина максимального потока, найденного ранее: поток $|f| = 7$. Совпадение не случайно: минимальный разрез и есть бутылочное горлышко сети.

ТЕОРЕМА 9.16 **Max-flow min-cut⁶⁴**

Величина максимального потока равна минимальной пропускной способности разреза.

Набросок доказательства

Докажем равенство с двух сторон.

Верхняя граница уже известна: $|f| \leq c(A, B)$ для любого разреза. Значит, максимальный поток не больше минимального разреза.

Достижимость. Покажем, что равенство достижимо. Алгоритм Форда–Фалкерсона останавливается, когда в остаточной сети нет увеличивающего пути. Пусть A — множество вершин, достижимых из s в остаточной сети, $B = V \setminus A$. Тогда $s \in A$, $t \in B$, и это — разрез. Все рёбра из A в B насыщены: иначе по ним осталась бы остаточная способность и вершина из B стала бы достижима. Все рёбра из B в A имеют нулевой поток: иначе обратное ребро в остаточной сети вело бы из A в B , и вершина из B была бы достижима. Следовательно, $f(A, B) = c(A, B)$, а по следствию $|f| = f(A, B)$. Поток равен пропускной способности разреза — максимален, и разрез минимален.

Пример Трассировка Форда–Фалкерсона

Сеть: вершины $\{s, a, b, t\}$. Рёбра с пропускными способностями: $s \rightarrow a$ (3), $s \rightarrow b$ (2), $a \rightarrow b$ (1), $a \rightarrow t$ (2), $b \rightarrow t$ (3).

Итерация 1. Остаточные способности равны исходным. Увеличивающий путь: $s \rightarrow a \rightarrow t$. Минимальная остаточная пропускная способность: $\Delta = \min(3, 2) = 2$. Поток: $f(s, a) = 2$, $f(a, t) = 2$. Остаточные:

- $c_f(s, a) = 1$
- $c_f(a, s) = 2$
- $c_f(a, t) = 0$
- $c_f(t, a) = 2$

Итерация 2. Путь $s \rightarrow b \rightarrow t$: $\Delta = \min(2, 3) = 2$. Поток: $f(s, b) = 2$, $f(b, t) = 2$. Остаточные:

- $c_f(s, b) = 0$
- $c_f(b, s) = 2$
- $c_f(b, t) = 1$
- $c_f(t, b) = 2$

Итерация 3. Путь $s \rightarrow a \rightarrow b \rightarrow t$: используем прямое ребро $a \rightarrow b$ (остаточная способность 1). $\Delta = \min(1, 1, 1) = 1$. Поток:

⁶⁴L. R. Ford, D. R. Fulkerson. «Maximal Flow Through a Network». Canadian Journal of Mathematics, 1956. Также известна как теорема Форда–Фалкерсона.

- $f(s, a) = 3$
- $f(a, b) = 1$
- $f(b, t) = 3$

Итоговый поток: $|f| = f(s, a) + f(s, b) = 3 + 2 = 5$. Минимальный разрез: например, $S = \{s, a\}, T = \{b, t\}$ с пропускной способностью $c(s, b) + c(a, t) + c(a, b) = 2 + 2 + 1 = 5$. Совпадает с величиной потока.

(Минимальных разрезов несколько: в финальной остаточной сети из s достигим лишь сам s , так что разрез $S = \{s\}$ тоже минимален и тоже имеет способность 5.)

Интуиция: поток $s \rightarrow t$ ограничен «бутылочным горлышком» — минимальным разрезом. Никакой поток не может превысить пропускную способность любого разреза: весь поток $S \rightarrow T$ проходит через рёбра разреза. И всегда существует поток, достигающий этого максимума.

ТЕОРЕМА 9.17 Интегральность потока

Если все пропускные способности сети — целые числа, то существует максимальный поток, принимающий целые значения на каждом ребре.

Доказательство

Докажем по индукции по шагам алгоритма Форда–Фалкерсона.

Алгоритм стартует с нулевого потока — целочисленного. На каждом шаге он пропускает вдоль увеличивающего пути поток, равный минимальной остаточной пропускной способности на пути — минимуму целых чисел. По индукции все значения потока остаются целыми на протяжении работы алгоритма. Когда увеличивающих путей не остаётся, построенный поток максимален и целочислен.

Интегральность — вот почему сведение задач к потоку ниже даёт точный ответ. Паросочетания и пути суть целочисленные объекты, и поток с целыми пропускными способностями может быть найден целым.

Применения — от транспортной логистики (максимальный грузопоток через сеть дорог) до двудольных паросочетаний. В последнем случае задача о максимальном паросочетании сводится к максимальному потоку добавлением источника, соединённого с левой долей, и стока, соединённого с правой. В паросочетании поток даёт число — размер ответа, а само паросочетание восстанавливается по загруженным рёбрам. Бывают сведения, где ответ — это множество, и его выдаёт не поток, а разрез.

Пример Выбор проектов с зависимостями

Фирма выбирает проекты: у каждого проекта есть прибыль, причём у инфраструктуры она отрицательная — её приходится оплачивать. Зависимость «проект j требует проекта i » означает: выбирая j , вынуждены выбрать и i . Нужно выбрать множество, замкнутое по зависимостям, с максимальной суммарной прибылью.

Сеть строится из задачи, а не подгоняется под неё. Для прибыльного проекта i — дуга $s \rightarrow i$ пропускной способности p_i : пока проект «на стороне источника», прибыль в кассе. Для убыточного — дуга $i \rightarrow t$ пропускной способности $-p_i$: платёж на стороне стока. Каждая зависимость « j требует i » — дуга $j \rightarrow i$ бесконечной пропускной способности.

Бесконечные дуги решают главное: конечный разрез не пересекает их, поэтому сторона источника без самого s — множество A — замкнута по зависимостям: если $j \in A$, то и требуемый для него i остался в A , иначе бесконечная дуга $j \rightarrow i$ пересекала бы границу. Прибыль множества A равна сумме всех положительных прибылей минус пропускная способность разреза $c(A, B)$: прибыльные проекты, ушедшие в B , теряют свой вклад, убыточные в A — добавляют платёж. Максимизировать прибыль — значит минимизировать разрез, и минимальный разрез сам является ответом.

Пусть проекты таковы: a с прибылью -3 ; b с $+8$, требующий a ; c с -2 ; d с $+1$, требующий c ; e с $+4$, требующий b . Наивный выбор всех прибыльных — b, d, e — не замкнут: тянет за собой a и c , итог $-3 + 8 - 2 + 1 + 4 = 8$. Сеть: дуги $s \rightarrow b$ (8), $s \rightarrow d$ (1), $s \rightarrow e$ (4), дуги $a \rightarrow t$ (3), $c \rightarrow t$ (2) и три бесконечные дуги $b \rightarrow a$, $d \rightarrow c$, $e \rightarrow b$.

Минимальный разрез: $A = \{s, a, b, e\}$, $B = \{c, d, t\}$. Из A в B идут $s \rightarrow d$ (1) и $a \rightarrow t$ (3) — пропускная способность 4. Поток той же величины собирается из путей $s \rightarrow b \rightarrow a \rightarrow t$ (3) и $s \rightarrow d \rightarrow c \rightarrow t$ (1) — по теореме max-flow min-cut разрез минимален. Ответ читается по стороне источника: проекты $\{a, b, e\}$ с прибылью $-3 + 8 + 4 = 9$. Перебор всех замкнутых множеств подтверждает максимум, но их может быть экспоненциально много — поток находит ответ за полиномиальное время, а интегральность гарантирует, что это точное множество, а не дробное решение.

Проверка Сетевые потоки и разрезы

1. Почему величина любого потока $s \rightarrow t$ не может превысить пропускную способность разреза?
2. Зачем в остаточной сети нужны обратные рёбра и что потеряет алгоритм Форда–Фалкерсона без них?

§ 9.15 Непересекающиеся пути и надёжность сети

Поток и разрез — два описания одной сети. Поток — величина глобальная: он складывается из поведения всех маршрутов сразу, и ни один маршрут не знает, сколько их всего. Разрез — картина локальная: конечный набор рёбер, отрезающий сток от источника, и именно этот малый набор ставит предел глобальному потоку. Теорема $\max\text{-flow min-cut}$ утверждает, что глобальный максимум и локальный минимум совпадают, и это совпадение — закономерность структуры, а не счастливая случайность одной задачи.

В паре «пути против разделителя» глобальное и локальное меняются местами, но схема сохраняется. Максимальное число непересекающихся путей между двумя узлами — глобальная мера надёжности сети. Минимальный набор вершин или рёбер, способный её разрушить, — локальная. Теорема Менгера утверждает равенство этих двух чисел, а паросочетание и вершинное покрытие дадут третий случай той же схемы.

Сколько независимых маршрутов можно проложить между двумя узлами сети, прежде чем повреждение их разъединит? Ответ — теорема Менгера. У неё две формы: рёберная (маршруты не делят каналы) и вершинная (маршруты не делят узлы). Формулировки и доказательства разные, поэтому разберём их отдельно.

ТЕОРЕМА 9.18 Теорема Менгера: рёберная форма⁶⁵

Для любых двух вершин u, v максимальное число попарно рёберно-непересекающихся путей, соединяющих их, равно минимальному числу рёбер, удаление которых разъединяет u, v . Каждое неориентированное ребро при этом заменяется двумя противоположными дугами.

Доказательство

Докажем сведением к теореме $\max\text{-flow min-cut}$.

Припишем каждому ребру пропускную способность 1 и объявим u источником, v стоком. По интегральности максимальный поток целочислен: единица потока по ребру означает использование этого ребра ровно одним путём. Закон сохранения разлагает поток на пути: идя от u по рёбрам с ненулевым потоком, доходим до v , и разные единицы не делят рёбра. Значит, величине максимального потока соответствует число рёберно-непересекающихся путей.

Минимальный разрез при единичных способностях — минимальный набор рёбер, разъединяющих u, v : каждое ребро разреза имеет пропускную способность 1, и его пропускная способность равна числу рёбер в нём. По теореме $\max\text{-flow min-cut}$ максимальный поток равен минимальному разрезу. Отсюда равенство числа путей и минимального числа разъединяющих рёбер.

⁶⁵K. Menger. «Zur allgemeinen Kurventheorie». Fundamenta Mathematicae, 1927.

ТЕОРЕМА 9.19 Теорема Менгера: вершинная форма

Для несмежных вершин u, v максимальное число попарно вершинно-непересекающихся путей, соединяющих их, — без общих внутренних вершин — равно минимальному числу вершин, удаление которых разъединяет u, v .

Доказательство

Докажем сведением вершинной формы к рёберной: расцепим вершины.

Каждую внутреннюю вершину w заменим парой $w_{\text{in}}, w_{\text{out}}$, соединённых ребром пропускной способности 1. Входящие рёбра перенаправим в w_{in} , исходящие — из w_{out} , исходным рёбрам припишем пропускную способность ∞ (их мы не хотим ограничивать).

Ребро $w_{\text{in}} \rightarrow w_{\text{out}}$ ограничивает пропуск через вершину одной единицей. Поэтому рёберно-непересекающиеся пути в расцеплённом графе соответствуют вершинно-непересекающимся путям в исходном: каждый путь проходит через вершину w ровно один раз, используя её ребро-перемычку. Условие несмежности u, v нужно, чтобы прямое ребро между ними не обходило разделитель: иначе один путь мог бы пройти напрямую, не задев ни одной внутренней вершины. Рёберная форма на расцеплённом графе даёт искомое равенство.

СЛЕДСТВИЕ 9.20 k -связность через пути

Граф является k -связным, если после удаления любых $k - 1$ вершин он остаётся связным. Граф является k -связным тогда и только тогда, когда между любой парой его вершин существуют k попарно вершинно-непересекающихся путей.

Для несмежной пары это прямое следствие вершинной формы Менгера.

Для смежной пары u, v удалим ребро uv : оставшийся граф $(k - 1)$ -связен, по Менгеру в нём есть $k - 1$ непересекающихся путей, и само ребро uv — k -й путь.

Пример Менгер: пути и разделитель

Граф на рис. 49 состоит из двух путей $u \rightarrow a \rightarrow c \rightarrow v, u \rightarrow b \rightarrow d \rightarrow v$ плюс рёбра $a - d, b - c, c - d$. Вершины u, v несмежны — условие вершинной формы выполнено.

Максимальное число попарно вершинно-непересекающихся путей между u, v равно двум: $u \rightarrow a \rightarrow c \rightarrow v, u \rightarrow b \rightarrow d \rightarrow v$. Пути $u \rightarrow a \rightarrow d \rightarrow v, u \rightarrow b \rightarrow c \rightarrow v$ делят внутренние вершины с первыми двумя и в набор не входят. Множество вершин, удаление которых разъединяет u, v , называют **разделителем**. Один из минимальных разделителей здесь — $\{a, b\}, \{c, d\}$ размера 2. Число путей равно размеру разделителя — равенство из теоремы Менгера.

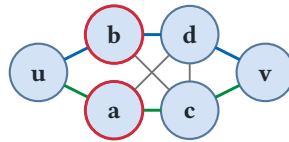


Рис. 49. Два вершинно-непересекающихся пути между u, v .

Число непересекающихся путей между парой вершин — мера отказоустойчивости сети. Если сеть выдерживает k разрывов каналов, не распадаясь, то между любыми двумя её узлами есть не менее $k + 1$ непересекающихся путей.

§ 9.16 Паросочетания и вершинное покрытие

Теорема Холла (раздел о двудольных графах) даёт необходимое и достаточное условие существования паросочетания, покрывающего долю X . Алгоритмически задача о максимальном паросочетании в двудольном графе решается сведением к максимальному потоку.

1. Соединяем источник s с каждой вершиной левой доли ребром пропускной способности 1.
2. Каждую вершину правой доли соединяем со стоком t ребром пропускной способности 1.
3. Каждое ребро исходного графа направляем слева направо с пропускной способностью 1.

Максимальный поток в такой сети даёт максимальное паросочетание.

АЛГОРИТМ Максимальное паросочетание (алгоритм Куна)

1. Для каждой вершины u левой доли:
 - Пытаемся найти увеличивающий чередующийся путь, начинающийся в u , с помощью DFS (или BFS).
 - Чередующийся путь — это путь, в котором рёбра попеременно не принадлежат и принадлежат текущему паросочетанию.
 - Если путь найден — инвертируем принадлежность рёбер на этом пути (не-паросочетанные становятся паросочетанными и наоборот). Размер паросочетания увеличивается на 1.
2. Повторяем для всех вершин левой доли.

Алгоритм Куна работает за $O(VE)$ — для каждой из $|X|$ вершин запускается DFS, просматривающий все рёбра. Этого достаточно для большинства практических задач при умеренных размерах графа.

Сведение к потоку даёт алгоритм и двойственную картину. У каждой комбинаторной задачи о «максимальном» есть двойственная задача о «минимальном», и теорема Кёнига — первая из таких пар.

ОПРЕДЕЛЕНИЕ 9.49 Вершинное покрытие

Вершинным покрытием графа называется множество вершин, содержащее хотя бы один конец каждого ребра.

ОПРЕДЕЛЕНИЕ 9.50 Минимальное вершинное покрытие

Минимальным вершинным покрытием называется покрытие наименьшего возможного размера.

Пример Вершинные покрытия

В двудольном графе на рис. 50 правая доля $\{y_1, y_2, y_3\}$ — вершинное покрытие: каждое ребро заканчивается в правой доле. В треугольнике K_3 двух вершин достаточно, чтобы покрыть все три ребра, а одной нет: ребро между двумя оставшимися осталось бы непокрытым.

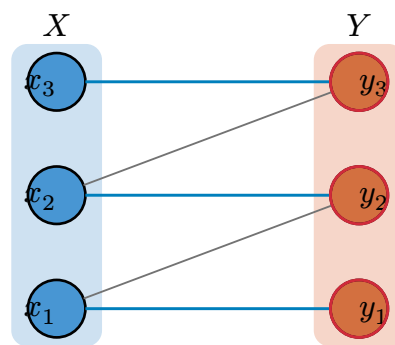


Рис. 50. Двудольный граф с максимальным паросочетанием и минимальным вершинным покрытием.

ТЕОРЕМА 9.21 Теорема Кёнига⁶⁶

В двудольном графе размер максимального паросочетания равен размеру минимального вершинного покрытия:

$$\nu(G) = \tau(G),$$

где $\nu(G)$ — размер максимального паросочетания, а $\tau(G)$ — размер минимального вершинного покрытия.

Набросок доказательства

Докажем равенство сведением к потоку, показав два неравенства.

Сведём паросочетание к потоку, как в разделе о паросочетаниях: источник s на каждую вершину левой доли X , каждая вершина правой доли Y на сток t , рёбра $s \rightarrow x, y \rightarrow t$ пропускной способности 1.

⁶⁶D. König. «Gráfok és matrixok». Matematikai és Fizikai Lapok, 1931.

Средним рёбрам $x \rightarrow y$ припишем пропускную способность $|X| + 1$: она не меняет величину потока (узкое место — единичные рёбра $s \rightarrow x$, поток не превосходит $|X|$) и не даёт минимальному разрезу пересекать эти рёбра.

По интегральности максимальный поток целочислен и соответствует максимальному паросочетанию: $\nu(G) = |f|$.

Первое неравенство $\tau(G) \leq \nu(G)$. Пусть (S, T) — минимальный разрез. Построим из него вершинное покрытие: вершины $x \in X$ вне S , вершины $y \in Y$ внутри S .

Ни одно ребро $x \rightarrow y$ не пересекает разрез: иначе его пропускная способность была бы не меньше $|X| + 1$, тогда как разрез из всех вершин X стоит ровно $|X|$.

Значит, для каждого ребра $x \rightarrow y$ либо $x \in S$, либо $y \in S$ — построенное множество действительно покрывает все рёбра.

Через разрез идут только рёбра $s \rightarrow x$ для x вне S , и $y \rightarrow t$ для y внутри S , каждое пропускной способности 1, — по единице за выбранную вершину. Поэтому пропускная способность разреза равна размеру покрытия: $\tau(G) \leq c(S, T) = |f| = \nu(G)$.

Второе неравенство $\tau(G) \geq \nu(G)$. Обратное неравенство получается простым счётом: рёбра паросочетания не делят концов, и каждому требуется своя вершина в покрытии.

В общих графах равенство не обязано выполняться. В треугольнике K_3 максимальное паросочетание имеет размер 1: любые два ребра делят вершину, так что в паросочетание входит не более одного ребра. А минимальное вершинное покрытие имеет размер 2: двух вершин достаточно, чтобы покрыть все три ребра. Двойственность «паросочетание = покрытие» — особенность двудольных графов. Именно она делает вершинное покрытие полиномиально разрешимым на двудольных графах, тогда как в общих графах задача NP-полна (глава 30).

Замечание Двойственность «max = min»

Max-flow min-cut, теорема Кёнига и теорема Менгера — формулировки одной двойственности: максимум одной величины равен минимуму двойственной. Поток равен разрезу, паросочетание — покрытию, число непересекающихся путей — минимальному сечению. Теорема Холла — критерий, а не равенство: она отвечает на вопрос «все ли вершины X можно покрыть» и тоже выводится из max-flow min-cut. В главе 8 мы уже встречали двойственность как способ описать структуру с двух сторон. Здесь максимум и минимум — два описания одной задачи, связанные равенством. Равенство «max = min» даёт сертификат оптимальности: предъявить минимальное покрытие — значит доказать, что большего паросочетания не существует.

§ 9.17 Планарность

Планарность — геометрическое ограничение с прямыми инженерными следствиями. При проектировании печатных плат проводники не могут пересекаться: пересечение означает короткое замыкание. При проектировании СБИС (сверхбольших интегральных схем) пересечения проводов на одном слое физически невозможны — для этого пришлось бы добавлять дополнительные слои металлизации, что удорожает производство. Даже при визуализации графов пересечения рёбер мешают восприятию: планарная укладка читается быстрее и точнее. Поэтому вопрос «является ли граф планарным» — это вопрос о том, можно ли реализовать схему на одном слое, можно ли развести провода без перемычек и можно ли нарисовать граф без визуального шума.

ОПРЕДЕЛЕНИЕ 9.51 Планарный граф

Граф называется **планарным**, если он может быть изображён на плоскости без пересечений рёбер.

ОПРЕДЕЛЕНИЕ 9.52 Грань

Грани — области, ограниченные рёбрами планарного графа.

Пример Планарные и непланарные графы

- K_4 (полный граф на 4 вершинах) планарен: его можно изобразить как треугольник с вершиной внутри, соединённой со всеми тремя углами, без пересечений.
- Любое дерево планарно — достаточно отводить ветви в стороны.
- K_5 на рис. 51 не планарен: при любой попытке укладки два ребра пересекутся.
- $K_{3,3}$ на рис. 52 также не планарен.
- Рисунок 53 показывает планарную укладку графа.

Замечание Почему планарность — свойство структуры, а не рисунка?

Определение планарности заслуживает пристального взгляда. Граф называется планарным, если *существует* его укладка на плоскости без пересечения рёбер. Слово «существует» — экзистенциальный квантор. Оно ничего не говорит о том, как проверить планарность по внутренним свойствам самого графа. Перед нами неконструктивное определение: мы знаем, что значит «быть планарным», но не знаем, как распознать планарность, не прибегая к перебору упадок.

Теорема Куратовского устраняет этот разрыв. Планарность эквивалентна отсутствию подграфов, гомеоморфных K_5 или $K_{3,3}$. Два запрещённых топологических минора — и вопрос решён. Определение, сформулированное через

внешнее представление (укладку), получает внутреннюю характеристику (запрещённые подструктуры).

Это паттерн: свойство, определённое через существование некоторого представления, оказывается эквивалентным запрету конечного набора конфигураций.

Формула Эйлера связывает число вершин, рёбер и граней любого планарного графа простым соотношением, из которого следуют все основные ограничения на планарность.

ТЕОРЕМА 9.22 Формула Эйлера⁶⁷

Для связного планарного графа:

$$V - E + F = 2,$$

где F — число граней (включая внешнюю).

Набросок доказательства

Индукция по числу рёбер E .

База: $E = 0$. Связный граф без рёбер может состоять только из одной вершины (иначе вершины были бы изолированы друг от друга). Тогда $V = 1, F = 1$ (только внешняя грань). $1 - 0 + 1 = 2$.

База для дерева: Если граф — дерево, то $E = V - 1, F = 1$ (дерево не разделяет плоскость). Тогда $V - (V - 1) + 1 = 2$.

Шаг: Пусть граф не является деревом, то есть содержит цикл. Удалим одно ребро из цикла. Число вершин не изменилось, число рёбер уменьшилось на 1, а две грани, разделённые удалённым ребром, слились в одну — число граней также уменьшилось на 1. Значение $V - E + F$ не изменилось. Продолжаем, пока граф не станет деревом, для которого формула уже проверена.

Та же формула связывает вершины, рёбра и грани любого выпуклого многогранника. Для куба: $8 - 12 + 6 = 2$. Для тетраэдра: $4 - 6 + 4 = 2$. Для додекаэдра: $20 - 30 + 12 = 2$. Эйлер опубликовал эту формулу в 1758 году, ещё до того, как понятие графа было формализовано, — он мыслил в терминах многогранников. Одно уравнение для многогранников и плоских графов — следствие топологической эквивалентности: плоскость, на которую проецируется планарный граф, гомеоморфна сфере без одной точки (стереографическая проекция).

⁶⁷L. Euler. «Elementa doctrinae solidorum» (Элементы учения о телах). 1758. Доказательство для многогранников. Позже обобщено на планарные графы.

СЛЕДСТВИЕ 9.23 Ограничение на число рёбер

Для простого планарного графа с $V \geq 3$:

$$E \leq 3V - 6.$$

Следствие: любой простой планарный граф имеет вершину степени ≤ 5 .

Доказательство

Пусть G — простой планарный граф с $V \geq 3$. Каждая грань ограничена не менее чем тремя рёбрами (в простом графе нет петель и кратных рёбер, поэтому грань не может быть ограничена одним или двумя рёбрами). При подсчёте суммы длин границ всех граней каждое ребро учитывается не более двух раз (оно разделяет не более двух граней). Следовательно, $3F \leq 2E$, откуда $F \leq 2\frac{E}{3}$.

Подставляем в формулу Эйлера $V - E + F = 2$: $V - E + 2\frac{E}{3} \geq 2$, откуда $V - \frac{E}{3} \geq 2$, и $E \leq 3V - 6$.

Для следствия: сумма степеней вершин равна $2E \leq 6V - 12$. Если бы все вершины имели степень ≥ 6 , сумма была бы $\geq 6V > 6V - 12$ — противоречие. Значит, существует вершина степени ≤ 5 .

Эти оценки — необходимые условия планарности:

- K_5 нарушает $E \leq 3V - 6$ ($E = 10, 3 \cdot 5 - 6 = 9$) и потому не планарен.
- $K_{3,3}$ этому неравенству удовлетворяет, но не планарен по другой причине: в графе без треугольников $E \leq 2V - 4$, а $K_{3,3}$ имеет $E = 9 > 2 \cdot 6 - 4 = 8$.

Это тоже следствие формулы Эйлера: в графе без треугольников каждая грань ограничена не менее чем четырьмя рёбрами, что даёт $E \leq 2V - 4$.

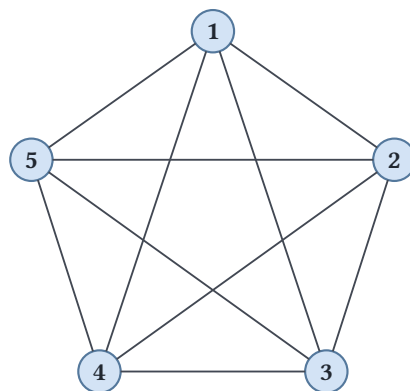


Рис. 51. Полный граф K_5 .

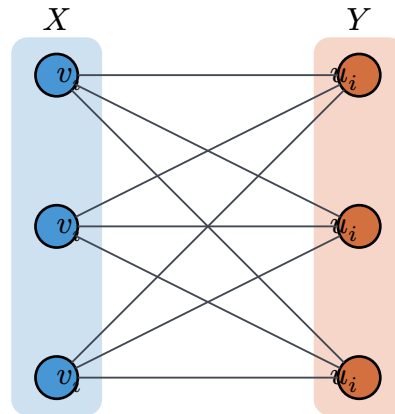
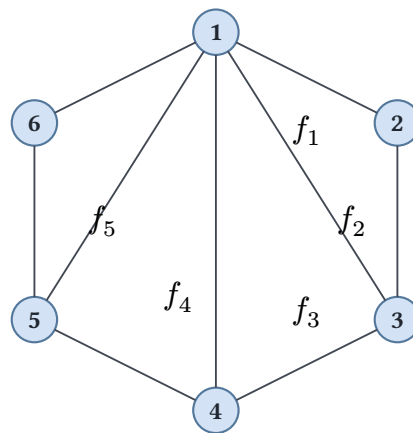
Рис. 52. Полный двудольный граф $K_{3,3}$.

Рис. 53. Планарный граф.

Ограничения на число рёбер не дают достаточного условия: $K_{3,3}$ удовлетворяет $E \leq 3V - 6$, но не планарен. Необходимое и достаточное условие даёт теорема Куратовского — через запрещённые подструктуры.

ОПРЕДЕЛЕНИЕ 9.53 Подразбиение ребра

Подразбиением ребра $\{u, v\}$ называется замена его на путь $u - w - v$ добавлением новой вершины w степени 2.

ОПРЕДЕЛЕНИЕ 9.54 Подразбиение графа

Граф H является **подразбиением** графа G , если H можно получить из G последовательным подразбиением рёбер (возможно, нулевым числом шагов).

Графы, получающиеся друг из друга подразбиением рёбер, называют гомеоморфными.

Интуитивно: если нельзя нарисовать K_5 или $K_{3,3}$ на плоскости, то добавить лишние вершины на рёбра этих графов и обойти запрет тоже не получится.

ТЕОРЕМА 9.24 Теорема Куратовского⁶⁸

Граф планарен тогда и только тогда, когда он не содержит подграфа, гомеоморфного K_5 или $K_{3,3}$ (то есть полученного из K_5 , $K_{3,3}$ подразбиением рёбер).

Набросок доказательства

Доказательство опирается на тонкую технику вершинной связности и выходит за рамки главы. Полное доказательство — в оригинальной статье Куратовского (1930).

Пример Подразбиения

Разобьём одно ребро K_5 : возьмём ребро $\{1, 2\}$ и заменим его на путь $1 - x - 2$, где x — новая вершина степени 2.

Полученный граф с шестью вершинами — подразбиение K_5 и, по теореме Куратовского, не планарен. Аналогично, замена нескольких рёбер $K_{3,3}$ на пути с промежуточными вершинами даёт подразбиение $K_{3,3}$ — тоже непланарный граф.

§ 9.18 Раскраска графов

История раскраски начинается с карт: сколько красок нужно, чтобы любые две граничащие страны получили разные цвета? Утверждение о четырёх красках можно отдать ребёнку вместе с карандашами: раскрась карту мира так, чтобы соседние страны различались. Карта мира, карта штатов, карта регионов любой страны — всё это красится четырьмя цветами, и после десятой попытки начинаешь подозревать, что иначе и быть не может. Тогда просыпается азарт изобретателя: хочется *изобрести* карту, для которой четырёх цветов не хватит. Именно так теорема и сопротивлялась математикам более века: придумывая всё более вычурные карты, никто не находил ни одной, требующей пятого цвета, — но и доказать невозможность не мог.

В терминах графов страны — вершины, общая граница — ребро, а минимальное число цветов — хроматическое число. Чтобы увидеть за картой граф, не нужно раскрашивать целые страны: поставь точку в центре каждой области и соедини линией области с общей границей. Карта превращается в сеть, и вопрос «хватит ли четырёх красок?» становится вопросом о раскраске вершин этой сети. Сведение не теряет сути: две карты, нарисованные по-разному, могут дать одну и ту же сеть, — значит, изучая сети, мы изучаем все карты сразу. Не всякая сеть получается из карты: если линии вынужденно пересекаются, как в K_5 , такую карту не нарисовать.

⁶⁸K. Kuratowski. «Sur le problème des courbes gauches en topologie». Fundamenta Mathematicae, 1930.

ОПРЕДЕЛЕНИЕ 9.55 Правильная k -раскраска

Раскраска вершин графа называется **правильной k -раскраской** (*proper k -coloring*), если смежные вершины получают разные цвета.

ОПРЕДЕЛЕНИЕ 9.56 Хроматическое число

Хроматическим числом графа G называется минимальное k , для которого существует правильная k -раскраска, обозначается

$$\chi(G).$$

Пример Хроматические числа

$\chi(C_4) = 2$ (цикл чётной длины, двудолен). $\chi(C_5) = 3$ (цикл нечётной длины, не двудолен). $\chi(K_n) = n$ (в полном графе все вершины попарно смежны, каждой нужен свой цвет). $\chi(\text{дерева}) = 2$ для любого дерева с $n \geq 2$ (все деревья двудольны). Рисунок 54 показывает правильную раскраску графа.

Двудольные графы (с хотя бы одним ребром) имеют хроматическое число $\chi(G) = 2$, полные — $\chi(K_n) = n$: это две крайности раскраски. Простейшая верхняя оценка получается жадным алгоритмом: красим вершины в фиксированном порядке, каждая — в наименьший цвет, свободный у её уже раскрашенных соседей.

АЛГОРИТМ Жадная раскраска

1. Зафиксировать порядок вершин $v_1, \dots, v_n$.
2. Идя по порядку, красить каждую вершину в наименьший цвет, отсутствующий у её уже раскрашенных соседей.

УТВЕРЖДЕНИЕ 9.25 Жадная оценка

Жадная раскраска использует не более $\Delta(G) + 1$ цветов, поэтому $\chi(G) \leq \Delta(G) + 1$, где $\Delta(G)$ — максимальная степень вершины.

Доказательство

Докажем подсчётом цветов у одной вершины. Когда очередь доходит до вершины v , среди её соседей раскрашено не более $\deg(v) \leq \Delta(G)$, и они заняли не более $\Delta(G)$ цветов. Из палитры в $\Delta(G) + 1$ цветов хотя бы один свободен, поэтому алгоритм не застревает ни на одной вершине и завершается с не более чем $\Delta(G) + 1$ цветами.

Оценка точна на полных графах: $\Delta(K_n) = n - 1$ и $\chi(K_n) = n$. Точна она и на циклах нечётной длины: $\Delta(C_5) = 2$ и $\chi(C_5) = 3$. Для большинства графов жадный алгоритм тратит меньше цветов, чем позволяет оценка, но может потратить и

больше необходимого: результат зависит от порядка вершин, а точное значение $\chi(G)$ находить NP-трудно.

Общая оценка через максимальную степень оставляет значительный зазор. Для планарных графов верно более точное утверждение.

ТЕОРЕМА 9.26 Теорема о четырёх красках

Любой планарный граф 4-раскрашиваем. Доказательство (Аппель и Хакен, 1976) — первое крупное математическое доказательство с применением компьютера: 1936 конфигураций проверены программно. В 2005 году Гонтье формализовал доказательство в Coq. Доказательство здесь не приводится: оно компьютерное и выходит за рамки главы.

Компьютерная природа доказательства вызвала спор: обязана ли математическая истина быть проверяемой человеком? Спор продолжается до сих пор — он остаётся редким случаем, когда событием стал метод доказательства. Трудность четырёх красок — в доказательстве. Версия в шесть цветов доказывается в несколько строк: взять самую маленькую карту-контрпример, удалить страну, перекрасить оставшееся и вернуть страну обратно. Версия в пять цветов требует уже тонкого рассуждения, а на четырёх эта схема вовсе рассыпается. Именно поэтому компьютерное доказательство Аппеля и Хакена стало событием: рукотворных проверок не хватает, перебор слишком велик для человека.

Теорема о четырёх красках относится к раскраске вершин. Аналог для рёберной раскраски даёт теорема Визинга. Рёберное хроматическое число $\chi'(G)$ — минимум цветов для правильной раскраски рёбер: рёбра с общей вершиной получают разные цвета.

ТЕОРЕМА 9.27 Теорема Визинга⁶⁹

Для рёберной раскраски:

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1.$$

Набросок доказательства

Нижняя оценка получается прямо из определения: рёбра, инцидентные вершине максимальной степени, требуют различных цветов.

Для верхней: индукция по числу рёбер. Удаляем ребро $e = \{u, v\}$, раскрашиваем оставшийся граф $\Delta(G) + 1$ цветами. Если у u, v имеется общий свободный цвет — красим e в него.

⁶⁹V. G. Vizing. «On an estimate of the chromatic class of a p-graph». Diskretnyi Analiz, 1964 (на русском: В. Г. Визинг, «Об оценке хроматического класса р-графа»).

Иначе строим чередующийся путь из цветов, отсутствующих у u, v , и перекрашиваем рёбра вдоль этого пути, освобождая общий цвет для e .

Пример Рёберная раскраска: обе границы Визинга

Теорема Визинга утверждает, что $\chi'(G)$ равно $\Delta(G)$ или $\Delta(G) + 1$, и обе границы встречаются. Цикл C_4 имеет $\Delta = 2$, и двух цветов достаточно: рёбра $(1, 2)$ и $(3, 4)$ красим в цвет a , рёбра $(2, 3)$ и $(4, 1)$ — в цвет b . У цикла C_5 тоже $\Delta = 2$, но двух цветов не хватает: чередуя цвета по циклу, мы дойдём до последнего ребра $(5, 1)$, которому нужен цвет, отличный и от цвета $(4, 5)$, и от цвета $(1, 2)$, а эти рёбра уже разных цветов.

Итак, $\chi'(C_4) = 2 = \Delta$ и $\chi'(C_5) = 3 = \Delta + 1$. Оба случая теоремы реально возникают.

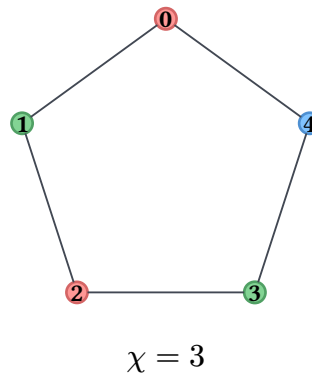


Рис. 54. Цикл C_5 с правильной 3-раскраской.

Раскраска графа решает задачу распределения регистров в компиляторе. Переменные, используемые одновременно, не могут занимать один регистр — конфликт. Граф интерференции регистров: вершины — переменные программы. Ребро между u, v , если u, v «живы» в одной точке программы (их время жизни пересекается).

Задача: назначить k доступных регистров вершинам графа так, чтобы смежные вершины получили разные регистры. Это в точности задача k -раскраски — NP-трудная в общем случае. На практике компиляторы используют эвристики. Алгоритм Чайтина (1982)⁷⁰ работает итеративно:

1. Удаляются вершины со степенью $< k$ и складываются в стек.
2. Когда удалять больше нечего, вершины раскрашиваются в обратном порядке (извлекая из стека).
3. Если на каком-то шаге все оставшиеся вершины имеют степень $\geq k$, одна из них «вытесняется» в память (spill) — для неё резервируется ячейка стека вместо регистра, и процесс повторяется заново.

⁷⁰G. Chaitin et al. «Register allocation via coloring». Computer Languages, 1981. G. Chaitin. «Register allocation & spilling via graph coloring». SIGPLAN Notices, 1982.

Та же модель — назначение радиочастот: вершины — передатчики, ребро — недопустимые помехи (географически близкие или работающие на соседних частотах), цвета — доступные частотные каналы. Та же задача возникает при составлении расписания экзаменов: вершины — экзамены, ребро — наличие общих студентов, цвета — временные слоты.

Проверка Раскраска графов

1. Почему жадная раскраска всегда укладывается в $\Delta(G) + 1$ цветов и на каких графах эта оценка достигается?
2. Что утверждает теорема Визинга и чем рёберная раскраска отличается от вершинной?
3. Как распределение регистров в компиляторе превращается в раскраску графа?

§ 9.19 Применения графов

Графы — рабочий инструмент дискретной математики и computer science. Рассмотрим несколько приложений.

9.19.1 Социальные сети

Социальный граф: вершины — люди, рёбра — дружеские связи, подписки или взаимодействия.

Центральность измеряет «влиятельность» вершины. Степень центральности (degree centrality) — количество связей: сколько людей знает данный человек. Промежуточность (betweenness centrality) вычисляется так: для каждой пары вершин (s, t) находятся все кратчайшие пути между ними. Центральность вершины v — доля этих путей, проходящих через v . Вершина с высокой промежуточностью — «мост» между сообществами, контролирующий поток информации.

Выделение сообществ (community detection): задача разбить граф на плотные подграфы с редкими связями между ними. Алгоритм Гирван–Ньюмана (2002) итеративно удаляет рёбра с наибольшей промежуточностью, постепенно разбивая граф на компоненты-сообщества. Каждое удаление увеличивает число компонент связности, и на каждом шаге пересчитывается промежуточность оставшихся рёбер. Альтернативный подход — максимизация модулярности (modularity): меры того, насколько плотность рёбер внутри предполагаемых сообществ превышает ожидаемую плотность в случайном графе с теми же степенями вершин.⁷¹

⁷¹M. Girvan, M. E. Newman. «Community structure in social and biological networks». PNAS, 2002. M. E. J. Newman. «Networks: An Introduction». 2010.

9.19.2 Маршрутизация в сетях

Интернет: маршрутизаторы — вершины, каналы связи — рёбра с весами (задержка или стоимость соединения).

Протокол OSPF (Open Shortest Path First) решает задачу внутридоменной маршрутизации: каждый маршрутизатор вычисляет кратчайшие пути до всех остальных в своей автономной системе.⁷² Механизм: каждый маршрутизатор хранит полную карту сети (link-state database) — взвешенный граф своей области. При изменении топологии (отказ канала, добавление маршрутизатора) обновлённая информация рассылается лавинно (flooding) всем участникам. Затем каждый маршрутизатор независимо запускает алгоритм Дейкстры.

Поддерживается множество посещённых вершин и min-heap граничных расстояний. На каждом шаге извлекается ближайшая непосещённая вершина, и расстояния до её соседей обновляются, если найден более короткий путь. Результат — дерево кратчайших путей от данного маршрутизатора до всех остальных. Пакеты направляются по следующему шагу этого дерева.⁷³

Междоменная маршрутизация (BGP — Border Gateway Protocol) принципиально иная. Автономные системы обмениваются информацией о доступных префиксах с учётом политик (какие пути предпочтительны по коммерческим соглашениям), а не только расстояний. BGP образует граф автономных систем, где поиск пути может быть осложнён противоречивыми политиками.

9.19.3 Разработка компиляторов

Компилятор использует несколько графовых представлений программы на разных этапах трансляции.

Граф потока управления (CFG): вершины — базовые блоки (последовательности инструкций без ветвлений), рёбра — возможные переходы (условные и безусловные ветвления, рёбра циклов). CFG строится на этапе промежуточного представления (IR) и служит основой для оптимизаций:

- Удаление недостижимого кода.
- Вынесение инвариантных вычислений из циклов.
- анализ доминирования: вершина u доминирует над v , если любой путь от входа к v проходит через u (дерево доминаторов строится за почти линейное время алгоритмом Ленгауэра–Тарьяна).

Граф вызовов (call graph): вершины — функции, ребро $f \rightarrow g$, если f вызывает функцию g . Используется для обнаружения рекурсии, вычисления побочных эффектов и оптимизации встраиванием (inlining): если функция вызывается из единственного места и достаточно мала, её тело подставляется в точку вызова, экономя накладные расходы на вызов.

⁷²J. Moy. «OSPF Version 2». RFC 2328, 1998.

⁷³E. W. Dijkstra. «A note on two problems in connexion with graphs». Numerische Mathematik, 1959.

Распределение регистров сводится к раскраске графа интерференции: вершины — переменные, «живые» одновременно. k регистров — k цветов. NP-трудная задача решается эвристиками (алгоритм Чайтина с вытеснением, описанный в разделе о раскраске графов).⁷⁴

9.19.4 PageRank и поисковые системы

Всемирная паутина — ориентированный граф: страницы — вершины, гиперссылки — дуги.

Алгоритм PageRank (Брин и Пейдж, 1998) моделирует поведение случайного пользователя. С вероятностью d (обычно 0.85) он переходит по случайной исходящей ссылке текущей страницы. С вероятностью $1 - d$ — «телепортируется» на случайную страницу из всей сети. Фактор телепортации предотвращает застревание в страницах без исходящих ссылок.

Формально, вектор PageRank r удовлетворяет системе линейных уравнений:

$$r(i) = \frac{1 - d}{N} + d \cdot \sum_{j \rightarrow i} \frac{r(j)}{\text{out}(j)},$$

где $\text{out}(j)$ — количество исходящих ссылок со страницы j , а N — общее число страниц. Это уравнение баланса марковской цепи: стационарное распределение случайного блуждания (*random walk*).

Вычисление — итерационный процесс (метод степеней, *power iteration*).

1. Начальное приближение: $r_0(i) = \frac{1}{N}$.
2. На каждом шаге t применяется переход по формуле выше, порождая r_{t+1} .
3. Итерации продолжаются, пока изменение $\|r_{t+1} - r_t\|$ не станет меньше заданного порога.

Метод сходится к единственному решению, потому что фактор телепортации делает марковскую цепь эргодичной (из любого состояния достижимо любое).

На графе из миллиардов вершин это соответствует распределённому умножению разреженной матрицы на вектор — основная вычислительная нагрузка поисковых систем.⁷⁵

§ 9.20 * Случайные графы

До сих пор мы строили графы вручную: выбирали вершины, проводили рёбра, проверяли свойства. Но многие графы в природе никто не проектировал. Социальная

⁷⁴A. Aho, M. Lam, R. Sethi, J. Ullman. «Compilers: Principles, Techniques, and Tools». 2-е изд., 2006.

⁷⁵S. Brin, L. Page. «The anatomy of a large-scale hypertextual Web search engine». Computer Networks, 1998.

сеть, интернет, пищевая цепь — никто не рисовал их матрицу смежности: эти графы *возникли*.

Изучать такие графы перебором вершин и рёбер бессмысленно: завтра сеть изменится, послезавтра изменится снова, поэтому нужен другой подход. Один из ответов — считать граф случайным: мы не знаем, кто с кем дружит, но можем оценить *вероятность* дружбы. Так устроена модель Эрдёша–Реньи $G(n, p)$.

В модели $G(n, p)$ на n помеченных вершинах каждое из $\frac{n(n-1)}{2}$ возможных рёбер появляется независимо с вероятностью p . При $p = 0$ граф пуст, при $p = 1$ — полон, а между крайностями свойства возникают и исчезают скачками.

Главный феномен модели — **порог связности**. Если $p = c \frac{\ln n}{n}$ с константой $c < 1$, то при $n \rightarrow \infty$ граф почти наверное несвязен: в нём остаются изолированные вершины.

Если $c > 1$, то граф почти наверное связан. Переход происходит в узкой окрестности $p = \frac{\ln n}{n}$: вероятность связности меняется от почти нуля до почти единицы — так вероятность управляет рождением связности. Полное доказательство — в главе 20.

За случайным графом стоит метод, выходящий далеко за пределы теории графов. Эрдёш ввёл *вероятностный метод*: чтобы доказать существование объекта с заданным свойством, достаточно показать, что случайный объект обладает этим свойством с положительной вероятностью. Не нужно строить объект явно — достаточно доказать, что вероятность его существования не равна нулю. Этим методом доказано существование графов с одновременно большим обхватом и большим хроматическим числом. Вероятностный метод — это *доказательство существования без построения*.

§ 9.21 Итоги главы

Граф — это рисунок, на котором значение имеет только связность: сдвигая станции метро и выпрямляя линии, мы ничего не теряем, потому что граф хранит лишь то, кто с кем соединён. Форма — то, как устроена связность целого, — здесь и есть главное, и на этом языке устройство измеряется степенью, расстоянием, деревьями и циклами. Локальные свойства начинают определять глобальные: чётность степеней решает существование эйлера цикла (критерий Эйлера), а отсутствие циклов нечётной длины — двудольность.

Два счётных приёма проходят через всю главу. Двойной счёт — когда одна величина подсчитана двумя способами — даёт лемму о рукопожатиях. Двойственность $\max = \min$ связывает поток с разрезом, паросочетание с вершинным покрытием, а теорема Менгера добавляет к ним непересекающиеся пути и сечения. Счётным балансом служит и формула Эйлера $V - E + F = 2$, из которой видно, что K_5 не планарен.

Храним граф матрицей или списками смежности — выбор представления определяет скорость всех алгоритмов. Поиск в ширину и в глубину дают два порядка обхода,

топологическая сортировка выстраивает зависимости в порядок сборки, а дерево всегда несёт ровно $n - 1$ ребро. Жадные алгоритмы Крускала и Прима решают задачу о минимальном остовном дереве, по-разному применяя один принцип — никогда не выбирать ребро, которое можно заменить более дешёвым, — а Дейкстра находит кратчайшие пути при неотрицательных весах.

Одна замена в формулировке меняет природу задачи: эйлеров цикл обходит каждое ребро, гамильтонов — каждую вершину, и от линейного алгоритма мы перескакиваем к NP-полноте. Двудольность, планарность и раскраска описываются локальными запретами и приводят к теореме о четырёх красках, а случайные графы дают вероятностный метод Эрдёша — доказательство существования без построения.

Диаграмма Хассе из главы 8 — ориентированный граф без циклов, булева алгебра из главы 10 — структура, изоморфная гиперкубу. Язык графов позволил смоделировать любую сеть и отличить полиномиально разрешимые задачи от NP-трудных. Следующий шаг — алгебра нулей и единиц, на которой строятся сами вычисления.

Проверка Проверьте себя

1. В чём разница между эйлеровым и гамильтоновым циклом и почему один проверяется быстро, а другой — нет?
2. Почему дерево на n вершинах имеет ровно $n - 1$ ребро? Что утверждает лемма о рукопожатиях и как из неё следует, что число вершин нечётной степени чётно?
3. Почему алгоритм Дейкстры требует неотрицательных весов? Чем минимальное остовное дерево отличается от дерева кратчайших путей из одного источника и почему Крускал и Прим, применяя жадность по-разному, возвращают остов минимального веса?
4. Чем BFS отличается от DFS: какой порядок обхода даёт каждый и для каких задач служит?
5. Почему граф двудольен тогда и только тогда, когда в нём нет циклов нечётной длины?
6. Что даёт формула Эйлера $V - E + F = 2$, как с её помощью убедиться, что K_5 не планарен, и какие пары задач связывает двойственность «max = min»?

Что дальше?

От графов — структур связей и их оптимизации — мы переходим к алгебре нулей и единиц. В главе 10 мы изучим булевы функции, критерий функциональной полноты Поста, нормальные формы и методы минимизации — математический фундамент цифровой схемотехники.

§ 9.22 Упражнения

Основные определения, степени и связность.

1. Сколько рёбер в полном графе K_n ? Для графа с вершинами $\{a, b, c, d\}$ и рёбрами $\{ab, ac, ad, bc, cd\}$ вычислите степень каждой вершины и проверьте лемму о рукопожатиях: сумма степеней равна $2|E|$.

2. * Докажите, что в любом графе число вершин нечётной степени чётно.

Деревья.

3. Нарисуйте все неизоморфные деревья с 5 вершинами. Сколько их? Затем: лес состоит из k деревьев на n вершинах — сколько в нём рёбер? Докажите формулу.
4. * Проверьте формулу Кэли для $n = 4$: перечислите все помеченные деревья на множестве вершин $\{1, 2, 3, 4\}$ и убедитесь, что их ровно $4^{4-2} = 16$.
5. * Утверждение: любое дерево с $n \geq 2$ вершинами имеет простой путь, проходящий через все вершины. «Доказательство» индукцией по n : база $n = 2$ — ребро соединяет обе вершины.

Шаг: пусть верно для n . Возьмём дерево с $n + 1$ вершиной и удалим лист v — останется дерево с n вершинами, в котором по предположению есть простой путь через все вершины.

Вершина v соединена с некоторой вершиной этого пути, поэтому допишем v в конец пути — путь через все $n + 1$ вершин готов. Утверждение ложно (приведите контрпример) — найдите ошибку в этом рассуждении.

Минимальное остовное дерево.

6. Вычислите минимальное остовное дерево графа на рис. 36 алгоритмом Крускала и алгоритмом Прима (старт из вершины 2). Проверьте, что оба дают один и тот же суммарный вес. Как изменится ответ, если вес ребра 2 — 5 уменьшить с 8 до 5?
7. * Докажите, что если все веса рёбер различны, то минимальное остовное дерево единственно. Подсказка: если бы было два, рассмотрите ребро минимального веса, входящее ровно в одно из них, и повторите обменный аргумент. Тем же обменным аргументом объясните корректность Прима: что даёт шаг «добавить лёгчайшее ребро на разрезе» и чем его разрез отличается от разреза в доказательстве Крускала?

Обходы.

8. Примените BFS к графу: вершины $\{a, b, c, d, e, f\}$, рёбра $ab, ac, bd, be, ce, cf, df$. Начните с a . Укажите расстояния от a до всех вершин и дерево кратчайших путей.
9. Оргграф: вершины $\{1, 2, 3, 4, 5, 6\}$, дуги $1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 6, 6 \rightarrow 4$. Выполните алгоритм Кана и убедитесь, что он выводит не все вершины: объясните, что это говорит о графе. Найдите компоненты сильной связности алгоритмом Косарайю, постройте конденсацию и её топологический порядок.
10. Приведите пример графа, который эйлеров, но не гамильтонов.

11. * Проверьте условие Дирака для графа Петерсена на рис. 30 и докажите, что он не гамильтонов.
12. * Объясните, почему алгоритм Дейкстры не работает при отрицательных весах рёбер. Приведите конкретный контрпример (граф и веса), на котором Дейкстра выдаёт неверный ответ.
13. Проследите алгоритм Дейкстры на графе: вершины $\{s, a, b, c, d\}$, рёбра $s - a$ (4), $s - b$ (1), $b - a$ (2), $a - c$ (1), $b - c$ (5), $c - d$ (3), $a - d$ (8). Сведите состояния в таблицу: после каждого извлечения из кучи записывайте текущие расстояния до всех вершин и отмечайте устаревшие записи. Восстановите по предшественникам кратчайший путь $s \rightarrow d$.

Двудольные графы, планарность и раскраска.

14. Какие из графов C_3, C_4, C_5 , звезды (центр и 3 листа), K_4 двудольны? Для двудольных укажите разбиение на доли, для остальных найдите хроматическое число и приведите минимальную раскраску.
15. Граф — путь $1 - 2 - 3 - 4 - 5$. Выполните жадную раскраску в порядке вершин 1, 2, 4, 5, 3 и покажите, что алгоритм использует три цвета — ровно оценку $\Delta + 1$ — хотя $\chi = 2$. Затем раскрасьте тот же путь в порядке 3, 2, 4, 1, 5 и объясните, чем определяется число цветов жадного алгоритма.
16. * Докажите, что K_5 не планарен, используя неравенство $E \leq 3V - 6$. Почему для $K_{3,3}$ этого неравенства недостаточно и нужна усиленная оценка $E \leq 2V - 4$?

Потоки и паросочетания.

17. Найдите максимальный поток в сети на рис. 44 и проверьте, что его величина равна пропускной способности минимального разреза. Затем найдите максимальное паросочетание и минимальное вершинное покрытие в полном двудольном графе $K_{3,3}$ и убедитесь, что их размеры равны (теорема Кёнига).
18. В примере Менгера на рис. 49 перечислите рёберно-непересекающиеся пути между u, v и найдите минимальный разрез.
19. * Объясните, почему сведение паросочетания к потоку с целыми пропускными способностями даёт именно паросочетание, а не дробный объект (интегральность потока).
20. * «Доказательство» того, что любой граф двудольен, индукцией по числу вершин. База: граф с одной вершиной — красим её в цвет 1, вторая доля пуста. Шаг: для графа с $n + 1$ вершиной удалим вершину v . Остаток по предположению двудольен, с долями A и B . Если все соседи v лежат в A , отнесём v к B . Если все в B — к A . Все рёбра вновь идут между долями, индукционный шаг завершён. Утверждение ложно (приведите контрпример) — найдите ошибку.

ПРОЕКТ Планировщик сборки

Система сборки читает граф зависимостей модулей: порядок сборки даёт топологическая сортировка, цикл зависимостей — ошибка проектирования, и обнаруживает её поиск компонент сильной связности. Соберите планировщик, который по списку модулей, их зависимостям и временам компиляции выдаёт план сборки, а при циклических зависимостях — точную диагностику, какой набор модулей зациклен.

① Модель графа

Представьте зависимости оргграфом: вершина — модуль, дуга $u \rightarrow v$ означает «сборка v требует собранного u ». Храните списками смежности и загрузите модель из раздела о топологической сортировке: модули `core`, `utils`, `app`, `tests` с их зависимостями.

② Топологическая сортировка

Реализуйте алгоритм Кана: очередь вершин нулевой входящей степени, вывод очередного модуля, уменьшение входящих степеней потребителей. Сверьте результат: каждая дуга ведёт от более раннего модуля к более позднему.

③ Циклы зависимостей

Если очередь опустела, а вершины остались, найдите компоненты сильной связности — алгоритмом Косараяу или однопроходным алгоритмом Тарьяна. Компонента из двух и более вершин — цикл зависимостей: сообщите имена модулей и постройте конденсацию, показав, какие группы блокируют друг друга.

④ Критический путь

Припишите каждому модулю время компиляции и найдите в DAG самый длинный путь — минимальное время сборки при неограниченном параллелизме. Динамика по топологическому порядку: ранний финиш модуля равен его времени компиляции плюс максимум ранних финишей его зависимостей. Восстановите сам путь — набор модулей, задержка любого из которых отодвигает конец сборки.

⑤ Отчёт

Выведите для входного графа порядок сборки либо список циклов с конденсацией, минимальное время сборки и критический путь. Проверьте планировщик на графе с циклом и на сгенерированном DAG из нескольких десятков модулей со случайными зависимостями и временами.

Итог: планировщик, который по графу зависимостей выдаёт корректный порядок сборки, диагностирует циклы через компоненты сильной связности и вычисляет критический путь с временами раннего старта модулей.

Х

Булева алгебра

Из двух операций и одного отрицания порождается 2^{2^n} функций от n переменных. При $n = 6$ это около $1.8 \cdot 10^{19}$ — больше, чем зёрен песка на Земле. Той же скудной тройки хватает, чтобы собрать любую цифровую схему, от сумматора до процессора. Уберите отрицание — и выразимость рушится: И и ИЛИ порождают только монотонные функции, даже НЕ из них не собрать. Добавление операции к полному набору ничего не меняет: полнота избыточна. Где граница между бесполезным и полным набором операций, и почему она такая резкая?

Условие в коде `if (has_access and not is_banned)` — не сокращённая запись рассуждения. Процессор вычисляет его как алгебраическое выражение: каждое И, каждое НЕ превращается в операцию над нулями и единицами. То же происходит в поисковом запросе «кофе И без кофеина», в побитовой маске `flags & MASK`, в проверке прав доступа. Логика, которой человек рассуждает, в машине работает как арифметика.

Булева алгебра извлекает алгебраическую структуру из логических операций. Конъюнкция становится умножением, дизъюнкция — сложением, отрицание — дополнением. С истиной и ложью можно работать как с числами: упрощать выражения, приводить к нормальным формам, проверять тождества подстановкой.

Любое условие в коде — булева функция. Побитовые `&`, `|`, `^`, `~` в C, Rust и Go — это булева алгебра, применённая к каждому биту отдельно.

В предыдущей главе (9) графы описывали связи между объектами: вершины и рёбра, пути и циклы, деревья и сети. Граф отвечает на вопрос, что с чем соединено. Множества, отношения, функции, порядки, графы — так был собран язык описания дискретных структур. Теперь на этом языке начинают строить: логика становится алгеброй. Логические связки из главы 1 и дистрибутивные решётки с дополнениями из главы 8 сходятся здесь в одной структуре. Булева алгебра — это дистрибутивная решётка с дополнениями, только на языке логики.

Первый шаг в этой вселенной сделал Джордж Буль. В 1847 году он опубликовал «The Mathematical Analysis of Logic», а в 1854-м — «The Laws of Thought»: рассуждение впервые удалось записать как вычисление. Восемьдесят лет спустя, в 1937 году, Клод Шеннон показал, что та же алгебра описывает релейные схемы: замкнутая цепь — истина, разомкнутая — ложь. Логика, которая казалась отвлечённой, легла в основание цифровой схемотехники. А Эмиль Пост в 1941 году ответил на вопрос, который

Буль оставил открытым: какие наборы операций достаточны, чтобы выразить все остальные.

Чтобы организовать вселенную функций, нужны представления. По таблице истинности строятся ДНФ и КНФ, и вопрос о том, какая форма компактнее, приводит к минимизации. Карта Карно сжимает формулу до минимума на глаз, алгоритм Квайна — МакКласки делает то же самое алгоритмически. Полином Жегалкина переводит функцию на язык XOR-сумм, а диаграммы решений (BDD) — на язык графов. Критерий Поста отвечает на вопрос о *функциональной полноте*: достаточно ли данного набора операций для выражения всех функций.

У алгебры есть прямая инженерная отдача. Раз одного NAND достаточно для любой схемы, производителю не нужно выпускать много типов деталей. В главе 11 булева функция станет физической: из этих операций соберутся вентили и схемы. А глава 14 покажет, что та же алгебра лежит в основании теории сложности.

За простотой двух операций стоит сдвиг в отношении к логике. Логика перестаёт быть способом рассуждать о высказываниях и становится *объектом*, с которым можно работать: упрощать, минимизировать, раскладывать по базисам. То, что начиналось как язык описания, оказывается языком вычисления.

ИСТОРИЯ От философской логики к электрическим схемам

Джордж Буль в «The Laws of Thought»⁷⁶ записал логические законы как алгебраические уравнения, которые можно решать и преобразовывать.

Буль был самоучкой: без формального университетского образования он овладел математикой по книгам и публиковал оригинальные работы в «Cambridge Mathematical Journal». К 34 годам он стал профессором математики в Queen's College, Cork — без диплома. «The Laws of Thought» — амбициозная попытка свести законы мышления к алгебраическим операциям. Современники отнеслись к этой попытке скептически: идея о том, что логическое рассуждение можно «вычислять», казалась не более чем метафорой.

Но его работа оставалась абстрактной системой без приложений за пределами математической логики.

В 1937 году Клод Шеннон, 22-летний магистрант MIT, написал диссертацию «A Symbolic Analysis of Relay and Switching Circuits». Он показал, что для синтеза любой переключательной схемы достаточно фиксированного набора операций. Любая булева функция может быть реализована комбинацией реле. Это открытие соединило математическую логику XIX века с электротехникой XX века.

Его диссертация считается рождением цифровой схемотехники как научной дисциплины. Абстрактная идея — что логическое рассуждение сводится к вычислению — впервые была реализована в железе: реле, а затем транзисторы, стали физическим носителем логических операций.

⁷⁶G. Boole. «The Laws of Thought». 1854.

ОБЗОР ГЛАВЫ

Сначала мы познакомимся с булевыми функциями и сосчитаем их: 2^{2^n} функций от n переменных, и среди них — базис из И, ИЛИ и НЕ. Критерий Поста даст исчерпывающий ответ на вопрос о полноте: пять замкнутых классов, и набор полон, если не прячется ни в один из них. Нормальные формы — ДНФ, КНФ, полином Жегалкина — приведут любую функцию к каноническому виду, а карта Карно и алгоритм Квайна — МакКласки сократят её до минимума. Диаграммы решений (BDD) предложат представление на языке графов, компактное для функций с регулярной структурой. Теорема Стоуна вернёт нас к аксиомам: всякая булева алгебра оказывается алгеброй подмножеств. Сквозной вопрос главы — какой набор операций минимально достаточен для любой схемы — и ответ на него окажется физическим: NAND-логика КМОП, из которой собраны современные чипы.

После этой главы вы сможете:

1. считать булевы функции и работать с их представлениями: таблица истинности, ДНФ, КНФ, полином Жегалкина;
2. проверять функциональную полноту набора операций критерием Поста;
3. минимизировать формулы картой Карно и алгоритмом Квайна — МакКласки;
4. пользоваться диаграммами решений для функций с регулярной структурой;
5. объяснять, почему одного NAND достаточно для любой схемы, и что из этого следует для производства.

§ 10.1 Булевы функции

10.1.1 Подсчёт булевых функций

Булева функция — это простейшая возможная функция: на вход подаётся набор нулей и единиц, на выходе — тоже 0 или 1. Несмотря на простоту, из таких функций собирается любая цифровая схема. Это и показал Шеннон в 1937 году.

ОПРЕДЕЛЕНИЕ 10.1 Булева функция

n -арной булевой функцией называется функция

$$f : \{0, 1\}^n \rightarrow \{0, 1\}.$$

Функция полностью определяется своей таблицей истинности с 2^n строками — по одной на каждую комбинацию значений n переменных. Сомневаясь в формуле, инженер возвращается к таблице истинности, как школьник к таблице умножения. Читать её строки удобно на языке «включено» и «выключено»: единица — включено, ноль — выключено, и та же пара слов описывает и высказывание, и положение реле, и бит в регистре.

Например, функция $f(x, y) = x \wedge y$ задаётся таблицей истинности:

x	y	$f(x, y)$
0	0	0
0	1	0
1	0	0
1	1	1

Число различных булевых функций от n переменных растёт как 2^{2^n} . Это быстрее любой экспоненты.

ТЕОРЕМА 10.1 Подсчёт булевых функций

Существует ровно

$$2^{2^n}$$

различных n -арных булевых функций.

Доказательство

Докажем подсчётом независимого выбора выхода для каждой строки таблицы.

Таблица истинности содержит 2^n строк — каждая строка соответствует одной входной комбинации. Для каждой из 2^n строк выход может быть независимо выбран равным 0 или 1. Это даёт по два варианта на каждую строку, итого 2^{2^n} комбинаций.

Пример Количество булевых функций для малых n

- $n = 0$: $2^{2^0} = 2$ функции — константы 0 и 1.
- $n = 1$: $2^{2^1} = 4$ функции — тождественная ($f(x) = x$), отрицание ($f(x) = \neg x$), константы 0 и 1.
- $n = 2$: $2^{2^2} = 16$ функций — классические логические вентили (AND, OR, XOR, NAND и т.д.).
- $n = 3$: $2^8 = 256$ функций.
- $n = 5$: $2^{32} \approx 4.3 \cdot 10^9$ функций — больше, чем можно перебрать вручную.

При $n = 2$ существует ровно 16 функций, и у каждой есть стандартное имя. Они перечислены ниже. Некоторые из них (AND, OR, NOT) образуют базис, через который выражаются все остальные.

Пример 16 бинарных булевых функций

Имя	Формула	Обозначение
Константа 0	0	ложь
Конъюнкция	$x \wedge y$	AND

Имя	Формула	Обозначение
Запрет y	$x \wedge \bar{y}$	—
Проекция x	x	—
Запрет x	$\bar{x} \wedge y$	—
Проекция y	y	—
Сложение по модулю 2	$x \oplus y$	XOR
Дизъюнкция	$x \vee y$	OR
Стрелка Пирса	$\downarrow$	NOR
Эквивалентность	$x \equiv y$	XNOR
Отрицание y	$\bar{y}$	NOT y
Обратная импликация	$y \rightarrow x$	—
Отрицание x	$\bar{x}$	NOT x
Импликация	$x \rightarrow y$	—
Штрих Шеффера	$\uparrow$	NAND
Константа 1	1	истина

Импликация выражается через базовые операции: $x \rightarrow y = \neg x \vee y$. Эквивалентность — через конъюнкцию двух импликаций: $x \equiv y = (x \rightarrow y) \wedge (y \rightarrow x) = (\neg x \vee y) \wedge (\neg y \vee x)$.

10.1.2 Функциональная полнота

Набор логических вентилей, из которого можно построить любую булеву функцию, называется *функционально полным*. На этом понятии держится проектирование аппаратуры: достаточно реализовать полный набор вентилей, чтобы собрать любую схему.

ОПРЕДЕЛЕНИЕ 10.2 Функциональная полнота

Множество булевых функций F называется **функционально полным**, если всякая булева функция может быть записана как композиция функций из F .

Пример Полные и неполные множества

- Множество NOT не полно: через одно отрицание нельзя выразить конъюнкцию — любая композиция $\neg(\neg(\dots\neg x\dots))$ даёт только x или $\neg x$.
- Множество AND, OR не полно: обе функции монотонны, их композиции также монотонны, а отрицание не монотонно.
- Множество NOT, AND полно: OR выражается через законы де Моргана: $x \vee y = \neg(\neg x \wedge \neg y)$.
- Множество NAND полно (следующая теорема).

NAND и NOR выделяются среди остальных вентилей: каждого из них по отдельности достаточно для построения всей булевой алгебры.

ТЕОРЕМА 10.2 Штрих Шеффера и стрелка Пирса⁷⁷

Набор {И-НЕ} (штрих Шеффера, $\uparrow$) по отдельности функционально полон. То же верно для набора {ИЛИ-НЕ} (стрелка Пирса, $\downarrow$).

Доказательство

Докажем построением: достаточно выразить НЕ, И и ИЛИ только через И-НЕ (для ИЛИ-НЕ аналогично).

Отрицание получается замыканием входов: $\neg x = (x \uparrow x)$, поскольку NAND переменной с собой даёт отрицание. Конъюнкция — двойным отрицанием И-НЕ: $x \wedge y = \neg(x \uparrow y) = (x \uparrow y) \uparrow (x \uparrow y)$, сначала получая NAND, затем инвертируя. Дизъюнкция — по закону де Моргана: $x \vee y = \neg x \uparrow \neg y = (x \uparrow x) \uparrow (y \uparrow y)$, инвертируя оба входа перед И-НЕ.

Поскольку множество {НЕ}, {И}, {ИЛИ} полно (любая функция представима в ДНФ или КНФ), NAND также полно.

Доказательство для NAND использовало стандартный приём: выразить множество {НЕ}, {И}, {ИЛИ} (классический полный набор), и полнота наследуется. Формулы из доказательства читаются как рецепты сборки. НЕ получается замыканием входов: подать один сигнал на оба входа И-НЕ — значит спросить вентиль про один бит сразу в двух экземплярах, и ответом будет его отрицание. И получается двойным отрицанием И-НЕ, ИЛИ — инвертированием входов перед И-НЕ. Так из одной детали собираются все три базовые операции, и полнота из алгебраического факта становится технологией. Тот же метод применим и к другим комбинациям вентиляей.

УТВЕРЖДЕНИЕ 10.3 Стандартные полные множества

- {НЕ}, {И}, {ИЛИ} — полный набор.
- {НЕ}, {И} функционально полны.
- {НЕ}, {ИЛИ} функционально полны.
- {И}, {ИЛИ} не полно.

Доказательство

Множество {НЕ}, {И}, {ИЛИ} полно: любая булева функция представима в ДНФ или КНФ, а эти формы используют только НЕ, И и ИЛИ.

⁷⁷H. M. Sheffer. «A set of five independent postulates for Boolean algebras, with application to logical constants». Transactions of the AMS, 1913. C. S. Peirce описал NOR-функцию ранее (1880), но его работа не была опубликована при жизни.

Из пары $\{\text{НЕ}\}, \{\text{И}\}$ выражается недостающий ИЛИ: $x \vee y = \neg(\neg x \wedge \neg y)$. Из пары $\{\text{НЕ}\}, \{\text{ИЛИ}\}$ выражается недостающий И: $x \wedge y = \neg(\neg x \vee \neg y)$. Поэтому обе пары также функционально полны.

$\{\text{И}\}, \{\text{ИЛИ}\}$ не полно, потому что обе функции монотонны: увеличение входов (замена 0 на 1) не уменьшает выход. Композиция монотонных функций монотонна, а отрицание не монотонно — следовательно, отрицание невыразимо через И и ИЛИ.

ЛОВУШКА Двойное отрицание не проникает в дизъюнкцию

Законы де Моргана и двойное отрицание легко перемешать в неверную цепочку. Возьмём произвольные a, b . По де Моргану $\neg(a \wedge b) = \neg a \vee \neg b$. Применив $\neg$ к обеим частям, получим $a \wedge b = \neg(\neg a \vee \neg b)$. Если теперь применить двойное отрицание к отдельным переменным, «вывод» $a \wedge b = a \vee b$ готов. Дыра в том, что $\neg(\neg a \vee \neg b) = a \wedge b \neq a \vee b$: отрицание не проникает внутрь дизъюнкции по частям. Проверка на паре $a = 1, b = 0$ мгновенно даёт абсурд: $0 = 1$.

За правилом стоит механическая картинка: черта отрицания — жёсткая плита, лежащая на всём выражении. Раскрывая $\overline{a \wedge b}$, плита ломается, обломки падают на a и на b по отдельности, а связка переворачивается: $\wedge$ становится $\vee$. Поэтому отрицание распределяется по операндам целиком, а не проникает внутрь по частям.

10.1.3 Критерий Поста

Критерий Поста (1941) даёт исчерпывающий ответ на вопрос о полноте произвольного набора функций: набор полон тогда и только тогда, когда он не содержится целиком ни в одном из пяти максимальных замкнутых классов. Это имеет прямое инженерное значение при выборе вентиляционной библиотеки для производства чипа.

УТВЕРЖДЕНИЕ 10.4 Замкнутые классы Поста

Пять максимальных классов булевых функций, каждый замкнут относительно композиции:

- $T_0 : f(0, \dots, 0) = 0$ — функции, **сохраняющие 0**.
- $T_1 : f(1, \dots, 1) = 1$ — функции, **сохраняющие 1**.
- $S : \neg f(\neg x_1, \dots, \neg x_n) = f(x_1, \dots, x_n)$ — **самодвойственные** функции. Значение функции на инвертированных входах противоположно значению на исходных.
- M : **монотонные** функции — увеличение значений входов (замена 0 на 1) никогда не уменьшает значения выхода.
- $L : f = a_0 \oplus a_1 x_1 \oplus \dots \oplus a_n x_n$ — **линейные** функции (сумма по модулю 2 подмножества переменных, возможно, с константой).

Каждый класс Поста фиксирует конкретный «дефект» — ограничение, запрещающее выражать функции определённого типа. Критерий Поста утверждает, что этих пяти дефектов достаточно: если набор избегает всех, он полон.

ТЕОРЕМА 10.5 Критерий Поста⁷⁸

Множество булевых функций F функционально полно тогда и только тогда, когда оно не содержится целиком ни в одном из классов T_0, T_1, S, M, L .

Доказательство

Докажем критерий в две части: сначала покажем, что каждый из пяти классов замкнут относительно композиции и является собственным максимальным, затем построим из пяти функций-свидетелей полную систему.

Каждый из пяти классов замкнут относительно композиции: подстановка функций класса в функцию того же класса даёт функцию, остающуюся в классе. Проверим для каждого.

Для T_0 : если $f, g_1, \dots, g_n \in T_0$, то $f(g_1(0, \dots, 0), \dots, g_n(0, \dots, 0)) = f(0, \dots, 0) = 0$. Для T_1 : если $f, g_1, \dots, g_n \in T_1$, то $f(g_1(1, \dots, 1), \dots, g_n(1, \dots, 1)) = f(1, \dots, 1) = 1$. Для S : если $f, g_1, \dots, g_n$ самодвойственны, то самодвойственна и их композиция. Обозначим $h((x)) = f(g_1((x)), \dots, g_n((x)))$. Тогда $\neg h(\neg(x)) = \neg f(g_1(\neg(x)), \dots, g_n(\neg(x))) = \neg f(\neg g_1((x)), \dots, \neg g_n((x)))$ по самодвойственности вложенных функций, а она означает $g_i(\neg y) = \neg g_i(y)$. Затем по самодвойственности внешней функции $\neg f(\neg g_1((x)), \dots, \neg g_n((x))) = f(g_1((x)), \dots, g_n((x))) = h((x))$. Для M : монотонность сохраняется при композиции — увеличение входов не уменьшает значений вложенных функций, поэтому не уменьшает и значения внешней функции. Для L : линейная функция от линейных функций раскрывается в линейную (подстановка суммы по модулю 2 в сумму по модулю 2 даёт сумму по модулю 2).

Каждый класс является собственным. Функция $\neg x$ не сохраняет ни 0, ни 1, поэтому не входит в классы T_0, T_1 . Функция $x \wedge y$ не самодвойственна: $\neg(\neg x \wedge \neg y) = x \vee y \neq x \wedge y$. Функция $\neg x$ не монотонна, поэтому не входит в класс M . Функция $x \wedge y$ не линейна, поэтому не входит в класс L . Все пять — максимальные замкнутые классы.

Необходимость. Если F полно, оно не может содержаться ни в одном собственном замкнутом классе. Действительно, если $F \subseteq C$ для замкнутого класса C , то и замыкание по композиции (все функции, выражимые из исходных) лежит в C . Поскольку C — собственный класс, он не содержит всех булевых функций, поэтому F не полно. Каждый из T_0, T_1, S, M, L является собственным замкнутым классом, поэтому полное F не может содержаться ни в одном из них.

⁷⁸E. L. Post. «The Two-Valued Iterative Systems of Mathematical Logic». Annals of Mathematics Studies, 1941. Русский перевод: Э. Пост. «Двузначные итеративные системы математической логики».

Достаточность. Пусть F не содержится ни в одном из пяти классов. Тогда в F существуют функции-«свидетели»: $f_0 \notin T_0, f_1 \notin T_1, f_S \notin S, f_M \notin M, f_L \notin L$. Покажем, как из них построить полную систему.

Шаг 1: константы. $f_0 \notin T_0 \Rightarrow f_0(0, \dots, 0) = 1$. Подставим x на все входы. Получим $g(x) = f_0(x, \dots, x)$. Тогда либо $g(x) = 1$ (константа 1 получена), либо $g(x) = \neg x$. Аналогично $f_1 \notin T_1 \Rightarrow f_1(1, \dots, 1) = 0$. Подстановка одной переменной на все входы даёт $h(x) = f_1(x, \dots, x)$ — константу 0 или $\neg x$.

Если уже получили $\neg x$ — отрицание построено. Если получили константы 0 и 1 — используем немонотонную функцию f_M . Немонотонность даёт два входа $a \leq b$ (покоординатно). На них значения функции противоположны: $f_M(a) = 1, f_M(b) = 0$. Подставим в f_M константы 0 и 1 на позиции, где $a_i = b_i$, а переменную x — на позиции, где $a_i = 0, b_i = 1$. Полученная функция имеет значения $g(0) = f_M(a) = 1, g(1) = f_M(b) = 0$, то есть $g(x) = \neg x$.

Остаётся случай, когда обе подстановки дали только $\neg x$ — отрицание есть, а констант нет. Константу строим из несамоудовлетворенной функции f_S . Несамодовлетворенность даёт вход, где $f_S(a) = f_S(\bar{a})$. Подставим x на позиции нулей этого входа, а на позиции единиц — $\neg x$. Полученная функция примет одно и то же значение на 0 и 1, то есть будет константой, а вторую константу даст отрицание.

Шаг 2: нелинейность даёт конъюнкцию. $f_L \notin L$ — её полином Жегалкина содержит моном с двумя и более переменными⁷⁹.

Имеем $f_L = \dots \oplus x_i x_j \dots \oplus$ (линейные члены и константа). Зафиксировав все переменные кроме x_i и x_j константами (полученными на шаге 1), получаем функцию вида $x_i x_j \oplus a x_i \oplus b x_j \oplus c$. Подстановка $x_i \mapsto x_i \oplus b, x_j \mapsto x_j \oplus a$ убирает линейные члены. Остаётся $x_i x_j \oplus c'$, то есть конъюнкция $x_i \wedge x_j$, возможно, с отрицанием. Если $c' = 1$, лишнее отрицание снимается с помощью NOT. Подстановка $x_i \mapsto x_i \oplus b$ при $b = 1$ заменяет переменную на её отрицание. Нелинейный моном $x_i x_j$ сохраняется, а линейные члены $a x_i$ и $b x_j$ сокращаются.

Возьмём $f_L = xy \oplus x$. Подстановка $y \mapsto y \oplus 1$ сокращает линейный член. Цепочка: $x(y \oplus 1) \oplus x = xy \oplus x \oplus x = xy$.

Шаг 3: полнота. Имея $\neg$ и $\wedge$, получаем $\vee$ по де Моргану: $x \vee y = \neg(\neg x \wedge \neg y)$. Система $\{\neg, \wedge, \vee\}$ полна (всякая булева функция представима в виде ДНФ или КНФ). Следовательно, F полно.

Интуиция доказательства: избегая всех пяти дефектов, F последовательно строит отрицание (из f_0, f_1 и немонотонной f_M), конъюнкцию (из нелинейной f_L) и получает полную систему.

⁷⁹Полином Жегалкина (АНФ) — представление булевой функции как XOR-суммы конъюнкций переменных без отрицаний, единственное для каждой функции (подробнее в разделе 10.7).

Замечание Почему критерий Поста даёт полный ответ?

Критерий Поста — редкий случай в математике: необходимое и достаточное условие, сформулированное конечным и обозримым образом.

Пять классов Поста — T_0, T_1, S, M, L — не являются произвольным набором ограничений. Каждый из них максимален: это собственный замкнутый класс, который нельзя расширить, не получив всё множество булевых функций. Максимальность означает, что класс достаточно широк, чтобы быть замкнутым относительно суперпозиции, и достаточно узок, чтобы не совпадать с полным множеством. Он — точная граница выразительности.

Теперь структурный вывод: любой набор функций, не содержащийся целиком ни в одном из пяти классов, способен породить все булевы функции. Именно максимальность превращает критерий Поста из эмпирического наблюдения в доказанную полноту классификации.

Пример Проверка полноты через критерий Поста

Проверим, является ли $\{\text{И-НЕ}\}$ ($\uparrow$) полным:

- T_0 : $\text{И-НЕ}(0, 0) = 1 \neq 0$ — не принадлежит T_0 .
- T_1 : $\text{И-НЕ}(1, 1) = 0 \neq 1$ — не принадлежит T_1 .
- S : проверим самодвойственность. С одной стороны, $\neg(\text{И-НЕ}(\neg x, \neg y)) = \neg(x \vee y) = \neg x \wedge \neg y$. С другой стороны, $\text{И-НЕ}(x, y) = \neg(x \wedge y) = \neg x \vee \neg y$. При $x = 1, y = 1$ левая часть = 0, правая = 0 — совпало. При $x = 1, y = 0$ левая часть = 0, правая = 1 — не совпало, значит, штрих Шеффера не самодвойственна.
- M : $\text{И-НЕ}(0, 1) = 1 > 0 = \text{И-НЕ}(1, 1)$ — увеличение входа уменьшает выход, не монотонна.
- L : $\text{И-НЕ}(x, y) = 1 \oplus xy$ — нелинейна из-за произведения переменных.

И-НЕ не принадлежит ни одному из пяти классов $\Rightarrow \{\text{И-НЕ}\}$ полно.

Примечание

И-НЕ и ИЛИ-НЕ являются универсальными вентилями в цифровой схемотехнике. Каждый из них функционально полон и при этом физически прост: в КМОП-технологии и NAND, и NOR требуют по 4 транзистора. NAND предпочтителен из-за более высокой подвижности электронов (что даёт лучшие частотные характеристики), поэтому NAND-вентили доминируют в современных интегральных схемах.

Один вентиль — NAND — достаточен для построения любой цифровой схемы: через NAND выражаются отрицание, конъюнкция и дизъюнкция (эти схемы мы построили в доказательстве теоремы о штрихе Шеффера). Из них собираются сумматоры, триггеры и, в конечном счёте, целые процессоры.

Пример Критерий Поста в другую сторону: неполный базис Жегалкина

Проверим, является ли полным множество $F = \{\text{И}, \text{XOR}\}$ — те самые операции, из которых строится полином Жегалкина в разделе 10.7. Оба направления критерия полезны: предыдущий пример показал полноту, этот пример покажет, как критерий ловит неполноту.

Пройдём по пяти классам Поста.

- T_0 : $\text{И}(0, 0) = 0$ и $\text{XOR}(0, 0) = 0$ — обе функции сохраняют 0, значит, $F \subset T_0$.
- T_1 : $\text{И}(1, 1) = 1$, но $\text{XOR}(1, 1) = 0 \neq 1$ — $F \not\subset T_1$.
- S : XOR не самодвойственна: $\neg(\text{XOR}(\neg x, \neg y))$ при $(x, y) = (0, 0)$ даёт $\neg(\text{XOR}(1, 1)) = 1 \neq 0 = \text{XOR}(0, 0)$ — $F \not\subset S$.
- M : И монотонна, но $\text{XOR}(0, 1) = 1$ и $\text{XOR}(1, 1) = 0$ — увеличение входа уменьшает выход — $F \not\subset M$.
- L : XOR линейна, но $\text{И}(x, y) = xy$ — произведение переменных — нелинейна, $F \not\subset L$.

Единственный класс, в который F уместается целиком, — T_0 . Критерий Поста даёт вывод: F не полно. Подтверждение без всякого критерия: обе функции сохраняют 0, поэтому любая их композиция тоже сохраняет 0, и константа 1 невыразима. Именно поэтому полином Жегалкина в разделе 10.7 требует константный член a_0 : без него полнота теряется.

В главе 1 мы упомянули, что пяти логических связок — $\neg, \wedge, \vee, \rightarrow, \equiv$ — более чем достаточно: хватило бы одной, штриха Шеффера $\uparrow$. Тогда это выглядело как любопытный факт о выразительности. Теперь мы понимаем *почему*: это прямое следствие функциональной полноты в булевой алгебре.

Эволюция идеи прослеживается в трёх шагах.

1. Сначала логические связки выступают как операции над истинностью высказываний (глава 1).
2. Затем булевы функции оказываются абстрактными объектами: 2^{2^n} функций, классифицируемых критерием Поста по пяти замкнутым классам (T_0, T_1, S, M, L).
3. Наконец функциональная полнота становится инженерным критерием: какой минимальный набор вентилях достаточен для построения любой цифровой схемы (глава 11).

Так критерий Поста, доказанный в 1941 году как результат о выразимости логических связок, стал математическим обоснованием выбора NAND в качестве универсального вентиля цифровой схемотехники.

10.1.4 Булева алгебра как алгебраическая структура

В языках системного программирования — C, Rust, Go — побитовые операторы реализуют булеву алгебру, применённую к каждому биту независимо.

- $x \& y$ — побитовая конъюнкция: i -й бит результата равен 1, только если оба i -х бита операндов равны 1.
- $x | y$ — побитовая дизъюнкция.
- $\sim x$ — побитовое дополнение (отрицание).
- $x \wedge y$ — побитовое XOR (сложение по модулю 2).

Целочисленная переменная (32 или 64 бита) — это булев вектор: $(b_{31}, \dots, b_0) \in \{0, 1\}^{32}$. Побитовая операция — это покомпонентное применение соответствующей булевой функции: результат i -го бита зависит только от i -х битов операндов. Все законы булевой алгебры — коммутативность, ассоциативность, дистрибутивность, де Моргана — выполняются побитово. Компилятор использует их для оптимизации: замена $(x | y) \& z$ на $(x \& z) | (y \& z)$ — легальная трансформация.

Аппаратно побитовые операции реализуются параллельно: 32-битное И — это 32 логических вентиля, работающих одновременно за один такт. Когда вы пишете `if (flags & MASK)`, процессор вычисляет побитовую конъюнкцию на вентилях из раздела 10.1.2. Цепочка булева алгебра $\rightarrow$ NAND-вентиль $\rightarrow$ побитовая операция $\rightarrow$ проверка флага в программе — это четыре уровня одной и той же структуры, от абстрактной математики до исполняемого кода.

Булевы функции можно изучать двумя способами: как таблицы истинности и как абстрактную алгебраическую структуру, заданную аксиоматически.

ОПРЕДЕЛЕНИЕ 10.3 Аксиомы булевой алгебры

Алгебраическая структура

$$(B, \wedge, \vee, \overline{(\cdot)}, 0, 1)$$

с двумя бинарными операциями $\wedge$ (конъюнкция) и $\vee$ (дизъюнкция), одной унарной операцией дополнения $\overline{(\cdot)}$ (черта над элементом) и двумя выделенными элементами 0 и 1 называется **булевой алгеброй**. Должны выполняться следующие условия для всех $a, b, c \in B$:

- **Коммутативность:** $a \wedge b = b \wedge a, a \vee b = b \vee a$.
- **Ассоциативность:** $(a \wedge b) \wedge c = a \wedge (b \wedge c), (a \vee b) \vee c = a \vee (b \vee c)$.
- **Дистрибутивность:** $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c), a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.
- **Нейтральные элементы:** $a \wedge 1 = a, a \vee 0 = a$.
- **Дополнение:** $a \wedge \bar{a} = 0, a \vee \bar{a} = 1$.

Примечание Почему законы доказываются перебором

Тождества булевой алгебры можно доказывать полным перебором — и это законное доказательство, а не эвристика. У переменной всего два значения, поэтому проверить формулу на нуле и на единице — значит проверить её везде. В арифметике такой возможности нет: вещественных чисел не перебрать, и тож-

дества приходится выводить цепочками преобразований. Булева алгебра в этом смысле уникальна: короткая проверка здесь заменяет длинное доказательство.

Пример Алгебра множеств как булева алгебра

Семейство всех подмножеств фиксированного множества U с операциями $\cap$ (пересечение), $\cup$ (объединение) и $\bar{X}$ (дополнение) образует булеву алгебру. Выделенные элементы — это $\emptyset, U$. Проверим аксиомы:

- $A \cap B = B \cap A, A \cup B = B \cup A$ (коммутативность) — верно.
- $(A \cap B) \cap C = A \cap (B \cap C)$ (ассоциативность) — верно.
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (дистрибутивность) — верно.
- $A \cap U = A, A \cup \emptyset = A$ (нейтральные элементы) — верно.
- $A \cap \bar{A} = \emptyset, A \cup \bar{A} = U$ (дополнение) — верно.

Пример Простейшая булева алгебра — $\{0, 1\}$

Простейшая булева алгебра — само множество $\{0, 1\}$ с операциями $\wedge, \vee, \neg$. Все аксиомы проверяются непосредственно: это модель, с которой мы работали с первой главы. Все остальные конечные булевы алгебры изоморфны алгебре подмножеств некоторого множества (теорема Стоуна, раздел 10.9).

Часть аксиом выглядит знакомо по школьной алгебре: $a \wedge 1 = a, a \vee 0 = a$ почти повторяют умножение на единицу и прибавление нуля. Дальше начинается то, от чего арифметика отвыкла: $a \wedge a = a$ — никакого a^2 , потому что и $0^2 = 0$, и $1^2 = 1$. Сколько ни прибавляй к истине истину, истиннее она не станет: $a \vee 1 = 1$. А распределительный закон работает в обе стороны. Вторая его половина $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ в арифметике была бы заведомой ложью. Её аналог $a + bc = (a + b)(a + c)$ неверен уже при малых числах. Законы булевой алгебры нельзя вывести из арифметики, их нужно читать по таблицам истинности.

Аксиомы обладают симметрией: каждая состоит из двух частей, переходящих друг в друга заменой $\wedge$ на $\vee$ и 0 на 1. Отсюда — принцип двойственности.

УТВЕРЖДЕНИЕ 10.6 Принцип двойственности

Всякое тождество булевой алгебры остаётся верным, если одновременно заменить $\wedge$ на $\vee$, $\vee$ на $\wedge$, 0 на 1, 1 на 0.

Доказательство

Докажем отражением доказательства: заменим каждую аксиому на её двойственную пару.

Каждая аксиома булевой алгебры состоит из двух частей, переходящих друг в друга при указанной замене. Поэтому доказательство любого тождества, ис-

пользующее только аксиомы, можно «отразить» — заменить каждую аксиому на её двойственную пару — получив доказательство двойственного тождества.

ЛОВУШКА Двойственность и законы де Моргана

Двойственное выражение получается одновременной заменой $\wedge$ на $\vee$, $\vee$ на $\wedge$, 0 на 1, 1 на 0. Два закона де Моргана двойственны друг другу: $\overline{A \wedge B} = \overline{A} \vee \overline{B}$, $\overline{A \vee B} = \overline{A} \wedge \overline{B}$. Частая ошибка — применить замену частично: поменять связку и забыть дополнения ($\overline{A \wedge B} = A \vee B$) либо дополнить операнды, не тронув связку ($\overline{A \wedge B} = \overline{A} \wedge \overline{B}$).

Проверка Аксиомы булевой алгебры и двойственность

1. Почему дистрибутивность в булевой алгебре действует в обе стороны, а в обычной арифметике — только в одну?
2. Принцип двойственности требует менять и связки, и константы. Почему замены одних связок недостаточно?

§ 10.2 * Клоны и замыкание

Критерий Поста делит наборы функций на полные и неполные, но за списком из пяти классов стоит более общая картина. Слово «максимальные» в их названии — термин теории порядка из главы 8: замкнутые классы вложены друг в друга, и пять классов Поста занимают в этом порядке верхний ярус. Если развернуть вложение целиком, замкнутые классы выстроятся в решётку, а критерий Поста окажется утверждением о её строении. Для инженера это каталог вентилярных библиотек: клон — все схемы, собираемые из данного набора деталей при любой глубине вложения.

ОПРЕДЕЛЕНИЕ 10.4 Замыкание

Замыканием множества F булевых функций называется наименьшее множество булевых функций, обозначается $[F]$, которое содержит F , все проекции $x_1, \dots, x_n \mapsto x_i$ и замкнуто относительно композиции: вместе с любыми $f, g_1, \dots, g_m$ оно содержит и $f(g_1, \dots, g_m)$.

ОПРЕДЕЛЕНИЕ 10.5 Клон

Множество булевых функций называется **клоном** (от англ. clone), если оно содержит все проекции $x_1, \dots, x_n \mapsto x_i$ и замкнуто относительно композиции.

Замыкание собирает из набора всё, что вообще можно собрать: функции, выраженные из F , и ничего кроме них.

Примечание Почему константы не бесплатны

В замыкании константы 0 и 1 не появляются по желанию: подстановка константы — это композиция с функцией-константой, и такая функция должна сначала попасть в $[F]$. Набор $\{И\}, \{XOR\}$ сохраняет ноль, и пока константа 1 не построена, ни одна композиция не нарушит это свойство — так класс T_0 работает в критерии Поста как препятствие. Свидетели f_0 и f_1 из доказательства критерия — функции, через которые константы всё же добываются. После этого запреты T_0 и T_1 теряют силу.

Пример Клоны и их замыкания

- Проекция образует клон — и наименьший из всех: композиция проекций снова даёт проекцию, а присутствие проекций в любом клоне предписано определением.
- Каждый из пяти классов Поста T_0, T_1, S, M, L — клон: замкнутость относительно композиции доказана в предыдущем разделе, а проекции удовлетворяют всем пяти условиям.
- Замыкание одиночного отрицания — проекции и их отрицания: $\neg\neg x = x$, других композиций из одного $\neg$ не получается. Констант в этом клоне нет: ни 0, ни 1 из отрицаний не собрать.
- Замыкание $\{И\}, \{XOR\}$ — весь класс T_0 . Обе функции сохраняют ноль, а T_0 — клон, поэтому $[\{И\}, \{XOR\}] \subseteq T_0$. У полинома Жегалкина свободный член — это и есть значение функции на нулевом наборе: все мономы с переменными на этом наборе обнуляются. У функций из T_0 значение на нулевом наборе равно нулю, значит, свободный член нулевой. Наоборот, всякий полином без свободного члена задаёт функцию из T_0 и собирается из переменных конъюнкциями и XOR-суммами, то есть лежит в замыкании. Выше, в примере с базисом Жегалкина, критерий Поста показал, что набор не полон. Теперь ответ называется точно: $[\{И\}, \{XOR\}] = T_0$.

УТВЕРЖДЕНИЕ 10.7 Решётка клонов

Клоны булевых функций, упорядоченные по включению, образуют решётку. Пересечение любого семейства клонов — клон. Наименьший клон — множество проекций, наибольший — множество всех булевых функций, а замыкание $[F]$ — пересечение всех клонов, содержащих F .

Доказательство

Докажем проверкой пересечения и указанием крайних элементов.

Пусть C_i — семейство клонов. Каждая проекция лежит в любом из C_i , поэтому лежит и в пересечении. Если $f, g_1, \dots, g_m$ лежат в пересечении, то по замкнуто-

сти каждого C_i там же лежит и $f(g_1, \dots, g_m)$, значит, пересечение замкнуто и содержит проекции — это клон.

Пересечение всех клонов, содержащих F , само клон и содержится в любом другом таком — это и есть $[F]$, наименьший клон над F . Снизу решётку замыкает клон проекций: по определению они входят в каждый клон. Сверху замыкает клон всех булевых функций: всё множество функций замкнуто и содержит любой клон. У пары клонов есть нижняя грань (пересечение) и верхняя (замыкание объединения), что и превращает систему клонов в решётку.

В этой решётке максимальные классы из критерия Поста получают точное место. Элемент, над которым в решётке нет ничего, кроме наибольшего элемента, называется *коатомом*. Двойственно определяются атомы — минимальные ненулевые элементы (они понадобятся в разделе 10.9). Максимальные собственные замкнутые классы — это коатомы решётки клонов. У решётки клонов двузначных функций коатомов ровно пять: T_0 , T_1 , S , M , L . В литературе по многозначным логикам их называют *предполными классами*. Цепи вложенных клонов могут продолжаться вниз неограниченно, но вверх обрываются: над любым собственным клоном стоит какой-нибудь коатом, а над коатомом — только верхний элемент.

УТВЕРЖДЕНИЕ 10.8 Критерий Поста на языке клонов

Для множества F булевых функций равносильны три условия:

- F функционально полно.
- Замыкание $[F]$ содержит всякую булеву функцию.
- F не содержится целиком ни в одном коатоме решётки клонов.

Набросок доказательства

Первое и второе условия равносильны по определению замыкания: полнота означает, что всякая булева функция записывается композицией функций из F , а все такие записи живут в $[F]$. Третье условие с первыми двумя связывает критерий Поста из предыдущего раздела: коатомы решётки клонов двузначных функций — в точности классы T_0 , T_1 , S , M , L , максимальность каждого установлена там же.

Остаётся вопрос о размере решётки. Булевых функций всех конечных арностей счётное число — счётное объединение конечных множеств, — поэтому клонов не больше, чем подмножеств счётного множества, то есть континуум (глава 7). Пост описал решётку целиком, и клонов оказалось счётное число: пять коатомов, их пересечения и бесконечные параметрические семейства классов, нисходящие цепями к клону проекций. Счётность — свойство именно двузначного мира: уже в трёхзначной логике замкнутых классов континуум. Некоторые из этих классов не

имеют даже конечного базиса⁸⁰.

Замечание Пост и решётки

Классификация замкнутых классов появилась раньше языка, которым она теперь описывается. Пост анонсировал свои результаты в начале 1920-х⁸¹ и опубликовал сводку лишь в 1941 году, а систематическая теория решёток сложилась в 1930-е, с работами Биркхоффа. Слово «клон» вошло в печать ещё позже, в монографии по универсальной алгебре 1965 года, авторство термина приписывается Филиппу Холлу⁸². Максимальные классы Пост выделил прямым анализом суперпозиций, без решёточной терминологии: структура в математике обычно обнаруживается раньше, чем имя для неё.

§ 10.3 Литералы, минтермы, макстермы

Любую булеву функцию можно представить в виде выражения, построенного из трёх операций: НЕ, И, ИЛИ. Но этого недостаточно для алгоритмической работы — нужно привести выражение к стандартному виду.

Базовые кирпичики нормальных форм — это *литералы* и элементарные конъюнкции/дизъюнкции.

Элементарные конъюнкции решают конкретную задачу. Таблица истинности — это 2^n строк, и каждая строка — независимый случай. Чтобы собрать функцию из этих случаев, нужен механизм, который «включается» на одной строке и «выключен» на всех остальных. Именно это делают *минтермы*: конъюнкция литералов истинна ровно на одной входной комбинации и ложна на всех остальных. Сложив (дизъюнкцией) минтермы для строк, где функция истинна, мы получаем ДНФ. Симметрично, *макстермы* «выключаются» на одной строке — их конъюнкция даёт КНФ. Минтермы и макстермы — две реализации одной идеи: представить функцию через атомарные детекторы строк.

ОПРЕДЕЛЕНИЕ 10.6 Литерал

Литерал — переменная или её отрицание: $x_i, \overline{x_i}$.

ОПРЕДЕЛЕНИЕ 10.7 Минтерм

Минтерм — конъюнкция литералов, в которую каждая переменная входит ровно один раз.

⁸⁰Ю. И. Янов, А. А. Мучник. «О существовании k -значных замкнутых классов, не имеющих конечного базиса». Доклады АН СССР, 1959.

⁸¹E. Post. «Introduction to a General Theory of Elementary Propositions». American Journal of Mathematics, 1921.

⁸²P. M. Cohn. «Universal Algebra». 1965.

Конъюнкция требует всех литералов разом — поэтому истинна лишь на одной строке. Для дизъюнкции достаточно одного истинного литерала, поэтому ложна она лишь на одной строке — когда все литералы ложны.

ОПРЕДЕЛЕНИЕ 10.8 Макстерм

Макстерм — дизъюнкция литералов, в которую каждая переменная входит ровно один раз.

И минтермов, и макстермов ровно 2^n — по одному на строку таблицы истинности. Оба свойства видны на примере с тремя переменными.

Пример Минтермы и макстермы для $n = 3$

Минтерм $x \wedge \bar{y} \wedge z$ истинен ровно на одном наборе. Это набор $(x, y, z) = (1, 0, 1)$.

Макстерм $x \vee \bar{y} \vee z$ ложен ровно на одном наборе. Это набор $(x, y, z) = (0, 1, 0)$.

Каждый минтерм и макстерм однозначно соответствует строке таблицы истинности.

Зачем нужна эта атомарная структура, станет ясно, когда мы перейдём к минимизации. Два минтерма, отличающиеся ровно в одной переменной, склеиваются: $xyz \vee xy\bar{z} = xy$. Один литерал исчезает. Именно на этом держится вся техника минимизации: карта Карно визуализирует смежность минтермов, алгоритм Квайна — МакКласки автоматизирует склеивание. Но прежде чем минимизировать, нужно научиться строить.

§ 10.4 ДНФ и КНФ

Нормальные формы были введены в главе 1 для формул логики высказываний. Здесь мы рассматриваем их с инженерной точки зрения — как представление булевых функций для аппаратной реализации и алгоритмической обработки. Ниже рассматриваются совершенные формы, их связь с таблицей истинности и переход к алгебраической нормальной форме (АНФ).

ОПРЕДЕЛЕНИЕ 10.9 Дизъюнктивная нормальная форма

Дизъюнкция конъюнкций литералов

$$(l_{1,1} \wedge \dots) \vee (l_{2,1} \wedge \dots) \vee \dots \vee (l_{m,1} \wedge \dots)$$

называется **дизъюнктивной нормальной формой (ДНФ)**.

ОПРЕДЕЛЕНИЕ 10.10 Терм

Каждая конъюнкция литералов в ДНФ называется **термом**.

Пример ДНФ: сумма произведений

$f(x, y, z) = (x \wedge \bar{y}) \vee (\bar{x} \wedge z) \vee (y \wedge z)$ — ДНФ из трёх термов. Функция истинна, если истинен хотя бы один терм:

- $x \wedge \bar{y}$: истинна, когда $x = 1, y = 0$
- $\bar{x} \wedge z$: истинна, когда $x = 0, z = 1$
- $y \wedge z$: истинна, когда $y = 1, z = 1$

ДНФ — это «сумма произведений»: функция истинна, если истинен хотя бы один терм. Но одной ДНФ недостаточно, и причина в двойственности: то, что ДНФ делает для единиц, КНФ делает для нулей. ДНФ «собирает» условия, при которых функция истинна. КНФ «исключает» условия, при которых она ложна. В одних задачах (SAT) естественнее КНФ, в других (синтез схем) — ДНФ. Перевод между ними нетривиален: дистрибутивность $\wedge$ относительно $\vee$ порождает экспоненциальный взрыв размера. На практике инженер выбирает ту форму, которая компактнее для конкретной функции.

ОПРЕДЕЛЕНИЕ 10.11 Конъюнктивная нормальная форма

Конъюнкция дизъюнкций литералов

$$(l_{1,1} \vee \dots) \wedge (l_{2,1} \vee \dots) \wedge \dots \wedge (l_{m,1} \vee \dots)$$

называется **конъюнктивной нормальной формой (КНФ)**.

ОПРЕДЕЛЕНИЕ 10.12 Дизъюнкт

Каждая дизъюнкция литералов в КНФ называется **дизъюнктом (clause)**.

Определения ДНФ и КНФ подводят к вопросу: для всякой ли булевой функции существует представление в этих формах? Ответ положительный, и построение алгоритмично.

ТЕОРЕМА 10.9 Существование нормальных форм

Всякая булева функция имеет представление как в ДНФ, так и в КНФ.

Доказательство

Докажем построением в две части: ДНФ соберём из единиц таблицы, КНФ — из нулей.

Построение ДНФ. Для каждой строки таблицы истинности, где $f(x_1, \dots, x_n) = 1$, строим минтерм: конъюнкцию литералов, в которой переменная x_i входит без отрицания при $x_i = 1$ и с отрицанием при $x_i = 0$. Этот минтерм равен 1 ровно на данной строке и 0 на всех остальных. Дизъюнкция всех таких минтермов равна 1 в точности на тех строках, где $f = 1$, то есть совпадает с f .

Построение КНФ. Для каждой строки, где $f(x_1, \dots, x_n) = 0$, строим макстерм: дизъюнкцию литералов, в которой переменная x_i входит с отрицанием при $x_i = 1$ и без отрицания при $x_i = 0$. Этот макстерм равен 0 ровно на данной строке и 1 на всех остальных. Конъюнкция всех таких макстермов равна 0 в точности на тех строках, где $f = 0$, то есть совпадает с f .

Если функция тождественно равна 0, её ДНФ — пустая дизъюнкция (константа 0). Если тождественно равна 1, её КНФ — пустая конъюнкция (константа 1).

Пример ДНФ и КНФ для XOR

Функция $x \oplus y$ истинна ровно на двух наборах. Это наборы $(0, 1), (1, 0)$.

ДНФ: $(\bar{x} \wedge y) \vee (x \wedge \bar{y})$ — дизъюнкция двух минтермов.

КНФ: $(x \vee y) \wedge (\bar{x} \vee \bar{y})$ — конъюнкция двух макстермов (функция равна 0 на $(0, 0)$ и $(1, 1)$).

В примере с XOR каждый терм использует обе переменные — и в ДНФ, и в КНФ. Это частный случай общего свойства: формы, в которых каждый терм содержит все n переменных, обладают полезной структурной простотой.

Отличие совершенных форм от обычных — в полноте термов. Обычная ДНФ — любая дизъюнкция конъюнкций. Термы могут использовать произвольные подмножества переменных. $x \vee (x \wedge \bar{y})$ — корректная ДНФ, но не совершенная. Совершенная форма требует, чтобы *каждый* терм/дизъюнкт содержал *все* n переменных. Это жёсткое ограничение, но оно даёт взамен единственность: СДНФ и СКНФ определены однозначно. У каждой функции ровно один канонический представитель в совершенной форме. Обычные же формы — это сокращённые, упрощённые варианты. Их много, и они не единственны.

ОПРЕДЕЛЕНИЕ 10.13 СДНФ

ДНФ, в которой каждая конъюнкция содержит все n переменных, называется **совершенной ДНФ** (СДНФ).

ОПРЕДЕЛЕНИЕ 10.14 СКНФ

КНФ, в которой каждая дизъюнкция содержит все n переменных, называется **совершенной КНФ** (СКНФ).

Совершенные формы единственны с точностью до порядка термов и дизъюнктов.

Пример Неединственность минимальной ДНФ

Функция $f(x, y, z)$, равная 1 на всех наборах, кроме $(0, 0, 0)$ и $(1, 1, 1)$, имеет два разных минимальных покрытия по три терма:

$$f = y\bar{z} \vee \bar{x}z \vee x\bar{y} = x\bar{z} \vee \bar{x}y \vee \bar{y}z.$$

Обе формулы содержат по три терма, и ни одну нельзя сократить, не потеряв эквивалентности. Неединственность минимальной ДНФ — штатная ситуация при минимизации.

ЛОВУШКА ДНФ не единственна, АНФ единственна

Единственна только СДНФ: обычных эквивалентных ДНФ много, и карта Карно ищет среди них минимальную. Полином Жегалкина (АНФ) единственен для каждой функции, поэтому он каноническая форма, а ДНФ — нет.

Пример Совершенные и несовершенные формы для импликации

Рассмотрим $f(x, y) = x \rightarrow y = \neg x \vee y$. Таблица истинности даёт $f = 0$ только на $(1, 0)$ и $f = 1$ на $(0, 0), (0, 1), (1, 1)$.

- СДНФ (дизъюнкция минтермов для строк с $f = 1$):

$$(\bar{x} \wedge \bar{y}) \vee (\bar{x} \wedge y) \vee (x \wedge y).$$

Содержит три минтерма.

- СКНФ (конъюнкция макстермов для строк с $f = 0$ — здесь единственная строка):

$$(\bar{x} \vee y).$$

Содержит один макстерм и компактна.

- Минимальная ДНФ — $\bar{x} \vee y$ — совпадает здесь со СКНФ и втрое короче СДНФ.

Это иллюстрирует, почему совершенные формы — лишь отправная точка, и нужна минимизация.

§ 10.5 СДНФ, АНФ и смена базиса

СДНФ уже раскладывает функцию по базису — но по минтермам, которых 2^n . Обобщим приём, взглянув на функции как на векторы.

Функции $\{0, 1\}^n \rightarrow \{0, 1\}$ образуют векторное пространство над $\text{GF}(2)$ — полем из двух элементов $\{0, 1\}$ с операциями XOR (сложение) и AND (умножение). Сложение в нём — поточечный XOR. Его размерность равна 2^n — по одной функции на строку таблицы истинности. Минтермы образуют базис этого пространства. Каждый минтерм — индикатор одной строки таблицы: он равен 1 ровно на одном наборе. СДНФ — разложение функции по этому базису: на каждом наборе истинен ровно один минтерм, поэтому OR по минтермам совпадает с XOR, и дизъюнкция собирает функцию как сумму в $\text{GF}(2)$.

Но базис из минтермов неудобен: каждый минтерм использует все n переменных. Всего таких минтермов 2^n . Для канонической формы над полем удобнее другой базис — мономы без отрицаний — с той же операцией сборки XOR (раздел 10.7).

Полином Жегалкина (АНФ) делает следующий шаг: разлагает функцию по *мономам* (произведениям переменных) как по базису, с операцией XOR в качестве сложения. В этом базисе каждая функция единственным образом представляется как XOR-сумма мономов (произведений подмножеств переменных, без отрицаний). Переход от СДНФ к АНФ — это смена базиса в пространстве размерности 2^n : минтермы (конъюнкции всех n переменных, с отрицаниями или без) заменяются на мономы (конъюнкции подмножеств переменных, без отрицаний).

Алгебраическая структура АНФ позволяет перенести методы линейной алгебры (матрицы, базисы, решение систем линейных уравнений) на булевы функции⁸³. Это используется в криптоанализе.

Пример Одна функция в двух базисах

Функция $f(x, y) = x \vee y$. СДНФ раскладывает её по минтермам:

$$f = (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \vee (x \wedge y),$$

а значения на наборах $(0, 0), (0, 1), (1, 0), (1, 1)$ — вектор $(0, 1, 1, 1)$ над $\text{GF}(2)$.

Полином Жегалкина раскладывает ту же функцию по мономам:

$$f = x \oplus y \oplus xy,$$

и на тех же наборах даёт тот же вектор: на $(1, 1)$ сумма $1 \oplus 1 \oplus 1 = 1$.

Переход от первого разложения ко второму — смена базиса в пространстве функций: минтермы $\bar{x}y, x\bar{y}, xy$ заменяются мономами x, y, xy .

§ 10.6 Минимизация

Совершенные нормальные формы, как правило, избыточны. Цель минимизации — найти представление с наименьшим числом литералов (или термов), эквивалентное исходной функции. Выигрыш прямой — меньше литералов, меньше вентиляей, меньше задержка, ниже энергопотребление.

10.6.1 Карты Карно

Метод, который мы сейчас рассмотрим, выглядит как раскрашивание клеточек в таблице. За этой визуальной простотой стоит нетривиальная алгебра. Каждая группа из 2^k смежных клеток — это алгебраический объект, а объединение таких групп решает NP-трудную задачу о минимальном покрытии. Метод Мориса Карно

⁸³И. И. Жегалкин. «О технике вычисления предложений в символической логике». Математический сборник, 1927.

(1953) — пример того, как правильное представление данных (код Грея для строк и столбцов) превращает алгебраическую манипуляцию в *геометрическую* интуицию. В течение двух десятилетий, до появления алгоритмических методов логического синтеза, карты Карно были основным инструментом инженеров-схемотехников.

Карта Карно — это визуальный метод минимизации, работающий для функций от небольшого числа переменных ($n \leq 4$). Его сила — в наглядности: группы смежных единиц на карте непосредственно видны как упрощённые конъюнкции.

ОПРЕДЕЛЕНИЕ 10.15 Карта Карно

Двумерная таблица истинности с упорядочиванием строк и столбцов в коде Грея называется **картой Карно**.

Благодаря коду Грея, соседние (смежные по стороне) клетки отличаются значением ровно одной переменной. Прямоугольная группа из 2^k смежных единичных клеток соответствует конъюнкции литералов. Число литералов равно $n - k$: это переменные, которые не меняются внутри группы.

Теперь формализуем то, что глаз видит на карте Карно: группа смежных единиц — это алгебраический объект.

ОПРЕДЕЛЕНИЕ 10.16 Простая импликанта (*prime implicant*)

Конъюнкция литералов называется **простой импликантой**, если она имплицирует функцию (всюду, где истинна конъюнкция, истинна и функция) и никакой её литерал нельзя удалить без нарушения импликации.

На карте Карно простая импликанта — это максимальная прямоугольная группа единичных клеток размера 2^k .

ОПРЕДЕЛЕНИЕ 10.17 Существенная простая импликанта (*essential prime implicant*)

Простая импликанта, покрывающая хотя бы одну единичную клетку, не покрываемую никакой другой простой импликантой, называется **существенной простой импликантой**.

Существенные импликанты обязаны входить в любое минимальное покрытие.

Пример Простые и существенные импликанты для функции большинства

Для функции большинства $f(x, y, z) = xy \vee xz \vee yz$, карта Карно которой изображена на рис. 55:

Простые импликанты — xy, xz, yz . Каждая из них — максимальная группа из $2^1 = 2$ клеток на карте Карно. Ни одну из них нельзя расширить (добавить ещё одну клетку), не потеряв импликацию.

Все три — существенные: импликанта xy покрывает клетку $xy\bar{z}$, которую не покрывают две другие. Аналогично xz покрывает $x\bar{y}z$, а yz — $\bar{x}yz$. Каждая существенная импликанта защищает «свою» единицу, которую больше никто не покрывает.

Минимальное покрытие обязано включать все три, что даёт формулу $xy \vee xz \vee yz$.

		x	
		0	1
yz	00		
	01		1
	11	1	1
	10		1

Рис. 55. Функция большинства $f(x, y, z) = xy \vee xz \vee yz$ на карте Карно.

Пример Группа из четырёх клеток — один литерал

Функция $f(x, y, z) = x$ равна 1 ровно на четырёх наборах — на всех, где $x = 1$. На карте Карно эти четыре клетки образуют один прямоугольник размера 2^2 . По правилу $n - k$ литералов ($n = 3$ переменных, $k = 2$) остаётся $3 - 2 = 1$ литерал — сам x . Четыре минтерма $x\bar{y}\bar{z}$, $x\bar{y}z$, $xy\bar{z}$, xyz склеиваются в один терм x . Группа из 2^k клеток всегда убирает k переменных из терма.

АЛГОРИТМ Минимизация по карте Карно

1. Выделить все простые импликанты — максимальные прямоугольные группы размера 2^k из смежных единичных клеток.
2. Отметить существенные простые импликанты — те, что покрывают хотя бы одну единицу, не покрываемую никакой другой импликантой. Существенные импликанты обязаны войти в минимальное покрытие.
3. Покрыть оставшиеся непокрытые единицы минимальным набором остальных импликант.
4. Неопределённые условия (don't cares, $\times$) можно считать как 0, так и 1 для получения более крупных групп.
5. Карты Карно эффективны для $n \leq 4$. Для большего числа переменных необходимы алгоритмические методы.

Разметка пустой карты Карно для четырёх переменных показана на рис. 56. Строки и столбцы упорядочены в коде Грея, так что соседние клетки отличаются ровно одной переменной.

		wx			
		00	01	11	10
yz	00				
	01				
	11				
	10				

Рис. 56. Пустая карта Карно для четырёх переменных.

10.6.2 Алгоритм Квайна — МакКласки

Для функций от многих переменных применяется табличный алгоритм Квайна — МакКласки. Он систематически находит все простые импликанты и затем решает задачу о покрытии.

АЛГОРИТМ Квайна — МакКласки

1. Записать все минтермы (входные наборы, на которых функция равна 1) в двоичном виде.
2. **Фаза склеивания:** перебираем пары минтермов, отличающихся ровно в одном бите. Склеиваем их, заменяя различающийся бит на прочерк (don't care). Повторяем итеративно, пока возможны склеивания. Несклеенные термы — простые импликанты.
3. **Фаза покрытия:** составить таблицу покрытия (строки — простые импликанты, столбцы — исходные минтермы) и выбрать минимальное по числу импликант подмножество строк, покрывающее все столбцы. Это задача о покрытии множества, в общем случае NP-трудная, но на практике решается эвристиками⁸⁴ для типичных размеров.

Проверка Минимизация булевых функций

1. Почему существенная простая импликанта обязана войти в любое минимальное покрытие?
2. Чем алгоритм Квайна — МакКласки отличается от карты Карно и когда он нужен?

⁸⁴R. Brayton et al. «Logic Minimization Algorithms for VLSI Synthesis». Kluwer, 1984. ABC: <https://github.com/berkeley-abc/abc>.

§ 10.7 Полином Жегалкина

ДНФ и КНФ выражают функцию через $\wedge$, $\vee$, $\neg$. Но существует и другой канонический вид — алгебраическая нормальная форма (АНФ), основанная на операции XOR (сложение по модулю 2).

ОПРЕДЕЛЕНИЕ 10.18 Полином Жегалкина

Представление булевой функции в виде

$$f(x_1, \dots, x_n) = \bigoplus_{S \subseteq \{1, \dots, n\}} a_S \cdot \prod_{i \in S} x_i,$$

где $a_S \in \{0, 1\}$ — коэффициенты, а XOR — сложение по модулю 2, называется **полиномом Жегалкина**.

Слагаемое полинома — это произведение (конъюнкция) некоторого подмножества переменных, умноженное на коэффициент 0 или 1.

Пример АНФ для базовых функций

- $x \wedge y = xy$ (коэффициент $a_{\{x,y\}} = 1$, остальные $a_S = 0$).
- $x \vee y = x \oplus y \oplus xy$ — три ненулевых коэффициента.
- $\neg x = x \oplus 1$ — два ненулевых коэффициента ($a_x = 1$, $a_0 = 1$).
- $x \oplus y$ уже записана в АНФ: $a_x = 1$, $a_y = 1$.

Определение полинома Жегалкина немедленно ставит вопрос: единственно ли такое представление? Ответ положительный — и это принципиальное отличие от ДНФ/КНФ, которые допускают множество эквивалентных форм.

ТЕОРЕМА 10.10 Единственность АНФ

Всякая булева функция имеет ровно один полином Жегалкина.

Набросок доказательства

Докажем подсчётом: число АНФ совпадает с числом булевых функций, а построение возможно для любой функции.

Полином Жегалкина от n переменных состоит из мономов, соответствующих подмножествам переменных. Каждый из 2^n возможных мономов может либо входить в полином (коэффициент 1), либо нет (коэффициент 0). Следовательно, существует ровно 2^{2^n} различных АНФ — столько же, сколько и булевых функций. Поскольку алгоритмы построения АНФ (метод неопределённых коэффициентов, метод треугольника Паскаля) дают представление для любой функции. Количество различных АНФ совпадает с количеством функций, поэтому представление единственно.

Единственность — сильное свойство: полином Жегалкина является канонической формой, как и СДНФ. Но в отличие от СДНФ, он использует $\oplus$, что делает его естественным для алгебраических манипуляций над полем $\text{GF}(2)$.

За единственностью стоит выбор операции сложения. XOR над полем $\text{GF}(2)$ — это сложение в поле, а многочлен над полем однозначно определяется своими коэффициентами. Дизъюнкция идемпотентна: $x \vee x = x$. Слагаемое в ДНФ можно продублировать, термы перегруппировать, а поглощаемый терм добавить — значение формулы останется прежним, поэтому эквивалентных ДНФ у функции бесконечно много. Среди них минимизация ищет запись с наименьшим числом литералов. Замена дизъюнкции на $\oplus$ уничтожает идемпотентность, а вместе с ней — и неоднозначность представления.

Существует три стандартных метода построения полинома Жегалкина:

- *Метод неопределённых коэффициентов* — для любой функции, заданной таблицей.
- *Эквивалентные преобразования* — для функции, уже записанной формулой.
- *Треугольник Паскаля* — самый быстрый для ручного вычисления по таблице истинности.

10.7.1 Метод неопределённых коэффициентов

Идея метода — записать АНФ в общем виде с неизвестными коэффициентами и найти их, подставляя конкретные значения переменных. Для функции n переменных общий вид содержит 2^n слагаемых — по одному на каждое подмножество переменных.

АЛГОРИТМ Метод неопределённых коэффициентов

1. Записать общий вид АНФ: $f(x_1, \dots, x_n) = \bigoplus_{S \subseteq \{1, \dots, n\}} a_S \cdot \prod_{i \in S} x_i$, где $a_S \in \{0, 1\}$ — неизвестные коэффициенты.
2. Подставить каждый из 2^n входных наборов в это равенство. Для каждого набора левая часть известна (значение функции), правая — линейное выражение над $\text{GF}(2)$ относительно a_S .
3. Решить полученную систему 2^n линейных уравнений с 2^n неизвестными над полем $\text{GF}(2)$.

Пример АНФ для $x \rightarrow y$ методом неопределённых коэффициентов

Функция $f(x, y) = x \rightarrow y = \neg x \vee y$. Таблица истинности:

- $f(0, 0) = 1$
- $f(0, 1) = 1$
- $f(1, 0) = 0$
- $f(1, 1) = 1$

Общий вид АНФ двух переменных: $f(x, y) = a_0 \oplus a_x x \oplus a_y y \oplus a_{xy} xy$.

Подставляем наборы:

- $(0, 0) : f(0, 0) = a_0 = 1 \Rightarrow a_0 = 1.$
- $(0, 1) : f(0, 1) = a_0 \oplus a_y = 1 \oplus a_y = 1 \Rightarrow a_y = 0.$
- $(1, 0) : f(1, 0) = a_0 \oplus a_x = 1 \oplus a_x = 0 \Rightarrow a_x = 1.$
- $(1, 1) : f(1, 1) = a_0 \oplus a_x \oplus a_y \oplus a_{xy} = 1 \oplus 1 \oplus 0 \oplus a_{xy} = 0 \oplus a_{xy} = 1 \Rightarrow a_{xy} = 1.$

Итак, $x \rightarrow y = 1 \oplus x \oplus xy$. Проверим: $1 \oplus x \oplus xy = (1 \oplus x) \oplus xy = \neg x \oplus xy$. Действительно, $\neg x \vee y = \neg x \oplus (xy)$ — известное тождество.

10.7.2 Метод эквивалентных преобразований

Если функция уже задана формулой, АНФ можно получить алгебраически — заменой операций $\vee$ и $\neg$ на выражения через $\oplus$ и $\wedge$. Метод опирается на два тождества над полем $\text{GF}(2)$:

УТВЕРЖДЕНИЕ 10.11 Выражение NOT и OR через XOR

$$\begin{aligned}\bar{x} &= x \oplus 1 \\ x \vee y &= x \oplus y \oplus xy.\end{aligned}$$

Доказательство

Докажем проверкой всех наборов.

Первое: $x \oplus 1$ равно $0 \oplus 1 = 1$ при $x = 0$ и $1 \oplus 1 = 0$ при $x = 1$, что совпадает с таблицей истинности отрицания.

Второе: проверим четыре набора (x, y) . Если $(x, y) = (0, 0)$, то $0 \oplus 0 \oplus (0 \wedge 0) = 0$, и $0 \vee 0 = 0$. Если $(x, y) = (0, 1)$, то $0 \oplus 1 \oplus (0 \wedge 1) = 1$, и $0 \vee 1 = 1$. Если $(x, y) = (1, 0)$, то $1 \oplus 0 \oplus (1 \wedge 0) = 1$, и $1 \vee 0 = 1$. Если $(x, y) = (1, 1)$, то $1 \oplus 1 \oplus (1 \wedge 1) = 0 \oplus 1 = 1$, и $1 \vee 1 = 1$. Значения совпадают на всех четырёх наборах.

АЛГОРИТМ Метод эквивалентных преобразований

1. Заменить каждое отрицание $\bar{\varphi}$ на $\varphi \oplus 1$.
2. Заменить каждую дизъюнкцию $\varphi \vee \psi$ на $\varphi \oplus \psi \oplus (\varphi \wedge \psi)$.
3. Раскрыть скобки, используя дистрибутивность $\wedge$ относительно $\oplus$: $x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$.
4. Упростить, используя $x \oplus x = 0$, $x \wedge x = x$.

Пример АНФ для $(x \vee y) \wedge \bar{z}$ методом преобразований

$$(x \vee y) \wedge \bar{z} = (x \oplus y \oplus xy) \wedge (z \oplus 1) = (x \wedge z) \oplus (y \wedge z) \oplus (xy \wedge z) \oplus x \oplus y \oplus xy.$$

Коэффициенты:

- $a_x = 1$

- $a_y = 1$
- $a_{xy} = 1$
- $a_{xz} = 1$
- $a_{yz} = 1$
- $a_{xyz} = 1$

остальные нули.

10.7.3 Метод треугольника Паскаля

Этот метод строит АНФ непосредственно по столбцу значений функции и не требует ни решения систем, ни алгебраических преобразований. Он работает для функции, заданной таблицей истинности, и особенно быстр при ручном вычислении.

АЛГОРИТМ Метод треугольника Паскаля

1. Выписать столбец значений функции f для всех 2^n наборов в лексикографическом порядке: от $(0, \dots, 0)$ до $(1, \dots, 1)$.
2. Построить треугольник: для каждой соседней пары значений вычислить XOR и записать результат между ними строкой ниже. Повторять, пока не останется один элемент.
3. Коэффициенты АНФ — это первые элементы каждой строки построенного треугольника (включая самую первую строку — исходный столбец).

Пример АНФ для $x \rightarrow y$ методом треугольника Паскаля

Таблица истинности $f(x, y) = x \rightarrow y$: значения (сверху вниз по наборам $(0, 0), (0, 1), (1, 0), (1, 1)$): 1, 1, 0, 1.

Строим треугольник: каждая следующая строка — XOR соседних элементов предыдущей.

1	1	0	1
	0	1	1
		1	0
			1

Первые элементы строк (жирным) дают коэффициенты в порядке двоичных индексов (константа, y , x , xy):

- $a_0 = 1$
- $a_y = 0$
- $a_x = 1$
- $a_{xy} = 1$

Итак, $f(x, y) = 1 \oplus x \oplus xy$ — совпадает с результатом метода неопределённых коэффициентов.

Взгляд на булевы функции через призму алгебры над полем $\text{GF}(2)$ — с операцией XOR вместо OR — открывает приложения далеко за пределами схемотехники.

10.7.4 Полином Жегалкина в криптоанализе

Полином Жегалкина применяется в криптоанализе — дисциплине, изучающей методы взлома криптографических алгоритмов. Большинство современных симметричных шифров (AES, Serpent) основаны на комбинации двух типов операций:

1. линейных — XOR, побитовые сдвиги.
2. нелинейных — S-блоки (таблицы подстановки, заменяющие m -битный вход на n -битный выход по фиксированной таблице).

Линейный криптоанализ (Matsui, 1993)⁸⁵: криптоаналитик ищет линейные соотношения между битами открытого текста, шифротекста и ключа, выполняющиеся с вероятностью, отличной от $\frac{1}{2}$. Каждый раунд шифра аппроксимируется линейной булевой функцией (XOR-суммой подмножества битов), и отклонение от $\frac{1}{2}$ накапливается по раундам.

Нелинейность булевой функции — это расстояние Хэмминга между ней и ближайшей *аффинной* функцией. (Расстояние Хэмминга — количество позиций, в которых различаются таблицы истинности. Аффинная функция — XOR-сумма подмножества переменных плюс, возможно, константа 1.) Чем выше нелинейность S-блока, тем меньше максимальная вероятность любой линейной аппроксимации — и тем больше данных нужно криптоаналитику для атаки.

Алгебраические атаки⁸⁶: если S-блок имеет АНФ низкой степени (мало переменных в мономах, немного слагаемых), то вся система шифрования сводится к системе полиномиальных уравнений над $\text{GF}(2)$. Решать такие системы можно алгоритмами типа XL или вычислением базиса Грёбнера.

Поэтому S-блоки современных шифров проектируются так, чтобы их АНФ имела высокую степень и много мономов. Конкретно, S-блок AES имеет АНФ степени 7 из 8 переменных — более сотни слагаемых в каждой координате⁸⁷.

§ 10.8 Диаграммы решений (BDD)

СДНФ представляет булеву функцию как дизъюнкцию минтермов. АНФ — как XOR-сумму мономов. Оба представления экспоненциальны в худшем случае: функция

⁸⁵M. Matsui. «Linear Cryptanalysis Method for DES Cipher». EUROCRYPT, 1993.

⁸⁶C. Cid, S. Murphy, M. Robshaw. «Algebraic Aspects of the Advanced Encryption Standard». 2006.

⁸⁷J. Daemen, V. Rijmen. «The Design of Rijndael». 2002.

большинства от n переменных требует экспоненциального размера и в ДНФ, и в КНФ, и в АНФ.

Существует третье каноническое представление — Binary Decision Diagram (BDD) — которое для многих практически важных функций оказывается компактным. BDD — это ориентированный ациклический граф, в котором каждая вершина помечена переменной и имеет два выхода: пунктирный (переменная равна 0) и сплошной (переменная равна 1). Листья помечены константами 0 или 1. Значение функции на заданном наборе переменных вычисляется спуском от корня к листу: на каждом шаге читаем значение переменной в вершине и идём по соответствующему ребру.

ОПРЕДЕЛЕНИЕ 10.19 BDD

BDD для булевой функции $f(x_1, \dots, x_n)$ — это ориентированный ациклический граф, в котором:

- Единственный корень (стартовая вершина).
- Каждая внутренняя вершина помечена переменной x_i и имеет два исходящих ребра: lo (пунктирное, $x_i = 0$) и hi (сплошное, $x_i = 1$).
- Листья помечены 0 или 1.

BDD сам по себе не является каноническим представлением: для одной функции можно построить много разных BDD, различающихся порядком переменных и наличием избыточных вершин. Два дополнительных ограничения — фиксированный порядок переменных и удаление избыточности — превращают BDD в каноническую форму.

ОПРЕДЕЛЕНИЕ 10.20 ROBDD

ROBDD (Reduced Ordered BDD) — BDD с двумя дополнительными ограничениями:

- **Упорядоченность** (Ordered): переменные на любом пути от корня к листу следуют в фиксированном порядке $x_1 < x_2 < \dots < x_n$ (некоторые переменные могут пропускаться).
- **Редуцированность** (Reduced): граф не содержит избыточных вершин (вершин с одинаковыми lo- и hi-потомками) и изоморфных подграфов. Разные вершины с одинаковой переменной и одинаковыми потомками склеиваются.

ROBDD является *каноническим представлением* булевой функции: для фиксированного порядка переменных каждая булева функция имеет ровно один ROBDD.

Построение BDD опирается на *разложение Шеннона*: для любой булевой функции $f(x_1, \dots, x_n)$ и переменной x_i выполняется

$$f = (\overline{x_i} \wedge f[0/x_i]) \vee (x_i \wedge f[1/x_i]),$$

Здесь $f[0/x_i]$ — функция f с подстановкой $x_i = 0$. А $f[1/x_i]$ — с подстановкой $x_i = 1$. Это разложение — комбинаторный факт: функция однозначно определяется

своим поведением на двух половинах, где переменная равна 0 или 1. В BDD оно представлено явно: lo-ребро ведёт к подграфу для $f[0/x_i]$, hi-ребро — к подграфу для $f[1/x_i]$.

Пример BDD для $f(x, y) = x \oplus y$

Разложение Шеннона по x :

- $f[0/x] = 0 \oplus y = y$,
- $f[1/x] = 1 \oplus y = \bar{y}$.

Корень BDD — вершина x . Пунктирное lo-ребро ($x = 0$) ведёт к подграфу для y : лист 0 при $y = 0$, лист 1 при $y = 1$. Сплошное hi-ребро ($x = 1$) ведёт к подграфу для $\bar{y}$: лист 1 при $y = 0$, лист 0 при $y = 1$. Оба подграфа ссылаются на одну и ту же переменную y , но с разными выходами — это разные узлы в ROBDD.

Готовый ROBDD для $x \oplus y$ показан на рис. 57. Три узла-переменных и два листа, причём константные листья 0 и 1 используются совместно обоими подграфами.

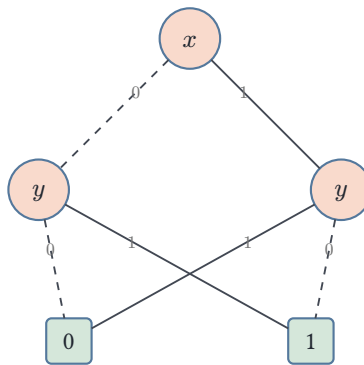


Рис. 57. ROBDD для $f(x, y) = x \oplus y$.

Сила BDD проявляется на функциях с регулярной структурой. BDD для суммы двух n -битных чисел с чередующимся порядком битов слагаемых $(a_1, b_1, a_2, b_2, \dots)$ имеет линейный размер, хотя её ДНФ экспоненциальна. Компактность BDD возникает из слияния одинаковых подграфов — две разные последовательности выборов, приводящие к одному результату, *схлопываются* в один узел. Экспоненциальное дерево решений превращается в граф, часто имеющий полиномиальный размер.



Сад расходящихся тропок

Борхес в рассказе «Сад расходящихся тропок»⁸⁸ описал ветвящуюся структуру решений за десятилетия до появления BDD. Но BDD отличаются от борхесовского сада слиянием: разные пути, ведущие к одному исходу, соединяются в общий узел.

⁸⁸J. L. Borges. «El jardín de senderos que se bifurcan». 1941.

Примечание BDD и верификация

ROBDD⁸⁹ лежат в основе символьной верификации (symbolic model checking). Множество состояний программы — это булева функция, закодированная как BDD.

Операции над множествами (объединение, пересечение, проверка пустоты) выполняются как операции над BDD за время, пропорциональное размеру BDD, а не количеству элементов множества. Именно это позволяет model checker'ам оперировать пространствами состояний, экспоненциально превосходящими размер доступной памяти.⁹⁰

Дальше — три детали, которые превращают ROBDD из канонического представления в практический инструмент: дополняющие рёбра, проблема порядка переменных и операции построения.

10.8.1 Дополняющие рёбра

Отрицание булевой функции — это та же диаграмма с переставленными листьями. Хранить отрицание отдельным подграфом расточительно: каждый узел удваивается.

ОПРЕДЕЛЕНИЕ 10.21 Дополняющие рёбра

Дополняющее ребро (complemented edge) несёт флаг отрицания: ребро с флагом указывает на узел, представляющий *отрицание* функции дочернего узла.

В ROBDD с дополняющими рёбрами константный лист один, а отрицание функции стоит ноль: достаточно переключить флаг на ребре. Число узлов сокращается примерно вдвое, а операция NOT становится бесплатной.

Дополняющие рёбра — стандартный приём современных библиотек диаграмм.

Пример Отрицание как перестановка листьев

Для функции $f(x, y) = x \wedge y$ ROBDD устроен так: корень x , пунктирное ребро ($x = 0$) ведёт в лист 0, а сплошное ($x = 1$) — в узел y , у которого пунктир ведёт в 0, а сплошной — в 1. Диаграмма для $\bar{f} = \bar{x} \vee \bar{y}$ имеет ту же форму: меняются только листья, 0 и 1 меняются местами.

Дополняющее ребро хранит одну диаграмму, а отрицание помечает флагом: отдельный подграф для $\bar{f}$ не нужен.

⁸⁹R. E. Bryant. «Graph-Based Algorithms for Boolean Function Manipulation». IEEE Transactions on Computers, 1986.

⁹⁰E. M. Clarke, O. Grumberg, D. Peled. «Model Checking». 1999.

10.8.2 Проблема порядка переменных

Размер ROBDD критически зависит от порядка переменных. Один порядок даёт линейную диаграмму, другой — экспоненциальную.

Пример Сумматор: линейный ROBDD

ROBDD для суммы двух n -битных чисел с порядком, чередующим биты слагаемых $(a_1, b_1, a_2, b_2, \dots)$, имеет линейный размер: перенос распространяется по цепочке узлов, и каждый узел соответствует одной позиции. ДНФ такой функции экспоненциальна — а диаграмма линейна.

Пример Умножитель: экспоненциальный ROBDD

ROBDD для произведения двух n -битных чисел экспоненциален при *любом* порядке переменных. Это доказанный факт: функция умножения не допускает компактной BDD. Порядок переменных — отдельная оптимизационная задача, решаемая эвристиками, которые опираются на структуру формулы.

10.8.3 Операции построения: apply

ROBDD строятся комбинированием меньших диаграмм. Операция $\text{apply}(f, g, \text{op})$ обходит диаграммы f и g параллельно и строит диаграмму f *op* g с использованием таблицы уникальности и кэша. Каждой булевой бинарной операции соответствует свой *apply*: объединение, пересечение, XOR — всё это *apply* с разными операциями на листьях. Благодаря каноничности построение выполняется за время, пропорциональное размеру результата, — и результат сразу редуцирован.

Ту же идею обобщают до тернарной операции $\text{ite}(f, g, h)$: результат равен g , когда f истинно, и h , когда f ложно. Бинарные операции становятся её частными случаями: $f \wedge g = \text{ite}(f, g, 0)$, $f \vee g = \text{ite}(f, 1, g)$, $f \oplus g = \text{ite}(f, |g|, g)$. Применение *ite* вместо *apply* — стандартный приём современных библиотек диаграмм.

ИСТОРИЯ Три рождения диаграмм

Идея диаграмм решений старше, чем кажется. С. Y. Lee в 1959 году описал *binary decision diagrams*⁹¹. Шелдон Акирс (Sheldon B. Akers) в 1978 году независимо ввёл их снова, уже под именем «Binary Decision Diagrams». Оба знали о каноничности, но не умели эффективно строить и комбинировать диаграммы. Рэндел Брайант (Randal E. Bryant) в 1986 году добавил редукцию и операции *apply* — и ROBDD стал практическим инструментом верификации. С тех пор диаграммы решений — основной способ представления булевых функций в символьной верификации и статическом анализе.

⁹¹C. Y. Lee. «Representation of Switching Circuits by Binary-Decision Programs». Bell System Technical Journal, 1959.

§ 10.9 * Теорема Стоуна о представлении

В разделе 10.1.4 мы определили булеву алгебру аксиоматически: коммутативность, ассоциативность, дистрибутивность, нейтральные элементы, дополнение. Алгебры подмножеств — семейства множеств с операциями $\cup, \cap, \bar{}$ — удовлетворяют этим аксиомам (мы проверили это в примере сразу после определения). Всякая ли булева алгебра изоморфна некоторой алгебре подмножеств? Или аксиомы описывают более широкий класс структур и алгебры подмножеств — лишь частный случай?

Ответ дал Маршалл Стоун в 1936 году. Его теорема утверждает: *всякая булева алгебра изоморфна некоторому полю множеств — подалгебре булеана подходящего множества. Аксиомы не описывают ничего, кроме алгебр подмножеств. Доказательство вводит конструкцию, ставшую стандартным инструментом алгебраической логики: ультрафильтры.*

10.9.1 Фильтры и ультрафильтры

В алгебре подмножеств $\mathcal{P}(X)$ есть естественное понятие «большого» семейства: все подмножества, содержащие фиксированную точку $x \in X$. Такое семейство замкнуто по пересечению — пересечение двух множеств, содержащих x , снова содержит x — и замкнуто вверх: если множество содержит x , то любое его надмножество тоже содержит x . Это и есть *фильтр*.

В абстрактной булевой алгебре точек нет, но структура та же. Фильтр собирает элементы, которые «почти равны единице» в том смысле, что их пересечение снова в фильтре, а всё, что выше элемента фильтра, тоже попадает в фильтр. Формально, для булевой алгебры $(B, \wedge, \vee, |, 0, 1)$:

ОПРЕДЕЛЕНИЕ 10.22 Фильтр

Подмножество $F \subseteq B$ называется **фильтром**, если:

- $1 \in F$.
- $a, b \in F \Rightarrow a \wedge b \in F$ — замкнутость по конъюнкции.
- $a \in F, a \leq b \Rightarrow b \in F$ — замкнутость вверх.

Простейший пример — *главный фильтр (principal filter)*: для фиксированного $a \in B$ множество $F_a = \{x \in B \mid a \leq x\}$ всех элементов, лежащих выше a . В алгебре подмножеств $\mathcal{P}(X)$ это семейство всех надмножеств фиксированного множества A : всё, что содержит A .

Фильтр называется *собственным (proper)*, если $0 \notin F$. Собственный фильтр можно расширять — добавлять новые элементы, не впуская 0. Когда расширять дальше невозможно — это ультрафильтр.

Ультрафильтр делит алгебру пополам: для каждого $a \in B$ ровно один из $a, \bar{a}$ лежит в U . Если ни a , ни $\bar{a}$ не принадлежат фильтру, можно добавить a — и фильтр останется собственным. Значит, он ещё не был максимальным.

ОПРЕДЕЛЕНИЕ 10.23 Ультрафильтр

Ультрафильтр — максимальный собственный фильтр: собственный фильтр $U \subseteq B$, такой что никакой собственный фильтр F не содержит U строго. Эквивалентно: для каждого $a \in B$ либо $a \in U$, либо $\bar{a} \in U$ (и никогда оба).

ЛЕММА 10.12 Лемма о расширении фильтра

Всякий собственный фильтр $F \subseteq B$ содержится в некотором ультрафильтре $U \supseteq F$.

Набросок доказательства

Рассмотрим семейство $\mathcal{F}$ всех собственных фильтров, содержащих F , частично упорядоченное включением. Возьмём цепь фильтров $F_1 \subseteq F_2 \subseteq \dots$ в $\mathcal{F}$. Их объединение — снова собственный фильтр. 1 принадлежит всем F_i , поэтому 1 в объединении. 0 не принадлежит ни одному F_i , поэтому 0 не в объединении. Для любых двух элементов цепи найдётся F_k , содержащий оба (цепь линейно упорядочена), а F_k замкнут по $\wedge$ и вверх. Значит, объединение цепи — верхняя грань цепи в $\mathcal{F}$.

Условия леммы Цорна выполнены: всякая цепь имеет верхнюю грань. Следовательно, в $\mathcal{F}$ существует максимальный элемент — ультрафильтр, содержащий F .

Зачем нам лемма о расширении? Доказательство теоремы Стоуна требует разделять элементы: если $a \neq b$, нужно построить ультрафильтр, содержащий один, но не другой. Лемма о расширении — инструмент для такого построения: мы начинаем с подходящего фильтра и расширяем его до ультрафильтра, сохраняя нужные свойства.

Сама лемма опирается на аксиому выбора (через лемму Цорна)⁹². Для конечных булевых алгебр аксиома выбора не нужна: фильтр можно расширять явно, перебирая элементы один за другим.

10.9.2 Теорема Стоуна

Обозначим $\text{Ult}(B)$ множество всех ультрафильтров булевой алгебры B . Определим отображение $\varphi : B \rightarrow \mathcal{P}(\text{Ult}(B))$:

Каждый элемент $a \in B$ присутствует в одних ультрафильтрах и отсутствует в других. Соберём все ультрафильтры, содержащие a .

$$\varphi(a) = \{U \in \text{Ult}(B) \mid a \in U\}.$$

⁹²Лемма Цорна — эквивалентная форма аксиомы выбора: если в частично упорядоченном множестве всякая цепь имеет верхнюю грань, то существует максимальный элемент.

Отображение φ вкладывает абстрактную булеву алгебру B в конкретную алгебру подмножеств $\mathcal{P}(\text{Ult}(B))$. Элементы B становятся множествами ультрафильтров. Операции $\wedge$, $\vee$ и дополнение переходят в $\cap$, $\cup$ и вычитание из $\text{Ult}(B)$.

ТЕОРЕМА 10.13 Стоун, 1936 — Теорема о представлении булевых алгебр

Для всякой булевой алгебры B отображение

$$\begin{aligned}\varphi : B &\rightarrow \mathcal{P}(\text{Ult}(B)), \\ \varphi(a) &= \{U \in \text{Ult}(B) \mid a \in U\}\end{aligned}$$

является изоморфизмом B на поле множеств $\text{Im}(\varphi) \subseteq \mathcal{P}(\text{Ult}(B))$. В частности, φ сохраняет все операции:

$$\begin{aligned}\varphi(a \wedge b) &= \varphi(a) \cap \varphi(b), \\ \varphi(a \vee b) &= \varphi(a) \cup \varphi(b), \\ \varphi(\bar{a}) &= \text{Ult}(B) \setminus \varphi(a), \\ \varphi(0) &= \emptyset, \\ \varphi(1) &= \text{Ult}(B).\end{aligned}$$

Доказательство

Докажем, что φ — инъективный гомоморфизм.

Гомоморфизм. Для каждой операции проверим сохранение.

Ультрафильтр содержит $a \wedge b$ тогда и только тогда, когда содержит и a , и b (замкнутость вверх и по $\wedge$), следовательно,

$$\varphi(a \wedge b) = \varphi(a) \cap \varphi(b).$$

Из свойства ультрафильтра содержать ровно один из x , $\bar{x}$ и из законов де Моргана следует, что

$$\varphi(a \vee b) = \varphi(a) \cup \varphi(b).$$

Каждый ультрафильтр содержит ровно один из a , $\bar{a}$, поэтому

$$\varphi(\bar{a}) = \text{Ult}(B) \setminus \varphi(a).$$

Наконец, 0 не принадлежит ни одному ультрафильтру, тогда как 1 принадлежит всем:

$$\varphi(0) = \emptyset, \varphi(1) = \text{Ult}(B).$$

Инъективность. Пусть $a \neq b$. В булевой алгебре $a \leq b \leftrightarrow a \wedge \bar{b} = 0$. Это же условие записывается как $a \vee b = b$. Если элементы не равны, хотя бы одно из $a \wedge \bar{b} \neq 0$, либо $\bar{a} \wedge b \neq 0$. Иначе выполнялись бы оба неравенства $a \leq b$ и $b \leq a$

одновременно, откуда следовало бы $a = b$. Без ограничения общности положим $a \wedge \bar{b} \neq 0$.

Рассмотрим главный фильтр $F_{a \wedge \bar{b}} = \{x \mid a \wedge \bar{b} \leq x\}$. Он собственный: $a \wedge \bar{b} \neq 0$, поэтому $0 \notin F_{a \wedge \bar{b}}$. По лемме о расширении, $F_{a \wedge \bar{b}}$ содержится в некотором ультрафильтре U .

Что попало в U ? $a \in U$ — поскольку $a \wedge \bar{b} \leq a$, а фильтр замкнут вверх. $\bar{b} \in U$ — поскольку $a \wedge \bar{b} \leq \bar{b}$. Из последнего: $b \notin U$ — ультрафильтр не может содержать и b , и $\bar{b}$ одновременно.

Итак, $a \in U, b \notin U$. Следовательно, $U \in \varphi(a), U \notin \varphi(b)$. Значит, $\varphi(a) \neq \varphi(b)$.

Теорема Стоуна даёт точный ответ на вопрос, с которого мы начали. Всякая булева алгебра изоморфна алгебре подмножеств — аксиомы из раздела 10.1.4 не описывают ничего, кроме алгебр подмножеств. Как следствие: любое тождество, истинное во всех алгебрах подмножеств, выводимо из аксиом, и наоборот — аксиомы полны.

Пример Конечная булева алгебра

Пусть $B = \{0, 1\}^2$ — булева алгебра с покомпонентными операциями. Элементы: 00, 01, 10, 11 — от нуля до единицы. Порядок: $a \leq b \leftrightarrow a_i \leq b_i$ для всех битов. Например, $01 \leq 11$, но 01 и 10 несравнимы.

Атомы — минимальные ненулевые элементы: 01 и 10. В конечной булевой алгебре каждый ультрафильтр — главный: он порождается ровно одним атомом.

Для атома 01:

$$U_{01} = \{x \in B \mid 01 \leq x\} = \{01, 11\}$$

Для атома 10:

$$U_{10} = \{10, 11\}$$

Других ультрафильтров в B нет.

Отображение φ сопоставляет каждому элементу множество содержащих его ультрафильтров. Результат — четыре множества, по одному на каждый элемент B :

Элемент a	$\varphi(a)$	Интерпретация
00	$\emptyset$	нуль не лежит ни в каком ультрафильтре
01	$\{U_{01}\}$	только первый ультрафильтр содержит 01
10	$\{U_{10}\}$	только второй ультрафильтр содержит 10
11	$\{U_{01}, U_{10}\}$	единица лежит в обоих ультрафильтрах

Проверим гомоморфизм: $\varphi(01 \wedge 10) = \varphi(00) = \emptyset = \{U_{01}\} \cap \{U_{10}\}$ (пересечение пусто, как и должно быть).

$$\varphi(01 \vee 10) = \varphi(11) = \{U_{01}, U_{10}\} = \{U_{01}\} \cup \{U_{10}\}.$$

$\text{Im}(\varphi) = \{\emptyset, \{U_{01}\}, \{U_{10}\}, \{U_{01}, U_{10}\}\}$ — алгебра всех подмножеств двухэлементного множества, то есть $\mathcal{P}(\{1, 2\})$.

Алгебра изоморфна $\mathcal{P}(\{1, 2\})$: четырёхэлементная булева алгебра — это всегда булеан двухэлементного множества.

СЛЕДСТВИЕ 10.14 Конечные булевы алгебры

Всякая конечная булева алгебра изоморфна $\mathcal{P}(X)$ для некоторого конечного множества X . В частности,

$$|B| = 2^n,$$

где n — число атомов (минимальных ненулевых элементов) алгебры B .

Набросок доказательства

Следствие теоремы Стоуна: алгебра изоморфна полю множеств, а конечное поле множеств — булеан множества своих атомов.

10.9.3 Теорема Стоуна и полнота аксиом

Аксиомы булевой алгебры из раздела 10.1.4 определяют, какие равенства между термами мы можем доказывать. Например, равенство $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ выводится из аксиом (это одна из них). Равенство $a \wedge (a \vee b) = a$ (закон поглощения) тоже выводится, хотя и не является аксиомой непосредственно. А равенство $a \wedge b = a \vee b$ невыводимо — оно ложно в алгебре $\{0, 1\}$.

Возникает вопрос: всё ли, что истинно во *всех* булевых алгебрах, выводимо из аксиом? Или существуют равенства, истинные в каждой конкретной модели, но недоказуемые формально?

Теорема Стоуна отвечает на этот вопрос утвердительно. Поскольку всякая булева алгебра изоморфна алгебре подмножеств, любое равенство, истинное во всех алгебрах подмножеств, истинно и во всех булевых алгебрах. По теореме Биркгофа о полноте эквациональной логики всякое равенство, истинное во всех булевых алгебрах, выводимо из аксиом. Аксиомы полны: они доказывают всё, что верно во всех моделях.

Тот же феномен в других алгебраических структурах: теорема Кэли (всякая группа — подгруппа группы перестановок) играет ту же роль для теории групп. Аксиомы не описывают ничего, что не представимо в конкретной форме: всякая булева алгебра вкладывается в алгебру подмножеств, всякая группа — в группу перестановок.

Замечание Двойственность Стоуна

Множество ультрафильтров $\text{Ult}(B)$ допускает топологию: база — семейство множеств $\{\varphi(a) \mid a \in B\}$. Полученное пространство — вполне несвязное компактное хаусдорфово (стоуново пространство). Возникает *двойственность Стоуна*: категория булевых алгебр эквивалентна категории стоуновых пространств. Каждой булевой алгебре соответствует её стоуново пространство ультрафильтров. Каждому стоунову пространству — булева алгебра его открыто-замкнутых подмножеств.

§ 10.10 Итоги главы

Булева алгебра превращает логические связки в операции: конъюнкция становится умножением, дизъюнкция — сложением, отрицание — дополнением. Логика, которой мы рассуждали, теперь работает как арифметика: условие в программе, поисковый запрос, побитовая маска — всё это вычисления над нулями и единицами. Над функциями от n переменных — их всего 2^{2^n} — можно вести вычисления, и уже при $n = 6$ их больше, чем зёрен песка на Земле.

Таблица истинности задаёт функцию однозначно, но представление можно выбирать. СДНФ, карта Карно, полином Жегалкина, диаграммы решений — каждый базис раскладывает функцию по-своему, и от выбора зависят единственность и размер представления. Полином Жегалкина единственен, а ДНФ допускает много эквивалентных форм — и минимизация сводится к склеиванию: карта Карно делает это на глаз, алгоритм Квайна — МакКласки — алгоритмически, а существенные импликанты образуют ядро покрытия.

Вопрос о полноте набора операций получает исчерпывающий ответ: пять замкнутых классов, и набор связок полон ровно тогда, когда не содержится целиком ни в одном из них. Поэтому одного NAND достаточно для любой схемы, а набора И и ИЛИ — нет: они порождают только монотонные функции, даже отрицание из них не собрать.

Теорема Стоуна вернула нас к аксиомам: всякая булева алгебра оказывается алгеброй подмножеств, и категория булевых алгебр эквивалентна категории стоуновых пространств. Инженерная отдача конкретна: КМОП-технология собирает NAND из четырёх транзисторов, и одного этого вентиля хватает для любой схемы. Следующий шаг — глава 11, где булева функция обретает физическую реализацию из вентилях.

Проверка Проверьте себя

1. Что означает функциональная полнота набора вентилях? Почему одного NAND достаточно, а набора $\{\text{И}\}, \{\text{ИЛИ}\}$ — нет?

2. Что такое критерий Поста и почему набор, не содержащийся ни в одном из пяти классов, полон?
3. Какие из представлений функции — СДНФ, ДНФ, полином Жегалкина, ROBDD — единственны?
4. Что такое минтерм и как из минтермов собирается СДНФ?
5. Почему на карте Карно соседние клетки отличаются ровно одной переменной и как это помогает склеивать термы?
6. Почему булевых функций от n переменных ровно 2^{2^n} , а не 2^n ?

Что дальше?

От абстрактной алгебры логики мы переходим к её физическому воплощению — логическим схемам. В главе 11 мы построим сумматоры, изучим универсальность NAND-вентилей и свяжем инженерный взгляд на вычисления с булевой основой, заложенной здесь.

§ 10.11 Упражнения

Булевы функции и таблицы истинности.

1. Постройте таблицу истинности для функции $f(x, y, z) = (x \vee \neg y) \wedge (y \oplus z)$. Сколько в ней строк и на скольких функция принимает значение 1?
2. Сколько существует различных булевых функций от трёх переменных? От четырёх? Объясните формулу 2^{2^n} .
3. * Среди 16 бинарных булевых функций найдите все, которые являются линейными (принадлежат классу L). Запишите каждую в виде $a_0 \oplus a_1x \oplus a_2y$.

Функциональная полнота и критерий Поста.

4. Выразите NOT, AND и OR через штрих Шеффера ($\uparrow$). Почему из этого следует, что множество {NAND} функционально полно?
5. Проверьте с помощью критерия Поста, является ли множество {XOR, NOT, 1} функционально полным. Для каждого из пяти классов укажите, принадлежит ли ему каждая функция множества.
6. * Проверьте через критерий Поста, является ли множество {IMPLY, 0} функционально полным. Для каждого класса, которому множество не принадлежит, укажите функцию-свидетеля.

Нормальные формы.

7. Постройте СДНФ и СКНФ для функции $f(x, y, z)$, равной 1 ровно на наборах $(0, 0, 0)$, $(0, 1, 1)$, $(1, 0, 1)$, $(1, 1, 0)$. Сравните размеры полученных форм.

8. Выразите импликацию $x \rightarrow y$ через $\neg$ и $\vee$. Постройте для неё СДНФ и СКНФ и сравните их размер.
9. * Докажите, что СДНФ функции $f(x_1, \dots, x_n)$ единственна с точностью до порядка термов. Указание: каждый минтерм определяется ровно одной строкой таблицы истинности.

Минимизация.

10. Минимизируйте функцию $f(x, y, z) = \neg x \neg y \neg z \vee \neg x y \neg z \vee x \neg y \neg z \vee x y \neg z \vee x y z$ с помощью карты Карно. Сколько получилось простых импликант и какие из них существенные?
11. * Функция большинства от трёх переменных $f(x, y, z) = xy \vee xz \vee yz$ изображена на карте Карно на рис. 55. Покажите, что все три импликанты — xy, xz, yz — существенные. Почему ни одну из них нельзя удалить из минимального покрытия?

Полином Жегалкина (АНФ).

12. Постройте полином Жегалкина для $f(x, y) = x \rightarrow y$ двумя способами: методом неопределённых коэффициентов и методом треугольника Паскаля. Убедитесь, что результаты совпадают.
13. Постройте полином Жегалкина для $f(x, y, z) = (x \vee \neg y) \wedge z$ методом эквивалентных преобразований. Сколько мономов содержит результат?
14. * Почему полином Жегалкина единственен для каждой функции, а ДНФ — нет? Объясните различие через свойства операций XOR и OR.

BDD.

15. Постройте ROBDD для $f(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$ с порядком переменных $x < y < z$. Сколько узлов в диаграмме? Сравните с числом строк таблицы истинности.
16. * Объясните, почему размер ROBDD критически зависит от порядка переменных. Приведите пример функции от трёх переменных, для которой разные порядки дают разное число узлов.
17. * Докажите, что функция $f(x_1, \dots, x_n) = x_1 \oplus x_2 \oplus \dots \oplus x_n$ линейна (принадлежит классу L). При каких n она самодвойственна? При каких n сохраняет 0, а при каких — сохраняет 1?

ПРОЕКТ Минимизатор ДНФ: алгоритм Квайна — МакКласки

Соберите минимизатор ДНФ, который по таблице истинности или списку минтермов находит минимальную по числу импликант формулу. Соберите его из двух фаз: склеивание минтермов, отличающихся одной переменной, даёт все простые импликанты, а выбор минимального покрытия таблицы — это задача о покрытии множества, в общем случае NP-трудная. Карта Карно справляется

на глаз для $n \leq 4$. Алгоритм Квайна — МакКласки работает таблично и для большего числа переменных.

① *Фаза склеивания*

Задайте функцию списком единичных наборов или таблицей истинности, а импликанту — кубом: задействованные переменные со значениями и безразличные позиции. Реализуйте итеративное попарное склеивание кубов с одинаковой маской безразличия, отличающихся ровно в одном задействованном бите, заменяя различающийся бит прочерком. Объясните, как проверить, что куб покрывает минтерм, и почему повторение склеивания до неподвижной точки даёт все простые импликанты. Проверьте индикатор нечётного числа единиц: почему его минтермы попарно отличаются минимум в двух переменных и склеек нет вовсе?

② *Минимальное покрытие*

Постройте таблицу покрытия: строки — простые импликанты, столбцы — единичные наборы. Отметьте существенные импликанты — те, что покрывают столбец, которому нет других покрытий — и объясните, почему они обязаны войти в любое минимальное покрытие. Затем покройте оставшиеся столбцы ветвями с границами: принудительные строки, отбрасывание доминируемых строк и столбцов, нижняя оценка размера, ветвление по столбцу минимальной степени. Разберитесь, когда перебор покрытия взрывается и какие сокращения режут ветви. Почему задача о покрытии множества NP-трудна, хотя для малых функций дерево ветвления остаётся маленьким?

③ *Don't-care*

Пусть безразличные наборы участвуют в склеивании, но не обязаны покрываться. Проверьте, что минимум при учёте безразличных наборов не больше минимума без них, и приведите пример, где безразличные наборы меняют ответ. Покажите, почему безразличные наборы могут укрупнить кубы и как это связано со счётом безразличных клеток как 0 или 1 на карте Карно.

④ *Проверка эквивалентности*

Для функций до восьми переменных переберите все 2^n наборов и убедитесь, что исходная и минимальная ДНФ совпадают, а на безразличных наборах значения не сверяются. Объясните, почему проверки на части наборов недостаточно.

⑤ *Доказательство минимальности*

Для функций от одного до четырёх переменных найдите минимум независимым способом — перебором по всем кубам, без опоры на Квайна — МакКласки — и сравните с размером вашей ДНФ. Сформулируйте, что именно перебирает независимая проверка, и объясните, почему она не годится для больших n .

Итог: минимизатор, который по таблице истинности или минтермам возвращает минимальную по числу импликант ДНФ, проверенную на эквивалентность на всех наборах и подтверждённую независимым перебором для малых функций.

XI

Схемы

Стоит нажать клавишу — и на экране появляется буква. Кажется, это происходит мгновенно, но за каждой буквой стоят миллиарды крошечных переключений транзисторов. Вычисление происходит в физическом устройстве: булева функция возвращает результат, только когда в схеме реально переключаются транзисторы.

В предыдущей главе (10) булевы функции были чистыми абстракциями: таблицы истинности, нормальные формы, критерий Поста. Там было важно, что функция вычисляет, а не как. Теперь вопрос в том, как именно: как заставить булеву функцию бежать по проводам? Ответ — логический *вентиль* (*gate*): транзисторная схема, которая вычисляет одну булеву функцию. Подали напряжение на входы — получили напряжение на выходе.

Булева функция от k переменных — это правило, которое для каждой из 2^k комбинаций входов решает, что вернуть: 0 или 1. Транзистор — это переключатель: проводит ток или нет. Соедините переключатели правильно — и они реализуют отрицание, конъюнкцию или дизъюнкцию. Четыре транзистора дают NAND — вентиль, из которого можно построить всё остальное.

Отдельный вентиль — это «кирпич». Из одного кирпича дом не построишь. Но тысячи вентилях, соединённых в схему, складывают числа, управляют памятью, выполняют инструкции процессора.

Сквозной пример главы — сложение. Сложить два числа — тоже булева функция: на входе $2k$ битов, на выходе $k + 1$ бит суммы. Полусумматор — схема из двух вентилях: XOR даёт младший бит суммы, AND — перенос. Полный сумматор принимает ещё и перенос с предыдущего разряда, и из таких блоков собирается многоразрядный сумматор. И вот здесь появляется первый инженерный вопрос: перенос бежит от младших разрядов к старшим, и цепочка сумматоров замедляет вычисление на длинных числах.

Складывая столбиком два 64-битных числа, перенос придётся протащить через все 64 разряда: каждый сумматор ждёт перенос от предыдущего. Параллельные схемы ускоряют перенос, но платят площадью — так у любой схемы появляются два ресурса, время и площадь, и инженер торгуется между ними. Глубина схемы — это время. Глубина как мера сложности параллельного вычисления — тема схемной сложности, которой глава 30 касается вскользь.

Прежде чем складывать, нужно условиться о языке: как числа вообще представлены в виде битов. Двоичная система — необходимость: вентиль понимает только 0 и 1,

два уровня напряжения. Шестнадцатеричная запись — компактная стенография для тех же битов, встречающаяся в каждом дампе памяти.

Схема — это ориентированный ациклический граф: сигнал течёт от входов к выходам через цепочку вентилях, без петель и обратных связей. Ацикличность гарантирует, что выход определяется исключительно текущими входами: комбинационная схема (*combinational circuit*) не помнит прошлого и ведёт себя как честная булева функция. Но чтобы *запомнить* хоть один бит, нужны обратные связи: выход возвращается на вход, и схема перестаёт быть функцией, становясь машиной с состояниями. Эту вторую область — память, конечные автоматы — откроет глава 23.

У теории булевых функций есть и практическая отдача для самих схем. NAND самодостаточен: любую булеву функцию можно собрать из одних NAND. Критерий Поста (глава 10) — общий критерий полноты: набор вентилях полон тогда и только тогда, когда он не содержится ни в одном из пяти замкнутых классов. Для производства это значит, что нужен только один тип детали — дешевле, проще, надёжнее. А вопрос «эквивалентны ли две схемы?» — это уже задача SAT: существует ли вход, на котором они разойдутся? Так схемы подводят к верификации и к границам теории сложности (глава 14).

Какова цена булевой функции в вентилях — и почему для почти любой функции она колоссальна?

ИСТОРИЯ От реле до закона Мура

Клод Шеннон (1937) показал, что булева алгебра описывает релейные схемы, но реле — медленные, шумные, механические. Переход к твердотельному переключателю без движущихся частей стал следующим шагом.

Джон Бардин, Уолтер Браттейн и Уильям Шокли в Bell Labs (1947) создали точечный транзистор — твердотельный усилитель и переключатель — и получили за него Нобелевскую премию по физике 1956 года. Транзистор был меньше, быстрее и надёжнее реле на порядки. Суть транзистора — в отсутствии движущихся частей: одно электричество включает и выключает другое. Никакой механики, только уровни напряжения, — поэтому переключение повторяется миллиарды раз в секунду без износа.

Джек Килби (Texas Instruments, 1958) и Роберт Нойс (Fairchild Semiconductor, 1959) независимо изобрели интегральную схему — несколько транзисторов на одном кристалле. Килби сделал первый прототип из одного куса германия, с соединениями из золотой проволоки — грубо, но работало. Нойс использовал кремний и планарный процесс (фотолитографию), что позволяло печатать соединения прямо на поверхности кристалла — и именно этот подход стал промышленным стандартом. Вместо пайки отдельных компонентов — химическое травление на кремниевой пластине. Вместо ручной сборки — массовое автоматизированное производство. Нобелевская премия пришла к Килби в 2000 году. Нойс к тому времени уже ушёл (1990), и премию посмертно не присуждают. Килби и Нойс

пришли к интегральной схеме независимо и разными путями, но промышленность приняла подход Нойса: кремний и планарный процесс оказались технологичнее.

Гордон Мур (1965) сформулировал эмпирический закон: число транзисторов на чипе удваивается примерно каждый год. В 1975 году Мур скорректировал оценку до двух лет. Этот экспоненциальный рост держался полвека и превратил вычислительные машины из комнатных монстров в карманные суперкомпьютеры. Булева алгебра, воплощённая в кремнии, превратила вычисление в промышленную технологию.

ОБЗОР ГЛАВЫ

В этой главе мы договоримся о языке: как числа живут в битах — двоичная, восьмеричная и шестнадцатеричная запись. Затем познакомимся с вентилями — кирпичами схем — увидим, какие семейства вентиляей нужны и почему одного NAND достаточно для всего (критерий Поста, глава 10). Из вентиляей соберём полусумматор и полный сумматор, соединим их в цепочку многоразрядного сложения и посмотрим, где перенос замедляет схему, а где параллельные вычисления его обгоняют. Код Грея покажет, как договорённость о кодировании влияет на физическую надёжность схемы, а верификация через SAT продемонстрирует, как тот же вопрос «эквивалентны ли две схемы?» упирается в NP-полноту.

После этой главы вы сможете:

1. переводить числа между двоичной, восьмеричной, шестнадцатеричной и десятичной системами;
2. читать схемы из вентиляей и объяснять, какие семейства вентиляей полны;
3. собирать полусумматор и полный сумматор и объяснять, почему цепочка переносов замедляет сложение;
4. пользоваться кодом Грея и объяснять, где он выигрывает;
5. сводить вопрос об эквивалентности схем к задаче выполнимости.

§ 11.1 Системы счисления

Прежде чем строить схемы, нужно условиться о языке: как числа представлены в виде последовательности битов.

В повседневной жизни мы используем десятичную систему: число $d_k d_{k-1} \dots d_0$ (где $0 \leq d_i \leq 9$) означает

$$d_k \cdot 10^k + d_{k-1} \cdot 10^{k-1} + \dots + d_0 \cdot 10^0.$$

Основание 10 — историческая случайность (десять пальцев на руках), а не математическая необходимость.

ОПРЕДЕЛЕНИЕ 11.1 Позиционная система счисления

Система записи, в которой число $d_k d_{k-1} \dots d_0$ с цифрами $0 \leq d_i < b$ представляет значение

$$\sum_{i=0}^k d_i b^i,$$

называется **позиционной системой счисления с основанием $b \geq 2$** .

Частные случаи основания:

- $b = 2$ — **двоичная** (binary) система;
- $b = 8$ — **восьмеричная** (octal) система;
- $b = 16$ — **шестнадцатеричная** (hexadecimal) система, цифры 0–9 и A–F для 10–15.

Пример Перевод из двоичной в десятичную

$1011_2 = 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 11_{10}$. $11111111_2 = 255_{10}$ — максимальное 8-битное число (1 байт).

Пример Перевод 13 в двоичную систему

Повторяем деление на 2 с остатком. Остатки, прочитанные снизу вверх, дают двоичную запись.

$$\begin{aligned} \frac{13}{2} &= 6, \text{ остаток } 1 \text{ (младший бит)} \\ \frac{6}{2} &= 3, \text{ остаток } 0 \\ \frac{3}{2} &= 1, \text{ остаток } 1 \\ \frac{1}{2} &= 0, \text{ остаток } 1 \text{ (старший бит)} \end{aligned}$$

Читаем снизу вверх: $13_{10} = 1101_2$.

Двоичную запись удобно читать как набор гирек: разряды — это веса 1, 2, 4, 8, 16, ..., а бит решает, включена гирька в сумму или нет. Число 1011_2 — это гирьки 8, 2 и 1, включённые вместе. Позже у картинки появится физический смысл: включённая гирька — это высокое напряжение на проводе, выключенная — низкое.

Двоичная система — родной язык цифровых схем: два значения (0 и 1) соответствуют двум уровням напряжения (низкому и высокому) в транзисторе. Восьмеричная и шестнадцатеричная — компактная запись двоичных данных: одна шестнадцатеричная цифра кодирует 4 бита (nibble), две — 8 бит (байт).

Пример Шестнадцатеричная запись

$10111110_2 = \text{BE}_{16} : 1011 = 11 = B, 1110 = 14 = E.$

Цвета в HTML: #FF00FF — это (255, 0, 255) в RGB (пурпурный).

Теперь, когда мы условились о языке нулей и единиц, посмотрим на исполнителей — логические вентили.

§ 11.2 Логические вентили и схемы

Схема собирается из вентилях — физических исполнителей булевых операций.

11.2.1 Логические вентили

Булевы функции, изученные в предыдущей главе, — это математические объекты. Чтобы построить реальное вычислительное устройство, эти функции необходимо воплотить в физических элементах. Именно для этого служат логические вентили — атомарные строительные блоки любой цифровой схемы.

ОПРЕДЕЛЕНИЕ 11.2 Логический вентиль

Вычислительный блок с двоичными входами и одним двоичным выходом, соответствующий фиксированной булевой функции, называется **логическим вентиляем**.

Вентиль AND принимает два сигнала и выдаёт 1 только если оба равны 1. Вентиль NOT принимает один сигнал и инвертирует его. Каждый вентиль — это физическая реализация соответствующей булевой функции: подаёшь напряжение на входы — получаешь напряжение на выходе в соответствии с таблицей истинности. Вентиль прячет транзисторы внутри: для схемы важны только входы и выход, а не соединения под корпусом. Это первый акт абстракции главы — тот же приём будет работать на каждом следующем уровне. Полусумматор станет «вентилем» побольше, а полный сумматор — «вентилем» ещё больше: внутри сложно, снаружи — только таблица истинности.

Пример От булевой функции к вентилю

Функция большинства $f(x, y, z) = xy \vee xz \vee yz$ может быть реализована схемой из трёх AND (на каждую конъюнкцию) и одного OR (для дизъюнкции результатов). Та же функция может быть реализована иначе — через NAND-вентили — и выбор реализации влияет на площадь, задержку и энергопотребление схемы. Логический вентиль — это точка соприкосновения абстрактной булевой алгебры с физическим миром транзисторов.

Стандартный набор вентиляей покрывает базовые булевы операции. Каждый вентиль имеет общепринятое схематическое обозначение, но с логической точки зрения нас интересует только реализуемая им функция.

Набор не случаен: в нём три роли. AND, OR и NOT — связи булевой алгебры из главы 10, тот базис, в котором мы выражали все остальные функции. NAND и NOR функционально полны по отдельности. В КМОП-технологии (от англ. CMOS — complementary metal-oxide-semiconductor) NAND и NOR — минимальные строительные блоки: их реализация дешевле, чем у AND и OR, и к ним мы ещё вернёмся. XOR для полноты не нужен — его можно собрать из AND, OR и NOT. Но XOR — незаменимый вентиль цифровых схем: он различает биты, контролирует паритет, сравнивает операнды.

ОПРЕДЕЛЕНИЕ 11.3 Стандартные вентили

Стандартными логическими вентилями называются:

- NOT (инвертор): $f(x) = \neg x$ — один вход, выход противоположен входу.
- AND: $f(x, y) = x \wedge y$ — выход 1, только если оба входа равны 1.
- OR: $f(x, y) = x \vee y$ — выход 1, если хотя бы один вход равен 1.
- NAND: $f(x, y) = \neg(x \wedge y)$ — отрицание AND.
- NOR: $f(x, y) = \neg(x \vee y)$ — отрицание OR.
- XOR (исключающее ИЛИ): $f(x, y) = x \oplus y$ — выход 1, если входы различны.

Пример Чтение таблиц истинности

Таблицу двухвходового вентиля удобно читать как одно четырёхбитное слово: $f(00)f(01)f(10)f(11)$. Например, AND — это 0001, OR — это 0111, XOR — это 0110. XOR отличается от OR ровно в последней строке: при обоих входах 1 OR даёт 1, а XOR даёт 0. NAND — это отрицание AND, NOR — отрицание OR. Полный перебор строк вынесен в сводную таблицу ниже, а здесь видна связь между вентилями.

Сводка стандартных вентиляей:

Вентиль	Обозначение	Функция	Таблица истинности
NOT	$\neg x$	Инвертирует вход	$f(0) = 1, f(1) = 0$
AND	$x \wedge y$	1 только если оба входа 1	$00 \rightarrow 0, 01 \rightarrow 0, 10 \rightarrow 0, 11 \rightarrow 1$
OR	$x \vee y$	1 если хотя бы один вход 1	$00 \rightarrow 0, 01 \rightarrow 1, 10 \rightarrow 1, 11 \rightarrow 1$
NAND	$\neg(x \wedge y)$	Отрицание AND	$00 \rightarrow 1, 01 \rightarrow 1, 10 \rightarrow 1, 11 \rightarrow 0$
NOR	$\neg(x \vee y)$	Отрицание OR	$00 \rightarrow 1, 01 \rightarrow 0, 10 \rightarrow 0, 11 \rightarrow 0$
XOR	$x \oplus y$	1 если входы различны	$00 \rightarrow 0, 01 \rightarrow 1, 10 \rightarrow 1, 11 \rightarrow 0$

Таблица 3. Стандартные логические вентили.

Среди этого набора особенно выделяются NAND и NOR. Полнота каждого доказана в главе 10: и штрих Шеффера, и стрелка Пирса по отдельности выражают отрицание, конъюнкцию и дизъюнкцию.

Примечание

В КМОП-технологии и NAND, и NOR собираются из четырёх транзисторов, а AND и OR — из шести: к каждому добавляется инвертор. Реальный выбор между NAND и NOR решает физика транзисторов, и промышленные схемы строятся на NAND (глава 10).

Проверка Логические вентили и полнота

1. Почему XOR не нужен для функциональной полноты, но без него не обходится ни одна схема?
2. Почему именно NAND, а не AND, считают основой современных интегральных схем?

11.2.2 Схемы как ориентированные ациклические графы

Отдельный вентиль — как деталь конструктора. Сам по себе он выполняет элементарную операцию, но не решает содержательной задачи. Чтобы вычислить сложную булеву функцию, вентили соединяют в схему — сеть, в которой выходы одних вентилях подаются на входы других. Это напоминает соединение труб в водопроводе: сигнал «течёт» от входов через цепочку вентилях к выходам, нигде не закливаясь. Если в водопроводе цикл означал бы бесконечную рециркуляцию, то в комбинационной схеме цикл означает неопределённость — сигнал не может стабилизироваться. Главное ограничение комбинационной схемы — отсутствие циклов. Это гарантирует, что сигнал распространяется строго от входов к выходам и результат определяется исключительно текущими значениями входов, без какой-либо памяти о прошлом.

ОПРЕДЕЛЕНИЕ 11.4 Логическая схема

Ориентированный ациклический граф (DAG) называется **логической схемой**, если в нём

1. истоки — входы (переменные, константы 0/1);
2. внутренние узлы — логические вентили;
3. стоки — выходы.

Пример Схема для $(x \wedge y) \vee \neg z$

Входы: x, y, z . Вентиль AND принимает x, y и выдаёт $x \wedge y$. Вентиль NOT принимает z и выдаёт $\neg z$. Вентиль OR принимает выходы AND и NOT и выдаёт $(x \wedge y) \vee \neg z$. Три вентиля, глубина 2 (AND и NOT работают параллельно на первом уровне, OR на втором). Схема является DAG: сигнал течёт строго слева направо, циклов нет.

При анализе и проектировании схем важно измерять их эффективность. Понятие эффективности схемы распадается на две конфликтующие меры. *Размер* схемы (число вентилях) — это стоимость: площадь кристалла, энергопотребление, цена производства. *Глубина* схемы (длиннейший путь от входа к выходу) — это скорость: задержка сигнала определяет максимальную тактовую частоту. В пределе можно сделать схему очень быстрой, но огромной (много параллельных вычислений), или очень компактной, но медленной (последовательные вычисления). Выбор между размером и глубиной — базовый инженерный компромисс, и мы увидим его на примере сумматоров: ripple-carry (компактный, медленный) против carry-lookahead (быстрый, большой).

ОПРЕДЕЛЕНИЕ 11.5 Размер

Количество вентилях называется **размером** схемы (аппаратные затраты, площадь кристалла).

ОПРЕДЕЛЕНИЕ 11.6 Глубина

Длиннейший путь от входа к выходу называется **глубиной** схемы (задержка распространения сигнала, тактовая частота).

Размер и глубина — характеристики схемы в целом. Есть ещё два числа, привязанных к отдельному вентилю, — от них зависит, какие схемы вообще удаётся собрать.

ОПРЕДЕЛЕНИЕ 11.7 Входная степень

Максимальное количество входов одного вентиля называется **входной степенью** (fan-in).

ОПРЕДЕЛЕНИЕ 11.8 Выходная степень

Максимальное количество вентиляей, управляемых одним выходом, называется **выходной степенью** (fan-out).

На практике fan-in ограничен физическими свойствами транзисторов: слишком большое число входов увеличивает задержку и потребление энергии. В КМОП-логике типичный fan-in не превышает 4–5. Fan-out ограничен нагрузочной способностью выхода: каждый следующий вход добавляет ёмкость, замедляющую переключение. При большом fan-out в схему вставляют буферы — дополнительные вентили, усиливающие сигнал ценой небольшой дополнительной задержки.

Пример Размер и глубина простых схем

- Схема для $(x \wedge y) \vee \neg z$: размер 3 (AND, NOT, OR), глубина 2 (AND и NOT на уровне 1, OR — на уровне 2).
- Схема $x \oplus y = (x \vee y) \wedge \neg(x \wedge y)$ из базовых вентиляей. Размер: 4 вентиля (OR, AND, NOT, AND). Глубина: 3 (OR и AND на уровне 1, NOT на уровне 2, финальный AND на уровне 3).
- Сумматор с последовательным переносом для n битов: размер и глубина $O(n)$ — перенос распространяется последовательно через все разряды.

ЛОВУШКА Задержка: не складывать параллельные пути

Глубина схемы — не сумма задержек всех вентиляей. Это длина самого длинного пути от входа к выходу — критический путь. Параллельные ветви засчитываются один раз: сигнал по короткой ветви приходит раньше и ждёт остальных. В схеме для $(x \wedge y) \vee \neg z$ три вентиля, но глубина два: AND и NOT работают одновременно, и только OR ждёт оба результата.

Примечание

Таблица истинности описывает, **что** вычисляется. Схема описывает, **как**. Одна и та же функция может быть реализована множеством различных схем. Задача проектировщика — найти схему, минимизирующую размер и глубину при заданных технологических ограничениях. Это нетривиальная оптимизационная задача: уже проверка эквивалентности двух схем является вычислительно трудной. Она сводится к проверке тавтологичности булевой формулы. Это coNP-полная задача. Классы сложности обсуждаются в главе 30.

11.2.3 Семейства схем

Одна схема имеет фиксированное число входов. Схема для 4-битного сложения не умеет складывать 8-битные числа — ей не хватает входов. Это ограничение *принципиально*: схема — не программа, она не параметризована размером входа. Но

практические функции — сложение, умножение, сравнение — определены для входов произвольной длины. Чтобы описать такую функцию в терминах схем, вводят понятие *семейства схем*: по одной схеме для каждого размера входа. Этот формализм, при всей своей громоздкости, стандартно связывает схемную сложность с теорией алгоритмов.

ОПРЕДЕЛЕНИЕ 11.9 Семейство схем

Последовательность схем $C_1, C_2, \dots$ называется **семейством схем**. Схема C_n из семейства имеет n входов.

ОПРЕДЕЛЕНИЕ 11.10 Равномерное семейство схем

Семейство схем, для которого существует алгоритм, строящий C_n по n , называется **равномерным**.

Равномерные семейства эквивалентны машинам Тьюринга по вычислительной мощности.

ОПРЕДЕЛЕНИЕ 11.11 Неравномерное семейство схем

Семейство схем, не обязанное порождаться алгоритмом, называется **неравномерным**.

Пример Семейство схем для проверки чётности

Функция $\text{PARITY}(x_1, \dots, x_n) = x_1 \oplus \dots \oplus x_n$ — XOR всех входов. Одна схема с фиксированным числом входов не подходит для произвольного n . Но семейство схем $C_2, C_3, C_4, \dots$ справляется:

- C_2 — один XOR-вентиль с двумя входами.
- C_3 — два XOR-вентилia (каскад).
- C_n — $n - 1$ XOR-вентилей, соединённых цепочкой или деревом.

Это семейство равномерно: алгоритм, строящий C_n , — просто цикл, генерирующий $n - 1$ вентилей и соединяющий их.

Без требования равномерности семейства схем становятся неестественно мощными: неразрешимые задачи могут быть «решены» неравномерными семействами схем экспоненциального размера. Зафиксируем n и рассмотрим язык, где ответ зависит от входа из n битов — например, «останавливается ли машина Тьюринга, код которой подан на вход, на пустой ленте». Для каждого n ответ — некоторая функция f_n от n битов. Её таблица истинности содержит 2^n строк. По таблице строится схема (экспоненциального размера), вычисляющая f_n на всех входах длины n — в том числе на входах, соответствующих неразрешимым случаям. Требование алгоритмической порождаемости C_n восстанавливает связь с обычной вычислимостью.

§ 11.3 Сумматоры

Сумма $2 + 3 = 5$ знакома с детства и вычисляется без размышлений. Сложить два числа в терминах нулей и единиц — значит решить, какие биты и как складываются. Алгоритм сложения столбиком не зависит от основания: в десятичной $8 + 2$ даёт 0 и перенос 1. Ровно так же $1 + 1$ в двоичной даёт 0 и перенос 1. Уметь нужно немногое — сложить один столбец и передать перенос соседнему. Из этого и вырастет вся конструкция главы: схема одного столбца, затем цепочка таких схем.

Процессор не знает арифметики — он знает только логические вентили. Сейчас мы увидим, как два десятка NAND-вентилей, правильно соединённых, начинают *складывать числа*. Иерархия видна уже на первом шаге: полусумматор — это два вентилей, полный сумматор — горстка, а n полных сумматоров, соединённых цепочкой, складывают числа произвольной разрядности. Никаких новых вентилей по ходу дела не появляется — новая способность рождается из композиции старых. Сложение — одна из основных операций любого процессора, и оно строится из тех же кирпичей, что и простейший вентиль.

Простота цепочки имеет цену: перенос идёт через все разряды по очереди, и чем больше разрядность, тем медленнее сложение. Ниже мы увидим и противоположное решение — параллельный перенос — которое быстрее, но требует больше вентилей.

11.3.1 Полусумматор и полный сумматор

Начнём с простейшей задачи: сложение двух однобитных чисел.

ОПРЕДЕЛЕНИЕ 11.12 Полусумматор (*half adder*)

Комбинационная схема с входами $A, B \in \{0, 1\}$ и выходами

$$S = A \oplus B$$

$$C = A \wedge B$$

называется **полусумматором**.

Слово *полу-* в названии точное: схема делает ровно половину работы колонки. Колонка складывает три бита — два слагаемых и входящий перенос — а полусумматор принимает только два. Недостающее звено — входящий перенос — достраивает полный сумматор.

Таблица истинности полусумматора — это таблица XOR для суммы и AND для переноса.

Пример Таблица истинности полусумматора

A	B	S (сумма)	C (перенос)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Схема полусумматора собрана из двух вентилях: XOR вычисляет сумму, AND — перенос. На рис. 58 показано, как сигналы текут от входов A и B к выходам S и C .

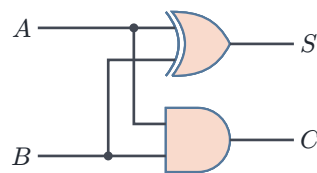


Рис. 58. Полусумматор.

Полусумматора достаточно для младшего разряда, но для всех последующих разрядов нужно складывать три бита: два бита слагаемых и входящий перенос из предыдущего разряда. Для этого служит полный сумматор.

ОПРЕДЕЛЕНИЕ 11.13 Полный сумматор (*full adder*)

Комбинационная схема с входами $A, B, C_{\text{in}} \in \{0, 1\}$ и выходами

$$S = A \oplus B \oplus C_{\text{in}}$$

$$C_{\text{out}} = (A \wedge B) \vee (C_{\text{in}} \wedge (A \oplus B))$$

называется **полным сумматором**.

Перенос C_{out} равен 1, когда не менее двух из трёх входов равны 1, — это функция большинства. Раскрыв $A \oplus B$, ту же формулу можно записать симметрично: $C_{\text{out}} = (A \wedge B) \vee (A \wedge C_{\text{in}}) \vee (B \wedge C_{\text{in}})$.

ЛОВУШКА XOR — сумма без переноса

XOR двух битов — это их арифметическая сумма без переноса: $1 \oplus 1 = 0$, $1 + 1 = 10_2$. В полном сумматоре бит суммы $s_i = a_i \oplus b_i \oplus c_i$. Перенос c_{i+1} вычисляется отдельной формулой. Поэтому при $1 + 1 + 1$ бит суммы равен 1, а перенос — 1: перенос не появляется на выходе XOR, его вычисляет отдельная часть схемы.

Полный сумматор достраивает недостающее звено: он принимает входящий перенос C_{in} и складывает три бита. На рис. 59 показана схема, собранная из двух полусумматоров и одного вентиля OR.

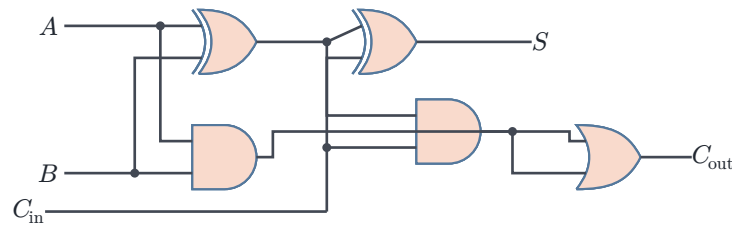


Рис. 59. Полный сумматор.

УТВЕРЖДЕНИЕ 11.1 Полный сумматор из двух полусумматоров

Соединим два полусумматора и вентиль OR: первый принимает A и B , второй принимает сумму первого $S_1 = A \oplus B$ и входящий перенос C_{in} , а OR объединяет переносы C_1, C_2 обоих полусумматоров. Выходом суммы служит сумма второго полусумматора. Такая схема — полный сумматор.

Доказательство

Докажем прямым вычислением: выпишем выходы сборки и сверим их с определением полного сумматора.

Первый полусумматор выдаёт $S_1 = A \oplus B$ и перенос $C_1 = A \wedge B$. Второй принимает S_1 и C_{in} и выдаёт сумму

$$S = S_1 \oplus C_{in} = A \oplus B \oplus C_{in}$$

и перенос $C_2 = S_1 \wedge C_{in} = (A \oplus B) \wedge C_{in}$. Выход OR равен

$$C_1 \vee C_2 = (A \wedge B) \vee (C_{in} \wedge (A \oplus B)),$$

и формулы совпадают с определением полного сумматора.

Остаётся проверить, что OR не теряет перенос. Перенос должен равняться 1, когда среди битов A, B, C_{in} хотя бы два равны 1. Случай $A = B = 1$ ловит C_1 , а случай «ровно один из битов A, B равен 1 и $C_{in} = 1$ » ловит C_2 : тогда $S_1 = 1$. Других способов набрать две единицы нет. Случаи не пересекаются: при $A = B = 1$ бит S_1 равен 0, и $C_2 = 0 \wedge C_{in} = 0$. Поэтому каждый вход с двумя единицами активирует ровно один из переносов, и $C_1 \vee C_2$ выдаёт правильный C_{out} .

Пример Таблица истинности полного сумматора

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1

A	B	C_{in}	S	C_{out}
1	1	1	1	1

Пример Полный сумматор на Rust

Формулы переносятся в код без потерь: сумма — XOR трёх входов, перенос — функция большинства.

```
/// Полный сумматор: биты a, b и перенос c_in
/// -> сумма и перенос на выход.
fn full_adder(a: bool, b: bool, c_in: bool) -> (bool, bool) {
    let sum = a ^ b ^ c_in;
    // функция большинства
    let c_out = (a && b) || (a && c_in) || (b && c_in);
    (sum, c_out)
}
```

11.3.2 Последовательный перенос и АЛУ

Имея полный сумматор, можно построить схему для сложения n -битных чисел: соединяем n полных сумматоров цепочкой, где перенос каждого разряда подаётся на вход следующего.

ОПРЕДЕЛЕНИЕ 11.14 Сумматор с последовательным переносом

Цепочка из n полных сумматоров для n -битного сложения называется **сумматором с последовательным переносом** (ripple-carry adder).

Перенос в цепочке распространяется через все разряды последовательно. Размер и глубина схемы — $O(n)$. Для 64-битного сложения это 64 последовательных переноса — существенная задержка для тактовой частоты процессора.

Пример Сумматор с последовательным переносом на Rust

Цепочка полных сумматоров — это цикл: каждый разряд складывает три бита и отдаёт перенос дальше.

```
/// Сложение n-битных чисел цепочкой полных сумматоров.
fn ripple_carry(a: &[bool], b: &[bool]) -> (Vec<bool>, bool) {
    let n = a.len();
    let mut sum = vec![false; n];
    let mut carry = false; // c_0 = 0, переноса снизу нет
    for i in 0..n {
        let (s, c_out) = full_adder(a[i], b[i], carry);
        sum[i] = s;
        carry = c_out; // перенос уходит в следующий разряд
    }
}
```

```

    (sum, carry) // carry: перенос из старшего разряда
}

```

Тактовая частота процессора определяется наихудшим путём задержки: за один такт сигнал должен успеть пройти от входов схемы до её выходов и стабилизироваться. Если самый длинный путь слишком длинный, такт приходится удлинять, а частоту — снижать. В многоразрядном сумматоре самый длинный путь — это цепочка переносов: старшие биты суммы не стабилизируются, пока перенос не дойдёт до них через все младшие разряды. Значит, 64 последовательных переноса напрямую ограничивают тактовую частоту. Опережающий перенос (*carry-lookahead*) — способ обойти этот предел: он вычисляет переносы параллельно и платит за это вентилями, зато допускает частоту на порядок выше.

Замечание Миелиновая оболочка и опережающий перенос

Нервный импульс движется двумя способами. В немиелинизированном волокне каждый участок мембраны деполяризуется — и только затем открываются каналы следующего. Сигнал проходит аксон шаг за шагом, задержка пропорциональна длине. Ripple-carry устроен так же: перенос идёт через цепочку сумматоров последовательно, задержка $O(n)$.

В миелинизированном волокне перехваты Ранвье разделены изолированными участками. Сигнал перепрыгивает от узла к узлу. Скачок вместо волны: задержка уменьшается в несколько раз. Carry-lookahead: генерация и распространение переноса вычисляются поблочно, затем дерево объединяет блоки — глубина падает с $O(n)$ до $O(\log n)$, и выигрыш растёт с разрядностью.

Миелиновая оболочка требует дополнительных белков. Опережающий перенос — дополнительных вентилях. В обоих случаях платят площадью и энергией за скорость.

УТВЕРЖДЕНИЕ 11.2 Корректность сумматора с последовательным переносом

Сумматор с последовательным переносом складывает два n -битных числа корректно: биты суммы и финальный перенос совпадают с результатом сложения.

Набросок доказательства

Индукция по разрядам от младшего к старшему.

База: в младший разряд перенос не приходит — $c_0 = 0$. Полный сумматор на позиции 0 выдаёт верный бит суммы и перенос: s_0, c_1 . Шаг: пусть на позицию i приходит корректный перенос c_i . Полный сумматор складывает три бита: a_i, b_i, c_i .

Бит суммы $s_i = a_i \oplus b_i \oplus c_i = (a_i + b_i + c_i) \bmod 2$. Перенос c_{i+1} равен 1 тогда и только тогда, когда $a_i + b_i + c_i \geq 2$. Это ровно то, что заложено в формулы полного сумматора. Перенос c_{i+1} поступает на вход следующего разряда, и шаг повторяется.

По индукции все n битов суммы и финальный перенос c_n верны: сумматор корректно складывает два n -битных числа.

Пример Трассировка ripple-carry: 0111 + 0001

Сложение двух 4-битных чисел на ripple-carry сумматоре:

1. **Разряд 0 (младший):** $1 + 1 = 0, c_1 = 1$.
2. **Разряд 1:** $1 + 0 + c_1 = 0, c_2 = 1$.
3. **Разряд 2:** $1 + 0 + c_2 = 0, c_3 = 1$.
4. **Разряд 3 (старший):** $0 + 0 + c_3 = 1, c_4 = 0$.

Результат: 1000 (десятичное 8). Перенос прошёл через все четыре разряда последовательно — каждый следующий полный сумматор ждёт переноса от предыдущего. Задержка пропорциональна числу разрядов: n полных сумматоров, глубина $O(n)$.

Инженерный ответ на эту проблему — параллелизм. Перенос в i -м разряде можно вычислить, не дожидаясь всех предыдущих. Для каждого разряда мы вычисляем два сигнала.

- **Генерация** (G_i): данный разряд рождает перенос, если оба бита равны 1.
- **Распространение** (P_i): данный разряд пропускает входящий перенос, если ровно один бит равен 1 ($P_i = A_i \oplus B_i$).

Если оба бита равны 1, перенос генерируется самим разрядом через G_i , и выбор XOR для P_i ничего не портит. XOR выбран потому, что он же даёт бит суммы. Теперь перенос из блока можно вычислить, зная только G, P его подблоков, без обхода всех разрядов.

Например, блок из четырёх разрядов генерирует перенос, если:

- Перенос генерируется в старшем разряде блока.
- Или генерируется в третьем и распространяется через четвёртый.
- Или генерируется во втором и распространяется через третий и четвёртый.
- и так далее:

$$G_{\text{block}} = G_3 \vee (P_3 \wedge G_2) \vee (P_3 \wedge P_2 \wedge G_1) \vee (P_3 \wedge P_2 \wedge P_1 \wedge G_0).$$

Через блок перенос проходит по такому же правилу: $P_{\text{block}} = P_3 \wedge P_2 \wedge P_1 \wedge P_0$. Выход блока равен $c_{\text{out}} = G_{\text{block}} \vee (P_{\text{block}} \wedge c_{\text{in}})$. Два блока комбинируются в один: $G = G_{\text{hi}} \vee (P_{\text{hi}} \wedge G_{\text{lo}}), P = P_{\text{hi}} \wedge P_{\text{lo}}$. Эта формула — булева функция фиксированной глубины (2 уровня логики: AND — OR), не зависящая от размера блока. Соединяя блоки в дерево по этим правилам, мы получаем все переносы за $O(\log n)$ шагов

вместо $O(n)$. За это приходится добавлять вентили для вычисления G, P на каждом уровне иерархии.

Замечание Глубина и время

Сигнал распространяется через вентили с конечной скоростью. Каждый уровень добавляет задержку в несколько пикосекунд — и эти задержки суммируются вдоль критического пути. Глубина схемы есть физическое время вычисления. Размер схемы есть физическая площадь кристалла — и, косвенно, стоимость и энергопотребление.

Компромисс между размером и глубиной, который мы видели на примере ripple-carry против carry-lookahead, — дихотомия, которая повторяется на каждом уровне абстракции: время против пространства. В схемной реализации она даёт $O(n)$ уровней задержки против $O(\log n)$ глубины с дополнительными вентилями: время против площади. Структура одна — меняется материал: процессорное время или кремний. К этой связи мы вернёмся, когда перейдём к классам P и NP в главе 30.

Примечание Вычитание через дополнительный код

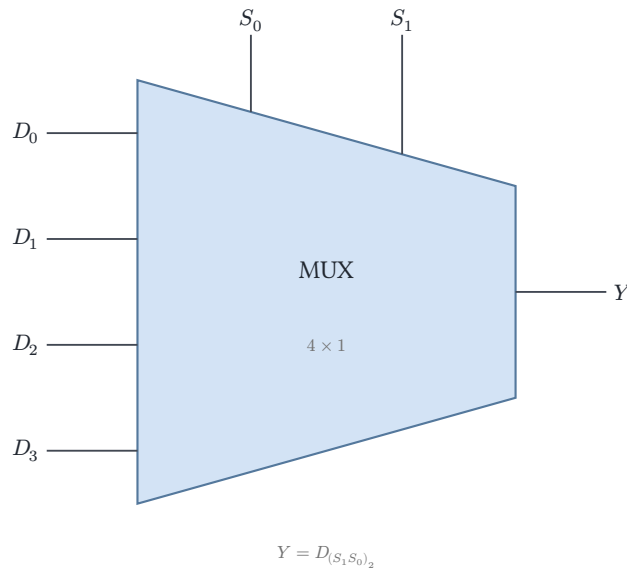
Вычитание реализуется через сложение с дополнительным кодом (*two's complement*): $A - B = A + (\overline{B} + 1)$. Инвертирование битов B даёт обратный код (*ones' complement*). Добавление единицы даёт дополнительный код. Таким образом, один и тот же сумматор может и складывать, и вычитать — выбор операции определяется управляющим сигналом.

Прежде чем собирать АЛУ, определим ещё две стандартные схемы.

ОПРЕДЕЛЕНИЕ 11.15 Мультиплексор

Комбинационная схема с 2^k входами данных, k адресными входами и одним выходом называется **мультиплексором** 2^k -к-1, если она выдаёт значение того входа данных, номер которого задан адресными битами.

Мультиплексор 4×1 ($k = 2$) — на рис. 60: четыре входа данных $D_0, \dots, D_3$, два адресных входа S_0, S_1 и один выход Y .

Рис. 60. Мультиплексор 4×1 .**ОПРЕДЕЛЕНИЕ 11.16** **Декодер**

Комбинационная схема с k входами и 2^k выходами называется **декодером** из k в 2^k , если она активирует ровно один выход — тот, номер которого совпадает с двоичным значением входа.

Пример Мультиплексор и декодер в действии

Мультиплексор 4×1 на рис. 60 выбирает один из четырёх входов данных. При адресе $S_1S_0 = 10$ на выход Y проходит D_2 : строка 10 — это двоичная запись номера 2. Остальные входы на выход не влияют.

Декодер 2×4 — обратная сторона той же идеи: k битов выбирают один из 2^k вариантов. На входе 01 активируется выход с номером 1, на входе 11 — выход с номером 3. Каждому адресу соответствует ровно один активный выход.

АЛУ (арифметико-логическое устройство) — вычислительное ядро процессора. Одноразрядный срез АЛУ подаёт операнды и код операции на параллельные блоки (AND, OR, ADD, SUB), а мультиплексор выбирает результат. n -битное АЛУ — n срезов с цепочкой переносов. Современные АЛУ включают умножители, баррель-сдвигатели и FPU — блоки плавающей арифметики (от англ. floating-point unit), работающие по стандарту IEEE 754⁹³.

⁹³J. L. Hennessy, D. A. Patterson. «Computer Architecture: A Quantitative Approach». 5-е изд., 2012, гл. 3.

§ 11.4 Код Грея

При передаче данных между асинхронными доменами или в механических энкодерах возникает проблема: если несколько битов изменяются одновременно, невозможно гарантировать, что все они переключатся строго в один и тот же момент. Промежуточные состояния могут давать ложные значения. Код Грея решает эту проблему, упорядочивая двоичные строки так, что соседние отличаются ровно в одном бите.

ОПРЕДЕЛЕНИЕ 11.17 Код Грея

Упорядочение всех 2^n двоичных строк длины n , при котором соседние строки (включая первую и последнюю) отличаются ровно в одном бите, называется **n -битным кодом Грея**.

3-битный код Грея: 000, 001, 011, 010, 110, 111, 101, 100. Каждая соседняя пара отличается ровно в одном бите. Последняя строка (100) и первая (000) тоже отличаются в одном бите — код циклический.

Код Грея строится рекурсивно.

УТВЕРЖДЕНИЕ 11.3 Рекурсивное построение

$$G_1 = [0, 1].$$

$$G_{n+1} = [0G_n, 1G_n^R],$$

где $G_n^R = \text{reverse}(G_n)$ — список в обратном порядке, а $0G_n$ означает приписывание 0 слева к каждой строке G_n .

Набросок доказательства

Докажем по индукции. База: $G_1 = [0, 1]$ — строки отличаются в одном бите, и первая с последней тоже. Переход: пусть G_n обладает свойством Грея. В $0G_n$ все строки начинаются с 0, внутри блока свойство сохраняется по индукции. В $1G_n^R$ все строки начинаются с 1, внутри блока свойство также выполнено (обращение порядка не нарушает попарного отличия в одном бите).

На стыке блоков: последняя строка $0G_n$ — это 0 плюс последняя строка G_n . Первая строка $1G_n^R$ — это 1 плюс последняя строка G_n (поскольку G_n^R начинается с конца G_n). Эти две строки отличаются только в первом бите (0 против 1).

Наконец, первая строка G_{n+1} и последняя: $0\dots 0$ против $1\dots 0$ — отличаются только в первом бите. Единица приписывается к первой строке G_n , которая есть $0\dots 0$. Значит, G_{n+1} — циклический код Грея.

Рекурсивная конструкция проще всего видна на конкретном примере.

Пример Коды Грея G_2, G_3

$$G_2 = [00, 01, 11, 10]$$

$$G_3 = [000, 001, 011, 010, 110, 111, 101, 100].$$

В G_3 каждая соседняя пара отличается ровно в одном бите, и последняя строка (100) отличается от первой (000) также ровно в одном бите — это циклический код Грея.

На практике требуется быстрое преобразование между двоичным представлением и кодом Грея, и обратно.

УТВЕРЖДЕНИЕ 11.4 Преобразование между двоичным кодом и кодом Грея

- **Двоичный в код Грея:** $g_{n-1} = b_{n-1}$ (старший бит сохраняется). Для $i < n - 1$:

$$g_i = b_i \oplus b_{i+1}.$$

- **Код Грея в двоичный:** $b_{n-1} = g_{n-1}$. Для $i < n - 1$:

$$b_i = g_i \oplus b_{i+1},$$

— накопительный XOR, начиная со старшего бита.

Набросок доказательства

Докажем в две стороны: сначала прямое преобразование двоичного кода в код Грея, затем обратное.

По построению G_n : строка $0G_n$ получается из G_n приписыванием 0, что сохраняет все XOR-отношения между битами. Строка $1G_n^R$ получается из G_n приписыванием 1 и обращением порядка. Индукцией по n проверяется, что $g_i = b_i \oplus b_{i+1}$ задаёт в точности рекурсивное построение.

В блоке $0G_n$ старший бит $g_{n-1} = 0$. Формула воспроизводит G_n . В блоке $1G_n^R$ старший бит $g_{n-1} = 1$. Формула вместе с обращением порядка воспроизводит $1G_n^R$.

Обратное преобразование — решение системы $g_i = b_i \oplus b_{i+1}$ относительно b_i . Последовательно раскрывая $b_i = g_i \oplus b_{i+1}$ от старшего бита к младшему, получаем накопительный XOR.

Пример Перевод в код Грея и обратно

Возьмём $b = 1010$: четыре бита $b_3 = 1, b_2 = 0, b_1 = 1, b_0 = 0$. Старший бит сохраняется: $g_3 = b_3 = 1$. Остальные — XOR с соседом слева:

$$g_2 = b_2 \oplus b_3 = 0 \oplus 1 = 1$$

$$g_1 = b_1 \oplus b_2 = 1 \oplus 0 = 1$$

$$g_0 = b_0 \oplus b_1 = 0 \oplus 1 = 1$$

Код Грея: 1111. Обратный перевод — накопительный XOR от старшего бита:

$$b_3 = g_3 = 1$$

$$b_2 = g_2 \oplus b_3 = 1 \oplus 1 = 0$$

$$b_1 = g_1 \oplus b_2 = 1 \oplus 0 = 1$$

$$b_0 = g_0 \oplus b_1 = 1 \oplus 1 = 0$$

Вернулись к 1010: прямой и обратный переводы взаимно обратны.

Оба преобразования выполняются за $O(n)$ и допускают эффективную аппаратную реализацию с помощью цепочки XOR-ов.

Практические применения кода Грея охватывают разные области:⁹⁴

1. Поворотные энкодеры: устранение ложных состояний при переключении.
2. Карты Карно: упорядочивание строк и столбцов для склеивания смежных клеток.
3. Генетические алгоритмы: коды Грея соседних целых отличаются в одном бите, что уменьшает разрушительный эффект кроссинговера.
4. Вложение в гиперкуб: последовательные коды Грея задают гамильтоновы циклы в n -мерном кубе, где каждое ребро — смена одного бита.

Проверка Код Грея

1. Почему вторая половина кода G_{n+1} строится в обратном порядке — $1G_n^R$, а не $1G_n$?
2. Почему обычный двоичный код даёт ложные состояния при переключении, а код Грея — нет?

§ 11.5 Верификация схем через SAT

Спроектировав схему, необходимо убедиться, что она работает правильно. Особенно остро эта задача стоит при оптимизации, где необходимо проверить эквивалентность оптимизированной схемы исходной спецификации. Обе задачи сводятся к одной конструкции — митеру.

ОПРЕДЕЛЕНИЕ 11.18 Митер (*miter*)

Пусть C_1 и C_2 — комбинационные схемы с общими входами и одинаковым числом выходов. Соединим соответствующие входы двух схем, на каждую пару

⁹⁴F. Gray. «Pulse Code Communication». US Patent 2,632,058, 1953.

соответствующих выходов поставим XOR, а все выходы XOR соберём одним OR. Полученная схема называется **митером**.

УТВЕРЖДЕНИЕ 11.5 Свойство митера

Митер выдаёт 1 ровно на тех входах, на которых C_1 и C_2 различаются.

Доказательство

Докажем эквивалентность в обе стороны.

Пусть на входе u митер выдаёт 1. Тогда OR получил единицу хотя бы с одного XOR, значит, соответствующие выходы C_1 и C_2 на u различны. Обратно: пусть схемы различаются на u . Хотя бы одна пара соответствующих выходов различна, её XOR выдаёт 1, и OR передаёт единицу на выход митера.

УТВЕРЖДЕНИЕ 11.6 Сведение эквивалентности схем к SAT

$C_1 \equiv C_2$ тогда и только тогда, когда митер невыполним.

Набросок доказательства

Каждый вентиль митера выражается КНФ через его таблицу истинности, а требование «выход митера равен 1» добавляется отдельной клаузой. По свойству митера эта КНФ выполнима тогда и только тогда, когда существует вход, на котором схемы различаются. Невыполнимость означает совпадение выходов на всех входах, то есть эквивалентность $C_1 \equiv C_2$.

Невыполнимость формулы — это дополнение задачи SAT. Решатель устанавливает её, исчерпав пространство поиска модели.

Пример Митер в действии

Сравним $f_1(x, y) = x \vee y$ и $f_2(x, y) = x \oplus y$. На входе $x = 1, y = 1$: $f_1 = 1, f_2 = 0$, XOR выходов даёт 1, и митер выдаёт 1 — схемы разошлись. На входе $x = 1, y = 0$: обе выдают 1, XOR даёт 0 — расхождения нет. Митер «ловит» ровно те входы, на которых схемы различаются: его выполнимость — доказательство различия, невыполнимость — эквивалентность.

В индустрии это называется формальной верификацией аппаратуры и входит в стандартный поток проектирования (design flow) процессорных компаний. Перед производством процессора (изготовление фотошаблона — маски — стоит миллионы долларов) необходимо гарантировать, что оптимизированная схема логически эквивалентна спецификации.

Процедура:

1. Спецификация (обычно на языке описания аппаратуры Verilog или VHDL на уровне регистровых передач — RTL) компилируется в эталонную логическую схему.
2. Оптимизатор синтеза преобразует её для уменьшения размера и задержки.
3. Строится митер — схема сравнения эталона и оптимизированной версии.
4. SAT-решатель доказывает невыполнимость митера (что означает эквивалентность) либо находит контрпример.

Современные SAT-решатели — программы, основанные на алгоритме CDCL (Conflict-Driven Clause Learning), — проверяют эквивалентность схем с миллионами вентилях за часы. Конференции FMCAD и CAV — основные площадки по формальным методам в проектировании аппаратуры⁹⁵.

Сведение верификации к SAT — улица с двусторонним движением. Аппаратура помогает SAT не меньше, чем SAT помогает аппаратуре. CDCL можно реализовать аппаратно на FPGA (от англ. field-programmable gate array — программируемая пользователем вентильная матрица). Там распространение булевых ограничений (Boolean Constraint Propagation, далее — BCP) выполняется параллельно. BCP занимает 80–90% времени в программном CDCL. В аппаратной реализации все дизъюнкты проверяются одновременно, что даёт ускорение на порядок.⁹⁶

§ 11.6 Схемная сложность

Схемы в этой главе мы проектировали под конкретные задачи и мерили их двумя числами — размером и глубиной. Сумматор с последовательным переносом имеет размер $O(n)$, и этого хватило для инженерного решения. Но размер конкретной схемы — ещё не ответ на принципиальный вопрос: каков минимальный размер схемы, вычисляющей данную функцию?

ОПРЕДЕЛЕНИЕ 11.19 Схемная сложность

Минимальный размер схемы, вычисляющей булеву функцию, называется **схемной сложностью** этой функции.

ОПРЕДЕЛЕНИЕ 11.20 Глубинная сложность

Минимальная глубина схемы, вычисляющей булеву функцию, называется **глубинной сложностью** этой функции.

⁹⁵A. Biere, M. Heule, H. van Maaren, T. Walsh (ред.). «Handbook of Satisfiability». 2-е изд., 2021, гл. 24.

⁹⁶I. Skliarova, A. B. Ferrari. «A Software/Reconfigurable Hardware SAT Solver». IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2004.

Замечание Почему сложность измеряют схемами?

Машина Тьюринга уже даёт меру сложности: время работы на входе длины n . Схемная мера отвечает на другой вопрос: сколько стоит функция, если под каждую длину входа строить отдельное устройство. Машина одна на все входы, и программа обслуживает короткие и длинные данные одним и тем же способом. Схему для каждой длины можно строить заново, не требуя алгоритма, порождающего все схемы семейства. Поэтому нижняя оценка для схем сильнее нижней оценки для машин: ей подчиняются даже устройства, для которых единый способ построения неизвестен.

Ответ упирается в сравнение мощностей — приём, знакомый по главе 7. Сколько всего существует функций? Сколько схем данного размера? Если функций несоизмеримо больше, чем схем, большинству функций больших схем не избежать.

Пример Функций больше, чем схем

Число булевых функций от n переменных — 2^{2^n} . Каждая функция задаётся таблицей из 2^n строк. Число схем размера s гораздо меньше: схема описывается списком из s вентилях. Каждый вентиль — с типом из t вариантов и двумя входами, выбранными из $n + s$ источников (исходные переменные или выходы предыдущих вентилях). Таких описаний не больше $(t(n + s)^2)^s$.

Сравним при скромных значениях. При $n = 6$ функций $2^{64} \approx 1.8 \cdot 10^{19}$. Схем из шести вентилях — не больше $(3 \cdot 12^2)^6 \approx 6.5 \cdot 10^{15}$ (три типа вентилях, два входа из $6 + 6$ источников). Функций больше в тысячи раз, и почти все они не реализуются схемой из шести вентилях. Так выглядит мощностной аргумент в его простейшей форме.

На рис. 61 это соотношение видно как геометрическая картина: множество схем размера s — крошечный островок внутри множества всех булевых функций.

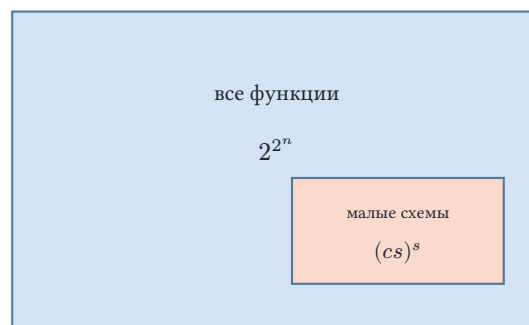


Рис. 61. Почти все функции лежат вне малых схем.

Обобщим подсчёт на все n . Ответ принадлежит Клоду Шеннону⁹⁷.

⁹⁷С. Е. Shannon. «The Synthesis of Two-Terminal Switching Circuits». Bell System Technical Journal, 1949.

ТЕОРЕМА 11.7 Мощностной аргумент Шеннона (*Shannon's counting argument*)

Почти все n -арные булевы функции требуют схем размера $\geq \frac{2^n}{2n}$.

Доказательство

Докажем мощностным подсчётом: сравним число схем ограниченного размера с числом 2^{2^n} всех булевых функций от n переменных.

Подсчёт схем. Зафиксируем базис из t типов вентиляей, где константа t не зависит от n , и разрешим каждому вентилю не более двух входов. Схема размера s описывается списком своих вентиляей. Каждый вентиль задаётся типом (t вариантов) и двумя входами, каждый из которых подключён к одной из n переменных или к выходу одного из s вентиляей. Разрешим подключение к любому из $n + s$ источников: каждая схема получит хотя бы одно такое описание, поэтому схем размера s не больше, чем описаний $(t(n + s)^2)^s$.

Суммирование по размерам. Положим $s_0 = \frac{2^n}{2n}$ и просуммируем оценку по всем размерам от 1 до s_0 . Слагаемое $(t(n + s)^2)^s$ растёт по s , поэтому каждое не превосходит значения при $s = s_0$, а слагаемых не больше s_0 . Итого схем размера не выше s_0 не больше, чем $s_0 \cdot (t(n + s_0)^2)^{s_0}$.

Оценка экспоненты. Прологарифмируем по основанию два. По построению $\log_2 s_0 = n - 1 - \log_2 n$. При $n \geq 7$ выполнено $s_0 \geq n$, откуда $\log_2(n + s_0) \leq 1 + \log_2 s_0 = n - \log_2 n$ и

$$\log_2 s_0 + s_0 \log_2 (t(n + s_0)^2) \leq 2^n + \log_2 s_0 - s_0(2 \log_2 n - \log_2 t).$$

Начиная с некоторой константы n_0 скобка $2 \log_2 n - \log_2 t$ положительна, а s_0 экспоненциально велико, поэтому вычтенное слагаемое превосходит $\log_2 s_0$.

Вывод. Число схем размера не выше $\frac{2^n}{2n}$ меньше числа всех функций 2^{2^n} , и их отношение стремится к нулю с ростом n . Значит, доля функций, вычисляемых схемами такого размера, стремится к нулю: почти все функции требуют схем размера $\geq \frac{2^n}{2n}$ — экспоненциального с точностью до множителя $\frac{1}{2n}$.

Замечание Верхняя оценка известна не хуже

Мощностной аргумент — только нижняя граница. Верхняя не отстаёт: Лупанов показал, что любую булеву функцию от n переменных реализует схема размера $(1 + o(1)) \cdot \frac{2^n}{n}$ ⁹⁸. Нижняя и верхняя границы совпадают с точностью до постоянного множителя. Схемная сложность *типичной* функции известна точно: она

⁹⁸О. Б. Лупанов. «О синтезе некоторых классов управляющих систем». Проблемы кибернетики, вып. 10, 1963.

порядка $\frac{2^n}{n}$. Значит, трудность не в том, чтобы узнать сложность функций вообще, а в том, чтобы предъявить хотя бы одну конкретную сложную функцию.

Результат Шеннона одновременно силён и разочаровывает. Почти все функции требуют больших схем, но доказательство лишь считает функции и схемы: оно не строит ни одной сложной функции. Мы знаем, что такие функции существуют — и не можем предъявить ни одну.

Пример Что значит доказать нижнюю оценку

Доказать нижнюю оценку для функции — значит показать, что никакая схема меньше некоторого размера её не вычисляет. Простейший случай — конъюнкция $f(x_1, \dots, x_n) = x_1 \wedge \dots \wedge x_n$. Каждый из n входов обязан влиять на выход, поэтому нужны $n - 1$ вентилях AND, соединённых в дерево.

Для почти всех функций аргумент Шеннона даёт порядок $\frac{2^n}{n}$. А для конкретных функций лучшая доказанная оценка — всего $5n - o(n)$. В этом разрыве между $\frac{2^n}{n}$ и $5n$ живёт открытая проблема.

Лучшая известная нижняя оценка для явно заданной функции линейная: $5n - o(n)$ ⁹⁹. Явно заданная функция — та, которую можно описать алгоритмически, а не доказать её существование мощностным аргументом. Десятилетия усилий не принесли ничего существенно лучшего: суперлинейная нижняя оценка для конкретной функции не доказана до сих пор. А суперполиномиальная нижняя оценка для функции из NP означала бы $P \neq NP$. Разрыв между знанием о функциях в целом и о функциях конкретных — одна из главных открытых проблем теории сложности. Она ждёт нас в главе 30.



Мощностной аргумент работает как перепись населения: он сообщает, сколько людей, но не сообщает, кто они. Чтобы назвать сложную функцию, нужен другой приём — тот, которого пока нет.

§ 11.7 * P/poly и монотонные схемы

Схемная сложность определена для функции с фиксированным числом входов. Но практические функции — сложение, умножение, сравнение — определены для входов любой длины. Для них нужен целый набор схем, по одной на каждый размер входа, — семейство схем (см. 11.2.3).

⁹⁹K. Iwama, H. Morizumi. «An explicit lower bound of $5n - o(n)$ for Boolean circuits». 2002. Предыдущий рекорд — $3n - o(n)$ — принадлежал N. Blum. «A Boolean Function Requiring $3n$ Network Size». Theoretical Computer Science, 1984.

Класс $P/poly$ собирает языки, для которых существует семейство схем полиномиального размера. Класс P содержится в $P/poly$: полиномиальная машина Тьюринга разворачивается в эквивалентную схему полиномиального размера. Но $P/poly$ содержит и неразрешимые языки: без требования алгоритмической порождаемости (см. 11.2.3) схема C_n может быть сколь угодно сложной в построении. В этом суть неравномерности.

На рис. 62 показано, как соотносятся классы: P лежит внутри $P/poly$, а тот — внутри всех языков, причём $P/poly$ вмещает и неразрешимые языки.

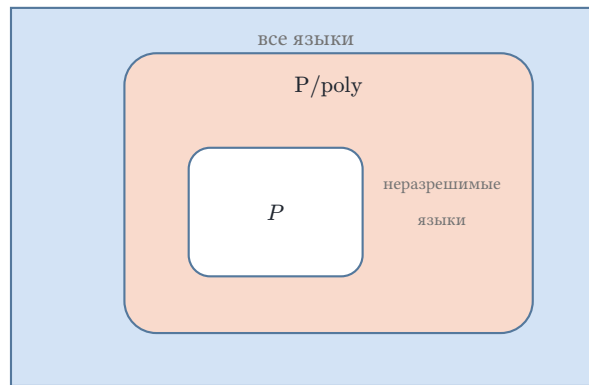


Рис. 62. Вложенность классов $P \subseteq P/poly$.

Теорема Карпа — Липтона¹⁰⁰ связывает схемы со всей иерархией классов. Если $NP \subseteq P/poly$, полиномиальная иерархия схлопывается до второго уровня. Доказать такую нижнюю оценку — значит ответить на вопрос о структуре всех классов сложности, а не только о поведении отдельных функций.

Пример CLIQUE: задача о знакомствах

Клика в графе — подмножество вершин, в котором каждая пара соединена ребром. В терминах знакомств: клика — компания, в которой все попарно знакомы. Функция CLIQUE отвечает на вопрос, есть ли в графе клика размера k . Она монотонна: добавив рёбра, мы не разрушим уже имеющуюся клику. Монотонность делает отрицания в схемах ненужными — и именно это открыло путь к нижним оценкам.

Известные результаты скромны. Для монотонных схем — без отрицаний — Александр Разборов доказал в 1985 году суперполиномиальную нижнюю оценку для CLIQUE ($n^{\Omega(\log n)}$)¹⁰¹, а Алон и Боппана в 1987 году — экспоненциальную¹⁰². Отрицание в общих схемах оказалось барьером, который не преодолен до сих пор.

¹⁰⁰R. M. Karp, R. J. Lipton. «Some Connections Between Nonuniform and Uniform Complexity Classes». STOC, 1980.

¹⁰¹А. А. Разборов. «О нижних оценках монотонной сложности некоторых булевых функций». Доклады АН СССР, 1985.

¹⁰²N. Alon, R. B. Boppana. «The Monotone Circuit Complexity of Boolean Functions». Combinatorica, 1987.

Инженерный опыт главы даёт частный ответ: сумматоры получились размера $O(n)$. Теория схемной сложности спрашивает о принципиальном: насколько малой может быть схема для заданной функции при растущем размере входа?

§ 11.8 Итоги главы

Схема — это булева функция, получившая физическую реализацию: вентиль вычисляет операцию, а соединение вентиля — композицию операций. У любой схемы два бюджета — размер и глубина, — и глубина есть физическое время вычисления: критический путь определяет задержку, а значит, и тактовую частоту. Компромисс между бюджетами виден уже на сумматорах: последовательный перенос экономит вентили, опережающий — время.

Инженерная сила подхода — в композиции. Полный сумматор собирается из двух полусумматоров и вентиля ИЛИ, а цепочка полных сумматоров складывает числа любой разрядности, не добавляя новых типов вентиля. Ацикличность превращает схему в булеву функцию — выход определяется только текущими входами, — а чтобы запомнить хоть один бит, нужны обратные связи.

Но у композиции есть предел. Мощностной аргумент Шеннона показывает, что почти все булевы функции требуют экспоненциально больших схем, а явные нижние оценки для общих схем остаются открытой проблемой — и этого не меняет даже полнота набора вентиля, ведь NAND-полнота отвечает на вопрос о выразимости, а не о размере. Код Грея показал, что договорённость о кодировании влияет на физическую надёжность схемы.

Комбинационные схемы — физическая реализация булевых функций из главы 10, и их возможности определяются свойствами самих функций: полнотой наборов вентиля и нижними оценками размера. Теперь мы умеем спроектировать схему по функции, оценить её размер и глубину, а вопрос об эквивалентности двух схем сводится к задаче выполнимости: невыполнимость митера и есть доказательство эквивалентности. Следующий шаг — защита того, что схемы вычисляют и передают: коды, исправляющие ошибки (глава 13).

Проверка Проверьте себя

1. Почему полный сумматор можно собрать из двух полусумматоров и одного вентиля ИЛИ?
2. Где в схеме возникает задержка и почему она ограничивает тактовую частоту?
3. Почему одного NAND достаточно для любой схемы?
4. Чем комбинационная схема отличается от схемы с памятью? Что гарантирует ацикличность?

5. В чём компромисс между сумматором с последовательным переносом и сумматором с опережающим переносом?
6. Что такое митер и как с его помощью проверяют эквивалентность двух схем?

Что дальше?

От физических схем мы переходим к защите информации от ошибок — кодам, исправляющим ошибки. В главе 13 мы познакомимся с расстоянием Хэмминга, кодами Хэмминга, Хаффмана и циклическими кодами — математическим фундаментом надёжной передачи данных.

§ 11.9 Упражнения

Системы счисления.

1. Переведите двоичное число 1101_2 в десятичную систему и десятичное число 23 в двоичную. Проверьте обратным переводом.
2. Переведите 10111110_2 в шестнадцатеричную запись. Почему одна шестнадцатеричная цифра кодирует ровно 4 бита?

Логические вентили и базовые схемы.

3. Запишите таблицы истинности для вентилях NAND, NOR и XOR.
4. Постройте комбинационную схему из базовых вентилях (AND, OR, NOT) для функции $f(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$. Определите размер и глубину схемы.
5. Выразите XOR через NAND. Сколько NAND-вентилей потребовалось?
6. * Реализуйте схему для $f(x, y, z) = (x \oplus y) \wedge \neg z$, используя только вентили NAND. Сравните размер с реализацией на базисе {AND, OR, NOT}.

Сумматоры.

7. Постройте таблицу истинности полусумматора. Выразите сумму и перенос S, C через XOR и AND.
8. Постройте схему полного сумматора из двух полусумматоров и одного вентиля OR. Объясните, почему $C_{\text{out}} = C_1 \vee C_2$, где C_1 и C_2 — переносы от двух полусумматоров.
9. * Сравните ripple-carry и carry-lookahead сумматоры: какой имеет меньшую глубину и почему? Какой платит за это большим размером?

Код Грея.

10. Переведите двоичное число 10110_2 в код Грея и обратно. Проверьте, что результат совпадает с исходным.

11. * Постройте список G_3 по формуле $G_{n+1} = [0G_n, 1G_n]$ — без обращения порядка. Укажите строки полученного списка, в которых нарушается свойство кода Грея.
Верификация и схемная сложность.
12. Постройте митер для проверки эквивалентности двух реализаций XOR: $f_1(x, y) = x \oplus y$, $f_2(x, y) = (x \vee y) \wedge \neg(x \wedge y)$. Почему невыполнимость митера означает эквивалентность схем?
13. * Объясните мощностной аргумент Шеннона: почему почти все булевы функции от n переменных требуют схем экспоненциального размера? Оцените, сколько существует различных схем размера не более 10 вентилях над базисом {NAND} (два входа), и сравните с числом булевых функций от 4 переменных.
14. * Мощностной аргумент доказывает: сложные функции существуют. Почему из него нельзя извлечь ни одной конкретной сложной функции? Где в доказательстве теряется явное построение?

ПРОЕКТ Многоразрядные сумматоры: последовательный, с пропуском и опережающий перенос

Соберите из логических вентилях три многоразрядных сумматора — с последовательным переносом, с пропуском переноса и с опережающим переносом — и убедитесь, что все складывают числа правильно. Постройте каждую схему как ориентированный ациклический граф из полусумматора, полного сумматора и формул генерации и распространения переноса. Измерьте глубину и размер всех схем и объясните, чем быстрый сумматор платит за скорость.

① Схема как граф вентилях

Реализуйте схему как ориентированный ациклический граф: узлы — входы, константы 0/1 и вентили AND, OR, XOR, NOT. Каждый новый вентиль ссылается только на уже добавленные узлы, поэтому номер узла задаёт топологический порядок, и симуляция по значениям входов проходит за один проход. Подсчитайте размер схемы (число вентилях) и глубину (длиннейший путь от входа к выходу), и продумайте, почему глубина — это длина критического пути, а не сумма задержек всех вентилях. На этом каркасе соберите полусумматор из двух вентилях: $S = A \oplus B$, $C = A \wedge B$. Затем соберите полный сумматор двумя способами: из двух полусумматоров и одного вентиля OR, как на рис. 59, и по формуле большинства для переноса. Проверьте, что обе реализации дают одинаковые суммы и переносы на всех комбинациях входов, и объясните, почему бит суммы $S = A \oplus B \oplus C_{\text{in}}$ в обеих схемах один и тот же, а перенос вычисляется по-разному. Соедините n полных сумматоров в цепочку: перенос каждого разряда подаётся на вход следующего. Проверьте, что размер и глубина растут линейно с разрядностью: $O(n)$. Выведите наружу перенос c_{n-1} в старший разряд и перенос c_n из него. Покажите, что беззнаковое переполнение — это $c_n = 1$, а знаковое — $c_n \oplus c_{n-1} = 1$.

② Перенос с пропуском (*carry-skip*)

Между последовательным и опережающим переносом лежит промежуточная конструкция. Разбейте разряды на группы фиксированного размера k . Внутри группы перенос идёт последовательно, как в ripple-carry, и заодно вычисляется, распространяет ли группа перенос насквозь. Между группами поставьте один вентиль пропуска: перенос из группы равен $c_{\text{out}} = G \vee (P \wedge c_{\text{in}})$, где G — перенос, порождённый внутри группы, а P — её суммарное распространение. Измерьте глубину и размер при разных k и найдите компромисс: глубина $O(k + \frac{n}{k})$, минимум вблизи $k \approx \sqrt{n}$. Сравните: при каком n пропуск уже выигрывает у цепочки, а при каком проигрывает из-за накладных расходов на группы.

③ Опережающий перенос

Для каждого разряда вычислите генерацию $G_i = A_i \wedge B_i$ и распространение $P_i = A_i \oplus B_i$. Разбейте разряды на блоки фиксированного размера и вычислите для каждого блока его генерацию и распространение: перенос, порождённый внутри блока, и перенос, проходящий блок насквозь. Два блока объединяются по правилам

$$G = G_{\text{hi}} \vee (P_{\text{hi}} \wedge G_{\text{lo}})$$

$$P = P_{\text{hi}} \wedge P_{\text{lo}},$$

а перенос из блока равен $c_{\text{out}} = G \vee (P \wedge c_{\text{in}})$. Соедините блоки в дерево и раздайте переносы сверху вниз: глубина схемы станет $O(\log n)$ вместо $O(n)$. Измерьте, сколько вентиля добавило ускорение, и объясните, почему выигрыш в глубине оплачен ростом размера. Постройте таблицу глубины и размера обоих сумматоров при $n = 8, 16, 32, 64$ и объясните, как с ростом n расходуется их соотношение: опережающий перенос мельче по глубине, но шире по вентилям. Изобразите этот компромисс графиком: точка на плоскости «глубина vs суммарное число вентиля» для каждой схемы и каждой разрядности. Возьмите логарифмический масштаб обеих осей — иначе точки разных разрядностей слипаются — и покажите, как точки расходятся с ростом n . Разберитесь, что изменится, если соединить блоки цепочкой вместо дерева, или взять блоки другого размера.

④ Знаковый режим

Сумматоры складывают беззнаковые числа. Те же схемы складывают знаковые, если читать биты как дополнительный код: строка $b_{n-1} \dots b_0$ означает $-b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$, поэтому $1000 \dots 0$ — это -2^{n-1} , а не 2^{n-1} . Реализуйте знаковое сложение: интерпретируйте входы и сумму как n -битный дополнительный код и верните флаг переполнения — результат не помещается в n бит. Проверьте граничные случаи при $n = 8$: $127 + 1$ переполняется в -128 , $-128 + (-1)$ переполняется в 127 , а $-1 + 1 = 0$ без переполнения. Проверьте, что знаковый режим работает на обоих сумматорах — цепочке и опережающем переносе. Объясните, почему биты суммы в знаковом и

беззнаковом режиме одни и те же, а переполнение — это расхождение переносов в старший разряд и из него.

⑤ **Матричный умножитель**

Сложение собирается из линейного числа вентилях, умножение — из квадратичного. Соберите матричный умножитель из частичных произведений $p_{i,j} = a_j \wedge b_i$ и сетки полных сумматоров, которая складывает строки со сдвигом. Проверьте произведение против целочисленного умножения: исчерпывающе для малых n и случайно вплоть до 64 бит. Измерьте глубину и число вентилях умножителя рядом с сумматорами в одной таблице: размер $O(n^2)$ против $O(n)$ у сложения.

⑥ **Верификация**

Реализуйте эталонное сложение на целых числах и сверяйте с ним суммы схемы: биты суммы и перенос из старшего разряда. Для малых n переберите все пары входов — их 2^{2n} — и проверьте каждую пару. Для больших n возьмите случайные пары и проверьте суммы, переносы и переполнение. Проверьте, что обе схемы дают одинаковые результаты на одних и тех же парах. Объясните, почему исчерпывающий перебор невозможен при большой разрядности и что случайная проверка может пропустить.

Итог: три корректные реализации многоразрядного сумматора — с последовательным переносом, с пропуском и с опережающим — матричный умножитель, знаковый режим в дополнительном коде с флагом переполнения, и измеренный компромисс «глубина vs вентилях» в таблице и на графике, а также ответ на вопрос, чем схема платит за скорость.

XII

Алгебраические структуры

На циферблате три часа дня. Прибавить десять часов — стрелка укажет час, а не тринадцать. Круг замыкает счёт: после двенадцати снова один.

Квадрат поворачивают на четверть оборота — он совпадает с самим собой. Поворот отменяется обратным поворотом, два поворота подряд снова дают поворот, четыре возвращают квадрат на место.

У часов и квадрата нет общих частей, но есть общий закон. И там, и там действия выполняют подряд и отменяют. Именно этот закон — а не стрелки и не вершины — составляет предмет алгебры.

Тот же закон узнаётся в системах, где нет ни циферблатов, ни вершин. Тасовки колоды: две подряд — снова перестановка, и у каждой перестановки есть обратная. Остатки при делении на простое: сложение и умножение по модулю не выводят за пределы системы, и у каждого ненулевого остатка есть обратный по умножению. Ходы кубика Рубика: любая последовательность ходов отменяется обратной последовательностью.

Каждую систему можно описывать отдельно — таблицами сложения, каталогами поворотов, шпаргалками раскладов. Но повторяющемуся закону тесно в частных описаниях: его записывают один раз, а теоремы о нём доказывают один раз — сразу для всех носителей. Теорема Лагранжа, доказанная для абстрактной структуры, одновременно работает и на остатках, и на тасовках, и на кубике.

Ставка выше, чем экономия чернил. На этих законах стоят практические машины: защита связи — арифметика остатков (глава 19), коды, исправляющие ошибки, — конечные поля (глава 13). Одна и та же абстракция работает и в криптографии, и в кодировании — системах, на вид не имеющих ничего общего.

Чем больше систем подпадает под один закон, тем меньше случайного остаётся в каждом отдельном доказательстве. И тем острее вопрос о самих правилах.

Какие правила превращают набор действий в структуру, и почему эта структура возникает сразу в нескольких непохожих местах?

ИСТОРИЯ Галуа и рождение абстрактной алгебры

Эварист Галуа прожил двадцать лет и почти всю сознательную жизнь посвятил одной задаче — условию разрешимости уравнений в радикалах. Уравнение второй степени решается через корень, третьей и четвёртой — через корни и

арифметику, а пятой — уже нет. Вопрос, почему для пятой степени общего рецепта не существует, занимал математиков с шестнадцатого века.

Галуа нашёл ответ, и ответ этот был про симметрии уравнений. Каждому уравнению он сопоставил группу — набор преобразований его корней, не меняющих формулу. Разрешимость уравнения оказалась связана со строением этой группы. Группа распадается на части определённого вида — уравнение решается в радикалах, не распадается — не решается.

Так алгебра повернулась от формул к структурам. Группы, кольца и поля стали изучать сами по себе, независимо от конкретных чисел, на которых они заданы. Поля, число элементов которых равно степени простого, носят имя Галуа.

ОБЗОР ГЛАВЫ

Глава выстраивает лестницу алгебраических структур и показывает, как каждая ступень добавляет возможность. Мы начнём с операции — правила, сопоставляющего паре элемент — и разберём, какие требования превращают её в структуру: ассоциативность, нейтральный элемент, обратимость. Затем добавим вторую операцию и дойдём до поля, где можно делить. Параллельно появятся инструменты, работающие на любой ступени: подгруппы и теорема Лагранжа, действие группы на множестве, гомоморфизмы и изоморфизмы, векторные пространства. В конце мы увидим, как эти структуры становятся кодами. Отдельная факультативная секция покажет боковые ступени лестницы — полурешётку и полукольцо.

После этой главы вы сможете:

1. распознавать операцию и проверять её свойства, определять полугруппу, моноид, группу;
2. выписывать таблицу Кэли группы и находить порядки элементов;
3. применять теорему Лагранжа и считать подгруппы и смежные классы;
4. отличать кольцо от поля, находить делители нуля и обратные элементы;
5. строить конечные поля $\text{GF}(p^m)$ по неприводимому многочлену и находить примитивный элемент;
6. объяснять изоморфизм структур и использовать векторные пространства над полем.

§ 12.1 Операция

Возьмём калькулятор и нажмём подряд две кнопки. Семь плюс пять — двенадцать, семь умножить на пять — тридцать пять. Оба раза мы подали на вход два числа, а на выходе получили одно. Форма у обеих кнопок одна: два значения внутрь, одно наружу. Правила у них разные, и в этой разнице живёт то, о чём пойдёт речь.

Тот же жест встречается далеко за пределами чисел. Программа берёт две строки и склеивает их: «ка» и «ша» дают «каша». Логическая схема складывает два бита по модулю два: $1 \oplus 1$ даёт 0. База данных объединяет два множества: $\{1, 2\}$ и $\{2, 3\}$ дают $\{1, 2, 3\}$. Повсюду одно и то же: два элемента на входе, один на выходе.

Математика собрала этот жест в одно понятие. *Бинарная операция* — правило, которое каждой упорядоченной паре элементов множества A ставит в соответствие элемент того же множества A . Слово «бинарная» указывает на два аргумента, а «операция» указывает на то, что из этих аргументов получается результат.

ОПРЕДЕЛЕНИЕ 12.1 Бинарная операция

Бинарной операцией на множестве A называется функция

$$f : A \times A \rightarrow A.$$

Упорядоченной паре (a, b) она сопоставляет результат, который записывают как $a \star b$.

Замкнутость встроена в определение в самом его виде. Сложили числа — получили число, подставили его в операцию ещё раз — снова число. Результат никогда не покидает множество, и потому операцию можно повторять сколько угодно, накапливая итог. Без замкнутости сворачивать список в одно значение было бы нелегко: промежуточный результат мог бы выпасть из A .

Пример Операции, которые мы постоянно используем

1. Сложение натуральных чисел: пара (a, b) даёт $a + b$, и результат снова натуральное число.
2. Конкатенация строк: пара ab, cd даёт $abcd$, и результат снова строка.
3. XOR на битах: пара $(0, 1)$ даёт 1, а пара $(1, 1)$ даёт 0, и результат снова бит.
4. Объединение множеств: множество $\{1, 2\}$ вместе с $\{2, 3\}$ даёт $\{1, 2, 3\}$, и результат снова множество.
5. Максимум на натуральных числах: пара $(3, 7)$ даёт 7, и результат снова натуральное число.

У всех этих операций одна форма, а поведение разное. Дело в том, какими свойствами операция обладает. Свойств четыре, и каждое — своё определение.

ОПРЕДЕЛЕНИЕ 12.2 Ассоциативность

Операция $\star$ на множестве A называется **ассоциативной**, если для всех $a, b, c \in A$ выполнено

$$(a \star b) \star c = a \star (b \star c).$$

При ассоциативности скобки не влияют на результат, и запись $a \star b \star c$ становится однозначной.

Сложение ассоциативно: $(3 + 4) + 5 = 3 + (4 + 5) = 12$. Вычитание не ассоциативно: $(10 - 3) - 2 = 5$, а $10 - (3 - 2) = 9$. Где ассоциативность есть, скобки можно смело опускать. Где её нет — порядок вычислений приходится фиксировать.

ОПРЕДЕЛЕНИЕ 12.3 Коммутативность

Операция $\star$ на множестве A называется **коммутативной**, если для всех $a, b \in A$ выполнено

$$a \star b = b \star a.$$

При коммутативности порядок аргументов не важен.

Коммутативны четыре операции:

1. сложение;
2. XOR;
3. объединение множеств;
4. максимум.

Конкатенация не коммутативна: $ab \star cd = abcd$, $a cd \star ab = cdab$.

ОПРЕДЕЛЕНИЕ 12.4 Нейтральный элемент

Элемент $e \in A$ называется **нейтральным** для операции $\star$, если для всех $a \in A$ выполнено

$$a \star e = e \star a = a.$$

Нейтральный элемент ничего не меняет.

Нейтральные элементы операций:

1. у сложения — ноль;
2. у конкатенации — пустая строка;
3. у XOR — бит 0;
4. у объединения — пустое множество;
5. у максимума на $\{0, 1\}$ — ноль, ведь $\max(0, a) = a$.

ОПРЕДЕЛЕНИЕ 12.5 Обратный элемент

Пусть e — нейтральный элемент для операции $\star$. Элемент a^{-1} называется **обратным** к a , если

$$a \star a^{-1} = a^{-1} \star a = e.$$

Обратные элементы:

1. у сложения на целых числах к a обратен $-a$;
2. у XOR каждый бит обратен сам себе, ведь $0 \star 0 = 0$ и $1 \star 1 = 0$;

3. у сложения на натуральных числах обратного нет — для 3 не существует такого $x \in \mathbb{N}$, что $3 + x = 0$.

Множество с ассоциативной операцией называется *полугруппой*. Если у неё есть ещё и нейтральный элемент, это *моноид*. Если обратный элемент есть у каждого элемента, это *группа*. Все три имени вырастают из одной операции и трёх её свойств. Коммутативность в лестницу не входит — она вернётся как признак абелевых групп.

На конечном множестве операцию можно записать одной таблицей целиком. Таблица Кэли — квадрат, в котором строки и столбцы подписаны элементами множества, а в клетке стоит результат операции. Строка a , столбец b , в пересечении — $a \star b$. Это таблица умножения операции: по ней операция восстанавливается без всяких формул.

Пример Таблица Кэли для XOR и максимума

Операция XOR на множестве $\{0, 1\}$ задаётся такой таблицей Кэли.

$\star$	0	1
0	0	1
1	1	0

В этом квадрате 0 — нейтральный элемент: применив XOR с нулём, мы ничего не меняем. На диагонали стоят нули, то есть каждый бит обратен самому себе. Значит, XOR на $\{0, 1\}$ обладает всеми четырьмя свойствами сразу. Такую операцию мы скоро назовём группой.

Операция максимума на том же множестве $\{0, 1\}$ выглядит иначе.

$\star$	0	1
0	0	1
1	1	1

Нейтральный элемент снова прост — 0. А вот обратных элементов есть не у всех: для 1 не найдётся такого x , что $\max(1, x) = 0$. Так максимум обладает тремя свойствами из четырёх, но до четвёртого — обратного элемента — не дотягивает.

Не всякое правило комбинирования — операция. Вычитание на натуральных числах ломается на первом же примере: $2 - 3 = -1$, а -1 не натуральное число. Результат выпадает из множества, и цепочка обрывается: вычитание не замкнуто на $\mathbb{N}$. Деление устроено ещё хуже: $\frac{1}{2}$ не целое число, а $\frac{1}{0}$ не определено вовсе. Такие правила — частичные функции, знакомые нам по главе 6: они определены лишь на части множества, поэтому операцией не являются.

Проверка Операция

1. Почему вычитание на натуральных числах не является бинарной операцией, а сложение является?

2. Какие из операций — конкатенация, XOR, объединение, максимум — коммутативны?
3. У максимума на $\{0, 1\}$ есть нейтральный элемент. Почему при этом у него нет обратных элементов?

§ 12.2 Полугруппа и моноид

Нам дали список из тысячи чисел и попросили получить одно число — сумму. Мы берём первое число, держим его в уме, добавляем второе, потом третье, и так до конца. Операция повторяется над уже накопленным значением, и весь процесс идёт только вперёд. Такую свёртку списка программисты называют **fold**: аккумулятор растёт, а отмены в нём не предусмотрено.

Проверим, зависит ли результат от того, как мы расставляем скобки. Пусть нужно сложить 1, 2, 3. Можно сначала взять $1 + 2 = 3$, а потом прибавить оставшуюся тройку: получится $(1 + 2) + 3 = 6$. Можно сгруппировать иначе: $1 + (2 + 3) = 1 + 5 = 6$. Результат один и тот же, и именно это свойство скобок мы сейчас выделим.

Свойство называется *ассоциативностью*. Операция ассоциативна, если результат не зависит от расстановки скобок. Множество с такой операцией заслуживает собственного имени, потому что накопление встречается повсюду.

ОПРЕДЕЛЕНИЕ 12.6 Полугруппа

Множество S вместе с бинарной операцией $\star : S \times S \rightarrow S$ называется **полугруппой**, если операция ассоциативна:

$$(a \star b) \star c = a \star (b \star c)$$

для всех $a, b, c \in S$.

Требование к операции всего одно — ассоциативность. Замкнутость уже встроена в запись: $a \star b \in S$ для любых $a, b \in S$. Полугруппа — это место, где можно накапливать. Отмены здесь не требуется: обратные элементы не обязаны существовать. Это отличает полугруппу от той конструкции, к которой мы перейдём в следующем разделе.

Полугруппы вокруг нас:

1. натуральные числа со сложением;
2. строки с конкатенацией;
3. множества с объединением.

Операции разные, а закон один — скобки не важны.

Теперь добавим элемент, который ничего не меняет. При сложении это ноль: $0 + a = a + 0 = a$. При конкатенации это пустая строка: склеить строку с пустой —

оставить её как есть. У операции $\max$ нейтральным служит наименьший элемент области: на множестве $\mathbb{N}_0 = \{0, 1, 2, \dots\}$ (натуральные с нулём) это ноль, ведь $\max(0, a) = a$. Такой элемент называют *нейтральным*, и его присутствие превращает полугруппу в моноид.

ОПРЕДЕЛЕНИЕ 12.7 Моноид

Полугруппа называется **моноидом**, если в ней существует нейтральный элемент $e \in S$ такой, что

$$e \star a = a \star e = a$$

для всех $a \in S$.

Разница между полугруппой и моноидом ровно одна, и это наличие нейтрального элемента. Различие имеет практический смысл. Чтобы свернуть список функцией `fold`, нужен стартовый аккумулятор. Если операция образует моноид, аккумулятором служит нейтральный элемент: свертка пустого списка даёт e , а свертка списка $[a, b]$ — значение $e \star a \star b$. В полугруппе без нейтрального стартовать не с чего, и свертка пустого списка просто не определена.

Пример Свёртка списка в Rust

В Rust свёртка — метод `fold` у итератора: он принимает начальное значение и замыкание, накапливающее результат.

```
let nums = vec![1, 2, 3, 4];

// Сложение образует моноид: нейтральный элемент --- ноль.
// Свёртка пустого списка даёт ноль.
let sum = nums.iter()
    .fold(0, |acc, x| acc + x); // 10
let sum_empty: i32 = Vec::<i32>::new().iter()
    .fold(0, |acc, x| acc + x); // 0

// Конкатенация строк --- тоже моноид: нейтральна пустая строка.
let joined = ["ка", "ша"].iter()
    .fold(String::new(), |acc, s| acc + s); // "каша"

// У `fold` есть только вариант с начальным значением.
// Итератор предоставляет отдельный `reduce`, который на пустом списке
// вернёт `None`.
let maybe: Option<i32> = vec![1, 2, 3, 4].into_iter()
    .reduce(|acc, x| acc + x); // Some(10)
let empty: Option<i32> = Vec::<i32>::new().into_iter()
    .reduce(|acc, x| acc + x); // None
```

Начальное значение — ровно нейтральный элемент: для сложения это ноль, для конкатенации — пустая строка. `reduce` без начального значения на пустом списке даёт `None`, потому что полугруппе не на чем стартовать.

У моноида нейтральный элемент единственный. Это свойство пригодится дальше, когда появятся сразу два нейтральных элемента: у сложения и у умножения.

УТВЕРЖДЕНИЕ 12.1 Единственность нейтрального элемента

Если в полугруппе существует нейтральный элемент, то он единственный.

Доказательство

Пусть e и e' — два нейтральных элемента.

Нейтральность e' даёт $e \star e' = e$, а нейтральность e даёт $e \star e' = e'$. Значит, $e = e \star e' = e'$, то есть нейтральный элемент единственный.

Пример Состояния автомата

В конечном автомате с множеством состояний Q каждый входной символ задаёт преобразование состояний. Цепочке символов соответствует композиция таких преобразований. Множество всех преобразований $Q \rightarrow Q$ с операцией композиции образует моноид. Ассоциативность наследуется от композиции функций, а нейтральным служит тождественное преобразование, переводящее каждое состояние в себя. Автомат использует лишь подмножество этих преобразований, порождённое его символами.

Проверка Полугруппа и моноид

1. Почему множество положительных целых чисел со сложением образует полугруппу, но нейтрального элемента не имеет?
2. Почему у строк нейтральным элементом обязана быть пустая строка, и может ли существовать второй, отличный от неё нейтральный элемент?
3. Чем отличается свертка списка в моноиде от свертки в полугруппе, где нейтрального элемента нет?

§ 12.3 Группа

Симметрия — идея о том, что фигуру можно преобразовать, и она останется собой. Поворот квадрата на четверть оборота даёт тот же квадрат: положение вершин меняется, фигура — нет. Отражение относительно диагонали тоже возвращает квадрат на место. Интуиция здесь прозрачна: преобразование сохраняет форму объекта.

Но эта интуиция шире, чем нужно. Любое отображение множества в себя формально «переводит фигуру в себя» — образ точки остаётся точкой фигуры, но несколько точек могут слипнуться в одну. Постоянное отображение, стягивающее весь квадрат в одну точку, таким преобразованием является, и при этом оно безвозвратно уни-

чтожает форму. Проекция на прямую переводит квадрат в его тень, и из тени фигуру не собрать.

Значит, дело в обратимой стороне преобразования. Обратимое преобразование можно отменить: если квадрат повернули, найдётся поворот, возвращающий его на место. Отражение отменяется самим собой, а поворот — обратным поворотом. Именно пара «преобразование и его отмена» превращает набор преобразований в алгебраическую структуру — группу.

Посмотрим на обратимые преобразования чуть пристальнее. Возьмём лист бумаги, лежащий на столе. Каждое перемещение листа — сдвиг, поворот, переворот — отменяется другим перемещением: сдвигом в обратную сторону, поворотом назад, повторным переворотом. Последовательность двух перемещений снова перемещение, так что набор замкнут: из него мы не выходим. И у любого перемещения есть обратное, так что движение можно отменить. Это и есть группа: множество, замкнутое относительно операции, где операция сочетается и у каждого элемента есть обратный.

Формально это три условия, и каждое проверяется явно. Замкнутость сюда не входит — она уже встроена в само понятие операции. Ассоциативность позволяет не думать о скобках при трёх и более сомножителях. Нейтральный элемент ничего не меняет, а обратный элемент отменяет сделанное. Первое условие мы уже видели в полугруппах, второе — в моноидах. Новое здесь — третье, требование обратного элемента. Операцию группы дальше записываем точкой $\cdot$, как умножение, а не звёздочкой $\star$.

ОПРЕДЕЛЕНИЕ 12.8 Группа

Множество G с бинарной операцией $\cdot$ называется **группой**, если выполнены три условия:

1. **Ассоциативность**: для всех $a, b, c \in G$ верно $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. **Нейтральный элемент**: существует такой элемент $e \in G$, что $e \cdot a = a \cdot e = a$ для всех $a \in G$.
3. **Обратный элемент**: для каждого $a \in G$ существует такой элемент $a^{-1} \in G$, что $a \cdot a^{-1} = a^{-1} \cdot a = e$.

Из трёх условий сразу следует возможность сокращать.

УТВЕРЖДЕНИЕ 12.2 Сокращение в группе

В группе из равенства $a \cdot x = a \cdot y$ следует $x = y$, а из $x \cdot a = y \cdot a$ следует $x = y$.

Доказательство

Умножим равенство $a \cdot x = a \cdot y$ слева на a^{-1} . Получим $a^{-1} \cdot a \cdot x = a^{-1} \cdot a \cdot y$, то есть $e \cdot x = e \cdot y$, а нейтральный элемент оставляет x и y на месте, и выходит $x = y$.

Второе равенство, $x \cdot a = y \cdot a$, умножается справа на a^{-1} и доказывается так же.

Классический пример группы — симметрии квадрата. Повернём квадрат на 90 градусов — он совпадёт с собой, хотя вершины поменяются местами. Перечислим преобразования, переводящие квадрат в себя. Четыре поворота: на 0, 90, 180 и 270 градусов. И четыре отражения: относительно двух средних линий и двух диагоналей. Итого восемь.

Обозначим поворот на 90 градусов через r , а отражение относительно вертикальной оси — через s . Повторяя r , получаем r^2 (поворот на 180 градусов) и r^3 (на 270 градусов), а r^4 возвращает квадрат на место. Произведение $a \cdot b$ понимаем как последовательное выполнение: сначала b , затем a . Каждый поворот и каждое отражение обратимы, поэтому все восемь преобразований образуют группу — её обозначают D_4 и называют группой симметрий квадрата. Вся её структура видна в таблице Кэли (рис. 4): в строке a и столбце b стоит произведение $a \cdot b$, то есть результат выполнения сначала b , затем a .

$\cdot$	e	r	r^2	r^3	s	rs	r^2s	r^3s
e	e	r	r^2	r^3	s	rs	r^2s	r^3s
r	r	r^2	r^3	e	rs	r^2s	r^3s	s
r^2	r^2	r^3	e	r	r^2s	r^3s	s	rs
r^3	r^3	e	r	r^2	r^3s	s	rs	r^2s
s	s	r^3s	r^2s	rs	e	r^3	r^2	r
rs	rs	s	r^3s	r^2s	r	e	r^3	r^2
r^2s	r^2s	rs	s	r^3s	r^2	r	e	r^3
r^3s	r^3s	r^2s	rs	s	r^3	r^2	r	e

Таблица 4. Таблица Кэли группы D_4 .

Столбец и строка, отвечающие e , повторяют шапку — нейтральный элемент ничего не меняет. Каждая строка и каждый столбец содержат каждый из восьми элементов ровно один раз. Это так называемое свойство латинского квадрата, и оно прямое следствие закона сокращения: уравнение $a \cdot x = b$ имеет единственное решение $x = a^{-1} \cdot b$. Если бы обратного у a не было, произведение $a \cdot x$ могло бы повторяться, и в строке случались бы пропуски.

Таблица не симметрична относительно диагонали, и это тоже не случайность. Посмотрим на клетки строки s и r : $s \cdot r = r^3s$, а $r \cdot s = rs$. Повернуть и отразить — одно, отразить и повернуть — другое, поэтому порядок сомножителей важен. Группа D_4 не коммутативна, и такие группы называют неабелевыми. Абелевой называют коммутативную группу, и это слово появится снова в кольцах и полях, где сложение коммутативно всегда.

Другой привычный пример — сложение остатков. Множество $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$ с операцией «сложить и взять остаток от деления на n » образует группу. Ноль — нейтральный элемент, а обратное к a равно $n-a$ (для $a=0$ к самому нулю).

Прибавив единицу n раз, мы пробежим все элементы $\mathbb{Z}_n$ по кругу, поэтому единица порождает всю группу.

ОПРЕДЕЛЕНИЕ 12.9 Образующий

Элемент g группы G называется **образующим**, если его повторные применения дают все элементы группы. Иначе говоря, каждый элемент G равен g^k для некоторого целого k .

В $\mathbb{Z}_8$ образующим служит единица: её повторные прибавления обходят все восемь элементов по кругу (рис. 63). Элемент 2 образующим не является — его повторные сложения дают 2, 4, 6, 0 и зацикливаются, не тронув 1, 3, 5, 7.

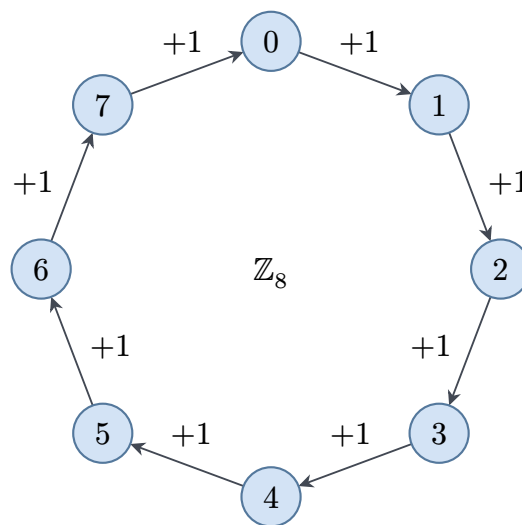


Рис. 63. Циклическая группа $\mathbb{Z}_8$ по сложению.

Группа, у которой есть образующий, называется циклической.

ОПРЕДЕЛЕНИЕ 12.10 Циклическая группа

Группа G называется **циклической**, если в ней существует образующий элемент.

Цикличность — сильное ограничение, и оно полностью описывает группу.

ТЕОРЕМА 12.3 Строение циклической группы

Всякая циклическая группа устроена так же, как $\mathbb{Z}_n$ по сложению для некоторого n , либо как $\mathbb{Z}$ по сложению, если она бесконечна. Слово «устроена так же» здесь означает «изоморфна» — структуры совпадают с точностью до переименования элементов.

Набросок доказательства

Фиксируем образующий g — тогда каждый элемент группы имеет вид g^k .

Сопоставим элементу g^k целое число k . Это соответствие сохраняет операцию, ведь $g^i \cdot g^j = g^{i+j}$, то есть произведению элементов отвечает сумма показателей.

Если степени g не повторяются, соответствие взаимно однозначно, и группа изоморфна $\mathbb{Z}$ по сложению. Если же $g^m = e$ для некоторого $m > 0$, берём наименьшее такое m — порядок g . Степени $g^0, g^1, \dots, g^{m-1}$ различны, а следующая возвращается к $g^0 = e$, поэтому показатели складываются по модулю m и группа изоморфна $\mathbb{Z}_m$.

Все группы вида $\mathbb{Z}_n$ циклические, и их образующих несколько: в $\mathbb{Z}_6$ это 1 и 5. Но не всякая группа циклическая: D_4 порядка 8 циклической не является, ведь в ней нет элемента порядка 8.

С умножением та же идея работает иначе. На $\mathbb{Z}_n$ умножение группу не образует: в $\mathbb{Z}_6$ числу 2 нет обратного, потому что $2 \cdot x$ всегда чётно и к единице по модулю 6 не привести. Обратимые по модулю n элементы — ровно взаимно простые с n , и способ находить обратный через соотношение Безу выведет глава 19. Для простого модуля p взаимно просты с p все числа от 1 до $p - 1$, и они образуют группу по умножению. Обозначается эта группа $\mathbb{Z}_p^*$.

Пример Мультипликативная группа $\mathbb{Z}_5^*$

Возьмём $p = 5$. Множество $\mathbb{Z}_5^* = \{1, 2, 3, 4\}$ замкнуто относительно умножения по модулю 5 (произведение двух ненулевых остатков на 5 не делится), а обратный элемент найдётся у каждого.

$$a^{-1} \bmod 5 : \quad 1 \rightarrow 1, \quad 2 \rightarrow 3, \quad 3 \rightarrow 2, \quad 4 \rightarrow 4.$$

Проверим одно из обратных: $3 \cdot 2 = 6 \equiv 1 \pmod{5}$. Число степеней до возвращения к единице называют *порядком* элемента, и мы определим его точно сразу после примера. Элемент 4 имеет порядок 2, так как $4^2 = 16 \equiv 1$. Степени двойки: $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8 \equiv 3, 2^4 = 16 \equiv 1$. Они пробегают все четыре элемента $\mathbb{Z}_5^*$, поэтому 2 — образующий и группа $\mathbb{Z}_5^*$ циклическая.

Посмотрим на степени элемента внимательнее. В конечной группе список $g, g^2, g^3, \dots$ обязан повториться: элементов конечное число, а степеней бесконечно много.

ОПРЕДЕЛЕНИЕ 12.11 Порядок элемента

Порядком элемента g конечной группы называется наименьшее положительное k , при котором $g^k = e$. Обозначается $\text{ord}(g)$.

Слово «порядок» здесь используется в двух близких смыслах, и их стоит развести. Порядок группы — это число её элементов. Это разные величины: порядок элемента g не превосходит порядок группы, а по теореме Лагранжа и делит его. В $\mathbb{Z}_5^*$ элемент

4 имеет порядок 2, в D_4 поворот r имеет порядок 4, а каждое отражение — порядок 2. Сам нейтральный элемент имеет порядок 1. Порядок элемента совпадает с порядком группы ровно тогда, когда элемент — образующий, и это ещё один взгляд на циклическую группу.

Замечание Почему период степеней упирается в обратимость

В конечном моноиде, как и в группе, степени любого элемента рано или поздно зацикливаются. Цикл, однако, не обязан проходить через нейтральный элемент. Возьмём умножение целых чисел и элемент ноль. Его степени — $0^0 = 1$ (нулевая степень — это нейтральный элемент), $0^1 = 0$, $0^2 = 0$, и дальше одни нули. Последовательность $1, 0, 0, \dots$ уходит в хвост из нулей и к единице не возвращается.

В группе такого хвоста нет. Пусть две степени совпали: $g^i = g^j$ при $i > j$. По закону сокращения умножим обе части на обратный к g^j элемент, то есть на g^{-j} , и получим $g^{i-j} = e$. Значит, у g есть положительная степень, равная e , и наименьшая такая степень — порядок — существует. После первого возвращения к e степени повторяются с начала: $g^{k+1} = g$, $g^{k+2} = g^2$, и так далее. В группе последовательность степеней — чистый круг через e .

В примере с нулём ломается шаг сокращения: у нуля нет обратного, поэтому из $0^i = 0^j$ нельзя вывести $0^{i-j} = e$.

Группы перестановок дают самое наглядное применение этой структуры. Перестановка набора из n объектов — это обратимое преобразование: её можно отменить обратной перестановкой. Все перестановки n объектов с операцией композиции образуют симметрическую группу S_n , и элементов в ней $n!$. Группа S_3 из шести перестановок трёх объектов устроена как D_3 , группа симметрий треугольника. Кубик Рубика — конечная группа перестановок: каждый поворот грани переставляет угловые и рёберные кубики, а любое достижимое положение — элемент этой группы. Число достижимых положений равно 43252003274489856000 — примерно 4.3×10^{19} . Самую сложную позицию можно разобрать за 20 ходов, и это число называют числом бога. Проверка достижимости положения — проверка принадлежности элементу группы, и она сводится к условиям чётности на перестановках.

Проверка Группа и обратимость

1. Почему постоянное отображение, стягивающее квадрат в точку, не считается симметрией?
2. Какие восемь элементов образуют группу D_4 и каков порядок каждого из них?
3. Сколько элементов в группе $\mathbb{Z}_n$ по сложению и какой элемент является её образующим?
4. Почему $\mathbb{Z}_6$ с умножением не является группой, а $\mathbb{Z}_5^*$ — является?
5. Как порядок элемента связан с существованием обратного к нему?

§ 12.4 Подгруппы и теорема Лагранжа

Вернёмся к группе вычетов по модулю 12: множество $\mathbb{Z}_{12} = \{0, 1, \dots, 11\}$ со сложением по модулю. Отберём из $\mathbb{Z}_{12}$ только чётные числа $\{0, 2, 4, 6, 8, 10\}$ и посмотрим, что получится. Сумма двух чётных вычетов снова чётна, поэтому сложение не выводит нас за пределы отобранного множества. Противоположный к чётному вычету тоже чётен: $-2 = 10$, $-6 = 6$ по модулю 12. Значит, внутри $\mathbb{Z}_{12}$ сидит своя маленькая группа на шесть элементов, устроенная по тем же правилам.

Такой отбор — подмножество, которое само является группой относительно той же операции — называют подгруппой. Подгруппа — кусок исходной группы, который живёт своей жизнью: сложение внутри него не требует выйти наружу, а обратный элемент всегда лежит рядом. Раз зашла речь о группах в группах, зафиксируем понятие строго.

ОПРЕДЕЛЕНИЕ 12.12 Подгруппа

Непустое подмножество $H \subseteq G$ называется **подгруппой** группы G , если для любых $a, b \in H$ выполнены оба условия:

1. замкнутость относительно операции: $a \cdot b \in H$;
2. замкнутость относительно обратного: $a^{-1} \in H$.

В определении нет ни слова про нейтральный элемент. Он появляется сам: если H непусто, возьмём $a \in H$, и тогда $a \cdot a^{-1} = e \in H$ по двум условиям сразу. Так что e оказывается в H бесплатно, и отобранное множество действительно группа.

Условие замкнутости относительно операции само по себе ещё не делает из множества подгруппу. В бесконечной группе целых $\mathbb{Z}$ со сложением неотрицательные числа $\{0, 1, 2, \dots\}$ замкнуты относительно сложения, но обратный ко 2 равен -2 , и в отобранном множестве его нет. В конечной группе картина иная: там одной замкнутости достаточно.

УТВЕРЖДЕНИЕ 12.4 Критерий подгруппы в конечной группе

Пусть G — конечная группа, а H — непустое подмножество, замкнутое относительно операции. Тогда H — подгруппа группы G .

Доказательство

Достаточно показать, что H замкнуто относительно обратного.

Возьмём $a \in H$ и рассмотрим степени $a, a^2, a^3, \dots$. Все они лежат в H , потому что $a \in H$ и H замкнута относительно операции.

В конечной группе этот список обязан повториться: найдутся $i > j$ с $a^i = a^j$. По закону сокращения умножим обе части на a^{-j} и получим $a^{i-j} = e$. Показатель $i - j$ положителен, так что e — тоже степень a , а значит $e \in H$.

Обратный к a — это a^{i-j-1} : произведение $a \cdot a^{i-j-1} = a^{i-j} = e$. Этот элемент — степень a , поэтому он лежит в H .

Пример Подгруппы $\mathbb{Z}_{12}$

Выпишем подгруппы $\mathbb{Z}_{12}$ и их порядки.

Подгруппа H	Порядок $ H $	Порождает элемент
$\{0\}$	1	0
$\{0, 6\}$	2	6
$\{0, 4, 8\}$	3	4
$\{0, 3, 6, 9\}$	4	3
$\{0, 2, 4, 6, 8, 10\}$	6	2
$\mathbb{Z}_{12}$	12	1

Подгруппа, порождённая элементом g , — это все его кратные $0, g, 2g, \dots$, пока не вернёмся к нулю. Для $g = 4$ это $0, 4, 8$: три элемента, и $3 \cdot 4 = 12 = 0$. Для $g = 6$ это $0, 6$: два элемента, и $2 \cdot 6 = 12 = 0$. Каждый порядок в таблице делит 12 — и это предвестие теоремы Лагранжа.

Шесть подгрупп $\mathbb{Z}_{12}$ складываются в решётку (рис. 64): узел $\langle g \rangle$ — подгруппа, порождённая элементом g , а ребро спускается от большей подгруппы к содержащейся в ней. Наверху — вся группа $\langle 1 \rangle = \mathbb{Z}_{12}$, внизу — тривиальная $\langle 0 \rangle = \{0\}$, а четыре промежуточных узла — подгруппы порядков 6, 4, 3 и 2.

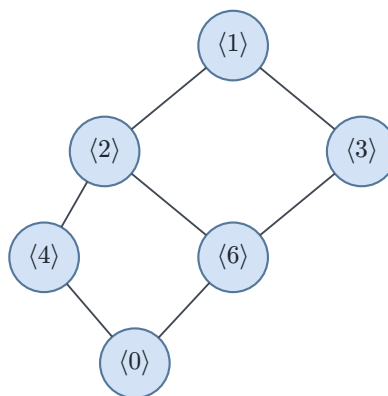


Рис. 64. Решётка подгрупп $\mathbb{Z}_{12}$.

Раз подгруппа живёт внутри группы, попробуем её сдвинуть.

ОПРЕДЕЛЕНИЕ 12.13 Смежный класс

Пусть H — подгруппа группы G , и пусть $a \in G$. Левый смежный класс подгруппы H по элементу a — это множество

$$aH = \{a \cdot h \mid h \in H\}.$$

В аддитивной записи, как в $\mathbb{Z}_{12}$, умножение читается как сложение: $a + H = \{a + h \mid h \in H\}$.

Смежный класс получается, когда каждый элемент подгруппы умножают на один и тот же элемент a . Смежные классы разбивают группу на части одинакового размера. Посмотрим, как это выглядит в $\mathbb{Z}_{12}$ для подгруппы $H = \{0, 4, 8\}$, у которой три элемента.

Смежный класс	Элементы	Число элементов
$0 + H$	$\{0, 4, 8\}$	3
$1 + H$	$\{1, 5, 9\}$	3
$2 + H$	$\{2, 6, 10\}$	3
$3 + H$	$\{3, 7, 11\}$	3

Каждый класс состоит из трёх элементов, а классов ровно четыре: $12 = 4 \cdot 3$. Дальнейшие сдвиги повторяют уже найденные классы: $4 + H = \{4, 8, 0\} = 0 + H$, и так далее. Два разных класса не имеют общих элементов, а вместе покрывают всю группу. Это и есть разбиение, и на нём стоит теорема Лагранжа.

ТЕОРЕМА 12.5 Теорема Лагранжа

Пусть G — конечная группа, а H — её подгруппа.

Тогда порядок $|H|$ делит порядок $|G|$.

Доказательство

Докажем, разбив G на попарно непересекающиеся смежные классы одинакового размера.

Покажем сначала, что каждый смежный класс $aH = \{a \cdot h \mid h \in H\}$ содержит ровно $|H|$ элементов. Отображение $h \mapsto a \cdot h$ из H в aH инъективно, ведь из $a \cdot h_1 = a \cdot h_2$ закон сокращения даёт $h_1 = h_2$, и сюръективно, ведь aH по определению состоит из элементов вида $a \cdot h$. Значит, это биекция, и $|aH| = |H|$ для любого a .

Теперь покажем, что два смежных класса либо совпадают, либо не пересекаются. Пусть aH и bH имеют общий элемент c , то есть $c = a \cdot h_1 = b \cdot h_2$ для некоторых $h_1, h_2 \in H$. Тогда $a = b \cdot h_2 \cdot h_1^{-1}$, а это произведение элемента b и двух элементов подгруппы H , поэтому $a \in bH$. Подставив это выражение в произвольный элемент $a \cdot h$ класса aH , получаем $a \cdot h = b \cdot (h_2 \cdot h_1^{-1} \cdot h)$, где произведение в

скобках лежит в H , и весь элемент попадает в bH . Значит, $aH \subseteq bH$, а то же рассуждение с переменной a и b местами даёт обратное включение, и классы совпадают.

Осталось сложить части в целое. Каждый элемент $a \in G$ лежит в своём классе, ведь $a = a \cdot e \in aH$, так что G — объединение попарно непересекающихся смежных классов по H . Если таких классов k , а каждый содержит ровно $|H|$ элементов, то $|G| = k \cdot |H|$, и $|H|$ делит $|G|$.

Число смежных классов подгруппы H в группе G имеет собственное имя.

ОПРЕДЕЛЕНИЕ 12.14 Индекс подгруппы

Индексом подгруппы H в группе G называется число её смежных классов, то есть $\frac{|G|}{|H|}$. Обозначается $[G : H]$.

Из теоремы Лагранжа сразу получается ограничение на порядок элемента.

СЛЕДСТВИЕ 12.6 Порядок элемента делит порядок группы

Для любого элемента g конечной группы G порядок $\text{ord}(g)$ делит порядок $|G|$.

Набросок доказательства

Рассмотрим подгруппу, порождённую g , то есть множество всех степеней g . Это действительно подгруппа: $g^i \cdot g^j = g^{i+j}$ — снова степень, а обратный к g^i — это g^{k-i} , ведь $g^i \cdot g^{k-i} = g^k = e$.

В этой подгруппе ровно $\text{ord}(g)$ элементов: степени $g^0, g^1, \dots, g^{k-1}$ различны, а $g^k = e$ возвращает в начало.

Остаётся применить теорему Лагранжа к этой подгруппе внутри G .

Из следствия о порядке элемента вытекает факт про группы простого порядка.

СЛЕДСТВИЕ 12.7 Группа простого порядка циклическая

Если порядок группы равен простому числу p , то группа циклическая, и любой её неединичный элемент — образующий.

Доказательство

Возьмём неединичный элемент g . Его порядок делит p и не равен единице, значит равен p .

Степени $g^0, g^1, \dots, g^{p-1}$ дают p различных элементов, то есть всю группу. Значит, g — образующий.

Из теоремы Лагранжа следует и утверждение теории чисел.

СЛЕДСТВИЕ 12.8 **Малая теорема Ферма**

Пусть p — простое, а a не кратно p . Тогда $a^{p-1} \equiv 1 \pmod{p}$.

Доказательство

Рассмотрим группу $\mathbb{Z}_p^*$ ненулевых остатков по умножению. Её порядок равен $p - 1$, и a — элемент этой группы.

По следствию из теоремы Лагранжа порядок элемента a делит порядок группы, то есть $p - 1$. Пусть $\text{ord}(a) = k$, и пусть $p - 1 = km$. Тогда $a^{p-1} = a^{km} = (a^k)^m = 1^m = 1$ в $\mathbb{Z}_p^*$. Значит, $a^{p-1} \equiv 1 \pmod{p}$.

Смежные классы нарезают группу на равные ломти. Классы сами можно превратить в элементы новой группы: правило $(a + H) + (b + H) = (a + b) + H$ корректно в $\mathbb{Z}_{12}$, потому что группа коммутативна. В общей форме правило требует нормальности подгруппы, и полную теорию факторгрупп разворачивает раздел о гомоморфизмах. В $\mathbb{Z}_{12}$ классы подгруппы $H = \{0, 3, 6, 9\}$ складываются в группу из трёх элементов: $(1 + H) + (1 + H) + (1 + H) = 3 + H = 0 + H$, и по кругу это арифметика $\mathbb{Z}_3$.

Те же равные части вскрывают одну криптографическую атаку. Порядок группы $\mathbb{Z}_p^*$ равен $p - 1$, и как во всякой циклической группе, в ней есть подгруппа каждого порядка, делящего $p - 1$. Если $p - 1$ раскладывается на малые простые множители (говорят, порядок *гладкий*), задача дискретного логарифма распадается по этим подгруппам, и каждая решается перебором в маленькой группе. Это атака Полига–Хеллмана, и защищаются от неё выбором простого p , у которого $p - 1$ имеет большой простой делитель.

Проверка Подгруппы и теорема Лагранжа

1. Почему в $\mathbb{Z}$ множество неотрицательных чисел не подгруппа, хотя оно замкнуто относительно сложения?
2. Сколько подгрупп порядка 2 в $\mathbb{Z}_8$, и почему ровно столько?
3. Почему в группе порядка 6 нет элемента порядка 4, хотя 4 меньше 6?

§ 12.5 Действие группы на множестве

В квадрате пронумеруем вершины 1, 2, 3, 4 по часовой стрелке. Раскрасим каждую вершину в чёрный или белый цвет. Повернём квадрат на 90° : вершина 1 перейдёт на место вершины 2, вершина 2 — на место вершины 3, и так далее. Раскраска повернулась вместе с квадратом, и её цвета теперь стоят у других вершин. Если одну

раскраску можно получить из другой поворотом, мы считаем их одной, и слово «та же самая» обретает точный смысл.

Симметрия превращает объект в неотличимый от него. Поворот квадрата сохраняет квадрат, хотя и переставляет его вершины. Такой поворот — это отображение множества вершин на себя, то есть перестановка: $1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4, 4 \rightarrow 1$. Перестановки можно выполнять подряд, и их композиция снова даёт перестановку: два поворота на 90° — это поворот на 180° . Четыре поворота — на $0, 90, 180$ и 270 градусов — образуют группу: поворот на 0° служит нейтральным элементом, а обратный к повороту на 90° — поворот на 270° . Группа симметрий — это набор преобразований, и у этого набора есть свои законы.

Поворот — преобразование одного и того же множества вершин. Группа, состоящая из преобразований множества X , *действует* на X .

ОПРЕДЕЛЕНИЕ 12.15 Действие группы на множестве

Пусть G — группа с нейтральным элементом e , и пусть X — множество. Отображение $G \times X \rightarrow X$, сопоставляющее паре (g, x) элемент $g \cdot x$, называется **действием** группы G на множестве X , если выполнены два условия:

1. $e \cdot x = x$ для всех $x \in X$;
2. $(gh) \cdot x = g \cdot (h \cdot x)$ для всех $g, h \in G$ и всех $x \in X$.

Вторая аксиома согласует последовательность преобразований с умножением в группе: сначала подействовать h , потом g — то же самое, что подействовать сразу произведением gh . Каждый элемент группы несёт преобразование. Для фиксированного g отображение $x \rightarrow g \cdot x$ — биекция X на себя, ведь элемент g^{-1} отменяет действие g : $g^{-1} \cdot (g \cdot x) = (g^{-1}g) \cdot x = e \cdot x = x$. Каждый элемент группы действует как перестановка множества X , и действие — это способ увидеть группу в преобразованиях конкретного множества.

Вернёмся к поворотам квадрата, действующим на вершины. Начнём с вершины 1 и приложим все повороты по очереди: поворот на 90° переводит 1 в 2, на 180° — в 3, на 270° — в 4. Вершина 1 дотягивается до всех четырёх вершин. Множество точек, в которые элемент x можно перевести действиями группы, называется его *орбитой*.

ОПРЕДЕЛЕНИЕ 12.16 Орбита

Орбитой элемента $x \in X$ относительно действия группы G называется множество $G \cdot x = \{g \cdot x \mid g \in G\}$.

Орбита собирает элементы, неотличимые с точки зрения группы. Если y лежит в орбите x , то x лежит в орбите y , потому что обратный элемент возвращает всё на место. Отношение «переводится в» — эквивалентность: нейтральный элемент делает его рефлексивным, обратный — симметричным, умножение — транзитивным. Поэтому орбиты разбивают X на непересекающиеся классы, и каждый элемент лежит ровно в одной орбите.

Некоторые точки группа оставляет на месте. Точка x называется *неподвижной* относительно элемента g , если $g \cdot x = x$. Поворот на 180° не оставляет неподвижной ни одной вершины: каждая вершина переходит в противоположную, а это другая вершина.

Точку, которую группа оставляет на месте, можно охарактеризовать отдельно.

ОПРЕДЕЛЕНИЕ 12.17 Стабилизатор

Стабилизатором точки $x \in X$ называется множество $\text{Stab}(x) = \{g \in G \mid g \cdot x = x\}$.

Стабилизатор образует подгруппу группы G . Если элемент неподвижен относительно всей группы, его стабилизатор совпадает с G , а орбита состоит из одной точки.

Пример Повороты квадрата на вершинах

Группа поворотов квадрата $G = \{e, r, r^2, r^3\}$, где r — поворот на 90° , действует на множестве вершин $\{1, 2, 3, 4\}$. Поворот r переводит $r \cdot 1 = 2, r \cdot 2 = 3, r \cdot 3 = 4, r \cdot 4 = 1$. Орбита вершины 1: $G \cdot 1 = \{e \cdot 1, r \cdot 1, r^2 \cdot 1, r^3 \cdot 1\} = \{1, 2, 3, 4\}$. Действие переводит каждую вершину в каждую, и потому орбита здесь одна — она содержит все четыре вершины. Стабилизатор вершины 1 тривиален: любой нетождественный поворот уводит вершину 1 из её угла, и на месте её держит только нейтральный элемент. Теорема об орбите и стабилизаторе (ниже) даёт $4 \cdot 1 = 4$ — порядок группы.

Связь между орбитой и стабилизатором — общий закон.

ТЕОРЕМА 12.9 Теорема об орбите и стабилизаторе

Пусть группа G действует на множестве X , и пусть $x \in X$. Тогда размер орбиты умножить на размер стабилизатора равен порядку группы:

$$|G \cdot x| \cdot |\text{Stab}(x)| = |G|.$$

Доказательство

Докажем, установив взаимно однозначное соответствие между смежными классами по стабилизатору и точками орбиты.

Стабилизатор $\text{Stab}(x)$ — подгруппа группы G , поэтому по теореме Лагранжа смежных классов по нему ровно $\frac{|G|}{|\text{Stab}(x)|}$. Сопоставим классу $g \cdot \text{Stab}(x)$ точку $g \cdot x$ орбиты и проверим корректность этого соответствия. Если $g_1 \cdot \text{Stab}(x) = g_2 \cdot \text{Stab}(x)$, то $g_2^{-1}g_1 \in \text{Stab}(x)$, откуда $g_2^{-1}g_1 \cdot x = x$ и $g_1 \cdot x = g_2 \cdot x$, то есть оба представителя дают одну и ту же точку.

Покажем, что соответствие взаимно однозначно. Оно инъективно, ведь из $g_1 \cdot x = g_2 \cdot x$ следует $g_2^{-1}g_1 \cdot x = x$, то есть $g_2^{-1}g_1 \in \text{Stab}(x)$, и классы совпадают. Оно сюръективно, поскольку каждая точка орбиты имеет вид $g \cdot x$ и потому служит образом класса $g \cdot \text{Stab}(x)$.

Итак, орбита взаимно однозначно соответствует смежным классам по стабилизатору, и элементов в ней столько же, сколько этих классов. По теореме Лагранжа число классов равно $\frac{|G|}{|\text{Stab}(x)|}$, откуда $|G \cdot x| \cdot |\text{Stab}(x)| = |G|$.

Пример Раскраски квадрата с точностью до поворота

Рассмотрим множество X всех раскрасок вершин квадрата в два цвета: таких раскрасок $2^4 = 16$. Каждый поворот переводит раскраску в другую раскраску, так что группа поворотов действует на X . Две раскраски лежат в одной орбите, если одну можно получить из другой поворотом, и именно такие раскраски мы считаем одинаковыми. Орбиты этого действия — типы раскрасок с точностью до поворота, и таких типов шесть:

- орбита из одной раскраски «все вершины чёрные», неподвижной относительно каждого поворота;
- орбита из одной раскраски «все вершины белые», неподвижной относительно каждого поворота;
- орбита из четырёх раскрасок с ровно одной чёрной вершиной;
- орбита из четырёх раскрасок с ровно одной белой вершиной;
- орбита из двух шахматных раскрасок, где чёрные вершины стоят напротив белых;
- орбита из четырёх раскрасок, где две чёрные вершины соседние.

Размеры орбит складываются в $1 + 1 + 4 + 4 + 2 + 4 = 16$, и шесть орбит делят все раскраски на типы.

Подсчёт орбит и есть ответ на вопрос, сколько раскрасок существует с точностью до симметрии. В примере мы перечислили орбиты вручную, но на большом множестве такой перебор нежизнеспособен. Помогает *лемма Бёрнсайда*: число орбит равно среднему числу неподвижных точек по всем элементам группы, $N = \left(\frac{1}{|G|}\right) \sum_{g \in G} |\text{Fix}(g)|$, где $\text{Fix}(g)$ — множество точек, неподвижных относительно g . Для раскрасок квадрата она даёт $\frac{16+2+4+2}{4} = 6$ — ровно те шесть типов, что мы перечислили. Мы разберём лемму Бёрнсайда в главе 18, а здесь важна предпосылка: лемма живёт в мире действий, и неподвижная точка — её рабочее понятие.

Замечание Сама по себе группа не действует

Одна и та же группа действует на разных множествах, и действие — отдельная информация от самой группы. Группа поворотов квадрата действует на вершины, на рёбра, на диагонали, на раскраски, и каждый раз это своё действие. Абстрактная группа задана умножением своих элементов, и она не знает, на что

ей действовать. Условия действия гарантируют согласованность: без них любой набор перестановок, никак не связанный с умножением в группе, сошёл бы за действие.

Проверка Действие группы

1. Почему раскраска «все вершины чёрные» лежит в орбите из одного элемента, а раскраска с одной чёрной вершиной — в орбите из четырёх?
2. Сколько орбит у действия поворотов квадрата на множестве его вершин, и почему стабилизатор каждой вершины тривиален?
3. Что можно сказать об орбите точки, стабилизатор которой совпадает со всей группой?

Орбиты — это классификация: одинаковое собрано в один класс, а число классов — ответ на вопрос «сколько с точностью до симметрии». Группа абстрактна, а действие конкретно: одна структура, много воплощений. Так абстрактная алгебра возвращает информацию о мире, и лемма Бёрнсайда — первое звено этой связи.

До сих пор речь шла об одной операции. Теперь добавим вторую, и структура обретает арифметику.

§ 12.6 Кольцо

На циферблате с шестью делениями считать можно по-разному. Сложение по модулю 6 мы уже освоили: прибавить 4 к 5 — значит пройти ещё четыре деления и остановиться на 3. Но арифметика одной операцией не ограничивается: числа ещё и умножают. Умножим по модулю 6: $2 \cdot 3 = 6$, а 6 на таком циферблате — это ноль. Два элемента, отличных от нуля, дали в произведении ноль.

У целых чисел такое невозможно: если произведение равно нулю, то хотя бы один сомножитель нулевой. По модулю 6 это правило ломается. При этом обе операции замкнуты, ассоциативны и согласованы дистрибутивным законом. Так возникает структура с двумя операциями и разными требованиями к ним.

В группе мы требовали от каждой операции обратимости: любой элемент можно было «отменить». У сложения обратимость остаётся — на циферблате вычитание по модулю 6 работает без оговорок. У умножения она пропадает: у двойки по модулю 6 нет обратного, и на неё нельзя разделить. Структура, где сложение ведёт себя как в группе, умножение — как в полугруппе, а связывает их дистрибутивность, называется *кольцом*.

ОПРЕДЕЛЕНИЕ 12.18 Кольцо

Множество R с двумя бинарными операциями — сложением $+$ и умножением $\cdot$ — называется **кольцом**, если выполнены три условия.

1. По сложению R — абелева группа: сложение ассоциативно и коммутативно, есть нейтральный элемент 0 , и у каждого $a \in R$ есть обратный $-a$ такой, что $a + (-a) = 0$.
2. По умножению R — полугруппа: умножение ассоциативно, то есть $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ для любых $a, b, c \in R$.
3. Умножение дистрибутивно относительно сложения: $a \cdot (b + c) = a \cdot b + a \cdot c$ и $(a + b) \cdot c = a \cdot c + b \cdot c$ для любых $a, b, c \in R$.

Умножение в кольце может быть некоммутативным, может обходиться без нейтрального элемента, и обратный по умножению не обязан существовать у каждого элемента. Во всех примерах этой главы у умножения нейтральный элемент всё же находится, и тексты, включающие его в определение, называют такое кольцо кольцом с единицей. Поле убирает все три свободы: умножение в нём коммутативно, у него есть нейтральный элемент 1 , и у каждого ненулевого элемента есть обратный.

По сложению $\mathbb{Z}$ — абелева группа, по умножению — полугруппа, а дистрибутивность знакома со школы: $2 \cdot (3 + 4) = 2 \cdot 3 + 2 \cdot 4$. Значит, целые числа с обычными операциями образуют кольцо, и в нём нет делителей нуля. Кольцом будет и $\mathbb{Z}_6$ — в нём два ненулевых элемента могут давать в произведении ноль, но аксиомы кольца это не нарушает. К кольцам относятся и многочлены: сложение и умножение многочленов с действительными коэффициентами ассоциативны и дистрибутивны, так что $\mathbb{R}[x]$ — тоже кольцо.

Новизна кольца в свободе выбора операций: они могут быть какими угодно, лишь бы аксиомы выполнялись. Разберём это на необычном примере.

Пример Булево кольцо

Возьмём множество $B = \{0, 1\}$ и зададим на нём две операции: сложение — исключающее ИЛИ ($\oplus$), умножение — конъюнкция ($\wedge$).

a	b	$a \oplus b$	$a \wedge b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Проверим аксиомы. Сложение $\oplus$ — абелева группа: нейтральный элемент это 0 , а каждый элемент обратен сам себе, потому что $a \oplus a = 0$. Умножение $\wedge$ ассоциативно и коммутативно, у него есть нейтральный элемент 1 . Дистрибутивность выполняется: $1 \wedge (b \oplus c) = b \oplus c = (1 \wedge b) \oplus (1 \wedge c)$, а при $a = 0$ обе стороны равны нулю. Значит, B с этими операциями — кольцо, хотя ни сложение, ни умножение здесь не похожи на свои числовые двойники. В этом кольце каждый элемент идемпотентен: $a \wedge a = a$, то есть умножить элемент на себя — значит ничего не изменить.

Булево кольцо напрямую связано с булевой алгеброй из главы 10. Носитель один и тот же — $\{0, 1\}$, умножение одно и то же — конъюнкция, а сложение различается: в кольце это $\oplus$, в алгебре — дизъюнкция $\vee$. Одна структура выражается через другую тождествами $a \vee b = a \oplus b \oplus (a \wedge b)$ и $\neg a = a \oplus 1$, так что это две формы одной двоичной логики.

В кольце вычетов идемпотентности нет: по модулю 6 имеем $2 \cdot 2 = 4$. Там же два ненулевых элемента могут дать в произведении ноль: $2 \cdot 3 = 0$.

ОПРЕДЕЛЕНИЕ 12.19 Делитель нуля

Ненулевой элемент a кольца называется **делителем нуля**, если найдётся ненулевой b такой, что $a \cdot b = 0$.

Пример Кольцо вычетов с делителями нуля

Вернёмся к $\mathbb{Z}_6$. По сложению $\mathbb{Z}_6$ — абелева группа (мы это уже знаем), по умножению — полугруппа, так что это кольцо. Умножение по модулю 6 скрывает сюрприз: $2 \cdot 3 = 6 \equiv 0 \pmod{6}$, при этом оба сомножителя отличны от нуля. Другой пример: $3 \cdot 4 = 12 \equiv 0 \pmod{6}$, значит и 4 — делитель нуля. Обратного к 2 по умножению не существует: каким бы ни было a , число $2 \cdot a$ по модулю 6 равно 0, 2 или 4, но никогда не 1. На 2 в $\mathbb{Z}_6$ делить нельзя, хотя делить на 5 можно: $5 \cdot 5 = 25 \equiv 1 \pmod{6}$, так что 5 обратен сам себе.

Деление на делитель нуля невозможно — и это не случайность.

УТВЕРЖДЕНИЕ 12.10 Делитель нуля не обратим

Делитель нуля не может иметь обратного по умножению.

Доказательство

Пусть $a \cdot b = 0$, причём оба сомножителя ненулевые.

Предположим от противного, что у a есть обратный a^{-1} . Умножим обе стороны равенства $a \cdot b = 0$ на a^{-1} слева: $(a^{-1} \cdot a) \cdot b = a^{-1} \cdot 0$. Слева получится b (произведение $a^{-1} \cdot a$ равно единице), справа ноль, то есть $b = 0$. Это противоречит выбору $b \neq 0$. Значит, обратного к a не существует.

В кольце обратный по умножению есть у единицы и, быть может, ещё у нескольких элементов. Чтобы делить всегда, нужен обратный у каждого ненулевого. Это и есть следующая структура — поле.

Пример Некоммутативное кольцо матриц

Матрицы 2×2 с действительными коэффициентами образуют кольцо $M_2(\mathbb{R})$. Сложение — покомпонентное, умножение — обычное матричное. Сложение коммутативно, дистрибутивность выполняется, а вот умножение — нет:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

Два произведения различаются, поэтому $M_2(\mathbb{R})$ — кольцо, но не коммутативное. В нём есть и делители нуля: ненулевые матрицы могут давать в произведении нулевую.

Проверка Кольцо и делители нуля

1. Почему $\mathbb{Z}_6$ — кольцо, а умножение в нём не образует группу?
2. Почему в $\mathbb{Z}$ нет делителей нуля, хотя в $\mathbb{Z}_6$ они есть? Что отличает эти два кольца?
3. Какие элементы $\mathbb{Z}_6$ обратимы по умножению, и почему их ровно два?
4. Чем булево кольцо $B = \{0, 1\}$ с операциями $\oplus$ и $\wedge$ отличается от обычных целых — при том, что это тоже кольцо?

§ 12.7 Поле

Кольцо собралось из двух операций, но деления среди них нет. Сложение образует группу, умножение ассоциативно, дистрибутивность связывает их. Так устроены целые числа и кольца вычетов. В кольце вычетов по модулю 6 у элемента 2 нет обратного: никакое число, умноженное на 2, не даёт остаток 1. И $2 \cdot 3 = 0$: ненулевые элементы могут умножаться в нуль. Чтобы кольцо стало полем, ему не хватает обратного элемента по умножению у каждого ненулевого числа. Умножение при этом обязано быть коммутативным, но в привычных кольцах оно уже такое. Потребуем обратимость — и деление становится законным.

Поле — это множество, в котором арифметика устроена так же, как в рациональных числах: сложение, вычитание, умножение и деление работают, а делить можно на всё, кроме нуля. Целые числа $\mathbb{Z}$ так не устроены: половина $\frac{1}{2}$ не является целым числом. Рациональные $\mathbb{Q}$, вещественные $\mathbb{R}$ и комплексные $\mathbb{C}$ — поля. Интереснее всего проявляется поле в конечном мире остатков. В кольце вычетов по модулю 7 каждый ненулевой остаток от 1 до 6 имеет пару, произведение которой даёт остаток 1. Значит, деление в остатках по модулю 7 определено всегда, и место, где это возможно, заслуживает отдельного имени.

ОПРЕДЕЛЕНИЕ 12.20 Поле

Полем называется множество F с двумя бинарными операциями $+$ и $\cdot$, для которых выполнены условия:

1. $(F, +)$ — абелева группа с нейтральным элементом 0;
2. $(F \setminus \{0\}, \cdot)$ — абелева группа с нейтральным элементом 1;
3. $a \cdot (b + c) = a \cdot b + a \cdot c$ для всех $a, b, c \in F$.

Первое условие говорит, что в поле можно складывать и вычитать, а второе — что ненулевые элементы можно умножать и делить. Нейтральный элемент первой группы, 0, не входит во вторую: на ноль делить нельзя, и в определении это отражено явно. Третье условие, дистрибутивность, связывает между собой сложение и умножение. Вместе они описывают то, что мы с детства называем обычной арифметикой, только на произвольном множестве.

Пример Поле $\mathbb{Z}_7$

a	1	2	3	4	5	6
a^{-1}	1	4	5	2	3	6

Проверим несколько строк: $2 \cdot 4 = 8 = 1 \pmod{7}$, $3 \cdot 5 = 15 = 1 \pmod{7}$, а $6 \cdot 6 = 36 = 1 \pmod{7}$. В каждом случае произведение даёт остаток 1, поэтому вторая строка — обратные элементы первой. Значит, кольцо вычетов по модулю 7 — поле.

ОПРЕДЕЛЕНИЕ 12.21 Порядок поля

Число элементов конечного поля называется его **порядком**, обозначается $|K|$.

Поле $\mathbb{Z}_7$ имеет порядок 7, поле $\mathbb{Z}_p$ для простого p — порядок p . Поле $\mathbb{Z}_7$ мы получили именно так: его элементы — остатки от 0 до 6, а сложение и умножение считаются по модулю семь. Для каждого простого p поле вычетов $\mathbb{Z}_p$ с такой арифметикой называют *простым полем*.

Пример Минимальное поле $\mathbb{Z}_2$

Наименьшее поле — $\mathbb{Z}_2$ из двух элементов: 0 и 1. Сложение и умножение задаются по модулю два:

$$1 + 1 = 0, \quad 1 \cdot 1 = 1.$$

Ненулевой элемент только один — единица, и она обратна самой себе. Именно это поле лежит в основании всей двоичной арифметики: биты — его элементы, XOR — сложение, AND — умножение.

Простоты модуля достаточно, чтобы каждый ненулевой остаток имел обратный.

ТЕОРЕМА 12.11 Поле вычетов по простому модулю

Кольцо вычетов $\mathbb{Z}_p$ является полем тогда и только тогда, когда p — простое.

Доказательство

Предположим сначала, что p простое. Возьмём ненулевой остаток a , то есть $1 \leq a \leq p-1$. Тогда $\gcd(a, p) = 1$, потому что p имеет ровно два делителя: 1 и p , а a меньше p и не равен нулю. По соотношению Безу (доказательство — в главе 19) найдутся целые x, y с $ax + py = 1$. Взяв равенство по модулю p , получаем $ax \equiv 1 \pmod{p}$, то есть x — обратный к a . Значит, каждый ненулевой элемент обратим, и $\mathbb{Z}_p$ — поле.

Теперь предположим, что p составное: $p = ab$, где $1 < a, b < p$. Тогда $a \cdot b = p \equiv 0 \pmod{p}$, хотя и a , и b — ненулевые остатки. Появился делитель нуля, поэтому $\mathbb{Z}_p$ полем не является.

Сама проверка даёт ненулевые элементы без обратного: в $\mathbb{Z}_6$ достаточно вспомнить, что $2 \cdot 3 = 0$, или вычислить напрямую, что у 2 нет обратного.

В поле делителей нуля нет. Если $a \cdot b = 0$ и $a \neq 0$, то умножение обеих частей на a^{-1} даёт $b = 0$. Произведение двух ненулевых элементов поля никогда не равно нулю, и именно этим $\mathbb{Z}_p$ отличается от $\mathbb{Z}_6$, где $2 \cdot 3 = 0$ при ненулевых сомножителях. Делить на ноль не сможет никакое поле: для любого a выполнено $0 \cdot a = 0$, поэтому равенство $0 \cdot a = 1$ не выполняется ни при каком a , и обратного к нулю не существует.

Деление в поле — это умножение на обратный элемент, а отдельная операция деления не вводится. Вместо того чтобы делить a на b , мы умножаем a на b^{-1} . Поэтому вся арифметика поля задаётся двумя операциями, $+$ и $\cdot$, а вычитание и деление оказываются производными.

Проверка Поле и делители нуля

1. В $\mathbb{Z}_9$ элемент 3 не имеет обратного. Объясните, какое свойство поля это нарушает.
2. Чему равен порядок поля $\mathbb{Z}_{17}$?
3. Разложите 4 в произведение делителей нуля по модулю 12 и объясните, почему отсюда следует отсутствие обратного.
4. Почему делить на ноль не сможет никакое поле, даже бесконечное?

§ 12.8 Конечные поля

Байт состоит из восьми битов и потому принимает $2^8 = 256$ значений. Над байтами хочется выполнять арифметику так же свободно, как над обычными числами:

складывать, умножать, делить. Но деление возможно только там, где на каждый ненулевой элемент можно разделить, то есть в поле.

Поля вычетов $\mathbb{Z}_p$ это умеют, однако у них ровно p элементов, и p обязано быть простым. Число 256 простым не является, и кольцо $\mathbb{Z}_{256}$ полем не становится: в нём есть делители нуля, например $2 \cdot 128 = 256 \equiv 0 \pmod{256}$. Арифметикой по модулю 256 байтовое деление не построить.

Поле с 256 элементами существует, и изобретение, которое его даёт, принадлежит Эваристу Галуа. Поля, число элементов которых равно степени простого числа, по традиции называют *полями Галуа*. Осталось понять, как такое поле построить. Строительство начинается с простого поля $\mathbb{Z}_p$, к которому присоединяют корень многочлена, не раскладывающегося над этим полем.

ОПРЕДЕЛЕНИЕ 12.22 Неприводимый многочлен

Многочлен $f(x)$ степени m с коэффициентами из $\mathbb{Z}_p$ называется **неприводимым** над $\mathbb{Z}_p$, если его нельзя представить произведением двух многочленов меньших степеней.

Если f неприводим, то кольцо $\mathbb{Z}_{p[x]}$ по модулю f оказывается полем $\frac{\mathbb{Z}_{p[x]}}{f(x)}$. Объявим $f(x)$ равным нулю — будем считать многочлены, отличающиеся на кратное f , одним и тем же — и рассмотрим все остатки от деления многочленов над $\mathbb{Z}_p$ на f . Среди остатков — ровно все многочлены степени меньше m , а их p^m , потому что каждый из m коэффициентов выбирается из p значений. Сложение выполняется покомпонентно по модулю p , умножение — как у многочленов с приведением результата по модулю f . Так из простого поля $\mathbb{Z}_p$ вырастает поле побольше, и им оказывается $\text{GF}(p^m)$. Запись $\text{GF}(p^m) = \text{GF}(p) \frac{[x]}{f(x)}$ — та же конструкция, которой из целых чисел получают $\mathbb{Z}_p = \frac{\mathbb{Z}}{p}$, только роль числа p играет многочлен. Скобки (p) и $(f(x))$ обозначают множество всех кратных: (p) — целые числа, делящиеся на p , а $(f(x))$ — многочлены, делящиеся на $f(x)$. Множество всех кратных одного элемента называют *идеалом*, порождённым этим элементом.

ОПРЕДЕЛЕНИЕ 12.23 Конечное поле

Поле K называется **конечным**, если множество его элементов конечно.

Порядок конечного поля устроен закономерно: не любое число может им быть. Чтобы это доказать, понадобится характеристика. Рассмотрим элемент 1 поля и будем прибавлять его к себе: $1, 1 + 1, 1 + 1 + 1, \dots$. В конечном поле этот ряд обязан повториться.

ОПРЕДЕЛЕНИЕ 12.24 Характеристика поля

Наименьшее p , при котором сумма p единиц равна нулю, называется **характеристикой** поля.

ТЕОРЕМА 12.12 **Порядок конечного поля**

Порядок конечного поля обязан быть степенью простого числа: если поле конечно, то $|K| = p^m$ для некоторого простого p и натурального m .

Набросок доказательства

Примем без доказательства, что характеристика — простое число. Суммы $0, 1, 2 \cdot 1, \dots, (p-1) \cdot 1$ образуют подполе из p элементов, ведущее себя ровно как $\mathbb{Z}_p$ по модулю p , — простое подполе.

Поле можно умножать на элементы этого подполя, поэтому оно является векторным пространством над $\mathbb{Z}_p$ (понятие из раздела 12.10). Если размерность этого пространства равна m , то в поле ровно p^m элементов.

Построение выше показывает, почему поле с p^m элементами существует.

Классификация конечных полей — ещё более сильное утверждение.

ТЕОРЕМА 12.13 **Классификация конечных полей**

Для любого простого p и натурального m существует конечное поле из p^m элементов, и любые два таких поля изоморфны. Всякое конечное поле изоморфно некоторому $\text{GF}(p^m)$.

Примечание

Слово «изоморфно» здесь означает «устроено одинаково, только переименовано». Строгий смысл изоморфизма появится в следующем разделе, а доказательство существования и единственности поля $\text{GF}(p^m)$ опирается на неприводимые многочлены и выходит за рамки этой главы.

Вот почему для байтов подходит именно $256 = 2^8$: двойка простая, а степень восемь задаёт поле нужного размера.

Неприводимость f — условие, без которого деление рассыпается. Если f раскладывается в произведение $f = gh$ с многочленами меньших степеней, то по модулю f произведение gh равно нулю, хотя оба сомножителя не нулевые — появляются делители нуля. В поле с делителями нуля делить нельзя: на ненулевой элемент, который сам является делителем нуля, обратного не существует.

Раз f неприводим, то ненулевой многочлен $a(x)$ степени меньше m взаимно прост с f . Расширенный алгоритм Евклида находит такие многочлены $u(x)$ и $v(x)$, что $u(x)a(x) + v(x)f(x) = 1$. Взяв это равенство по модулю f , получаем $u(x)a(x) = 1$ в $\text{GF}(p^m)$, и $u(x)$ оказывается обратным к $a(x)$.

Практика в таком поле проста: сложить — значит сложить коэффициенты, умножить — перемножить многочлены и взять остаток, разделить — найти обратный расширенным Евклидом и умножить.

Пример Поле $\text{GF}(4)$

Возьмём $p = 2$ и $m = 2$. Многочлен $f(x) = x^2 + x + 1$ неприводим над $\mathbb{Z}_2$. Квадратный многочлен раскладывается на множители тогда и только тогда, когда у него есть корень, а здесь $f(0) = 1$ и $f(1) = 1$, то есть корней нет. Остатки от деления на f — четыре многочлена степени не выше единицы: $0, 1, x, x + 1$.

Обозначим $\alpha = x$. Правило поля $\alpha^2 + \alpha + 1 = 0$ даёт $\alpha^2 = \alpha + 1$.

Таблица сложения — это поразрядный XOR коэффициентов:

+	0	1	α	$\alpha + 1$
0	0	1	α	$\alpha + 1$
1	1	0	$\alpha + 1$	α
α	α	$\alpha + 1$	0	1
$\alpha + 1$	$\alpha + 1$	α	1	0

Таблица 5. Сложение в $\text{GF}(4)$.

Таблица умножения строится тем же правилом $\alpha^2 = \alpha + 1$:

·	0	1	α	$\alpha + 1$
0	0	0	0	0
1	0	1	α	$\alpha + 1$
α	0	α	$\alpha + 1$	1
$\alpha + 1$	0	$\alpha + 1$	1	α

Таблица 6. Умножение в $\text{GF}(4)$.

Сумма двух элементов снова лежит в множестве, произведение ненулевых элементов тоже, а на каждый ненулевой элемент находится обратный — перед нами поле. Степени α таковы: $\alpha^1 = \alpha$, $\alpha^2 = \alpha + 1$, $\alpha^3 = 1$, поэтому порядок элемента α равен трём, и его степени пробегают все три ненулевых элемента поля.

Полю с четырьмя элементами рано или поздно становится тесно, и для байтов нужен $m = 8$. Над $\mathbb{Z}_2$ берут неприводимый многочлен $f(x) = x^8 + x^4 + x^3 + x + 1$, и остатки от деления на него — ровно байты. Каждый байт из восьми битов задаёт коэффициенты многочлена степени меньше восьми, а сложение байтов оказывается просто XOR. В таком поле живут шифр AES и коды Рида–Соломона, и именно в нём делить байты становится так же естественно, как делить обычные числа.

Ненулевые элементы $\text{GF}(q)$ образуют по умножению группу $\text{GF}(q)^*$ порядка $q - 1$. Галуа показал, что эта группа циклическая: найдётся элемент, чьи степени пробегают все ненулевые элементы поля.

ОПРЕДЕЛЕНИЕ 12.25 **Примитивный элемент**

Образующий циклической группы $\text{GF}(q)^*$ называется **примитивным элементом** поля $\text{GF}(q)$. Его степени $g^1, g^2, \dots, g^{q-1}$ пробегают все ненулевые элементы поля.

Примечание

Примитивным бывает не любой элемент поля. В поле из 256 элементов с многочленом $x^8 + x^4 + x^3 + x + 1$ элемент x имеет порядок 51, а примитивен $x + 1$ с порядком $255 = 2^8 - 1$. Поиск примитивного элемента — отдельная задача перебора степеней, и на неё опираются конструкции, где нужны все ненулевые степени одного элемента.

Проверка **Конечные поля: многочлены и расширение**

1. Почему арифметика по модулю 256 полем не становится, а построение над $\mathbb{Z}_2$ по неприводимому многочлену степени восемь полем становится?
2. Убедитесь по таблице умножения $\text{GF}(4)$, что элемент $\alpha + 1$ имеет порядок три.
3. Что сломается в конструкции, если вместо $f(x) = x^2 + x + 1$ взять $f(x) = x^2$ над $\mathbb{Z}_2$?
4. Сколько элементов в $\text{GF}(8)$ и любой ли ненулевой элемент примитивен?

Всякое поле с одним и тем же числом элементов устроено одинаково, и эта одинаковость делает строгим понятие изоморфизма. Как переносить операции между структурами, не ломая их, — тема следующего раздела.

§ 12.9 **Гомоморфизм и изоморфизм**

Каждая алгебраическая структура — это множество с операциями, и операции составляют её содержание. Множество остатков по модулю семь и множество вращений правильного семиугольника — разные множества: их элементы не совпадают буквально. Тем не менее каждое несёт по бинарной операции, и обе операции подчиняются одним правилам. Вопрос этого раздела — когда две структуры устроены одинаково.

Ответ не лежит в сравнении носителей. Две группы на разных множествах различны как множества, и требовать их совпадения — значит отказаться от сравнения структур вовсе. Сравнить по числу элементов тоже нельзя: группы $\mathbb{Z}_4$ и $\mathbb{Z}_2 \times \mathbb{Z}_2$ (пары остатков, складываемые покомпонентно по модулю два) имеют по четыре элемента, но их таблицы операции различаются — в первой есть элемент порядка четыре, во второй его нет. Между этими крайностями лежит решение: сопоставить

элементам одной структуры элементы другой так, чтобы операции согласовались. Такое сопоставление называют гомоморфизмом, а биективное — изоморфизмом.

Слово «сохранять» здесь получает точный смысл: отображение коммутирует с операцией. Вычислить в исходной структуре и перейти в новую — либо перейти в новую и вычислить — должно привести к одному результату. Это условие задаёт верность переноса.

Начнём с вычислительной задачи. Пусть сегодня среда, а нас спрашивают, какой день недели наступит через сто дней. Сто дней — это четырнадцать недель и два дня, поэтому ответ — пятница. Сто можно было сократить и сразу: сто при делении на семь даёт остаток два. Сокращение не мешает дальнейшим действиям, в этом его ценность.

Проверим это на умножении. Произведение двенадцати и девяти равно 108, а 108 при делении на семь даёт остаток три. Сведём множители заранее: двенадцать даёт остаток пять, девять — остаток два. Произведение пяти и двух равно 10, и остаток снова три. Результат совпал.

Это свойство отображения $\varphi : \mathbb{Z} \rightarrow \mathbb{Z}_7$, которое переводит целое число в остаток от деления на семь. Остаток от произведения равен остатку от произведения остатков:

$$\varphi(x \cdot y) = \varphi(x) \cdot \varphi(y).$$

То же верно и для сложения. Отображение, которое согласуется с операцией, называют **гомоморфизмом**.

ОПРЕДЕЛЕНИЕ 12.26 Гомоморфизм

Пусть $(A, \star)$ и $(B, \cdot)$ — множества с бинарными операциями. Отображение $\varphi : A \rightarrow B$ называется **гомоморфизмом**, если для любых $x, y \in A$ выполнено

$$\varphi(x \star y) = \varphi(x) \cdot \varphi(y).$$

В группах сохранение нейтрального элемента вытекает из самого определения. Подставим e_A в условие гомоморфизма: $\varphi(e_A) = \varphi(e_A \star e_A) = \varphi(e_A) \cdot \varphi(e_A)$. Сократим это равенство на $\varphi(e_A)$ по закону сокращения и получим $e_B = \varphi(e_A)$. В моноиде, где сокращать нельзя, это рассуждение ломается, и сохранение нейтрального элемента приходится требовать отдельно.

Вернёмся к $\varphi : \mathbb{Z} \rightarrow \mathbb{Z}_7$. Оно сохраняет сложение, умножение и единицу, поэтому это гомоморфизм колец. Оно сюръективно: каждый остаток встречается как образ целого числа. Инъективности нет: числа 1 и 8 дают один остаток. Элементы с нулевым остатком, то есть кратные семи, образуют **ядро**.

Ядро — важная характеристика гомоморфизма.

ОПРЕДЕЛЕНИЕ 12.27 Ядро гомоморфизма

Ядром гомоморфизма групп $\varphi : G \rightarrow H$ называется множество тех $x \in G$, для которых $\varphi(x)$ равен нейтральному элементу группы H . Обозначается $\ker(\varphi)$.

Ядро — подгруппа группы G , притом особая: оно *нормально*.

ОПРЕДЕЛЕНИЕ 12.28 Нормальная подгруппа

Подгруппа H группы G называется **нормальной**, если для любого $g \in G$ выполнено $gHg^{-1} = H$. Это равносильно совпадению левого и правого смежных классов: $gH = Hg$ для всех $g \in G$.

Умножение смежных классов через представителей, $(gH) \cdot (kH) = (g \cdot k)H$, корректно ровно для нормальных подгрупп. В этом случае классы образуют группу — факторгруппу $\frac{G}{H}$. Ядро гомоморфизма нормально. Если $\varphi(x) = e$, то $\varphi(g \cdot x \cdot g^{-1}) = \varphi(g) \cdot \varphi(x) \cdot \varphi(g^{-1}) = \varphi(g) \cdot e \cdot \varphi(g^{-1}) = \varphi(g \cdot g^{-1}) = \varphi(e) = e$, так что $g \cdot x \cdot g^{-1}$ лежит в ядре для любого $g \in G$. В коммутативной группе каждая подгруппа нормальна, поэтому факторгруппа $\frac{\mathbb{Z}_{12}}{H}$ из раздела о смежных классах определена корректно.

Смежные классы по ядру устроены так же, как элементы образа. Здесь связь между ядром и образом превращается в отдельную теорему.

ТЕОРЕМА 12.14 Первая теорема об изоморфизме

Пусть $\varphi : G \rightarrow H$ — гомоморфизм групп. Тогда факторгруппа $\frac{G}{\ker(\varphi)}$ изоморфна образу φ . Здесь $\frac{G}{\ker(\varphi)}$ — группа смежных классов по ядру, где произведение классов задаётся через произведение их представителей.

Набросок доказательства

Каждому классу $g \cdot \ker(\varphi)$ сопоставим образ $\varphi(g)$. Выбор представителя роли не играет: если $g_1 \cdot \ker(\varphi) = g_2 \cdot \ker(\varphi)$, то $g_2^{-1} \cdot g_1 \in \ker(\varphi)$, и потому $\varphi(g_1) = \varphi(g_2 \cdot (g_2^{-1} \cdot g_1)) = \varphi(g_2) \cdot e = \varphi(g_2)$. Соответствие биективно: разные классы дают разные образы, а каждый элемент образа достигим.

Произведению классов $g \cdot \ker(\varphi)$ и $k \cdot \ker(\varphi)$ отвечает $\varphi(g) \cdot \varphi(k) = \varphi(g \cdot k)$, то есть образ произведения, а нейтральному классу — нейтральный элемент $\varphi(e) = e$. Значит, это изоморфизм.

Первая теорема об изоморфизме сжимается в коммутативную диаграмму (рис. 65), в которой из G в H ведут два пути, и оба дают одно и то же. Прямой путь — гомоморфизм φ , а обходной — сначала проекция π на факторгруппу, затем изоморфизм $\tilde{\varphi}$ на образ, затем вложение ι образа в H .

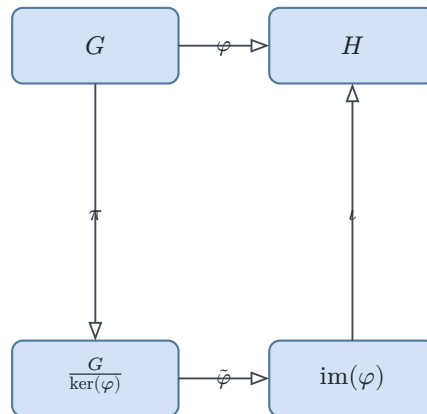


Рис. 65. Первая теорема об изоморфизме.

Пример Три примера гомоморфизмов

1. **Сведение по модулю.** Отображение $\varphi : \mathbb{Z} \rightarrow \mathbb{Z}_n$ с $\varphi(x) = x \bmod n$ сохраняет сложение, умножение и единицу:

$$\varphi(x + y) = \varphi(x) + \varphi(y), \quad \varphi(x \cdot y) = \varphi(x) \cdot \varphi(y), \quad \varphi(1) = 1.$$

Оно сюръективно, а его ядро — множество чисел, делящихся на n .

2. **Отражение.** Пусть ρ — отражение плоскости относительно прямой, проходящей через начало координат. Отражение — биекция, и оно линейно: $\rho(u + v) = \rho(u) + \rho(v)$. Значит, ρ — автоморфизм, то есть биективный гомоморфизм аддитивной группы векторов в себя: отразить сумму — то же, что отразить каждое слагаемое и сложить результаты.
3. **Тождество.** Отображение $\text{id}(x) = x$ — гомоморфизм, причём биективный, то есть автоморфизм. Постоянное отображение $f : G \rightarrow H$ с $f(x) = e_H$ тоже гомоморфизм: $f(xy) = e_H = e_H e_H = f(x)f(y)$.

Гомоморфизм переносит операцию, но не обязан быть обратимым. Обратимым его делает биективность, и такие отображения получают отдельное имя.

ОПРЕДЕЛЕНИЕ 12.29 Изоморфизм

Биективный гомоморфизм $\varphi : A \rightarrow B$ называется **изоморфизмом**. Если между A и B существует изоморфизм, структуры называют **изоморфными**.

Изоморфизм, переводящий структуру в неё саму, называют автоморфизмом. Изоморфизм устанавливает взаимнооднозначное соответствие между структурами: каждой операции одной из них отвечает ровно одна операция другой, и все они согласованы. Две изоморфные группы на разных носителях как множества различны, но с точностью до переименования совпадают. Утверждение, сформулированное через операции и верное в одной структуре, верно и в любой изоморфной ей. Изоморфизм обязан сохранять и отношения, если структура их несёт: изоморфизм решёток переводит $x \leq y$ в $\varphi(x) \leq \varphi(y)$.

Для конечных полей это утверждение сильнее: любые два поля с p^m элементами изоморфны, поэтому существует ровно одно поле $\text{GF}(p^m)$, и глава 13 опирается на этот факт. То же явление встретится в курсе ещё дважды. В главе 22 любые две алгебры Дедекинда изоморфны, и отсюда следует категоричность аксиом Пеано: натуральный ряд — единственная структура с точностью до переименования (раздел 22.2). В теории решёток (глава 8) изоморфизм позволяет говорить «решётка содержит подрешётку, изоморфную M_3 или N_5 », где «изоморфную» означает «устроенную так же», а имена элементов несущественны.

Гомоморфизм полей устроен жёстче, чем гомоморфизм колец: его ядро всегда тривиально.

УТВЕРЖДЕНИЕ 12.15 Гомоморфизм полей инъективен

Всякий гомоморфизм полей инъективен.

Доказательство

Докажем от противного.

Допустим, ядро содержит ненулевой элемент a , и покажем, что тогда оно совпадает со всем полем. Для любого $b \in F$ выполнено $\varphi(b \cdot a) = \varphi(b) \cdot \varphi(a) = \varphi(b) \cdot 0 = 0$, поэтому $b \cdot a \in \ker(\varphi)$. Умножение на a — биекция поля на себя, ведь у a есть обратный, так что произведения $b \cdot a$ пробегает всё поле F . Значит, ядро совпадает со всем F .

Но тогда ядро содержало бы единицу 1, а по определению гомоморфизма полей $\varphi(1) = 1 \neq 0$, то есть 1 в ядре лежать не может. Противоречие, значит ядро нулевое, и φ переводит в ноль только ноль, то есть инъективен.

Примечание

Гомоморфизм полей — это вложение, а не произвольное отображение. Сюръективность не требуется: поле $\mathbb{R}$ вкладывается в поле рациональных функций над $\mathbb{R}$, но не совпадает с ним. Для колец вывод неверен: сведение $\mathbb{Z} \rightarrow \mathbb{Z}_7$ имеет ядро $7\mathbb{Z}$ и не инъективно.

Проверка Гомоморфизм и изоморфизм

1. Почему $\mathbb{Z}_4$ и $\mathbb{Z}_2 \times \mathbb{Z}_2$ неизоморфны, хотя у них одинаковое число элементов?
2. Докажите, что обратное отображение к изоморфизму — тоже гомоморфизм.
3. Какое ядро у гомоморфизма $\mathbb{Z} \rightarrow \mathbb{Z}_3$, где $\mathbb{Z}_3$ — кольцо остатков по модулю три?
4. Почему для класса групп сохранения одной операции достаточно, а для кольца нужны обе?

§ 12.10 Векторные пространства над полем

Пусть у нас есть несколько сообщений одинаковой длины, и каждое сообщение — строка из нулей и единиц. Мы уже умеем складывать такие строки: поэлементно по модулю два. Раз $1 + 1 = 0$, каждый элемент обратен сам себе, и вычитание здесь совпадает со сложением.

Над строками определена операция: их можно складывать между собой. Для сложения выполнены те же законы, что и для сложения чисел, — только всё вычисляется по модулю два. По сложению этот набор строк — группа.

Но строкам можно предложить и вторую операцию: умножать строку на отдельный бит. Умножение на единицу ничего не меняет, а умножение на ноль превращает строку в одну из всех нулей. Со сложением и скалярным умножением строки несут уже две операции.

В общей постановке роль «битов» играет произвольное поле, а роль строк — векторы. Такая пара и образует *векторное пространство* (*vector space*).

ОПРЕДЕЛЕНИЕ 12.30 Векторное пространство

Пусть F — поле. Множество V называется векторным пространством над F , если на нём заданы две операции: сложение векторов $v + w$ и умножение вектора на скаляр av , где $a \in F$ и $v \in V$. Относительно сложения V является коммутативной группой. Скалярное умножение удовлетворяет аксиомам:

1. **Дистрибутивность по сложению векторов:** $a(v + w) = av + aw$ для всех $a \in F, v, w \in V$.
2. **Дистрибутивность по сложению скаляров:** $(a + b)v = av + bv$ для всех $a, b \in F, v \in V$.
3. **Совместимость:** $a(bv) = (ab)v$ для всех $a, b \in F, v \in V$.
4. **Единица:** $1v = v$ для всех $v \in V$, где 1 — единица поля F .

Элементы V называются векторами, элементы F — скалярами.

Каждая аксиома фиксирует одно правило обращения с координатами. Дистрибутивность разрешает умножить каждую координату на скаляр, а потом сложить результаты — в любом порядке. Совместимость разрешает перемножить скаляры между собой, а результат умножить на вектор. Аксиома $1v = v$ отсекает вырожденный случай, когда единица поля действовала бы на векторы как ноль и скалярное умножение всё обнуляло бы.

Группа требовала одной операции и обратимости. Векторное пространство добавляет поле скаляров и вторую операцию, согласованную с первой. Без скалярного умножения векторы остались бы просто группой по сложению.

Самый привычный пример — обычная плоскость. Точки плоскости — это $\mathbb{R}^2$, пары действительных чисел. Сложение выполняется по координатам, а умножение

на число растягивает или сжимает вектор. Собственно, класс всех векторных пространств над $\mathbb{R}$ и есть то место, где живёт линейная алгебра.

Для нас сейчас важнее другой пример: пространство двоичных строк, к которому мы обратимся в главе 13.

Пример Двоичные строки как векторное пространство

Рассмотрим множество всех строк длины n из нулей и единиц. Обозначим его $\text{GF}(2)^n$. Сложение двух строк — поэлементное сложение по модулю два. Скаляры берём из поля $\text{GF}(2) = \{0, 1\}$, и в этом поле всего два элемента. Умножение строки на скаляр тривиально: $1v = v$, а $0v$ — строка из всех нулей.

Скалярное умножение «на единицу» ничего не меняет, поэтому вся структура пространства спрятана в сложении. Сложение в $\text{GF}(2)^n$ — это ровно то самое «исключающее ИЛИ», которое мы привыкли писать как хог.

В Rust вектор в $\text{GF}(2)^8$ — это просто байт, а сложение векторов — побитовый XOR.

```
// Вектор в GF(2)^8: байт --- строка из восьми битов.
let a: u8 = 0b1010_1100;
let b: u8 = 0b0110_1001;

// Сложение векторов --- поэлементный XOR.
let sum = a ^ b; // 0b1100_0101

// Нулевой вектор --- байт из всех нулей, он ничего не меняет.
let zero: u8 = 0;
assert_eq!(a ^ zero, a);

println!("{:08b} ^ {:08b} = {:08b}", a, b, sum);
```

Для $\text{GF}(2)^n$ и для $\mathbb{R}^n$ законы одни и те же, хотя в первом случае мы работаем с полем из двух элементов, а во втором — с полем действительных чисел. Доказав свойство один раз в терминах аксиом, мы получаем его сразу для всех полей и всех размерностей.

Чтобы говорить о размерности, нужно понять, когда один вектор выражается через другие. Вектор, раскладывающийся по остальным, не добавляет к набору ничего нового — именно это и есть зависимость векторов.

ОПРЕДЕЛЕНИЕ 12.31 Линейная комбинация

Вектор v называется **линейной комбинацией** векторов $v_1, \dots, v_m$, если

$$v = a_1 v_1 + a_2 v_2 + \dots + a_m v_m$$

для некоторых скаляров $a_1, \dots, a_m$.

ОПРЕДЕЛЕНИЕ 12.32 **Линейная зависимость**

Набор векторов называется **линейно зависимым**, если один из них выражается через остальные линейной комбинацией.

ОПРЕДЕЛЕНИЕ 12.33 **Линейная независимость**

Набор векторов называется **линейно независимым**, если ни один из его векторов не выражается через остальные.

Пример

В $\text{GF}(2)^3$ векторы 100 и 010 линейно независимы: ни один не выражается через другой. Вектор 110 выражается через них: $110 = 100 + 010$.

Линейная независимость — то свойство семейства векторов, которое превращает его в *матроид* из главы 17. Аксиомы наследственности и замены для независимых наборов — ровно те, что там проверялись. Так векторные пространства порождают пример матроида.

Набор векторов, через который выражается всё пространство, называется его **базисом**.

ОПРЕДЕЛЕНИЕ 12.34 **Базис**

Набор векторов $v_1, \dots, v_n$ называется **базисом** векторного пространства V , если он линейно независим и каждый вектор из V выражается через него линейной комбинацией.

ОПРЕДЕЛЕНИЕ 12.35 **Размерность**

Размерностью векторного пространства V называется число векторов в его базисе, обозначается $\dim V$.

Размерность — это число независимых свободных координат, которыми пространство задаётся. Плоскость $\mathbb{R}^2$ двумерна, потому что любая её точка задаётся двумя числами. Пространство $\text{GF}(2)^n$ имеет размерность n : строка задаётся n битами. Если в базисе n векторов, то поле из q элементов порождает пространство из q^n векторов, и для $\text{GF}(2)^n$ это число равно 2^n .

В $\text{GF}(2)^n$ есть стандартный базис — векторы $e_1, \dots, e_n$, у каждого из которых на i -й позиции стоит единица, а на остальных — нули. Через них раскладывается любая строка: она просто перечисляет, какие e_i складываются. Такая запись — координаты вектора, и они совпадают с самими битами строки.

Однако базис не обязан быть единственным. Любые n линейно независимых векторов из $\text{GF}(2)^n$ образуют базис, и размерность не зависит от выбора базиса.

Примечание

$\text{GF}(2^3)$ и $\text{GF}(2)^3$ — разные объекты, хотя в каждом по восемь элементов. Показатель внутри скобок задаёт размер поля — $\text{GF}(2^3)$ содержит $2^3 = 8$ элементов. Верхний индекс снаружи задаёт размерность — $\text{GF}(2)^3$ содержит $2^3 = 8$ векторов. Разница — в умножении. В поле каждый ненулевой элемент обратим, а в пространстве покомпонентное умножение имеет делители нуля, ведь $(1, 0, 0) \cdot (0, 1, 0) = (0, 0, 0)$.

Обратимся к вопросу, который прямо ведёт нас к кодам.

ОПРЕДЕЛЕНИЕ 12.36 Подпространство

Подмножество $W \subseteq V$ называется **подпространством** векторного пространства V , если оно само является векторным пространством над тем же полем.

Проверять все аксиомы не нужно — подпространству достаточно замкнутости относительно сложения и умножения на скаляр, а остальные аксиомы наследуются из V .

Пример Подпространства $\text{GF}(2)^3$

Возьмём пространство всех восьми строк длины три. Перечислим его подпространства. Первое — вырожденное: подпространство $W = \{000\}$, состоящее из одного нулевого вектора. Далее — семь одномерных подпространств. Каждому ненулевому вектору v соответствует подпространство $\{000, v\}$ из двух элементов, и различных векторов v здесь ровно семь.

Например, $W = \{000, 101\}$ — подпространство: сумма двух его векторов даёт 000, а скалярное умножение на ноль или единицу не выводит из набора. Вот пример двумерного подпространства. Векторы 100 и 010 линейно независимы. Их линейные комбинации — это 000, 100, 010 и сумма 110. Множество $W = \{000, 100, 010, 110\}$ — подпространство.

Таких двумерных подпространств, плоскостей, в $\text{GF}(2)^3$ тоже семь — каждая плоскость есть ортогональное дополнение ровно одной прямой. Всего же подпространств шестнадцать: одно нулевое, семь прямых, семь плоскостей и всё пространство.

Подпространства делают линейную структуру кодами. В следующей главе кодом станет набор сообщений, который можно проверить на ошибки. Линейные коды устроены ровно как подпространства $\text{GF}(2)^n$: код — это подпространство пространства всех строк длины n . Раз код — подпространство, для него выполняются все свойства векторного пространства, включая базис и размерность. Размерность кода — это число информационных бит: сколько битов сообщения код проносит без изменений. Общая длина кодового слова при этом равна размерности объёмлющего пространства.

Нить прослеживается сама: скалярное умножение требует поля, и мы получили векторное пространство. Векторное пространство требует понятия независимости, и мы получили базис. Из базиса вырастает размерность, а подпространства дают коды. Поле из предыдущего раздела позволило говорить о линейной структуре, в которой живут и коды, и матроиды.

Проверка Векторные пространства

1. Перечислите все аксиомы векторного пространства и объясните, зачем нужна аксиома $1v = v$.
2. Почему в $\text{GF}(2)^3$ одномерных подпространств ровно семь?
3. Почему для подпространства достаточно проверить замкнутость отдельно относительно сложения и относительно умножения на скаляр?
4. Объясните разницу между базисом и размерностью на примере $\text{GF}(2)^3$.

§ 12.11 * Полурешётки и полукольца

Лестница структур не заканчивается полем. Рядом с главной дорогой стоят ступени попроще: одна операция без обратных или две операции без вычитания. При первом чтении раздел можно пропустить, но оба понятия понадобятся в приложениях ниже.

ОПРЕДЕЛЕНИЕ 12.37 Полурешётка

Полугруппа называется **полурешёткой**, если её операция коммутативна и идемпотентна: $a \star a = a$ для каждого a .

Идемпотентность согласуется с интуицией объединения: применить операцию к элементу с самим собой — значит ничего не изменить. Порядок на полурешётке возникает, если положить $a \leq b$ при $a \star b = b$. Ассоциативность даёт транзитивность, идемпотентность — рефлексивность, коммутативность — антисимметрию, а операция $a \star b$ оказывается наименьшей верхней гранью пары. Так коммутативная идемпотентная полугруппа несёт на себе частичный порядок, и решётки из главы 8 — это полурешётки с двумя согласованными операциями.

ОПРЕДЕЛЕНИЕ 12.38 Полукольцо

Множество S с двумя операциями — сложением $+$ и умножением $\cdot$ — называется **полукольцом**, если выполнены четыре условия.

1. По сложению S — коммутативный моноид с нейтральным элементом 0.
2. По умножению S — моноид с нейтральным элементом 1.
3. Умножение дистрибутивно относительно сложения с обеих сторон.
4. Ноль поглощает умножение: $0 \cdot a = a \cdot 0 = 0$ для всех $a \in S$.

Полукольцо — это кольцо без вычитания: обратных по сложению не требуется, поэтому отрицательных элементов в нём нет. Натуральные числа с обычными арифметическими операциями — полукольцо, которому до кольца не хватает отрицательных. Булеан любого множества с объединением и пересечением — тоже полукольцо. Сумма и произведение типов из приложений ниже — ещё одно полукольцо, уже не на числах и не на множествах. Ближе всего к этой главе булево полукольцо: носитель $\{0, 1\}$ тот же, что у булева кольца из раздела о кольцах, но сложением служит $\vee$, а не $\oplus$.

§ 12.12 Приложения

Мы построили лестницу структур, но не сказали, зачем она нужна. Сейчас разберём шесть реальных систем и в каждой найдём свою ступень. Каждое применение пройдем целиком — от задачи до алгебраического закона, на котором оно стоит.

12.12.1 Живые переменные в компиляторе

Компилятор переводит программу в машинный код, а машинный код считает в регистрах — немногих быстрых ячейках памяти внутри процессора. Регистров мало, переменных в программе много, и на каждый регистр претендует сразу несколько переменных. Если две переменные никогда не бывают нужны одновременно, они могут делить один регистр по очереди, и программа станет быстрее. Чтобы решить, кто с кем делит регистр, компилятор должен знать, когда каждая переменная ещё нужна, а когда про неё можно забыть.

ОПРЕДЕЛЕНИЕ 12.39 Живая переменная

Переменная *жива* в точке программы, если её значение ещё будет прочитано на каком-нибудь пути из этой точки. Если значение больше никогда не прочитают, переменная *мертва*, и занятый ею регистр можно отдать другой.

Анализ идёт по программе назад, от конца к началу, и тащит за собой множество живых переменных. В конце функции живёт то, что функция возвращает. Каждый оператор меняет множество двумя движениями.

1. Оператор *убивает* переменную, которую присваивает, — старое значение больше никому не нужно.
2. Оператор *порождает* переменные, которые читает, — они обязаны дожить до этой строки.

Там, где два пути программы сходятся, множества объединяются, ведь переменная жива, если она нужна хотя бы на одном из путей.

Возьмём программу и прогоним по ней анализ.

```

1:  a := b + c
2:  d := a + 1
3:  if d > 0 then
4:      e := a
5:  else
6:      e := b
7:  f := e + c
8:  return f

```

Строка 1 читает b и c и пишет в a , строка 2 читает a и пишет в d , строка 3 читает d . Строки 4 и 6 пишут в e , читая соответственно a и b , строка 7 читает e и c и пишет в f , а строка 8 возвращает f . Разбор идёт с конца, и таблицу удобно читать снизу вверх.

Точка программы	Живые переменные
перед строкой 8, на выходе	$\{f\}$
перед строкой 7	$\{e, c\}$
перед строкой 6	$\{b, c\}$
перед строкой 4	$\{a, c\}$
после строки 3, на развилке	$\{a, b, c\}$
перед строкой 3	$\{a, b, c, d\}$
перед строкой 2	$\{a, b, c\}$
перед строкой 1, на входе	$\{b, c\}$

На выходе живо одно имя — f , потому что его возвращают. Строка 7 пишет в f и читает e и c , поэтому на её входе f сменяется парой e и c . Строка 6 пишет в e и читает b , поэтому перед ней живут b и c , а строка 4 пишет в e и читает a , поэтому перед ней живут a и c .

После строки 3 пути из строк 4 и 6 сходятся, и компилятор берёт объединение $\{a, c\} \cup \{b, c\} = \{a, b, c\}$. Строка 3 читает d , поэтому к объединению добавляется d , а строки 2 и 1 убирают по одной переменной и добавляют прочитанные. На входе функции живёт $\{b, c\}$ — аргументы, которые пришли снаружи, а всё вычисленное внутри компилятор волен выбрасывать и переиспользовать.

Объединение множеств ассоциативно, коммутативно и идемпотентно — $A \cup A = A$. Живые множества с операцией объединения — полурешётка из факультативного раздела выше. Анализ — это итерация до неподвижной точки. Мы начинаем с пустых множеств, прогоняем все операторы и повторяем, пока множества не перестанут меняться. Переменных в функции конечное число, множеств тоже конечное число, а каждое движение множества только растёт, поэтому итерация обязана остановиться. В разобранном выше примере циклов нет, поэтому хватило одного прохода, а повторять нужно, когда в потоке программы есть обратные рёбра. На вычисленных множествах компилятор строит граф, в котором две переменные соединены ребром, если живут одновременно, и раскрашивает его — каждый цвет получает отдельный регистр. Заодно присваивания мёртвым переменным удаляются целиком.

Классическое изложение — в «Драконьей книге»¹⁰³. Анализ живых переменных работает внутри LLVM, GCC и rustc.

12.12.2 Свёртка на кластере и MapReduce

Вернёмся к свёртке списка функцией `fold` из раздела о моноиде. Там аккумулятор накапливал сумму по одному числу, и порядок накопления не играл роли, потому что сложение ассоциативно. Ассоциативность разрешает переставить скобки, а значит — разбить список на куски, свернуть каждый кусок отдельно и сложить частичные суммы. Сумма не зависит от того, сложили ли мы сначала первую сотню чисел, а потом остальные, или наоборот.

Пока список помещается в одну машину, это экономия нескольких тактов. Когда данные не помещаются ни в одну память, это единственный способ вообще посчитать ответ. Кластер — сотни машин, каждая держит свой кусок данных, и ни одна не видит целого. MapReduce¹⁰⁴ — схема, по которой такой кластер сворачивает данные в два этапа.

1. На этапе *map* каждая машина проходит по своему куску и превращает каждую запись в пару «ключ — значение». Ключ — это пометка, по которой значения потом соберутся вместе.
2. На этапе *reduce* все пары с одинаковым ключом стекаются в одну кучу, и машины сворачивают каждую кучу одной операцией.

Стандартный пример — подсчёт слов. Каждая машина получает свой кусок текста, и на этапе *map* превращает каждое встреченное слово в пару (слово, 1). На этапе *reduce* для каждого слова складываются все единицы, и получается, сколько раз слово встретилось во всём корпусе.

Операция, которой складывают единицы, — это сложение, и оно ассоциативно. Порядок, в котором машины объединяют частичные суммы, никто не фиксирует, и ассоциативность гарантирует, что ответ от порядка не зависит.

Ассоциативность — то самое условие, без которого параллельная свёртка невозможна. Свернём тысячу чисел вычитанием, и разные машины дадут разные ответы в зависимости от порядка, в котором им пришли частичные результаты. $(10 - 3) - 2 = 5$, а $10 - (3 - 2) = 9$. Сложение, умножение, максимум и объединение множеств годятся, а вычитание и деление — нет. На том же принципе построен Apache Spark, который сворачивает данные поверх распределённых структур, и его операции группировки — ровно свёртки ассоциативной операцией.

12.12.3 Счётчик на нескольких машинах и CRDT

Представьте счётчик, который хранит, например, число просмотров страницы. Один сервер не справляется с наплывом запросов, поэтому счётчик держат сразу на

¹⁰³A. Aho, M. Lam, R. Sethi, J. Ullman. «Compilers: Principles, Techniques, and Tools». 2nd ed., 2006.

¹⁰⁴J. Dean, S. Ghemawat. «MapReduce: Simplified Data Processing on Large Clusters». OSDI, 2004.

нескольких серверах, и каждый обрабатывает свою долю. Запрос «добавь единицу» прилетает на какой-то из серверов, тот увеличивает свою копию, а время от времени серверы обмениваются копиями, чтобы прийти к согласию. Центрального сервера, который вёл бы один-единственный правильный счётчик, нет.

Наивный способ — хранить на каждом сервере число и при обмене как-то их сливать. Если брать максимум, две единицы, прибавленные на разных серверах, схлопнутся в одну. Если складывать, то при повторном обмене те же единицы прибавятся второй раз, и счётчик раздуется. Причина в том, что две операции «прибавить единицу» нельзя слить безнаказанно, как числа.

Выход в том, чтобы хранить вклад каждой машины отдельно, а общее значение считать суммой вкладов. Счётчик G-Counter — это таблица, в которой для каждой машины записано, сколько раз именно она нажала на счётчик. Машина A хранит $\{A : 3, B : 5\}$ — значит, A нажала три раза, B — пять, а общее значение 8. Прибавить единицу — увеличить только свой вклад. Примирить два счётчика — взять попарный максимум одинаковых вкладов.

```
use std::collections::HashMap;

// Вклад каждой машины: имя машины -> сколько раз она нажала.
#[derive(Clone, Debug)]
struct GCounter {
    counts: HashMap<&'static str, u64>,
}

impl GCounter {
    fn new() -> Self {
        GCounter { counts: HashMap::new() }
    }

    // Машина прибавляет единицу только к своему вкладу.
    fn increment(&mut self, machine: &'static str) {
        *self.counts.entry(machine).or_insert(0) += 1;
    }

    // Примирение --- попарный максимум вкладов.
    fn merge(&mut self, other: &GCounter) {
        for (&m, &n) in &other.counts {
            let mine = self.counts.entry(m).or_insert(0);
            *mine = (*mine).max(n);
        }
    }

    // Общее значение --- сумма всех вкладов.
    fn value(&self) -> u64 {
        self.counts.values().sum()
    }
}
```

Пример Три нажатия и пять нажатий

Машина A нажала три раза и хранит таблицу $\{A : 3\}$, машина B нажала пять раз и хранит $\{B : 5\}$. После обмена у каждой таблица $\{A : 3, B : 5\}$, и общее значение $3 + 5 = 8$. Обменяются ещё раз — попарный максимум таблиц $\{A : 3, B : 5\}$ и $\{A : 3, B : 5\}$ снова даёт $\{A : 3, B : 5\}$, и ничего не задваивается.

ОПРЕДЕЛЕНИЕ 12.40 CRDT

CRDT (*conflict-free replicated data type*) — структура данных, которую несколько машин правят одновременно, а примиряют без центрального сервера и без конфликтов. Примирение обязано быть ассоциативным, коммутативным и идемпотентным, то есть операцией полурешётки.

Максимум идемпотентен, ведь $\max(x, x) = x$, поэтому применить примирение дважды — всё равно что один раз. Максимум к тому же ассоциативен и коммутативен, поэтому серверы могут обмениваться в любом порядке и в любых сочетаниях. Эти три свойства дают тот же вид структуры, что и у объединения живых множеств, — полурешётку, только с операцией максимума вместо объединения. Сам тип данных описан Шапиро и соавторами¹⁰⁵. Счётчик G-Counter — простейший CRDT, и он работает в базах данных Riak и Redis.

12.12.4 Обмен ключами Диффи–Хеллмана

Два собеседника, Алиса и Боб, хотят договориться об общем секрете, например о ключе шифра. Канал между ними открыт, и всё, что они передают, слышит Ева. Как двум людям, у которых не было общего секрета, создать его на глазах у подслушивающего?

Протокол Диффи–Хеллмана делает это возведением в степень в циклической группе. Алиса и Боб открыто договариваются о простом p и образующем g мультипликативной группы $\mathbb{Z}_p^*$ — группы ненулевых остатков по модулю p . Образующий — элемент, степени которого пробегают всю группу, мы говорили о нём в разделе о группах. Протокол состоит из четырёх шагов.

1. Каждый придумывает секрет и держит его в тайне. Алиса загадывает a , Боб — b .
2. Алиса шлёт Бобу $g^a \bmod p$, Боб шлёт Алисе $g^b \bmod p$.
3. Каждый возводит полученное в свой секрет. Алиса считает $(g^b)^a$, Боб — $(g^a)^b$.
4. Оба приходят к одному числу, потому что $(g^a)^b = (g^b)^a = g^{a \cdot b} \bmod p$.

Это число и есть общий секрет. Рисунок 66 собирает протокол в схему: по открытому каналу между Алисой и Бобом летят только степени g^a и g^b , а Ева подслушивает канал.

¹⁰⁵M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. «Conflict-Free Replicated Data Types». SSS, 2011.

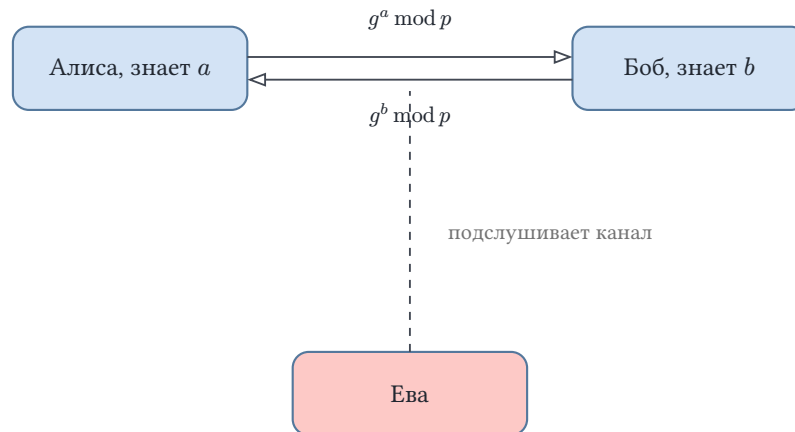


Рис. 66. Обмен ключами Диффи–Хеллмана.

Пример Протокол на маленьких числах

Возьмём $p = 23$ и $g = 5$. Алиса выбирает $a = 6$ и шлёт $5^6 \bmod 23 = 8$, Боб выбирает $b = 15$ и шлёт $5^{15} \bmod 23 = 19$. Алиса вычисляет $19^6 \bmod 23 = 2$, Боб — $8^{15} \bmod 23 = 2$. Общий секрет — 2.

Ева видит g , g^a и g^b , но секрет по ним не восстановит. Ей нужно найти a по известным g и g^a , то есть решить уравнение $5^x \equiv 8 \pmod{23}$. Это задача дискретного логарифма. Логарифм называется дискретным, потому что степени пробегают конечную группу и решать приходится перебором, а не формулой. В нашем примере перебор короткий — $5^1 = 5$, $5^2 = 2$, $5^3 = 10$, $5^4 = 4$, $5^5 = 20$, $5^6 = 8$, и на шестом шаге ответ найден. В настоящем протоколе p занимает сотни бит, группа огромна, и перебрать её за разумное время невозможно — на этом и держится защита. Протокол предложен Диффи и Хеллманом¹⁰⁶ и лежит в основе HTTPS и SSH, а его вариант на эллиптических кривых — в мессенджере Signal. Тонкости модулярной арифметики и атаки — в главе 19.

12.12.5 Контрольная сумма CRC

Файл передали по кабелю, и один бит перевернулся с нуля на единицу. Как приёмнику заметить порчу, не прося передать файл заново?

Запишем файл как многочлен. Каждый бит — коэффициент при своей степени x , и биты читаются слева направо, от старшей степени к младшей. Битами 1, 1, 0, 1 задаётся многочлен $x^3 + x^2 + 1$. Арифметика коэффициентов — над полем $\mathbb{Z}_2$, где $1 + 1 = 0$, поэтому вычитание совпадает со сложением, и оба суть операция XOR.

ОПРЕДЕЛЕНИЕ 12.41 Контрольная сумма CRC

Контрольная сумма CRC (*cyclic redundancy check*) — остаток от деления многочлена, представляющего данные, на фиксированный многочлен-образующую.

¹⁰⁶W. Diffie, M. Hellman. «New Directions in Cryptography». IEEE Transactions on Information Theory, 1976.

Например, для CRC-8 образующая — $x^8 + x^2 + x + 1$.

Возьмём сообщение 1, 1, 0, 1 и образующую $x^3 + x + 1$, которой отвечают биты 1, 0, 1, 1. Степень образующей равна трём, поэтому к сообщению дописывают три нуля и делят на образующую. Деление выполняется как обычное деление столбиком, только вычитание заменено на XOR.

Пример CRC трёхбитного сообщения

Сообщение 1, 1, 0, 1 — это многочлен $x^3 + x^2 + 1$, а после дописывания трёх нулей — $x^6 + x^5 + x^3$. Делим $x^6 + x^5 + x^3$ на $x^3 + x + 1$ по модулю два, шаг за шагом.

1. первый член частного — x^3 , вычитаем $x^6 + x^4 + x^3$ и получаем $x^5 + x^4$;
2. второй — x^2 , вычитаем $x^5 + x^3 + x^2$ и получаем $x^4 + x^3 + x^2$;
3. третий — x , вычитаем $x^4 + x^2 + x$ и получаем $x^3 + x$;
4. четвёртый — 1, вычитаем $x^3 + x + 1$ и получаем 1.

Остаток 1 — это контрольная сумма, биты 0, 0, 1.

Передачик дописывает остаток к сообщению и шлёт 1, 1, 0, 1, 0, 0, 1. Приёмник делит всё полученное на образующую. Если биты не повреждены, остаток равен нулю — контрольная сумма подобрана ровно так, чтобы сообщение делилось нацело. Перевернётся один бит, и вместо 1, 1, 0, 1, 0, 0, 1 придёт 1, 1, 1, 1, 0, 0, 1, а остаток от деления на $x^3 + x + 1$ окажется равным $x^2 + x$, то есть 1, 1, 0, и приёмник по ненулевому остатку узнаёт об ошибке.

Деление столбиком не делают вручную — его выполняет сдвиговый регистр. Это цепочка ячеек, по которой биты текут слева направо, а в позициях, где у образующей стоит единица, помещён вентиль XOR, подмешивающий биты образующей. Такой регистр собирается из нескольких десятков вентилях и почти ничего не стоит, поэтому CRC встроена в кадры Ethernet, архивы ZIP и файлы PNG. Классическое описание циклических кодов — Петерсон и Браун¹⁰⁷.

12.12.6 Суммы типов и отсутствие null

Языки с указателями долго позволяли любой переменной принять значение null — «здесь ничего нет». Тони Хоар, придумавший null-указатель, назвал его ошибкой на миллиард долларов¹⁰⁸. Программы падали, потому что программист забывал проверить, а вдруг значение — null. Сумма типов убирает эту ошибку на уровне языка.

¹⁰⁷W. Peterson, D. Brown. «Cyclic Codes for Error Detection». Proceedings of the IRE, 1961.

¹⁰⁸C. A. R. Hoare. «Null References: The Billion Dollar Mistake». QCon London, 2009.

ОПРЕДЕЛЕНИЕ 12.42 Сумма типов

Сумма типов — тип, значение которого принадлежит ровно одному из перечисленных вариантов. Каждое значение несёт метку, по которой видно, какой это вариант.

В Rust $\text{Option}(T)$ — сумма типа T и особого значения None , означающего «ничего». Значение типа $\text{Option}(\text{i32})$ — это либо число, либо None . Функция, которая может не найти ответ, возвращает Option , и вызывающий обязан рассмотреть оба варианта, прежде чем пользоваться значением.

```
fn safe_divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 {
        None
    } else {
        Some(a / b)
    }
}

let q = match safe_divide(4.0, 2.0) {
    Some(x) => x,
    None => return,
};
```

Конструкция `match` обязана перечислить все варианты, поэтому забыть про None невозможно — компилятор откажется собирать программу. Деление на ноль больше не ломает программу тихо. Оно превратилось в значение None , которое нельзя проигнорировать.

Имя «сумма» проверяется подсчётом значений.

1. У $\text{Option}(T)$ значений $1 + |T|$ — одно значение None плюс по одному на каждое значение T .
2. У $\text{Result}(T, E)$ — типа «либо результат, либо ошибка» — значений $|T| + |E|$.
3. У пары (T, U) значений $|T| \cdot |U|$ — первый элемент выбирают $|T|$ способами и для каждого — $|U|$ способов для второго.

Сумма типов складывает мощности вариантов, а пара их перемножает. Набор типов с этими двумя операциями образует полукольцо из факультативного раздела выше. На суммах типов построены Rust, Swift и Kotlin, и они убрали из программ целый класс аварий, связанных с `null`.

Булево кольцо замыкает ту же линию на уровне железа — полусумматор и полный сумматор из главы 11 суть операции поля $\text{GF}(2)$. Поле ведёт дальше, к кодам Рида-Соломона из главы 13, которые восстанавливают стёртые символы по значениям многочлена над конечным полем.

§ 12.13 Связь с кодами

Подпространства $\text{GF}(2)^n$ дают линейные коды: код — это подпространство, а его базис задаёт, какие сообщения кодируются. Чем больше размерность кода, тем больше данных он проносит. Но подпространство само по себе не исправляет ошибки. Защиту добавляет расстояние: чем дальше разнесены кодовые слова, тем больше искажений код замечает и чинит.

Поле $\text{GF}(2^m)$ понадобилось кодам, потому что они работают с байтами, а не с отдельными битами. Кольца для кода мало: восстановление по потерянному значению — это интерполяция, а она опирается на деление на ненулевые элементы. В кольце $\mathbb{Z}_{256}$ делить можно не всегда, и код на нём не построить.

Код Рида–Соломона кодирует сообщение значениями многочлена над $\text{GF}(q)$. Сообщение задаёт коэффициенты многочлена степени меньше k , а кодовое слово — его значения в n точках поля. Получателю достаточно любых k из этих значений: через них проходит ровно один многочлен степени меньше k , и интерполяция его восстанавливает. Число элементов поля задаёт, сколько символов помещается в одно кодовое слово, а циклическая группа $\text{GF}(q)^*$ — порядок, в котором берутся значения.

Мост к кодам замкнут: операция — структура — поле — векторное пространство — код. Та же лестница ведёт и к криптографии: поле $\text{GF}(2^8)$ живёт в AES, а мультипликативная группа $\mathbb{Z}_p^*$ — в обмене ключами Диффи–Хеллмана. Алгебра, выстроенная в этой главе, оказывается общим языком для защиты информации.

§ 12.14 Итоги главы

Глава началась с вопроса, как из простых операций вырастают структуры. Ответ выстроился лестницей, и каждая ступень добавляла одну возможность.

Операция — правило, сопоставляющее паре элемент. Полугруппа добавляет ассоциативность: накапливать можно, не думая о скобках. Моноид добавляет нейтральный элемент: накоплению есть с чего начать. Группа добавляет обратимость: накопленное можно отменить, и именно обратимость превращает набор преобразований в язык симметрий. Дальше понадобились две операции. Кольцо соединяет сложение и умножение дистрибутивностью, а поле добавляет деление: каждый ненулевой элемент обратим. Так арифметика целых чисел дорастает до арифметики, где можно делить, — и до конечных полей, где это деление устроено по модулю многочлена.

Понятия, возникшие по пути, работают и за пределами этой лестницы. Подгруппы и теорема Лагранжа показывают, как группа распадается на равные части, и вскрывают криптографическую атаку Полига–Хеллмана. Действие группы на множестве описывает симметрии и готовит подсчёт с точностью до симметрии. Гомоморфизм и изоморфизм позволяют говорить об одной структуре на разных носителях, и это

делает единственным поле $\text{GF}(p^m)$. Векторные пространства над полем связывают алгебру с линейной структурой, в которой живут коды и матроиды.

Вся лестница целиком — на рис. 67: от полугруппы до поля пять ступеней, и каждая стрелка подписана возможностью, которую добавляет переход.

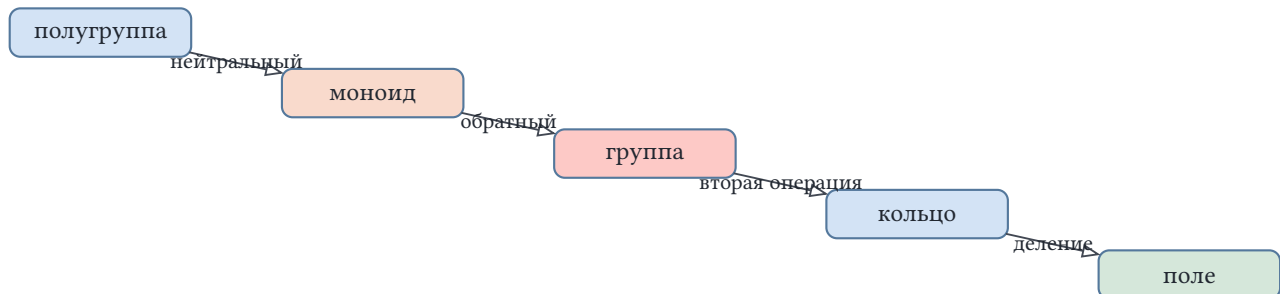


Рис. 67. Лестница структур: полугруппа, моноид, группа, кольцо, поле.

Лестница операций ведёт к кодам, исправляющим ошибки, и к криптографии, защищающей обмен. В следующей главе эти структуры встретятся в деле: линейные коды над $\text{GF}(2)$ и коды Рида–Соломона над $\text{GF}(2^8)$.

Проверка Проверьте себя

1. Чем моноид отличается от полугруппы и почему свертке пустого списка нужен нейтральный элемент?
2. Почему группа симметрий квадрата не коммутативна, хотя каждая строка её таблицы Кэли содержит все элементы ровно по разу?
3. Что гарантирует теорема Лагранжа и какой запрет она накладывает на порядки элементов?
4. Почему кольцо $\mathbb{Z}_6$ не является полем, а $\mathbb{Z}_5$ — является?
5. Как поле $\text{GF}(4)$ строится из $\mathbb{Z}_2$ и что такое примитивный элемент?
6. Что значит «структуры изоморфны» и почему поле $\text{GF}(2^8)$ единственно с точностью до изоморфизма?

Что дальше?

За пределами этой главы алгебраические структуры расступаются в ширину и вглубь. Теория Галуа — та самая, с которой глава начиналась — объясняет, какие уравнения решаются в радикалах: разрешимость оказывается строением группы симметрий корней. Теория представлений сажает группы и алгебры в матрицы: дискретное преобразование Фурье из анализа схем — представление циклической группы, и этот мост работает в сжатии сигналов и квантовых вычислениях. Гомологическая алгебра измеряет, насколько последовательность модулей уклоняется от точности, и стала рабочим языком топологии и теории категорий из последней главы книги. От абстрактных структур мы переходим к их первому делу — защите информации. В главе 13 изученные здесь поля и векторные пространства становятся кодами: линейные коды как подпространства $\text{GF}(2)^n$, коды Рида–Соломона как значения многочлена над конечным полем.

§ 12.15 Упражнения

Операция и структуры.

1. Постройте таблицу Кэли для XOR на $\{0, 1\}$. Покажите, что это группа, и найдите порядок каждого элемента.
2. Является ли множество $\{1, 2, 3\}$ с операцией максимума моноидом? А группой? Обоснуйте.
3. * Покажите, что операций на множестве из трёх элементов с фиксированным нейтральным элементом ровно $3^4 = 81$, и приведите пример неассоциативной операции с нейтральным элементом.

Группы.

4. Выпишите все элементы группы симметрий треугольника D_3 и таблицу Кэли. Найдите порядок каждого элемента.
5. В группе $\mathbb{Z}_{12}$ по сложению найдите все подгруппы и порядки их элементов.
6. * Сколько подгрупп порядка 4 в группе $\mathbb{Z}_{16}$? Почему ровно столько?
7. Докажите следствие из теоремы Лагранжа: группа простого порядка p циклическая, и каждый её неединичный элемент — образующий. Подсказка: порядок элемента делит p , а допустимых значений два.
8. * Малая теорема Ферма: для простого p и a , не кратного p , верно $a^{p-1} \equiv 1 \pmod{p}$. Выведите её из теоремы Лагранжа, применив её к группе $\mathbb{Z}_p^*$.

Кольца и поля.

9. Вычислите обратные элементы в $\mathbb{Z}_7$ и проверьте каждое умножением.
10. Является ли кольцо вычетов по модулю 9 полем? Найдите ненулевой элемент без обратного.
11. * Постройте поле $\text{GF}(4)$: таблицы сложения и умножения, найдите примитивный элемент.
12. Найдите все обратимые элементы кольца $\mathbb{Z}_{12}$ и объясните, почему их число равно $\varphi(12)$ (функция Эйлера).
13. * Постройте поле $\text{GF}(8)$ над $\mathbb{Z}_2$ по неприводимому многочлену $f(x) = x^3 + x + 1$. Найдите таблицу умножения и порядок каждого ненулевого элемента, определите примитивный элемент.

Гомоморфизм и векторные пространства.

14. Проверьте, что отображение $\mathbb{Z} \rightarrow \mathbb{Z}_5$ с $x \mapsto x \bmod 5$ — гомоморфизм колец, и найдите его ядро.
15. Перечислите все подпространства $\text{GF}(2)^3$ и их размерности.
16. Покажите, что группы $\mathbb{Z}_4$ и $\mathbb{Z}_2 \times \mathbb{Z}_2$ неизоморфны, хотя обе имеют порядок четыре. Подсказка: сравните порядки элементов.
17. * Теорема об орбите и стабилизаторе: размер орбиты равен $\frac{|G|}{|\text{Stab}(x)|}$. Проверьте её на действии группы поворотов квадрата на множестве вершин и на множестве раскрасок.

ПРОЕКТ Калькулятор конечных полей

Соберите инструмент для работы с конечными полями: арифметику простого поля $\mathbb{Z}_p$ и поля расширения $\text{GF}(p^m)$. Инструмент пригодится, чтобы проверять вычисления из этой главы и готовиться к кодам.

① Арифметика простого поля

Реализуйте $\mathbb{Z}_p$: сложение и умножение по модулю простого p , а также поиск обратного элемента расширенным алгоритмом Евклида. Проверьте, что каждый ненулевой элемент обратим, если p — простое, и что в $\mathbb{Z}_6$ необратимость возникает у элементов, не взаимно простых с модулем.

② Расширение до $\text{GF}(p^m)$

Реализуйте многочлены над $\mathbb{Z}_p$ и умножение по модулю неприводимого многочлена $f(x)$ степени m . Проверьте неприводимость нескольких многочленов и убедитесь, что в $\text{GF}(4)$ с $f(x) = x^2 + x + 1$ выполняется $\alpha^2 = \alpha + 1$.

③ Примитивный элемент

Реализуйте поиск примитивного элемента поля: элемента, чьи степени пробегает все ненулевые элементы. Проверьте, что в $\text{GF}(4)$ элемент α примитивен, а в поле для байтов элемент x имеет порядок 51.

④ Таблица степеней и логарифмов

Постройте по примитивному элементу таблицы степеней и логарифмов: каждому ненулевому элементу сопоставьте его логарифм k — степень α^k . Умножение сведите к сложению логарифмов по модулю $p^m - 1$, деление — к вычитанию, и проверьте, что таблица воспроизводит арифметику поля. Объясните, почему именно так устроено умножение в поле для байтов (AES) из главы о кодах.

⑤ Связь с кодом

Используйте поле $\text{GF}(2^m)$, чтобы закодировать сообщение многочленом и вычислить его значения в степенях примитивного элемента — прообраз кода Рида–Соломона.

Итог: калькулятор конечных полей: арифметика $\mathbb{Z}_p$ с обратными элементами по Евклиду, поле расширения $\text{GF}(p^m)$ как многочлены по модулю неприводимого, примитивный элемент, таблицы степеней и логарифмов для быстрого умножения и прообраз кодирования многочленом — готовый инструмент для проверки вычислений главы и перехода к кодам.

XIII

Коды и информация

Когда канал передаёт данные, он их искажает. Бит переворачивается, несколько соседних битов портятся разом, и принятое сообщение перестаёт совпадать с отправленным. Сбой этот не зависит от отправителя или получателя — он присущ самому каналу. Задача в том, чтобы получатель восстановил сообщение, даже когда часть пути прошла с ошибками.

Одна и та же задача возникает всюду, где данные покидают отправителя. Провод, радиоволна, дорожка диска — всё это каналы, и все они ошибаются. Запись на диск — тоже передача, только из прошлого в будущее. Отправитель пишет биты, а каналом служит время и износ носителя. На вход сообщение подаётся в одном виде, а на выход приходит в другом.

Интуиция предлагает защиту простого вида — избыточность. Если один бит b может быть испорчен, она велит передать его трижды, записав вместо 0 — 000, вместо 1 — 111. Одна ошибка затрагивает лишь одну копию из трёх, и две оставшиеся её перевешивают. Голосование по большинству восстанавливает исходный бит. Направление у этой интуиции верное, а пропорция — нет: чем больше защиты, тем надёжнее приём. Из трёх переданных битов полезен один, две трети канала потрачены на защиту, а не на данные. Десять полезных битов требуют тридцати в канале, сто — трёхсот. Защита всегда оплачивается полезной скоростью. Чем больше избыточности, тем меньше полезных бит в канале за единицу времени. Разумная постановка задачи — минимизировать плату при заданной надёжности.

Повторение измеряет исправляющую способность числом копий: три копии исправляют одну ошибку, пять — две. Но за этим числом не видно, откуда взялось само правило «три» или «пять»: повторение не говорит, достаточен ли запас и насколько он далёк от минимально возможного. Нужна мера, которая скажет, что трёх повторов достаточно, а пяти — избыточно. Такая мера есть — расстояние между кодовыми словами: им измеряют, насколько они разнесены, — так же, как метром измеряют расстояние до цели. Без неё инженер, наращивающий повторение, движется вслепую, и любое число копий выглядит одинаково оправданным.

Есть и второй изъян. Повторение предполагает, что ошибки изолированы. Между тремя копиями бита вклинивается не больше одной ошибки. Реальные каналы искажают целыми сериями. Всплеск помех портит сразу участок сообщения, и на трёх копиях подряд такая серия оставляет меньше половины. Голосование ошибается само.

Значит, защита держится на трёх вещах. Во-первых, на мере того, насколько разнесены кодовые слова. Чем дальше друг от друга стоят кодовые слова, тем больше искажений между ними остаётся исправимым. Во-вторых, на конструкциях, тратящих избыточность экономно, а не трёхкратным повторением. Это коды, занимающие канал заметно меньше. Во-третьих, на границе, ниже которой ни одна конструкция не спасает.

Надёжность кода определяется расстоянием между его словами, а не числом повторений: три копии — частный случай, где два слова разнесены на три позиции. Те же три лишних бита, если распорядиться ими иначе, защищают больше данных при том же расстоянии. Код-повторение — простая и дорогая форма избыточности: он защищает, платя полной ценой.

Граница, ниже которой ни одна конструкция не спасает, существует, и она жёсткая. Из неё следует, что цена защиты определяется каналом, а не навыком. Для каждого канала есть скорость, ниже которой сообщение передаётся со сколь угодно малой вероятностью ошибки, а выше — нет. Тем самым компромисс между надёжностью и эффективностью становится измеримым.

Избыточность — лишь одна грань вопроса, сколько бит действительно нужно сообщению. Защита эти биты добавляет, сжатие убирает лишние, оставляя только несущие информацию. Обе операции двигают одно и то же отношение — долю полезного в переданном — и обе упираются в границы, которые устанавливает теория информации: энтропию для сжатия, пропускную способность канала для защиты.

Как защитить информацию от ошибок — и какова цена этой защиты?

ИСТОРИЯ Шеннон и Хэмминг: рождение теории кодирования

Клод Шеннон опубликовал «A Mathematical Theory of Communication» в 1948 году — две части в Bell System Technical Journal. В этой работе он ввёл понятие информации, энтропии и пропускной способности канала, а также доказал свою теорему кодирования. Для любого канала с шумом существуют коды, передающие информацию со сколь угодно малой вероятностью ошибки — если скорость передачи ниже пропускной способности. Доказательство было неконструктивным: случайный код достаточно большой длины с высокой вероятностью окажется хорошим, но как его построить явно — Шеннон не говорил.

Ричард Хэмминг работал в той же Bell Labs и, по собственному признанию, придумал свой код от раздражения. Компьютер с перфокартами останавливался при каждой ошибке чтения, и выходные проходили впустую — Хэмминг решил, что машина должна находить и исправлять ошибки сама. Его статья «Error Detecting and Error Correcting Codes» (1950¹⁰⁹) дала первый явный код, исправляющий одну ошибку. С неё начинается алгебраическая теория кодирования — явные конструкции, стремящиеся к границе, указанной Шенноном¹¹⁰.

¹⁰⁹R. W. Hamming. «Error Detecting and Error Correcting Codes». Bell System Technical Journal, 1950.

¹¹⁰C. E. Shannon. «A Mathematical Theory of Communication». Bell System Technical Journal, 1948.

ОБЗОР ГЛАВЫ

В этой главе мы научимся измерять расстояние между кодовыми словами — расстояние Хэмминга и кодовое расстояние — и по нему судить, сколько ошибок код обнаружит и сколько исправит. Код Хэмминга откроет дверь в линейные коды и конечные поля $GF(2^m)$. Граница Шеннона покажет, где кончается сама возможность надёжной передачи, а код Хаффмана научит сжимать данные под заданные частоты. В конце циклические коды и коды Рида–Соломона объяснят, как многочлены находят повреждённый пакет в Ethernet и восстанавливают музыку на исцарапанном диске. Комбинаторные границы Хэмминга и Синглтона замкнут основную часть главы и протянут нить к счёту объектов в главе 18.

После этой главы вы сможете:

1. измерять расстояние между кодовыми словами и определять по кодовому расстоянию, сколько ошибок код обнаруживает и исправляет;
2. строить код Хэмминга и работать в конечных полях;
3. оценивать границу надёжной передачи по Шеннону;
4. строить код Хаффмана по частотам символов;
5. объяснять, как циклические коды и коды Рида–Соломона защищают Ethernet и компакт-диски.

§ 13.1 Расстояние Хэмминга и геометрия кода

Прежде чем исправлять ошибки, нужно научиться измерять расстояние между кодовыми словами. Инструмент для этого — расстояние Хэмминга.

ОПРЕДЕЛЕНИЕ 13.1 Расстояние Хэмминга

Расстоянием Хэмминга между двумя строками одинаковой длины называется число позиций, в которых они различаются, обозначается

$$d(x, y) = |\{i \mid x_i \neq y_i\}|.$$

Расстояние Хэмминга — метрика, то есть оно неотрицательно ($d(x, y) \geq 0$), симметрично и удовлетворяет неравенству треугольника. Геометрический смысл в том, что для превращения строки x в строку y нужно перевернуть ровно $d(x, y)$ битов.

Пример Расстояние Хэмминга

1. $d(10110, 10001) = 3$: различаются биты в позициях 3, 4 и 5
2. $d(0000, 1111) = 4$
3. $d(x, x) = 0$

ОПРЕДЕЛЕНИЕ 13.2 Кодовое расстояние

Кодовым расстоянием d кода C называется минимальное расстояние Хэмминга между любой парой различных кодовых слов, обозначается

$$d = \min(\{d(x, y) \mid x, y \in C, x \neq y\}).$$

ОПРЕДЕЛЕНИЕ 13.3 Параметры кода

Код характеризуется тройкой параметров $(n, |C|, d)$:

- n — длина кода;
- $|C|$ — число кодовых слов;
- d — кодовое расстояние.

Такой код называется $(n, |C|, d)$ -кодом.

Для линейных кодов пишут (n, k, d) , где k — размерность кода. В этом случае $|C| = 2^k$.

ЛОВУШКА Расстояние Хэмминга — позиции, а не ошибки

Расстояние Хэмминга $d(c_1, c_2)$ — это число позиций, в которых слова c_1 и c_2 различаются, а не разность числа единиц и не число ошибок. Ошибка возникает, когда строки читают как числа. Разность числа единиц равна $3 - 2 = 1$, хотя разошлись три позиции — 3, 4 и 5. Число ошибок — характеристика кода. Кодовое расстояние d задаёт границы обнаружения и исправления. Обнаружение возможно до $d - 1$ ошибок. Исправление — до $\lfloor \frac{d-1}{2} \rfloor$.

УТВЕРЖДЕНИЕ 13.1 Обнаружение и исправление ошибок

Кодовое расстояние определяет исправляющую способность кода:

- Если $d \geq s + 1$: код *обнаруживает* до s ошибок. Принятое слово с $\leq s$ ошибками не может быть другим кодовым словом.
- Если $d \geq 2t + 1$: код *исправляет* до t ошибок.

Набросок доказательства

Докажем от противного.

Пусть $d \geq 2t + 1$, и рассмотрим шары радиуса t вокруг каждого кодового слова, то есть множества строк на расстоянии не больше t от центра.

Если два таких шара пересекаются, то существует строка на расстоянии $\leq t$ от двух разных кодовых слов. По неравенству треугольника расстояние между этими кодовыми словами не больше $2t$, что противоречит условию $d \geq 2t + 1$.

Следовательно, шары не пересекаются, и принятое слово попадает ровно в один шар, после чего декодируем его в центр этого шара.

Кодовые слова — «звёзды» в пространстве всех двоичных строк. Чем дальше разнесены звёзды, тем больше ошибок код может исправить, и принятое слово декодируем к ближайшей звезде. Пространство всех двоичных строк огромно, но кодовыми словами объявлена лишь малая его часть. Декодер возвращает принятую строку к ближайшему кодовому слову — как текстовый редактор исправляет опечатку к ближайшему правильному слову. Опечатка «програма» узнаваема, хотя написана неверно, и редактор подставит «программа», а не далёкое «прогресс». Каждая звезда на рис. 68 окружена шаром радиуса t , и все строки внутри шара декодируются к его центру.

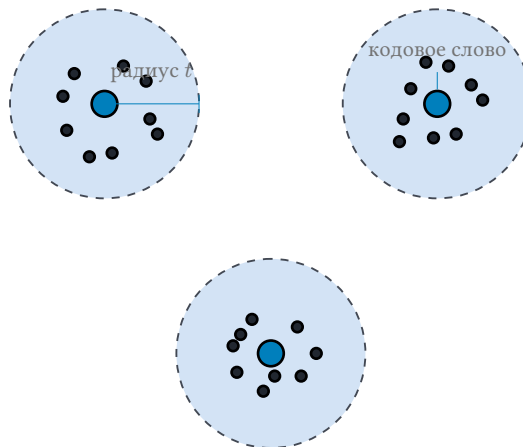


Рис. 68. Шары Хэмминга радиуса t вокруг кодовых слов.

Пример Параметры кодов

Код Хэмминга $(7, 4, 3)$: длина $n = 7$, $2^4 = 16$ кодовых слов, расстояние $d = 3$. $d \geq 2 \cdot 1 + 1 = 3$ — исправляет одну ошибку. Тривиальный код-повторение $(3, 1, 3)$: каждый бит повторяется трижды. $n = 3$, два кодовых слова $\{000, 111\}$, расстояние $d = 3$ — также исправляет одну ошибку.

Оба кода исправляют одну ошибку, но платят за это по-разному. Сравним цену защиты.

Код	n	Полезных бит на слово	Исправляет
Код-повторение $(3, 1, 3)$	3	1	1
Код Хэмминга $(7, 4, 3)$	7	4	1

При одинаковой надёжности код Хэмминга передаёт примерно в $\frac{12}{7} \approx 1.71$ раза больше данных в том же канале. Вот цена наивного подхода «повторю побольше» — он защищает не хуже, но сжигает канал впустую.

§ 13.2 Код Хэмминга (7, 4)

Код Хэмминга — исторически первый код, исправляющий ошибки. Параметры кода — $n = 7, k = 4, d = 3$, то есть четыре бита данных и исправление одной ошибки. Структура кодового слова (биты нумеруются с 1, биты 1, 2, 4 — проверочные, биты 3, 5, 6, 7 — информационные):

позиция: 1 2 3 4 5 6 7
бит: $p_1 p_2 d_1 p_4 d_2 d_3 d_4$

Проверочные биты вычисляются так, чтобы XOR битов в каждой из трёх контрольных групп равнялся нулю:

- Проверочный бит p_1 контролирует биты 1, 3, 5, 7. $p_1 = d_1 \oplus d_2 \oplus d_4$
- Проверочный бит p_2 контролирует биты 2, 3, 6, 7. $p_2 = d_1 \oplus d_3 \oplus d_4$
- Проверочный бит p_4 контролирует биты 4, 5, 6, 7. $p_4 = d_2 \oplus d_3 \oplus d_4$

Работа проверочных групп напоминает игру в двадцать вопросов, где каждый проверочный бит — вопрос, делящий множество позиций пополам. Вопрос «ошибка в правой половине?» даёт один бит ответа, а три таких вопроса складываются в двоичный номер позиции ошибки.

Проверочный бит p_i контролирует позиции, в двоичной записи которых бит номер i равен 1. Позиция $3 = 011$ входит в первые две группы, а позиция $7 = 111$ — во все три. На рис. 69 столбцы проверочных битов тянут линии вниз к данным своих контрольных групп, а кружок на линии отмечает включение бита в группу. Бит d_4 присутствует во всех трёх группах.

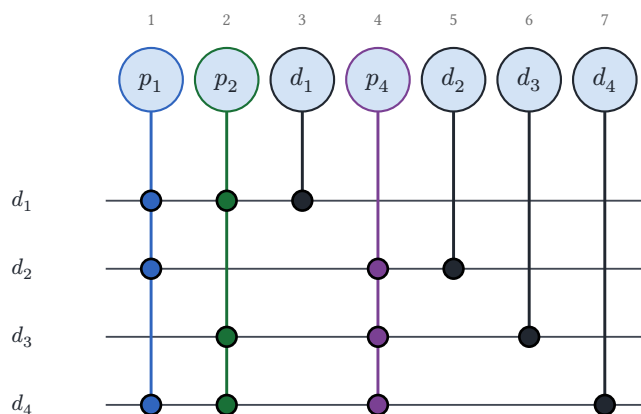


Рис. 69. Контрольные группы кода Хэмминга (7, 4).

Пример Кодирования

Данные: $d_1 = 1, d_2 = 0, d_3 = 1, d_4 = 1$.

$$p_1 = 1 \oplus 0 \oplus 1 = 0$$

$$p_2 = 1 \oplus 1 \oplus 1 = 1$$

$$p_4 = 0 \oplus 1 \oplus 1 = 0$$

Кодовое слово: $(p_1, p_2, d_1, p_4, d_2, d_3, d_4) = (0, 1, 1, 0, 0, 1, 1)$.

Примечание Код Хэмминга на Rust

Весь расчёт — три XOR-выражения. Кодирование выглядит так:

```
/// Код Хэмминга (7,4): 4 бита данных -> 7 бит кодового слова.
pub fn encode(data: [bool; 4]) -> [bool; 7] {
    let [d1, d2, d3, d4] = data;
    let p1 = d1 ^ d2 ^ d4;
    let p2 = d1 ^ d3 ^ d4;
    let p4 = d2 ^ d3 ^ d4;
    [p1, p2, d1, p4, d2, d3, d4]
}
```

Синдром — те же три XOR-выражения, собранные в число $(s_4 s_2 s_1)_2$:

```
/// 0 для корректного слова, иначе -- позиция одиночной ошибки.
pub fn syndrome(word: [bool; 7]) -> u8 {
    let [b1, b2, b3, b4, b5, b6, b7] = word;
    let s1 = b1 ^ b3 ^ b5 ^ b7;
    let s2 = b2 ^ b3 ^ b6 ^ b7;
    let s4 = b4 ^ b5 ^ b6 ^ b7;
    (s4 as u8) << 2 | (s2 as u8) << 1 | (s1 as u8)
}
```

Декодирование — исправление по синдрому, и если $s \neq 0$, переворачиваем бит с номером s . В коде это индекс $s - 1$, ведь Rust индексирует с нуля. Результат декодирования — структура с данными, числом исправленных ошибок и позицией ошибки:

```
/// Результат декодирования: данные, число исправленных ошибок, позиция ошибки.
pub struct Decoded {
    pub data: [bool; 4],
    pub corrected: usize,
    pub error_position: Option<u8>,
}

/// Выбрать биты данных из кодового слова.
fn data_bits(word: [bool; 7]) -> [bool; 4] {
    [word[2], word[4], word[5], word[6]]
}

/// Исправить одиночную ошибку: перевернуть бит на позиции синдрома.
pub fn decode(word: [bool; 7]) -> Decoded {
    let s = syndrome(word);
    if s == 0 {
        return Decoded { data: data_bits(word), corrected: 0, error_position: None };
    }
    let mut fixed = word;
```

```
fixed[(s - 1) as usize] = !fixed[(s - 1) as usize];
Decoded { data: data_bits(fixed), corrected: 1, error_position:
Some(s) }
}
```

При приёме вычисляется *синдром* — три бита, каждый как XOR битов своей контрольной группы:

$$s_1 = p_1 \oplus d_1 \oplus d_2 \oplus d_4$$

$$s_2 = p_2 \oplus d_1 \oplus d_3 \oplus d_4$$

$$s_4 = p_4 \oplus d_2 \oplus d_3 \oplus d_4$$

Название «синдром» пришло из медицины, где синдром — это совокупность симптомов, по которым врач распознаёт скрытую болезнь. Ненулевые проверочные биты — симптомы, а их совокупность — диагноз, то есть номер искажённого бита.

- Если все $s_i = 0$ — ошибок нет.
- Если синдром не равен нулю, номер искажённого бита равен $s_1 + 2s_2 + 4s_4$. Синдром читается как число $(s_4s_2s_1)_2$.
- Переворачиваем этот бит — ошибка исправлена.

ЛОВУШКА Совпадение синдрома с позицией — следствие выбора контрольных групп

Синдром — это три бита, каждый из которых — XOR битов своей контрольной группы. Для кода (7, 4) синдром численно совпадает с позицией ошибки, и (1, 1, 0) указывает на бит 3. Синдром (1, 1, 1) — на бит 7. Но это совпадение — следствие того, что столбцы проверочной матрицы H — двоичные номера позиций 1, ..., 7. При другом выборе групп то же значение синдрома укажет на другую позицию.

Синдром как двоичное число $(s_4s_2s_1)_2$ — это номер позиции с ошибкой. Все семь ненулевых синдромов соответствуют семи позициям кодового слова.

s_1	s_2	s_4	Синдром $(s_4s_2s_1)_2$	Позиция
1	0	0	1	1
0	1	0	2	2
1	1	0	3	3
0	0	1	4	4
1	0	1	5	5
0	1	1	6	6
1	1	1	7	7

Нулевой синдром (0, 0, 0) означает, что ошибок нет. Здесь s_1 — младший бит номера, а s_4 — старший, поэтому синдром просто читается как бинарная запись позиции.

Пример Исправление ошибки

Передано $(0, 1, 1, 0, 0, 1, 1)$. В позиции 7 (бит d_4) ошибка: принято $(0, 1, 1, 0, 0, 1, 0)$. $s_1 = 0 \oplus 1 \oplus 0 \oplus 0 = 1$

$$s_2 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$s_4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

Синдром $(s_1, s_2, s_4) = (1, 1, 1)$ указывает на позицию $1 + 2 + 4 = 7$. Переворачиваем бит 7: $0 \rightarrow 1$. Исправленное слово: $(0, 1, 1, 0, 0, 1, 1)$.

Пример Две ошибки: ложное исправление

Код с расстоянием 3 исправляет одну ошибку. Две ошибки он только *замечает* — синдром ненулевой. Но исправить их не может, ведь синдром указывает на неверную позицию, и декодер портит здоровый бит.

Передано $(0, 1, 1, 0, 0, 1, 1)$, ошибки в позициях 4 и 6. Принято $(0, 1, 1, 1, 0, 0, 1)$.

$$s_1 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$s_2 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$$

$$s_4 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$

Синдром $(s_1, s_2, s_4) = (0, 1, 0)$ имеет значение $0 + 2 \cdot 1 + 4 \cdot 0 = 2$. Декодер переворачивает бит 2: получается $(0, 0, 1, 1, 0, 0, 1)$. Данные получаются $(1, 0, 0, 1)$. Верные данные: $(1, 0, 1, 1)$. Две ошибки замаскировались под одну — в неверной позиции.

Пример Три ошибки: невидимое повреждение

Три ошибки могут остаться незамеченными вовсе. Если вектор ошибок сам является кодовым словом, XOR переводит переданное слово в другое кодовое слово, и синдром равен нулю.

Передано $(0, 1, 1, 0, 0, 1, 1)$, ошибки в позициях 1, 2 и 3. Принято $(1, 0, 0, 0, 0, 1, 1)$.

$$s_1 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$

$$s_2 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

$$s_4 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

Синдром нулевой — декодер сообщает «ошибок нет», хотя данные испорчены: $(0, 0, 1, 1)$. Правильные данные: $(1, 0, 1, 1)$. Вектор ошибок $(1, 1, 1, 0, 0, 0, 0)$ — кодовое слово. Оно соответствует данным $(1, 0, 0, 0)$, поэтому принятое слово тоже кодовое. За пределами гарантий $d - 1$ код за результат не отвечает.

Замечание Генетический код как помехоустойчивый

Генетический код использует избыточность для защиты от ошибок — с той же целью, что и код Хэмминга. Шестьдесят четыре кодона кодируют всего двадцать аминокислот — триплетный код вырожден, несколько разных кодонов соответ-

ствуют одной аминокислоте. Чаще всего различие между такими кодонами — в третьем нуклеотиде.

Точечная мутация в третьей позиции кодона часто не меняет аминокислоту, ведь кодоны GAA и GAG кодируют глутамат. Ошибка не имеет последствий — как одиночный переворот бита, который код Хэмминга исправляет на приёме.

Вырожденность генетического кода — следствие эволюции. Но математическая структура одна и та же — множество точек в пространстве, разнесённых на безопасное расстояние друг от друга, что в ядре клетки, что в оптоволокне.

Проверка Расстояние Хэмминга и избыточность

1. Почему расстояние Хэмминга между строками 10110 и 10001 равно 3, а не разности числа единиц $3 - 2 = 1$?
2. Код Хэмминга (7, 4, 3) и код-повторение (3, 1, 3) (каждый бит повторяется трижды) исправляют по одной ошибке. Чем код-повторение проигрывает коду Хэмминга?

§ 13.3 Расширенный код Хэмминга (8, 4, 4)

Пример с двумя ошибками выше показал цену расстояния 3. Двойная ошибка маскируется под одиночную и молча портит данные. Расширенный код Хэмминга снимает эту проблему одним дополнительным битом. К кодовому слову добавляется общий бит чётности, и кодовое расстояние вырастает до 4. Действительно, два кодовых слова базового кода различаются минимум в трёх позициях. Если число различий нечётно, то общие биты чётности у них тоже разные, так что полное расстояние не меньше 4. Минимум 4 достигается на парах, различающихся ровно в трёх позициях, поэтому кодовое расстояние расширенного кода равно 4. Код по-прежнему исправляет одну ошибку, но теперь отличает двойную от одиночной.

Синдром	Чётность	Что это и что делать
$s = 0$	сошлась	Ошибок нет
$s = 0$	не сошлась	Перевёрнут сам бит чётности (позиция 8) — исправляем его
$s \neq 0$	не сошлась	Одна ошибка — исправляем позицию s
$s \neq 0$	сошлась	Две ошибки — синдром указывает на неверную позицию, не исправляем

Пример Две ошибки в расширенном коде

Передано расширенное слово (0, 1, 1, 0, 0, 1, 1, 0). Данные: (1, 0, 1, 1). Ошибки в позициях 4 и 6: принято (0, 1, 1, 1, 0, 0, 1, 0).

Синдром первых семи бит: $(s_1, s_2, s_4) = (0, 1, 0)$ — значение 2. Чётность всего слова сошлась: единиц четыре.

По таблице выше это «две ошибки», ведь синдром указывает на бит 2, но мы его не трогаем — исправление было бы ложным. Декодер сообщает, что слово повреждено и данные不可信, вместо того чтобы молча исправить не тот бит.

§ 13.4 Линейные коды и $\text{GF}(2)$

Код Хэмминга — пример *линейного кода*, у которого множество кодовых слов замкнуто относительно XOR (сумма двух кодовых слов — снова кодовое слово). Это позволяет описать код линейной алгеброй над полем $\text{GF}(2)$.

ОПРЕДЕЛЕНИЕ 13.4 Линейный код

Множество $C \subset \text{GF}(2)^n$ называется **линейным кодом**, если оно содержит нулевое слово и замкнуто относительно покомпонентного XOR, то есть является линейным подпространством пространства $\text{GF}(2)^n$.

Двоичные строки длины n образуют векторное пространство $\text{GF}(2)^n$, и линейный код выбирает в нём подпространство. Запись $\text{GF}(2)^3$ — это векторное пространство всех троек битов, а не поле из восьми элементов. В нём $2^3 = 8$ векторов — самих троек (a, b, c) — и над ним определена только покомпонентная операция, без умножения с переносом. Поле из восьми элементов обозначается иначе, $\text{GF}(8)$ или $\text{GF}(2^3)$, и появляется в главе позже. Структура всех подпространств упорядочена включением и образует решётку (lattice), в которой каждое подпространство задаётся своим базисом. В главе 8 мы познакомились с решётками на примере булеана. На рис. 70 показаны все координатные подпространства $\text{GF}(2)^3$, где вершины — подпространства, а рёбра — включение.

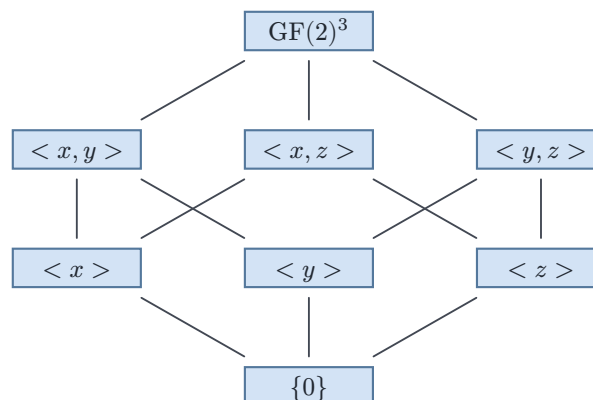


Рис. 70. Решётка координатных подпространств $\text{GF}(2)^3$ по включению.

Код задаётся *проверочной матрицей* (parity-check matrix) H . Её размер — $(n - k) \times n$. Слово c является кодовым, если $Hc = 0$. Все операции — по модулю 2, слово рассматривается как вектор-столбец. Для кода Хэмминга $(7, 4)$ матрица H имеет размер 3×7 . Её столбцы — двоичные записи чисел от 1 до 7:

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Строки H соответствуют трём контрольным группам кода Хэмминга: первая — биты s_1 (позиции 1, 3, 5, 7), вторая — s_2 (2, 3, 6, 7), третья — s_4 (4, 5, 6, 7). Столбец j — двоичная запись числа j , поэтому синдром $H \cdot r$ численно совпадает с номером искажённой позиции.

Синдром $s = H \cdot r$ вычисляется по принятому слову r и указывает на позицию ошибки. Если ошибка в бите i , синдром s равен столбцу i проверочной матрицы H .

Пример Синдром через матрицу

В матричном представлении r — вектор-столбец, а синдром — произведение $H \cdot r$:

$$H \cdot r = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Каждая строка результата — сумма соответствующих битов r по модулю 2, то есть ровно тот XOR, что считают три контрольные группы:

$$s_1 = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

$$s_2 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$s_4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

Синдром $(s_1, s_2, s_4) = (1, 1, 1)$ — это столбец 7 матрицы H , двоичная запись числа 7. Столбцы H — двоичные номера позиций, поэтому синдром численно совпадает с номером искажённого бита. Матричное умножение и три XOR-проверки дают один и тот же ответ.

Проверочная матрица отвечает на вопрос, является ли слово кодовым. Порождающая решает обратную задачу: по сообщению строит кодовое слово.

ОПРЕДЕЛЕНИЕ 13.5 Порождающая матрица

Порождающей матрицей (n, k) -линейного кода называется матрица G размера $k \times n$, если её строки образуют базис кода. Кодирование сообщения $m \in \text{GF}(2)^k$ задаётся умножением $c = mG$.

Каждая строка G — кодовое слово, поэтому она ортогональна каждой строке H , то есть $HG^T = 0$.

Пример Порождающая матрица кода Хэмминга $(7, 4)$

Три формулы проверочных битов собираются в одну матрицу:

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Умножение $c = mG$ при $m = (d_1, d_2, d_3, d_4)$ даёт $c = (p_1, p_2, d_1, p_4, d_2, d_3, d_4)$ — те же три XOR-выражения контрольных групп, что и в примере кодирования выше. Для сообщения $(1, 0, 1, 1)$ сложение первой, третьей и четвёртой строк даёт $(0, 1, 1, 0, 0, 1, 1)$.

ОПРЕДЕЛЕНИЕ 13.6 Вес слова (*Hamming weight*)

Весом слова x называется число его ненулевых позиций, обозначается $\text{wt}(x)$.

ТЕОРЕМА 13.2 Кодовое расстояние линейного кода

Кодовое расстояние линейного кода равно минимальному весу его ненулевого слова:

$$d = \min(\{\text{wt}(c) \mid c \in C, c \neq 0\}).$$

Набросок доказательства

Докажем двумя неравенствами.

Пусть c — ненулевое слово наименьшего веса. Нулевое слово принадлежит линейному коду, и пара $c, 0$ даёт $d \leq d(c, 0) = \text{wt}(c)$.

Обратно, для любых различных x и y из C сумма $x + y$ — ненулевое кодовое слово, и её вес равен числу позиций, в которых x и y различаются: $\text{wt}(x + y) = d(x, y)$. Каждый такой вес не меньше минимального веса ненулевого слова, поэтому и минимум по парам d не меньше его.

Теорема сокращает перебор: для кода Хэмминга $(7, 4)$ расстояние ищется среди 15 ненулевых слов, а не среди $\binom{16}{2} = 120$ пар. Заодно объясняется расстояние кода-повторения $(3, 1, 3)$: его ненулевое слово 111 имеет вес 3.

ОПРЕДЕЛЕНИЕ 13.7 Скорость кода

Скоростью кода $R = \frac{k}{n}$ называется доля информационных битов.

ОПРЕДЕЛЕНИЕ 13.8 Избыточность кода

Избыточностью $(1 - R)$ называется доля служебных битов.

Пример Скорости различных кодов

- Код Хэмминга $(7, 4)$: скорость $R = \frac{4}{7} \approx 0.57$. Избыточность: $\frac{3}{7} \approx 0.43$. Три проверочных бита дают коду способность исправить одну ошибку в семи битах.
- Код-повторение $(3, 1)$: скорость $R = \frac{1}{3} \approx 0.33$. Избыточность: $\frac{2}{3} \approx 0.67$ — две трети канала заняты повторами.
- Код Рида–Соломона $RS(255, 239)$: скорость $R = \frac{239}{255} \approx 0.937$. Избыточность всего 6.3% — исправляет до 8 символьных ошибок.

§ 13.5 Конечные поля $GF(p^m)$

Линейные коды над $GF(2)$ исправляют битовые ошибки. Однако ошибки часто приходят пачками — байтами, а не отдельными битами. Для этого нужно поле, у которого $2^8 = 256$ элементов. Но 256 — число не простое. А поля над $\mathbb{Z}_p$ устроены так, что в них ровно p элементов, где p — простое. Значит, пришлось бы довольствоваться полями $\mathbb{Z}_2 = \{0, 1\}$, $\mathbb{Z}_3 = \{0, 1, 2\}$ и так далее — и не было бы поля с 256 элементами.

Так возникает вопрос о том, бывают ли поля с непростым числом элементов. Ответ Галуа утвердителен, и число элементов всегда — степень простого. Поле $GF(256)$ существует, потому что $256 = 2^8$, и строится оно над $\mathbb{Z}_2$ как расширение степени 8. Для кодов Рида–Соломона, работающих с байтовыми символами, нужно поле $GF(q)$ с $q = p^m$.

УТВЕРЖДЕНИЕ 13.3 Классификация конечных полей

Порядок любого конечного поля равен степени простого числа. Для любой степени простого числа p^m существует поле с таким числом элементов. Все такие поля изоморфны.

Степени простого необходимы и достаточны — это теорема: других конечных полей, кроме $GF(p^m)$, не существует.

Обозначение $\mathbb{Z}_p$ — это поле вычетов по модулю p , то есть числа $0, 1, \dots, p-1$ со сложением и умножением по модулю p . Для $p = 2$ получается $\mathbb{Z}_2 = \{0, 1\}$, где $1 + 1 = 0$. Простое поле $\mathbb{Z}_p$ — самое маленькое поле, и p задаёт его размер.

Поле $\text{GF}(p^m)$ строится как множество многочленов над простым полем $\mathbb{Z}_p$. Берутся многочлены степени меньше m :

$$\text{GF}(p^m) = \{a_0 + a_1x + \dots + a_{m-1}x^{m-1} \mid a_i \in \mathbb{Z}_p\}.$$

Многочлены выбраны потому, что путь повторяет построение $\mathbb{Z}_p$ из целых чисел — берётся знакомый объект и вводится новое отношение, которое порождает больше элементов. Целые числа $\mathbb{Z}$ факторизуются по модулю p — получается $\mathbb{Z}_p$ из p элементов. Многочлены над $\mathbb{Z}_p$ факторизуются по неприводимому $f(x)$ степени m — получается поле из p^m элементов. Степень многочлена ограничена сверху числом m , поэтому элементов ровно p^m .

Сложение — по коэффициентному по модулю p . Умножение — как обычное умножение многочленов, но результат берётся по модулю неприводимого многочлена $f(x)$. Степень f равна m . Неприводимость означает, что f не раскладывается в произведение многочленов меньшей степени над $\mathbb{Z}_p$.

Неприводимость нужна ровно для того, чтобы у каждого ненулевого многочлена степени меньше m существовал обратный. Такой многочлен взаимно прост с f , и его обратный строится расширенным алгоритмом Евклида.

Построение поля повторяет путь от целых чисел к $\mathbb{Z}_p$. Многочлен $f(x)$ объявляется равным нулю, и это равенство даёт правило упрощения произведений. Старшие степени x сводятся к младшим. В $\mathbb{Z}_p$ число p объявляется равным нулю, поэтому остатки лежат в диапазоне $0, 1, \dots, p-1$. Здесь роль p играет $f(x)$. Каждый многочлен приводится к остатку степени меньше m . Запись

$$\text{GF}(p^m) = \text{GF}(p) \frac{[x]}{f(x)}$$

$$\text{как } \mathbb{Z}_p = \frac{\mathbb{Z}}{p}$$

читается как «поле, где многочлены делятся на $f(x)$ и берётся остаток» — так же, как $\mathbb{Z}_p$ — целые числа, взятые по модулю p .

Пример Поле $\text{GF}(4)$

Возьмём $p = 2, m = 2$. Неприводимый многочлен: $f(x) = x^2 + x + 1$. Проверим: $f(0) = 1, f(1) = 1$ — корней в поле $\text{GF}(2)$ нет, значит неприводим.

Элементы: $\{0, 1, x, x+1\}$. Это все многочлены степени ≤ 1 над $\text{GF}(2)$. Обозначим $\alpha = x$. Тогда $\alpha^2 = \alpha + 1$. Ведь $\alpha^2 + \alpha + 1 = 0$, и так поле определяет умножение.

Таблица сложения (XOR коэффициентов):

+	0	1	α	$\alpha + 1$
0	0	1	α	$\alpha + 1$
1	1	0	$\alpha + 1$	α
α	α	$\alpha + 1$	0	1
$\alpha + 1$	$\alpha + 1$	α	1	0

Таблица 7. Сложение в $GF(4)$.

Таблица умножения ($\alpha^2 = \alpha + 1$):

$\times$	0	1	α	$\alpha + 1$
0	0	0	0	0
1	0	1	α	$\alpha + 1$
α	0	α	$\alpha + 1$	1
$\alpha + 1$	0	$\alpha + 1$	1	α

Таблица 8. Умножение в $GF(4)$.

Мультипликативная группа $GF(4)^* = \{1, \alpha, \alpha + 1\}$ циклическая:

- $\alpha^1 = \alpha$
- $\alpha^2 = \alpha + 1$
- $\alpha^3 = 1$

α — примитивный элемент, и его степени порождают все ненулевые элементы поля.

Для любого конечного поля $GF(q)$ мультипликативная группа $GF(q)^*$ циклическая. Примитивный элемент — образующий этой группы. Именно на этом свойстве держится конструкция кодов Рида–Соломона, где кодовое слово — это значения многочлена в степенях примитивного элемента.

Замечание Сколько полей в $GF(256)$ и какой выбирают?

AES-полином неприводим, поэтому $GF(256)$ — корректное поле, но его корень α не примитивен, ведь порядок α равен 51, а не 255. Примитивный элемент в этом поле — например, элемент с номером 3, и его степени порождают все 255 ненулевых элементов. Преимущество AES-полинома — удобная редукция в аппаратуре, ведь умножение сводится к сдвигу и условному XOR, без деления многочленов. В кодах Рида–Соломона используют то же поле, но конкретная таблица умножения зависит от выбранного неприводимого многочлена.

§ 13.6 Граница Шеннона

Защита оплачивается избыточностью, и у этой цены есть нижняя граница. Шеннон поставил вопрос о максимальной скорости, с которой можно передавать информа-

цию через канал с шумом, сохраняя вероятность ошибки сколь угодно малой. Ответ даёт теорема Шеннона для дискретного канала с шумом.

Пусть канал передаёт биты и с вероятностью p переворачивает бит независимо от остальных (двоичный симметричный канал).

ОПРЕДЕЛЕНИЕ 13.9 Пропускная способность канала

Пропускной способностью двоичного симметричного канала с вероятностью ошибки p называется величина

$$C = 1 - H(p),$$

где $H(p) = -p \log_2 p - (1 - p) \log_2 (1 - p)$ — энтропия бинарного распределения.

Пример Значения пропускной способности

Канал без ошибок: $p = 0$, $H(0) = 0$, $C = 1$ — каждый бит канала несёт ровно бит информации. Канал полностью зашумлён: $p = 0.5$, $H(0.5) = 1$, $C = 0$ — канал бесполезен, ведь принятый бит не зависит от переданного. При $p = 1$ все биты переворачиваются, и достаточно инвертировать выход, чтобы канал снова стал идеальным, то есть $C = 1$, как и при $p = 0$.

Пропускная способность симметрична относительно p и $1 - p$ и убывает к нулю при приближении к $p = 0.5$:

p	$C = 1 - H(p)$	Что это значит
0	1	канал без ошибок
0.01	0.92	почти идеальный
0.1	0.53	умеренный шум
0.2	0.28	шум заметен
0.3	0.12	большая часть — шум
0.5	0	канал бесполезен
1	1	все биты перевёрнуты — инвертируем выход

ТЕОРЕМА 13.4 Теорема Шеннона о кодировании для канала с шумом (1948)¹¹¹

Для любой скорости $R < C$ существует последовательность кодов с длиной $n \rightarrow \infty$, скоростью R и вероятностью ошибки, стремящейся к нулю. Для любой скорости $R > C$ надёжная передача невозможна.

¹¹¹C. E. Shannon. «A Mathematical Theory of Communication». Bell System Technical Journal, 1948. Вторая теорема Шеннона (о канале с шумом). Доказательство неконструктивно — существование кода устанавливается вероятностным методом.

Набросок доказательства

Докажем в две стороны.

Доказательство существования (прямое утверждение) использует вероятностный метод. Рассматривается множество всех кодов длины n со скоростью R . Каждый код — это набор из $M = 2^{nR}$ кодовых слов, выбранных из всех возможных двоичных строк. Код выбирается случайно — каждое кодовое слово выбирается независимо и равновероятно из всех 2^n двоичных строк.

Для фиксированного кодового слова вычисляется вероятность того, что при передаче через канал с шумом p оно будет декодировано неверно. Затем математическое ожидание средней вероятности ошибки по случайному выбору кода оценивается сверху. При $R < C = 1 - H(p)$ эта оценка экспоненциально убывает с ростом n . Из малости математического ожидания следует, что существует хотя бы один код с вероятностью ошибки не выше среднего. Доля кодов с большой вероятностью ошибки пренебрежимо мала — почти все случайные коды хороши.

Обратное утверждение ($R > C$ влечёт невозможность надёжной передачи) доказывается через неравенство Фано, по которому скорость не может превышать пропускную способность, не жертвуя надёжностью.

Из доказательства не извлечь сам код, ведь оно неконструктивно. Множество всех кодов имеет размер двойной экспоненты 2^{q^n} для алфавита размера q , поэтому перебор невозможен. Построение явных кодов, приближающихся к границе Шеннона и допускающих эффективное декодирование, — долгое время открытая инженерная задача. В 2009 году Арикан построил полярные коды, достигающие пропускной способности двоичного симметричного канала с декодированием за $O(n \log n)$.

Теорема Шеннона меняет статус инженерной задачи. До неё качество связи выглядело свойством мастерства: чем искуснее код, тем надёжнее передача, и неизвестно, где кончается это мастерство. Теперь задача разделена на закон и инженерию: пропускная способность зависит только от вероятности ошибки в канале, а искусство измеряется тем, насколько код близок к границе. Термодинамика сделала тот же ход: коэффициент полезного действия тепловой машины ограничен температурами нагревателя и холодильника, а не искусством механика. Инженерия начинается там, где предел уже известен: остаётся приближаться к нему и измерять, какой ценой.

При этом теорема говорит о кодах как об объектах — множествах слов в пространстве строк — и ничего не говорит о способах их построения. Прямая часть гарантирует существование хорошего кода, обратная запрещает скорости выше C , и обе обходятся без алгоритмов.

Замечание Неконструктивность как двигатель теории

Доказательство через усреднение устанавливает, что объект существует, но не даёт его координат. Именно эта неконструктивность породила исследовательскую программу на семь десятилетий — построить семейства кодов, которые одновременно приближаются к границе Шеннона и допускают эффективное декодирование. Коды Рида–Соломона, свёрточные коды, турбо-коды, LDPC-коды, полярные коды — каждое поколение подбиралось к границе с новой стороны. Будь доказательство конструктивным, инженерам осталось бы только реализовать предписанное, и теория кодирования была бы закрытой главой.

§ 13.7 Код Хаффмана (сжатие без потерь)

Код Хэмминга защищает от ошибок, добавляя избыточность. Код Хаффмана решает обратную задачу — *убрать* избыточность, сжать данные до минимально возможного размера без потери информации.

Сопоставим часто встречающимся символам короткие кодовые слова, а редким — длинные. Азбука Морзе делает то же самое — буква Е (самая частая в английском) кодируется одной точкой, а Q — тире-тире-точка-тире.

ОПРЕДЕЛЕНИЕ 13.10 Префиксный код

Код называется **префиксным**, если ни одно кодовое слово не является префиксом другого.

Префиксные коды декодируются однозначно и без задержки — читаем биты, пока не получим кодовое слово, и сразу знаем, что это конец символа. Алгоритм Хаффмана строит оптимальный префиксный код для заданных частот символов. Средняя длина кода складывается из частот, умноженных на длины слов: $L = \sum_i f_i l_i$.

ТЕОРЕМА 13.5 Оптимальность кода Хаффмана

Для любых частот символов код, построенный алгоритмом Хаффмана, имеет минимальную среднюю длину среди всех префиксных кодов.

Набросок доказательства

Докажем индукцией по числу символов, опираясь на обменный аргумент.

Сначала покажем, что среди оптимальных деревьев есть такое, где два редчайших символа a и b — соседние листья на максимальной глубине. Возьмём в оптимальном дереве пару соседних листьев x и y наибольшей глубины. Если редчайшие символы стоят выше, обменяем местами x с a и y с b : вклад $f_x l_x + f_a l_a$ заменится на $f_a l_x + f_x l_a$ и не возрастает, поскольку $f_a \leq f_x$ и $l_a \leq l_x$. Для

пары y и b вычисление то же, поэтому обмен сохраняет оптимальность и ставит редчайшие символы в соседнюю пару на максимальной глубине.

Теперь сольём a и b в один символ веса $f_a + f_b$ — это первый шаг алгоритма. Слияние уменьшает среднюю длину ровно на $f_a + f_b$, и по предположению индукции алгоритм строит оптимальный код для сокращённого алфавита из $n - 1$ символа. Обратное разворачивание прибавляет те же $f_a + f_b$ к любому префиксному коду, поэтому оптимум для сокращённого алфавита даёт оптимум для исходных n символов.

Пример Код Хаффмана для пяти символов

Символы и частоты: $A = 0.40, B = 0.25, C = 0.20, D = 0.10, E = 0.05$. Алгоритм «жадный», строит дерево снизу вверх.

1. Создаём лес из пяти деревьев, каждое — один символ с весом f_i .
2. Выбираем два дерева с наименьшими весами: $D = 0.10, E = 0.05$, объединяем в узел DE с весом 0.15.
3. Следующие два наименьших: $C = 0.20, DE = 0.15$, объединяем в узел CDE с весом 0.35.
4. Затем $B = 0.25, CDE = 0.35$: узел BCDE с весом 0.60.
5. Наконец $A = 0.40, BCDE = 0.60$: корень с весом 1.00.
6. Проходим от корня к листьям: левый переход — 0, правый — 1.

Кодовые слова (от корня): $A = 0, B = 10, C = 110, D = 1110, E = 1111$.

Средняя длина: $0.40 \cdot 1 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.10 \cdot 4 + 0.05 \cdot 4 = 2.10$ битов на символ. Для сравнения, равномерный код для 5 символов требует $\lceil \log_2 5 \rceil = 3$ битов на символ.

Пример Декодирование по префиксу

Слова из примера выше: $A = 0, B = 10, C = 110, D = 1110, E = 1111$. Последовательность 1101111 читается слева направо без возвратов: 110 — это C , сразу за ним 1111 — это E . Другого разбиения нет: после 11 код ещё не завершён (ни A , ни B не начинаются с двух единиц), а 110 — уже полное слово. Так префиксность устраняет неоднозначность: конец слова виден в момент его прочтения.

Построенное дерево показано на рис. 71 — каждый путь от корня к листу читается как кодовое слово, а метки 0 и 1 на рёбрах — биты этого слова.

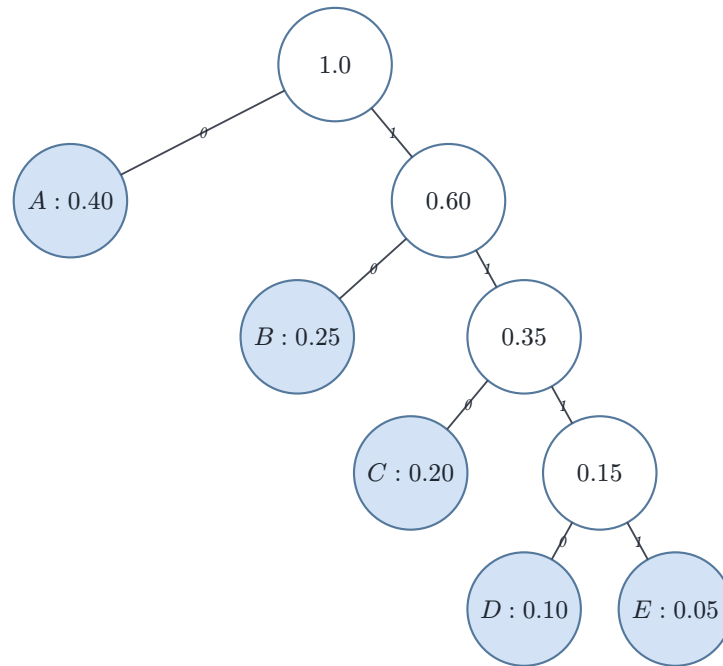


Рис. 71. Дерево Хаффмана для пяти символов.

Замечание Код Хаффмана не единственный

Первая свобода — выбор, какому из двух сливаемых поддеревьев достаётся 0, а какому 1. Слияние D и E можно закодировать как $D = 0, E = 1$ или $D = 1, E = 0$: средняя длина останется прежней, а сами коды станут другими. Именно поэтому у разных реализаций один и тот же файл может сжиматься слегка разными кодами.

Вторая свобода — сама структура дерева, и появляется она только при *равных весах*. Если два поддерева имеют одинаковую частоту, их можно слить в любом порядке, и оба дерева будут оптимальными. При частотах $A = 0.40, B = 0.25, C = 0.20, D = 0.10, E = 0.05$ все веса различны, поэтому структура кода определена однозначно — меняется лишь расстановка 0 и 1.

Код Хаффмана используется в JPEG, MP3, DEFLATE (zip/gzip) и многих других форматах. В сочетании с кодами, исправляющими ошибки, возникает двуслойная схема — сжать данные кодом Хаффмана, затем защитить кодом Хэмминга (или более мощным кодом).

Проверка Код Хаффмана

1. Почему код Хаффмана обязан быть префиксным и что сломается при декодировании, если одно слово станет префиксом другого?
2. Частому символу достался короткий код, редкому — длинный: $A = 0, E = 1111$. Почему такая асимметрия снижает среднюю длину по сравнению с равномерным кодом?

§ 13.8 * Словарное сжатие: LZ77 и LZ78

Код Хаффмана видит только частоты: популярным символам достаются короткие слова, редким — длинные, и на этом его кругозор кончается. Главный же резерв реальных данных лежит в повторениях. Частотный код помнит, сколько раз буква встретилась, и не помнит, где она стояла. В слове «ABRACADABRA» кусок «ABRA» проходит дважды, в программе имена переменных повторяются сотни раз, в тексте — целые обороты. Платить за второй экземпляр «ABRA» полновесными битами — всё равно что второй раз покупать ту же книгу. Словарное сжатие заменяет повторный кусок ссылкой на то место, где этот кусок уже проходил: канал передаёт адрес, а не копию.

Якоб Зив и Абрахам Лемпель опубликовали в 1977 году схему, доводящую эту идею до машины.¹¹² Кодер держит *скользящее окно* — уже закодированный хвост входа, внутри которого разрешено искать совпадения. Окно скользит за курсором: отработанное уходит из вида, свежее поступает в обращение. Очередной кусок входа кодер замещает тройкой — ссылкой в окно плюс одним новым символом.

ОПРЕДЕЛЕНИЕ 13.11 Тройка LZ77

Тройкой LZ77 называется запись (o, l, a) , где o — смещение от текущей позиции назад до начала совпадения с уже закодированной частью, l — длина совпадения, a — первый символ, следующий за совпадением. Запись $(0, 0, a)$ означает, что совпадения не было и символ a передаётся буквально.

Кодер работает жадно: на каждом шаге он ищет самое длинное совпадение окна с входом. Свежий символ не обязан быть новым для окна: в этой позиции совпадение продлить не удалось, хотя сам символ может встречаться в окне в других местах. Декодер устроен ещё проще: тройка говорит «вернись на o символов назад и переписи l штук», затем к выходу добавляется свежий символ — декодер копирует байты из собственного выхода. Длина совпадения может превосходить смещение: тройка $(1, 9, x)$ означает девять копий одного байта, и декодер успевает дописывать хвост одновременно с чтением. Так серия из девяти одинаковых байтов ужимается в три числа. Окно — это память машины о недавнем: чем оно шире, тем дальше кодер видит повторения, и тем дороже поиск по нему.

Пример LZ77 для «ABRACADABRA»

Пусть окном служит весь уже закодированный префикс, а совпадение выбирается самое длинное. Одиннадцать букв сворачиваются в шесть троек. Если совпадение доходит до конца входа, тройка остаётся без свежего символа.

¹¹²J. Ziv, A. Lempel. «A Universal Algorithm for Sequential Data Compression». IEEE Transactions on Information Theory, 1977.

Тройка	Совпало	Свежий символ
$(0, 0, A)$	—	A
$(0, 0, B)$	—	B
$(0, 0, R)$	—	R
$(3, 1, C)$	A	C
$(2, 1, D)$	A	D
$(7, 4)$	ABRA	конец входа

Финальная тройка — главная добыча: в семи символах назад лежит точная копия всего хвоста, и четыре буквы кодируются парой чисел. Декодер идёт по тройкам слева направо и строит строку из собственного выхода: к тройке $(7, 4)$ у него записаны первые семь букв, и он переписывает четыре из их начала. Арифметика при этом честно показывает цену механики: при фиксированной разрядности — 3 бита на смещение, 3 на длину, 8 на символ — пять троек и финальная пара занимают $5 \cdot 14 + 6 = 76$ бит против $11 \cdot 8 = 88$ бит исходной строки. На коротких входах выигрыш призрачен — двенадцать бит со всей строки. Зато механика видна как на ладони: тройке всё равно, что кодировать — буквы, байты архива или машинное слово. Настоящая экономия приходит на длинных данных, где совпадения в десятки символов многократно окупают пару чисел.

Через год тот же дуэт предложил вариант с растущим словарём фраз — LZ78.¹¹³ Окно вечно уносит отработанное на свалку, а словарь копит: выход кодируется парой из номера уже выученной фразы и нового символа, и эта же пара немедленно зачисляется в словарь новой фразой. Номер 0 зарезервирован за пустой фразой, с которой словарь стартует. Для той же строки LZ78 выдаёт семь пар: $(0, A)$, $(0, B)$, $(0, R)$, $(1, C)$, $(1, D)$, $(1, B)$, $(3, A)$. Словарь после кодирования содержит семь фраз: A , B , R , AC , AD , AB , RA , и декодер восстанавливает его синхронно с кодером, ведь каждая пара добавляет ровно одну фразу. На коротком слове LZ78 не успел выучить куски вроде ABRA, которые LZ77 ловит целиком. На длинных входах словарь накапливает именно такие часто повторяющиеся куски. Практические реализации ограничивают словарь и время от времени строят его заново — иначе память росла бы вместе с входом.

Пора назвать меру. Хаффман оптимален для источника с заданной таблицей частот, но у настоящего текста таблицы нет — есть источник, порождающий символы с вероятностями, возможно зависящими от предыстории. Мерой служит энтропия источника H : среднее число бит на символ, ниже которого код без потерь опуститься не может. Сжать данные — значит приблизить длину кода на символ к H , и весь вопрос в том, насколько близко можно подойти, не зная вероятностей заранее.

¹¹³J. Ziv, A. Lempel. «Compression of Individual Sequences via Variable-Rate Coding». IEEE Transactions on Information Theory, 1978.

ТЕОРЕМА 13.6 Асимптотическая оптимальность LZ78

Пусть источник стационарен — его вероятностные правила не меняются со временем — и эргодичен — по одной длинной реализации восстанавливаются все его средние характеристики. Тогда длина кода LZ78 в битах на закодированный символ сходится к энтропии источника H . Тот же предел достигает LZ77, когда окно растёт вместе с входом.

Кодер нигде не получает частоты заранее: словарь собирается по ходу кодирования, поэтому такие коды называются *универсальными*. Гарантия асимптотическая: размер словаря, скорость кодирования, работа с типичными файлами остаются инженерией поверх неё, но сам предел задан этой теоремой.

Теперь у избыточности названы оба источника — неравные частоты и повторения — и у каждого свой инструмент. DEFLATE — формат, на котором стоят gzip, zip и PNG, — прогоняет данные через LZ77 с окном $2^{15} = 32768$ байт, а затем кодирует Хаффманом получившийся поток литералов и ссылок.¹¹⁴ Спецификация разрешает совпадения длиной от 3 до 258 символов и доверяет Хаффману две таблицы: одну для литералов и длин, другую для смещений. Хаффману после LZ77 есть что ужимать: типичный текст чаще ссылается на недавние куски, чем на далёкие, и символы этого потока отнюдь не равновероятны. Так закрывается линия главы: защита добавляет избыточность по расчёту, сжатие убирает её без потерь. Частотная избыточность достаётся коду Хаффмана, повторяемая — словарю LZ, а в сочетании с кодами из основного текста выстраивается вся башня: LZ выучивает повторения, Хаффман ужимает остаток, код Хэмминга добавляет расчётную избыточность.

§ 13.9 Циклические коды

Циклический код — линейный код, замкнутый относительно циклического сдвига:

$$(c_0, \dots, c_{n-1}) \rightarrow (c_{n-1}, c_0, \dots, c_{n-2}).$$

Это свойство радикально упрощает реализацию, ведь сдвиговые регистры с линейной обратной связью (LFSR) сами считают и кодовое слово, и синдром — минимум вентилей, никаких матричных умножений.

Циклические коды описываются многочленами над $\text{GF}(2)$. Кодовое слово представляется многочленом $c(x) = c_0 + c_1x + \dots + c_{n-1}x^{n-1}$. Циклический сдвиг на одну позицию — это умножение на $x \bmod (x^n - 1)$. Старший коэффициент «заворачивается» в младший разряд.

Код замкнут относительно таких сдвигов, поэтому замкнут относительно умножения на x , а значит — относительно умножения на любой многочлен. Отсюда

¹¹⁴P. Deutsch. «DEFLATE Compressed Data Format Specification version 1.3». RFC 1951, 1996.

множество кодовых слов порождено одним многочленом $g(x)$. Кодовые слова — в точности многочлены, кратные $g(x)$.

Покажем, что одного многочлена достаточно. Выберем в коде ненулевой многочлен $g(x)$ наименьшей степени. Разделим любое кодовое слово $c(x)$ на $g(x)$ с остатком, то есть $c = qg + r$, где степень r меньше степени g . Остаток — разность двух кодовых слов, ведь c слово по условию, а qg — кратное g и тоже слово, поскольку код замкнут относительно умножения на многочлены и относительно сложения. Если бы r был ненулевым, он был бы кодовым словом степени меньше, чем у g — вопреки выбору g . Значит, $r = 0$, и каждое слово делится на g нацело, так что один многочлен порождает весь код.

Замкнутость кода требует, чтобы и циклические сдвиги самого $g(x)$ оставались кратными $g(x)$. Это выполняется для всех сдвигов ровно тогда, когда $g(x) \mid x^n - 1$.

ОПРЕДЕЛЕНИЕ 13.12 Порождающий многочлен

(n, k) -циклический код задаётся **порождающим многочленом**

$$g(x) = g_0 + g_1x + \dots + g_{n-k}x^{n-k},$$

где $g_0 = 1$, $g_{n-k} = 1$, и $g(x)$ делит $x^n - 1$ над двоичным полем.

Кодирование сообщения $m(x)$ степени меньше k даёт кодовое слово $c(x) = m(x) \cdot g(x)$. Альтернативно $c(x) = x^{n-k}m(x) - r(x)$, где $r(x) = x^{n-k}m(x) \bmod g(x)$ — остаток, помещаемый в проверочные позиции.

Пример Циклический код Хэмминга (7, 4)

Код Хэмминга (7, 4) эквивалентен циклическому коду с порождающим многочленом $g(x) = 1 + x + x^3$. Проверим делимость $g(x) \mid x^7 - 1$ над двоичным полем: $x^7 - 1 = (1 + x + x^3)(1 + x + x^2 + x^4)$ (проверяется умножением — удвоенные члены обнуляются). Сообщение $m = (1, 0, 1, 1)$. Ему соответствует многочлен $m(x) = 1 + 0 \cdot x + 1 \cdot x^2 + 1 \cdot x^3 = 1 + x^2 + x^3$. Кодовое слово: $c(x) = m(x) \cdot g(x) = (1 + x^2 + x^3)(1 + x + x^3) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 \pmod{2}$. Учтено: $x^i + x^i = 0$. Коэффициенты: $(1, 1, 1, 1, 1, 1, 1)$ — кодовое слово. Первые 4 бита — не исходное сообщение (используется несистематическая форма), но после декодирования восстановится.

Восстановление сообщения — деление на $g(x)$. Раз кодовое слово устроено как $c(x) = m(x) \cdot g(x)$, сообщение находится делением на $g(x)$. В примере выше $c(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6$. Деление даёт $c(x) \div (1 + x + x^3) = 1 + x^2 + x^3$. Это исходный многочлен сообщения. Деление выполняется столбиком над двоичным полем, где на каждом шаге старший член делимого гасится старшим членом делителя, а поскольку $x^i + x^i = 0$, промежуточные члены попарно сокращаются.

Циклические коды — основа реальных протоколов. CRC (Cyclic Redundancy Check) обнаруживает ошибки в Ethernet и файловых системах. Коды БЧХ (Боуза–Чоудхури–Хоквингема, 1960) и Рида–Соломона исправляют ошибки в CD/DVD, QR-кодах и спутниковой связи.

Пример CRC-32 — обнаружение ошибок в Ethernet

CRC-32 использует порождающий многочлен

$$g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1.$$

Отправитель вычисляет остаток $r(x)$ от деления данных на $g(x)$. Остаток добавляется в конец пакета. Получатель делит принятые данные на $g(x)$, и если остаток ненулевой, ошибка обнаружена. CRC-32 обнаруживает любую одиночную ошибку, любую серию подряд идущих испорченных битов длиной до 32 и любые двойные ошибки на пакетах практической длины. Применение: Ethernet (IEEE 802.3), PNG, gzip.

§ 13.10 Коды Рида–Соломона

Так код перестаёт замечать, сколько бит испорчено внутри символа, и считает только число испорченных символов. Именно это и нужно при царапине, ведь она портит длинный участок, но покрывает несколько символов целиком.

Значения многочлена дают исправление потому, что многочлен степени меньше k однозначно задаётся своими значениями в k точках, и это интерполяция. Если отправить его значения в $n > k$ точках, то для восстановления многочлена достаточно любых k из них. Остальные $n - k$ значений — избыточность, ведь они позволяют восстановить данные, даже если часть значений потеряна или искажена. Задача только в том, чтобы понять, какие значения повреждены, а какие целы.

ОПРЕДЕЛЕНИЕ 13.13 Код Рида–Соломона

Пусть $\text{GF}(q)$ — конечное поле с примитивным элементом α . Параметры: $n = q - 1$, $1 \leq k \leq n$. Сообщение $(m_0, \dots, m_{k-1})$ задаёт многочлен

$$m(x) = m_0 + m_1x + \dots + m_{k-1}x^{k-1}$$

степени меньше k .

Кодовое слово — значения этого многочлена в n последовательных степенях примитивного элемента:

$$c = (m(\alpha^0), m(\alpha^1), m(\alpha^2), \dots, m(\alpha^{n-1})).$$

Отправитель вычисляет значения $m(x)$ в n точках, получатель принимает их. Если бы не было ошибок, получатель взял бы любые k значений и восстановил $m(x)$

интерполяцией. При ошибках часть значений неверна — и тогда в дело вступает избыточность.

Кодовое расстояние $d = n - k + 1$ следует из подсчёта корней. Многочлен степени меньше k имеет не более $k - 1$ корней. Значит, два различных многочлена совпадают не более чем в $k - 1$ точках. Их кодовые слова различаются минимум в $n - (k - 1) = n - k + 1$ позициях. Так код достигает границы Синглтона и является MDS-кодом (Maximum Distance Separable).

Представьте, что мы пересылаем другу четыре числа, но один пакет теряется по дороге. По трём уцелевшим числам четвёртое не восстановить. Но можно отправить шесть значений одного многочлена степени 3. Многочлен степени 3 однозначно задаётся значениями в четырёх точках, поэтому друг восстановит его по любым четырём уцелевшим пакетам. Избыточность здесь — значения того же многочлена в дополнительных точках.

Примечание Потеря пакета — не то же, что искажение

Метафора с потерянным пакетом удобна, но прячет важное различие. Искажённый бит приходит на приём как есть, просто с неверным значением. Код знает позицию, но не знает, верно ли значение — восстановление идёт по избыточности. Потерянный пакет не приходит вовсе, и у получателя нет ни значения, ни уверенности в самой позиции. Если позиция известна (получатель знает, **какой** пакет пропал), задача легче и называется *стиранием*, и код восстанавливает даже больше, чем при искажениях. Если позиция неизвестна, задача — это вопрос доставки, а не исправление ошибок, и на этом уровне коды не работают. Ответ на него дают протоколы, а не алгебра: нумерация пакетов, подтверждения, повторная передача.

Пример RS над GF(8) — маленький, но показательный

$q = 8 = 2^3$ с примитивным элементом α . Здесь α — корень многочлена $x^3 + x + 1 = 0$, поэтому $\alpha^3 = \alpha + 1$. Параметры: $n = q - 1 = 7$, $k = 3$, $d = 7 - 3 + 1 = 5$.

Код $RS(7, 3, 5)_8$: три символа сообщения кодируются семью символами. Исправляет до 2 символьных ошибок. Каждый символ — 3 бита, так что две символьные ошибки — это до 6 испорченных бит.

Сообщение $(m_0, m_1, m_2) = (1, \alpha, \alpha^2)$. Многочлен: $m(x) = 1 + \alpha x + \alpha^2 x^2$.

Используем $\alpha^3 = \alpha + 1$, тогда $\alpha^4 = \alpha \cdot (\alpha + 1) = \alpha^2 + \alpha$, $\alpha^5 = \alpha \cdot (\alpha^2 + \alpha) = \alpha^2 + \alpha + 1$, $\alpha^6 = \alpha \cdot \alpha^5 = \alpha^2 + 1$. Подставляем:

$$\begin{aligned}
c_0 &= m(1) = 1 + \alpha + \alpha^2 = \alpha^5, \\
c_1 &= m(\alpha) = 1 + \alpha^2 + \alpha^4 = 1 + \alpha^2 + (\alpha^2 + \alpha) = \alpha^3, \\
c_2 &= m(\alpha^2) = 1 + \alpha^3 + \alpha^6 = 1 + (\alpha + 1) + (\alpha^2 + 1) = \alpha^5, \\
c_3 &= m(\alpha^3) = 1 + \alpha^4 + \alpha^8 = 1 + (\alpha^2 + \alpha) + \alpha = \alpha^6, \\
c_4 &= m(\alpha^4) = 1 + \alpha^5 + \alpha^{10} = 1 + (\alpha^2 + \alpha + 1) + (\alpha + 1) = \alpha^6, \\
c_5 &= m(\alpha^5) = 1 + \alpha^6 + \alpha^{12} = 1 + (\alpha^2 + 1) + (\alpha^2 + \alpha + 1) = \alpha^3, \\
c_6 &= m(\alpha^6) = 1 + \alpha^7 + \alpha^{14} = 1 + 1 + 1 = 1.
\end{aligned}$$

Полное кодовое слово:

$$c = (\alpha^5, \alpha^3, \alpha^5, \alpha^6, \alpha^6, \alpha^3, 1).$$

Избыточность позволяет восстановить данные, даже если часть значений искажена. Верно и обратное — *любой* многочлен степени меньше k , согласующийся с принятым словом хотя бы в k позициях, правильный, если ошибок было не больше $\frac{n-k}{2}$. Проверим это на нашем примере — внесём две ошибки и восстановим сообщение.

Пример Декодирование RS(7, 3, 5) по интерполяции

Передано $c = (\alpha^5, \alpha^3, \alpha^5, \alpha^6, \alpha^6, \alpha^3, 1)$. Ошибки в позициях 1 и 5: в позиции 1 значение α^3 заменено на 0, в позиции 5 — α^3 на 1. Принято:

$$r = (\alpha^5, 0, \alpha^5, \alpha^6, \alpha^6, 1, 1).$$

Будем перебирать тройки позиций и восстанавливать многочлен степени меньше 3 интерполяцией по трём значениям. Возьмём позиции 0, 2 и 3 — они не повреждены. Интерполяция по (α^0, α^5) , (α^2, α^5) , (α^3, α^6) даёт многочлен с коэффициентами $(1, \alpha, \alpha^2)$:

$$m(x) = 1 + \alpha x + \alpha^2 x^2.$$

Проверим, что m согласуется с r во всех пяти неповреждённых позициях. Позиция 1 (значение 0) отличается от $m(\alpha) = \alpha^3$ — там ошибка. Позиция 5 (значение 1) отличается от $m(\alpha^5) = \alpha^3$ — тоже ошибка. Остальные значения совпадают.

Значит, найденный многочлен — правильный, а сообщение — $(1, \alpha, \alpha^2)$. Две ошибки исправлены.

Если ошибок не больше $t = \frac{n-k}{2}$, то принятое слово согласуется с правильным многочленом минимум в $n - t$ позициях, а это больше k . Правильный многочлен — единственный степени меньше k , согласующийся с r хотя бы в $k + t$ позициях. Любой другой кандидат отступил бы от r в большем числе позиций. Поэтому перебор троек гарантированно находит один и тот же многочлен, пока ошибок не больше двух.

На практике перебор по всем $\binom{n}{k}$ наборам слишком дорог, ведь для больших n их миллионы. Алгоритм Берлекэмп–Мэсси (1968) находит тот же многочлен, но за $O(n^2)$, не перебирая наборы. Он строит многочлен локаторов ошибок по синдромам, как решает линейную рекурренту (глава 21). Корни локаторов указывают позиции ошибок, а значения ошибок восстанавливаются из тех же синдромов. Идея та же, что в нашем примере, — интерполяция по неповреждённым значениям. Меняется только способ поиска.

Примечание RS-коды на практике

1. **QR-коды:** RS над $GF(256)$ с разными уровнями коррекции (L, M, Q, H) — от восстановления 7% до 30% повреждённой площади.
2. **CD/DVD/Blu-ray:** каскадный код (*concatenated code*) CIRC (Cross-Interleaved Reed-Solomon Code) — два слоя RS с перемежением, защита от царапин и пыли.
3. **Спутниковая связь (DVB):** $RS(204, 188, 17)_{256}$ — 188 байт данных + 16 проверочных, исправляет до 8 байтовых ошибок.
4. **RAID 6:** RS над $GF(256)$ для восстановления данных при отказе двух дисков одновременно.

§ 13.11 Коды и комбинаторика

Кодовые слова — точки двоичного куба $\{0, 1\}^n$. Исправление ошибок сводится к комбинаторному вопросу о том, сколько непересекающихся шаров радиуса t можно упаковать в n -мерный куб. Ответ даёт граница Хэмминга:

$$|C| \cdot \sum_{i=0}^t \binom{n}{i} \leq 2^n.$$

Сумма — объём шара радиуса t , то есть число строк на расстоянии не больше t от центра. Умноженный на число кодовых слов, он не может превышать общее число строк.

УТВЕРЖДЕНИЕ 13.7 Граница Хэмминга

$$|C| \cdot \sum_{i=0}^t \binom{n}{i} \leq 2^n.$$

Набросок доказательства

Докажем подсчётом.

У кода, исправляющего t ошибок, шары радиуса t вокруг кодовых слов не пересекаются. Объём шара радиуса t в двоичном кубе $\{0, 1\}^n$ — это все строки на расстоянии не больше t от центра, и их число равно $\sum_{i=0}^t \binom{n}{i}$.

Сумма объёмов всех шаров не превышает объём всего пространства 2^n , поэтому $|C| \cdot \sum_{i=0}^t \binom{n}{i} \leq 2^n$.

Код, достигающий границы Хэмминга, называется *совершенным*. Код Хэмминга $(7, 4)$ — совершенный, ведь шары радиуса 1 вокруг 16 кодовых слов покрывают всё пространство $\{0, 1\}^7$ без зазоров.

УТВЕРЖДЕНИЕ 13.8 Граница Синглтона

$$d \leq n - k + 1.$$

Набросок доказательства

Докажем, показав, что после удаления $d - 1$ координат кодовые слова остаются попарно различными.

Удалим первые $d - 1$ координат каждого кодового слова. Два различных кодовых слова не могут совпасть в оставшихся $n - d + 1$ координатах. Иначе они различались бы менее чем в d позициях, что противоречит кодовому расстоянию d .

Следовательно, все $|C|$ слов различны в этих $n - d + 1$ позициях. В двоичном алфавите $|C| \leq 2^{n-d+1}$. Для линейного кода $|C| = 2^k$. Отсюда $2^k \leq 2^{n-d+1}$ и $d \leq n - k + 1$.

Самые известные примеры кодов, достигающих границы Синглтона (MDS-коды), — коды Рида–Соломона. Они оптимальны по расстоянию, и при фиксированных n и k ни один код не может иметь большее расстояние.

ТЕОРЕМА 13.9 Граница Варшавова–Гилберта

Существует код длины n над алфавитом из q символов с кодовым расстоянием d . Число кодовых слов не меньше $\frac{q^n}{\sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i}$. Гарантия существования — мощностной аргумент, как и у Шеннона.

Набросок доказательства

Докажем жадным построением.

Выбираем кодовые слова по одному, и каждое новое слово должно отстоять от всех уже выбранных на расстояние не меньше d . На алфавите из q символов выбранное слово запрещает не более $V = \sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i$ слов, то есть все строки в шаре радиуса $d - 1$ вокруг него.

Пока выбрано меньше $\frac{q^n}{V}$ слов, общее число запрещённых слов меньше q^n , и остаётся хотя бы одна незапрещённая строка.

Значит, жадный процесс не застревает и строит код длины n с расстоянием d и числом кодовых слов не меньше $\frac{q^n}{V}$.

Пример Граница Варшамова–Гилберта в числах

Пусть $n = 7, q = 2, d = 3$. Шар радиуса $d - 1 = 2$ содержит $V = 1 + 7 + 21 = 29$ строк. Граница обещает код с числом слов не меньше $\frac{128}{29} \approx 4.4$, то есть не менее пяти. Жадный процесс это подтверждает, ведь пока выбрано не больше четырёх слов, запрещено не более $4 \cdot 29 = 116$ строк из 128, и свободная строка остаётся — пятое слово добавить можно.

Примечание Коды и комбинаторные виды

Перечисление линейных кодов заданной длины и размерности над $\text{GF}(q)$ — это задача комбинаторных видов. Количество k -мерных подпространств в n -мерном пространстве над $\text{GF}(q)$ даётся q -биномиальным коэффициентом (гауссовым биномом):

$$\text{Gauss}(n, k, q) = \frac{(q^n - 1)(q^n - q) \cdots (q^n - q^{k-1})}{(q^k - 1)(q^k - q) \cdots (q^k - q^{k-1})}.$$

Формально подстановка $q = 1$ в q -биномиальный коэффициент даёт обычный биномиальный коэффициент. Хотя q определён только для степеней простого, алгебраическое выражение имеет смысл как предел при $q \rightarrow 1$.

§ 13.12 * Обобщённые RS-коды и связь с БЧХ

В основном разделе мы построили RS-коды через значения многочлена в степенях примитивного элемента. Это классическая конструкция. В более общей форме можно выбирать произвольные различные точки поля.

ОПРЕДЕЛЕНИЕ 13.14 Обобщённый код Рида–Соломона

Пусть $\text{GF}(q)$ — конечное поле. Параметры: $1 \leq k \leq n \leq q$. Выберем n различных элементов $\alpha_1, \dots, \alpha_n \in \text{GF}(q)$.

Сообщение $(m_0, \dots, m_{k-1})$ задаёт многочлен $m(x) = m_0 + m_1x + \dots + m_{k-1}x^{k-1}$ степени меньше k .

Кодовое слово — вектор значений в выбранных точках:

$$c = (m(\alpha_1), m(\alpha_2), \dots, m(\alpha_n)).$$

Код $\text{RS}(n, k)_q$ имеет длину n , размерность k и расстояние $d = n - k + 1$.

Это обобщение определения из раздела о кодах Рида–Соломона, где точки были зафиксированы как $\alpha^0, \dots, \alpha^{n-1}$, а здесь их можно выбрать произвольно. Если взять n различных степеней примитивного элемента — получится классический код основного раздела.

Пример RS(15, 11, 5) над GF(16)

$q = 16 = 2^4$. Поле: $\text{GF}(16) = \text{GF}(2)_{x^4+x+1}^{[x]}$.

Параметры: $n = 15, k = 11, d = 5$.

Избыточность: 4 символа из 15 (около 27%).

Код исправляет до 2 символьных ошибок. Каждый символ — 4 бита. Две ошибки в разных символах — до 8 испорченных бит. Если же ошибки групповые и умещаются в два символа — код исправляет их полностью, независимо от числа испорченных бит внутри символа.

RS-коды — частный случай кодов БЧХ. Код БЧХ задаётся требованием, чтобы кодовый многочлен имел корнями $\delta - 1$ последовательных степеней примитивного элемента. Для RS-кода поле символов совпадает с полем, содержащим эти корни, и именно поэтому RS-код достигает границы Синглтона. В общей теории БЧХ эти поля разнесены, и поле символов с полем корней — разные расширения $\text{GF}(p)$.

Свойство MDS — причина инженерного долголетия RS-кодов. QR-код использует RS над $\text{GF}(256)$, и даже если треть изображения повреждена, данные восстанавливаются интерполяцией по уцелевшим точкам. В компакт-дисках два слоя RS с перемежением (CIRC) защищают от царапин, где внутренний слой исправляет случайные ошибки, а внешний — групповые. Спутники Voyager (запуск 1977) передавали снимки Юпитера и Сатурна каскадом из свёрточного кода и RS(255, 223) — через миллиарды километров.

При заданной избыточности RS-код исправляет максимально возможное число ошибок — граница Синглтона это гарантирует. Современные коды (турбо, LDPC) превосходят RS по приближению к границе Шеннона, но *не по расстоянию* при фиксированной избыточности. В каждой точке этого компромисса RS-код остаётся эталоном, с которым сравнивают новые конструкции.

§ 13.13 * Свёрточные коды и алгоритм Витерби

Все коды, рассмотренные до сих пор, — блочные. Берём k бит, кодируем в n бит, передаём. Следующий блок независим от предыдущего. Так работают код Хэмминга и коды Рида–Соломона.

Но голос в GSM, видеопоток со спутника, сигнал WiFi — непрерывные. Делить их на блоки значит либо ждать заполнения (задержка), либо терять эффективность на неполных блоках.

Пример Кодер (7, 3) — два выходных бита на один входной

Классический свёрточный кодер с памятью на два предыдущих бита: $k = 2$, четыре состояния. Состояние — два последних входных бита (s_1, s_0) перед сдвигом. Два выхода — XOR-ы текущего входа u и битов состояния:

$$c_1 = u \oplus s_1 \oplus s_0,$$

$$c_2 = u \oplus s_0.$$

Полиномы кодера: $g_1 = 1 + x + x^2$, $g_2 = 1 + x$.

Закодируем входной поток $u = 1, 0, 1, 1$, начиная с нулевого состояния:

u	Состояние (s_1, s_0)	$c_1 c_2$	Состояние после
1	(0, 0)	1, 1	(1, 0)
0	(1, 0)	1, 0	(0, 1)
1	(0, 1)	0, 0	(1, 0)
1	(1, 0)	0, 1	(1, 1)

Выходной поток: 11, 10, 00, 01. После каждого бита состояние сдвигается — новый бит становится старшим, а прежний старший — младшим. Именно поэтому приёмник не может декодировать бит, не зная предыдущих, ведь кодовое слово каждого бита зависит от его соседей.

Приёмник видит поток, искажённый шумом. Нужно восстановить наиболее вероятный путь через решётку — тот, что ближе всего к принятому сигналу по расстоянию Хэмминга. Алгоритм Витерби (1967) делает это динамическим программированием.

- Для каждого момента времени и каждого из 2^k состояний он хранит лучший ведущий путь и его метрику — накопленное расстояние до принятого сигнала.
- С приходом нового бита метрики обновляются: для каждого состояния выбираем лучший из двух входящих переходов.

Сложность — $O(2^k \cdot n)$. Здесь n — длина потока.

Свёрточный код дополняет блочный, а не заменяет его. В каскадных схемах — как на Voyager — свёрточный код работает внутренним, исправляя рассеянные ошибки потока. RS-код — внешним, добывая оставшиеся групповые. Расстояние Хэмминга — общая метрика для обоих слоёв. Алгоритм Витерби минимизирует именно его, а минимизация расстояния эквивалентна минимизации вероятности ошибки.

§ 13.14 * Пакетная передача и протоколы

Все коды главы работают с одной предпосылкой. Получатель принимает символы, возможно искажённые, но сам факт приёма известен. Для каналов без этой предпосылки нужен другой слой — доставка. Данные режутся на пакеты, пакеты идут по

сети, и сеть может потерять пакет, продублировать его или переставить местами. Помехоустойчивый код здесь бессилен. Он исправляет значение в известной позиции, а не возвращает потерянную позицию.

Доставку обеспечивает стек протоколов, а не алгебра. На уровне сети (IP) каждый пакет несёт адрес отправителя и получателя и идёт по сети независимо от остальных. Порядок прихода не гарантирован. Пакеты могут прийти в любом порядке, потеряться или прийти дважды. Транспортный уровень добавляет то, чего не хватало.

UDP — простейший транспорт. Он отправляет пакет (дейтаграмму) и не проверяет, дошёл ли он, и за это получает скорость и простоту ценой отсутствия гарантии доставки. TCP строится поверх IP и восстанавливает надёжность. Каждому байту присваивается порядковый номер, получатель подтверждает приём (ACK), отправитель ждёт подтверждения. Если подтверждение не пришло за таймаут — пакет отправляется снова. Так TCP гарантирует, что данные придут в порядке и без потерь, и платит за это задержкой и повторной передачей. Потеря пакета не исчезает, она становится наблюдаемой и исправляется.

Связь с кодами главы двойная. Коды (CRC, Хэмминг, RS) защищают данные внутри пакета от искажения, TCP защищает доставку пакета. Первое исправляет значение, второе — наличие. Обе задачи решаются на разных уровнях и не заменяют друг друга.

§ 13.15 Итоги главы

Код — это множество точек в пространстве слов, и вся теория кодирования вырастает из одной метрики — расстояния Хэмминга. По минимальному расстоянию между кодовыми словами мы сразу узнаём, сколько ошибок код обнаружит ($d - 1$) и сколько исправит ($\lfloor \frac{d-1}{2} \rfloor$) — исправляющая способность оказывается геометрией, ведь шары вокруг кодовых слов не должны пересекаться. Совершенный код Хэмминга $(7, 4)$ показывает границу этой геометрии — его шар радиуса 1 покрывает всё пространство.

Граница Шеннона ставит предел самой возможности надёжной передачи — при скорости выше пропускной способности канала никакая изобретательность не спасёт. Доказательство при этом неконструктивно — оно говорит, что хороший код существует, но не предъявляет его. Код Хаффмана показывает обратное направление — сжатие данных без потерь, где жадный алгоритм строит оптимальный префиксный код для заданных частот символов.

За эффективными кодами стоит алгебра. Линейные коды, циклические коды и коды Рида–Соломона используют один и тот же аппарат — конечные поля — который обслуживает и Ethernet (CRC), и компакт-диски. Свёрточные коды с декодированием Витерби работают в GSM и WiFi, а каскадные схемы — как на Voyager — сочетают внутренний и внешний код.

Проверить, что слово кодовое, в линейном коде — одно умножение на проверочную матрицу. Поиск же ближайшего кодового слова при исправлении ошибок остаётся переборным, и вычислительная сторона таких задач — предмет главы 14. Комбинаторные границы на параметры кодов (Хэмминга, Синглтона) опираются на технику главы 18. Теперь мы можем выбрать код под задачу — исправляющий ошибки или сжимающий данные — и оценить, насколько он близок к теоретическому пределу.

Проверка Проверьте себя

1. Почему кодовое расстояние d даёт исправление $\lfloor \frac{d-1}{2} \rfloor$ ошибок, а не $d - 1$?
2. Чем сжатие Хаффмана отличается от кодов, исправляющих ошибки, и почему две схемы применяют вместе?
3. Что гарантирует граница Шеннона, и почему доказательство не предъявляет самого кода?
4. Почему код Хэмминга $(7, 4)$ исправляет ровно одну ошибку, и как синдром указывает на искажённый бит?
5. Зачем кодам Рида–Соломона конечные поля, а не арифметика целых чисел?
6. В чём разница между обнаружением и исправлением ошибки, и какую из двух задач решает CRC?

Что дальше?

От защиты информации мы переходим к логической задаче, стоящей за верификацией схем — задаче булевой выполнимости. В главе 14 мы изучим алгоритм 2-SAT, искусство кодирования задач в КНФ, архитектуру CDCL-решателей и технологию, на которой держится верификация hardware и software.

§ 13.16 Упражнения

Расстояние Хэмминга и кодовое расстояние.

1. Вычислите расстояние Хэмминга между парами строк: 1100101 и 1010110. Затем: 0001111 и 1110000. Затем: 1010101 и 0101010.
2. Найдите кодовое расстояние множества $C = \{00000, 00111, 11001, 11110\}$. Сколько ошибок этот код может обнаружить? Сколько — исправить?
3. * Докажите, что не существует двоичного кода длины $n = 5$ с $|C| = 4$ кодовыми словами и кодовым расстоянием $d = 4$. Указание: считая одно слово равным 00000 (XOR не меняет расстояний), покажите, что оставшиеся слова веса ≥ 4 не могут быть попарно на расстоянии ≥ 4 .

Код Хэмминга $(7, 4)$.

4. Закодируйте сообщение $d = (1, 1, 0, 1)$ кодом Хэмминга $(7, 4)$. Выпишите вычисление каждого проверочного бита и итоговое кодовое слово.

5. Принято слово $r = (0, 1, 1, 1, 0, 0, 1)$. Используя синдромное декодирование, определите, произошла ли ошибка, и если да — в какой позиции. Исправьте ошибку и восстановите информационные биты.
6. * Проверьте, что код Хэмминга $(7, 4)$ — совершенный: вычислите объём шара радиуса 1 в $\{0, 1\}^7$. Убедитесь, что $16 \cdot \text{объём} = 2^7$.

Линейные коды и конечные поля.

7. Является ли множество $C = \{0000, 0101, 1010, 1111\}$ линейным кодом над $\text{GF}(2)$? Если да — найдите порождающую матрицу и кодовое расстояние. Если нет — объясните, какое свойство нарушается.
8. В поле $\text{GF}(4)$ с неприводимым многочленом $x^2 + x + 1$ примитивным элементом служит $\alpha = x$ (класс многочлена x по модулю $x^2 + x + 1$). Вычислите $\alpha^2, \alpha^3, (\alpha + 1)(\alpha + 1)$. Убедитесь, что $\alpha^3 = 1$.
9. * Вычислите гауссов биномиальный коэффициент $\text{Gauss}(4, 2, 2)$ — число двумерных подпространств в $\text{GF}(2)^4$. Приведите явный перечень базисов всех этих подпространств.

Код Хаффмана.

10. Даны символы с частотами: $A = 0.40, B = 0.25, C = 0.20, D = 0.10, E = 0.05$. Постройте дерево Хаффмана и выпишите кодовое слово для каждого символа. Вычислите среднюю длину кода и сравните с длиной равномерного кода.
11. * Почему код Хаффмана обязан быть префиксным? Приведите пример кода, который не является префиксным, и покажите, что одна и та же битовая последовательность допускает два разных декодирования.

Граница Шеннона.

12. Вычислите пропускную способность двоичного симметричного канала: $C = 1 - H(p), p = 0.5, 1$. Объясните смысл каждого результата.
13. * Почему доказательство теоремы Шеннона неконструктивно — то есть не предъявляет сам код? Что гарантирует существование хорошего кода, если его нельзя построить явно?

Циклические коды и коды Рида–Соломона.

14. Проверьте делимость $g(x) = 1 + x + x^3 \mid x^7 - 1$ над двоичным полем. Найдите кодовое слово циклического кода с порождающим многочленом $g(x)$ для сообщения $t(x) = 1 + x$.
15. Для кода Рида–Соломона $\text{RS}(7, 3, 5)_8$ проверьте, что достигается граница Синглтона $d = n - k + 1$. Сколько символьных ошибок он исправляет?
16. * Код-повторение $(n, 1, n)$ состоит из двух кодовых слов: все нули и все единицы. При каких n этот код является совершенным? Решите уравнение границы Хэмминга как равенство относительно n и $t = \lfloor \frac{n-1}{2} \rfloor$.

17. * Кодовое слово кода Рида–Соломона $RS(n, k)$ над полем $GF(q)$ есть значения многочлена степени меньше k в n точках поля. Здесь $n = q - 1$. Докажите, что любые два различных кодовых слова отличаются не менее чем в $n - k + 1$ позициях. Указание: рассмотрите разность соответствующих многочленов.

ПРОЕКТ **Кодек Хэмминга: исправление одиночных и обнаружение двойных ошибок**

Соберите работающий кодек Хэмминга: кодирование четырёх битов данных в семь, исправление одиночной ошибки по синдрому и расширение до кода $(8, 4)$ с общим битом чётности, который отличает одиночную ошибку от двойной. Проверьте контрольные группы, синдром и границу Хэмминга полным перебором ошибок и симуляцией канала с шумом.

① **Арифметика $GF(2)$, кодирование и синдром**

Реализуйте арифметику поля $GF(2)$: бит как `bool`, сложение — $\oplus$, умножение — логическое «и», умножение матрицы на вектор, транспонирование, вес и расстояние Хэмминга. Задайте код Хэмминга $(7, 4)$ порождающей матрицей G и проверочной матрицей H и продумайте, почему столбцы H удобно выбрать двоичными номерами позиций и как этот выбор определяет синдром. Реализуйте кодирование $c = m \cdot G$ и проверьте, что $Hc = 0$ для всех кодовых слов. Сопоставьте матричный путь с тремя XOR-выражениями контрольных групп и убедитесь, что они дают одни и те же слова. Вычислите синдром принятого слова как $s = Hr$ и объясните, почему нулевой синдром означает отсутствие одиночной ошибки.

② **Исправление одиночной ошибки**

Переберите все кодовые слова и все позиции одиночной ошибки: синдром должен указывать на искажённый бит, и переворот этого бита восстанавливает слово. Продумайте, почему одиночная ошибка всегда исправляется, а двойная — нет: куда указывает синдром двух ошибок и что при этом происходит с данными. Вычислите кодовое расстояние кода и свяжите его с гарантиями исправления и обнаружения ошибок.

③ **Граница Хэмминга и совершенность**

Вычислите объём шара радиуса t в $\{0, 1\}^n$ и проверьте границу Хэмминга

$$|C| \cdot \sum_{i=0}^t \binom{n}{i} \leq 2^n$$

для кода $(7, 4)$ и его расширения $(8, 4)$. Объясните, почему код $(7, 4)$ достигает границы — шары радиуса 1 покрывают всё пространство без зазоров — а расширенный код границы не достигает: код $(8, 4)$ платит за обнаружение двойных ошибок и перестаёт быть совершенным.

④ **Расширенный код $(8, 4)$ с битом чётности**

Добавьте к кодовому слову общий бит чётности и получите код, исправляющий одиночную и обнаруживающий двойную ошибку. Продумайте, как

бит чётности различает одиночную и двойную ошибку: что показывает синдром в каждом случае и почему двойная ошибка больше не маскируется под одиночную. Переберите все одиночные и двойные ошибки и разведите четыре исхода декодирования: слово чистое, ошибка исправлена, ошибка в самом бите чётности, двойная ошибка обнаружена.

⑤ **Канал с шумом**

Соберите симуляцию канала: вносите заданное число ошибок в случайные позиции и независимо переворачивайте биты с вероятностью p . Сведите статистику исходов в таблицу: чисто, исправлено, пропущено, обнаружено — для обоих кодов и разного числа внесённых ошибок. Продумайте, зачем симуляции детерминированный генератор с фиксированным посевом: прогон должен воспроизводиться до единицы.

Итог: работающий кодек: кодирование и исправление одиночной ошибки кодом $(7, 4)$, обнаружение двойной кодом $(8, 4)$, статистика симуляции канала и проверка границы Хэмминга с объяснением, почему один код совершенный, а другой — нет.

XIV

SAT

Две схемы должны делать одно и то же. Инженер, заменяющий проверенную схему оптимизированной, обязан ответить на этот вопрос уверенно — ни один вход не должен развести их выходы. Ответ не найти перебором, ведь у схемы с сотней входов 2^{100} комбинаций, и просмотреть их все не успеет ни одна машина. Нужен способ рассуждать о схеме целиком, не подставляя входы по одному.

В прошлой главе (13) данные защищались от ошибок в пути: шум портит биты, код замечает повреждение и чинит его. Но код сторожит передачу, а производство остаётся вне его ведения: он предполагает, что вычисление уже сделано верно. Здесь вопрос о самом вычислении — верно ли оно, прежде чем данные отправлены. Защита от ошибок и проверка правильности — разные службы, и вторая не проще первой. Тот же вопрос о существовании стоит за множеством инженерных задач.

Можно ли составить расписание экзаменов так, чтобы ни один студент не сдавал два экзамена в один день? Есть ли решение у sudoku? Найдётся ли вход, на котором программа ошибается? Всё это — вопросы вида: существует ли объект с нужными свойствами? Общее у них одно — во всех случаях объект есть набор битов, а свойство — логическое условие.

Булева алгебра (глава 10) даёт язык, на котором такие вопросы записываются точно. Каждое ограничение становится дизъюнктом, ограничения соединяются конъюнкцией, и вся задача — одна формула в *конъюнктивной нормальной форме*. Вопрос «есть ли решение» превращается в вопрос: существует ли набор значений переменных, при котором формула истинна? Это и есть задача булевой выполнимости, SAT. Вопрос об эквивалентности схем (глава 11) — тоже SAT: схемы расходятся ровно тогда, когда формула, описывающая их различие, выполнима.

Формулировка обманчиво проста. За ней стоит граница, которая делит задачи на два мира. Формулы с дизъюнктами из двух литералов решаются быстро: 2-SAT сводится к *графу импликаций* и разбирается за линейное время. Стоит добавить третий литерал — и задача переходит в число самых трудных: 3-SAT NP-полна. Резкая смена сложности при добавлении одного литерала объясняется структурой дизъюнкта. Но формулу надо ещё записать: от выбора переменных зависит, решит ли решатель задачу за секунды или не решит вовсе.

NP-полнота означает, что полиномиальный алгоритм для SAT дал бы полиномиальные алгоритмы для всех задач класса NP. Список таких задач давно перерос комбинаторику: sudoku, сапёр, тетрис, FreeCell, кроссворды и сворачивание белков

— одна и та же трудность, и быстрый способ решать sudoku был бы быстрым способом сворачивать белки. Существует ли такой алгоритм — это и есть вопрос $P = NP$, и ответа пока нет. Систематически классы сложности разбираются в главе 30. Для практики NP-полнота — не приговор. Верификация микросхем, ограниченная проверка моделей, разрешение зависимостей пакетов — индустрия применяет SAT каждый день. Современные решатели на алгоритме CDCL справляются с формулами из миллионов переменных — когда у формул есть структура.

NP-полнота описывает худший случай — намеренно построенные формулы, которые в инженерии не встречаются. Инженерная формула наследует структуру породившей её задачи, и решатель эту структуру находит. Что отличает формулы, которые решаются за секунды, от тех, на которых решатель застревает навсегда?

ИСТОРИЯ NP-полная задача, которую научились решать

Мартин Дэвис и Хилари Патнэм¹¹⁵ предложили первую систематическую процедуру для проверки выполнимости булевых формул — алгоритм Davis-Putnam, основанный на резолюции и элиминации переменных. Алгоритм страдал от экспоненциального роста памяти: каждое применение правила резолюции порождало новые дизъюнкты.

Дэвис, Логеман и Лавленд¹¹⁶ модифицировали его в DPLL: замена резолюции на поиск с возвратом (backtracking) и распространение единичного дизъюнкта (unit propagation). DPLL стал стандартным подходом на три десятилетия, но на трудных формулах всё равно требовал экспоненциального времени.

Стивен Кук¹¹⁷ и Леонид Левин¹¹⁸ независимо доказали NP-полноту SAT. Теорема Кука–Левина утверждает: если SAT решается за полиномиальное время, то $P = NP$. SAT стал эталонной трудной задачей — и одновременно главной целью алгоритмических атак.

В 1996 году Жуан Маркес Силва и Карем Сакаллах предложили Conflict-Driven Clause Learning (CDCL) — механизм обучения на конфликтах с нехронологическим возвратом. Отличие CDCL от DPLL — в смене парадигмы: главную роль играет обучение на конфликтах. DPLL забывает конфликты сразу после возврата. CDCL — запоминает.

Анализируя граф импликаций, решатель извлекает *причину* конфликта в виде нового дизъюнкта — и добавляет его в формулу. Этот дизъюнкт логически следует из исходной формулы (он не сужает пространство решений), но запрещает ту конкретную комбинацию присваиваний, которая привела к конфликту.

¹¹⁵M. Davis, H. Putnam. «A Computing Procedure for Quantification Theory». Journal of the ACM, 1960.

¹¹⁶M. Davis, G. Logemann, D. Loveland. «A Machine Program for Theorem-Proving». Communications of the ACM, 1962.

¹¹⁷S. A. Cook. «The Complexity of Theorem-Proving Procedures». STOC, 1971.

¹¹⁸Л. А. Левин. «Универсальные задачи перебора». Проблемы передачи информации, 1973.

CDCL-решатели (GRASP, Chaff, MiniSat, Glucose, CaDiCaL) научились решать формулы с миллионами переменных. NP-полная задача оказалась практически решаемой для структурированных промышленных формул. CDCL-решатели стали стандартным инструментом формальной верификации и комбинаторной оптимизации.

ОБЗОР ГЛАВЫ

В этой главе мы уточним, что такое КНФ-формула, дизъюнкт и выполнимость. Затем увидим, как длина дизъюнкта решает судьбу задачи: 2-SAT сводится к графу импликаций и решается за линейное время, а 3-SAT NP-полна. Теорема Кука–Левина сделает это утверждение точным, а сведения — раскраска графа, расписание экзаменов, sudoku — покажут, как прикладная задача становится КНФ-формулой. Разбор решателя CDCL объяснит, откуда берётся его эффективность на промышленных формулах с миллионами переменных. SMT — обобщение SAT на арифметику и другие теории — вынесено в отдельную главу 15. Здесь мы вернёмся к вопросу $P = NP$, который SAT сделал точным.

После этой главы вы сможете:

1. приводить задачу к КНФ и проверять выполнимость формулы;
2. решать 2-SAT через граф импликаций за линейное время;
3. объяснять NP-полноту 3-SAT и строить простые сведения к ней;
4. объяснять работу CDCL: распространение, обучение на конфликтах, возврат;
5. понимать, почему промышленные формулы с миллионами переменных решаются, несмотря на NP-полноту.

§ 14.1 Задача SAT

14.1.1 Определения SAT

Задача выполнимости булевой формулы — одна из старейших алгоритмических проблем. От верификации аппаратуры до решения головоломок — всё сводится к SAT.

Формулы обычно приводятся к конъюнктивной нормальной форме (КНФ), с которой и работают алгоритмы.

ОПРЕДЕЛЕНИЕ 14.1 КНФ-формула

КНФ-формула — формула вида $\varphi = C_1 \wedge \dots \wedge C_m$, где каждый дизъюнкт $C_i = (l_{i,1} \vee \dots \vee l_{i,k_i})$.

ОПРЕДЕЛЕНИЕ 14.2 Литерал

Литерал — переменная x или её отрицание: $\bar{x}$.

Пример КНФ-формула

$\varphi = (x \vee \bar{y}) \wedge (\bar{x} \vee y \vee z) \wedge (\bar{z})$. Здесь три дизъюнкта:

- $C_1 = (x \vee \bar{y})$ (два литерала)
- $C_2 = (\bar{x} \vee y \vee z)$ (три литерала)
- $C_3 = (\bar{z})$ (один литерал)

Единичный дизъюнкт (*unit clause*) $(\bar{z})$ немедленно принуждает $z = 0$ в любом выполняющем наборе.

Теперь, когда определение КНФ готово, сформулируем саму задачу SAT.

ОПРЕДЕЛЕНИЕ 14.3 SAT

SAT (задача булевой выполнимости) — задача определения, существует ли набор значений переменных, при котором формула φ истинна.

ОПРЕДЕЛЕНИЕ 14.4 Модель

Модель формулы φ — выполняющий набор значений переменных, при котором формула истинна.

ОПРЕДЕЛЕНИЕ 14.5 Выполнимая формула

Формула **выполнима**, если модель существует. В противном случае формула **невыполнима**.

Резолюция — система вывода для КНФ, исторически связанная с SAT: алгоритм Дэвиса–Патнэма основан на ней. Обзор систем вывода — в главе 2. Здесь резолюция служит рабочим инструментом.

§ 14.2 Резолюция

Таблицы истинности для формулы от n переменных требуют 2^n строк. С ростом n этот метод перестаёт работать. Нужен метод, проверяющий логические следствия без перебора всех интерпретаций.

Принцип резолюции (resolution) — это система вывода, где для проверки противоречивости набора дизъюнктов достаточно одного правила вывода — резолюции. Он работает на КНФ и лежит в основе автоматических доказателей теорем (SAT-решателей) и логического программирования (Prolog).

Здесь и далее черта над переменной означает отрицание: $\bar{x} = \neg x$. Надчёркивание удобнее в дизъюнктах: отрицание пишется над переменной, и отдельный знак перед ней не нужен.

ОПРЕДЕЛЕНИЕ 14.6 Правило резолюции

Из двух дизъюнктов, содержащих контрарную пару литералов (одну и ту же переменную с противоположными знаками), выводится новый дизъюнкт — дизъюнкция всех литералов обоих дизъюнктов, кроме контрарной пары.

Формально:

$$C_1 \vee x, C_2 \vee \bar{x} \vdash C_1 \vee C_2.$$

ОПРЕДЕЛЕНИЕ 14.7 Резольвента

Резольвента — дизъюнкт, выводимый из двух исходных правилом резолюции.

Интуиция: если переменная истинна, то дизъюнкт $C_2 \vee \bar{x}$ истинен только тогда, когда истинен C_2 . Если переменная ложна, то дизъюнкт $C_1 \vee x$ истинен только тогда, когда истинен C_1 . В любом случае истинна дизъюнкция $C_1 \vee C_2$. Резолюция — это формализация разбора случаев «переменная либо истинна, либо ложна, и в обоих случаях что-то следует».

Пример Применение правила резолюции

Из дизъюнктов $(x \vee y \vee \bar{z})$ и $(\bar{x} \vee \bar{y} \vee w)$ с контрарной парой $\frac{x}{\bar{x}}$ получаем резольвенту $(y \vee \bar{z} \vee \bar{y} \vee w)$.

Эта резольвента — тавтология: она содержит контрарную пару $y, \bar{y}$. Тавтологическую резольвенту отбрасывают: она всегда истинна, не добавляет ограничений и не может привести к пустому дизъюнкту.

Процедура доказательства методом резолюции:

АЛГОРИТМ Метод резолюции

1. Привести посылки и отрицание заключения к КНФ.
2. Многократно применять правило резолюции к парам дизъюнктов, добавляя резольвенты в множество.
3. Если на каком-то шаге выводится *пустой дизъюнкт* (*empty clause*) ($\square$), множество противоречиво — исходные посылки влекут заключение.
4. Если пустой дизъюнкт не выводится, а новых резольвент нет — множество выполнимо, заключение не следует.

Пустой дизъюнкт возникает как резольвента пары (x) и $(\bar{x})$ — двух единичных дизъюнктов с контрарными литералами. Это противоречие в чистом виде.

Пример Доказательство через резолюцию

Посылки: $p \rightarrow q \equiv \bar{p} \vee q$, $q \rightarrow r \equiv \bar{q} \vee r$. Заключение: $p \rightarrow r \equiv \bar{p} \vee r$.

Приводим отрицание заключения к КНФ: $p \wedge \bar{r}$ — два единичных дизъюнкта: (p) , $(\bar{r})$.

Исходное множество дизъюнкторов: $\{(\bar{p} \vee q), (\bar{q} \vee r), (p), (\bar{r})\}$.

Резолюция:

1. Из пары (p) , $(\bar{p} \vee q)$ получаем резольвенту (q) .
2. Из пары (q) , $(\bar{q} \vee r)$ получаем резольвенту (r) .
3. Из пары (r) , $(\bar{r})$ получаем пустой дизъюнкт $\square$.

Проведённое опровержение удобно читать по графу: узлы — дизъюнкты, рёбра — шаги резолюции, отсекаемая переменная подписана на ребре. Резольвенты извлекаются из входных дизъюнкторов, пока не получен пустой дизъюнкт $\square$ — как на рис. 72.

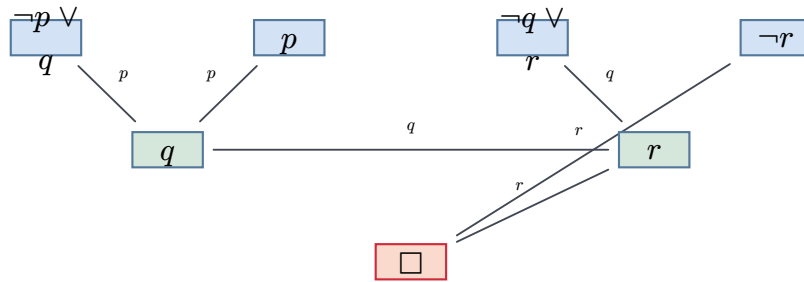


Рис. 72. Резолюционное опровержение.

ТЕОРЕМА 14.1 **Корректность и полнота резолюции¹¹⁹**

Множество дизъюнкторов невыполнимо тогда и только тогда, когда из него методом резолюции выводится пустой дизъюнкт.

Набросок доказательства

Докажем эквивалентность в обе стороны.

Корректность. Резольвента $C_1 \vee C_2$ дизъюнкторов $C_1 \vee x$ и $C_2 \vee \bar{x}$ есть их логическое следствие: при $x = 1$ истинен второй дизъюнкт и с ним его часть C_2 , при $x = 0$ истинна часть C_1 первого. Любая модель исходного множества выполняет каждую выведенную резольвенту, поэтому из вывода пустого дизъюнкта, который не выполняет ни одна интерпретация, следует невыполнимость исходного множества. Тавтологические резольвенты можно отбрасывать: они истинны при всех интерпретациях и на вывод пустого дизъюнкта не влияют.

¹¹⁹J. A. Robinson. «A Machine-Oriented Logic Based on the Resolution Principle». Journal of the ACM, 1965.

Полнота. Пусть множество дизъюнктов невыполнимо. Покажем индукцией по числу n его переменных, что пустой дизъюнкт выводим. При $n = 0$ всякий дизъюнкт множества пуст, и невыполнимое множество уже содержит пустой дизъюнкт. Пусть $n \geq 1$ и для множеств с меньшим числом переменных утверждение верно. Зафиксируем переменную x и разделим дизъюнкты на три группы: содержащие x , содержащие $\bar{x}$ и не содержащие ни того, ни другого. Из дизъюнктов первой группы вычеркнем x , из второй — $\bar{x}$, и каждую из двух половин соединим с третьей группой. Оба полученных множества невыполнимы: модель любого из них вместе с подходящим значением x выполняла бы все дизъюнкты исходного множества, что противоречит невыполнимости. По предположению индукции пустой дизъюнкт выводим из каждой половины. Каждый шаг такого вывода поднимается до шага над дизъюнктами исходного множества: резольвента исходных дизъюнктов либо совпадает с резольвентой укороченных, либо отличается от неё одним лишним литералом x или $\bar{x}$. Поэтому из исходного множества выводим либо пустой дизъюнкт, либо единичный дизъюнкт (x) , а из второй половины — $(\bar{x})$. Если пустой дизъюнкт не выведен уже на этом шаге, применяем правило к паре (x) и $(\bar{x})$ и получаем его одним шагом.

На практике наивная резолюция порождает экспоненциально много дизъюнктов. Современные SAT-решатели используют эвристики (упорядочение литералов, стратегии предпочтения), и полный перебор резольвент им не нужен.

Логическое программирование (Prolog) основано на SLD-резолюции — ограниченной форме резолюции. Ему посвящена отдельная глава (глава 16). В SLD-резолюции один из дизъюнктов — правило Хорна (возможно, однолитеральный факт), а второй — цель, дизъюнкт без положительных литералов. Правило Хорна — это дизъюнкт с ровно одним положительным литералом:

$$\bar{p}_1 \vee \dots \vee \bar{p}_k \vee q,$$

что соответствует импликации $p_1 \wedge \dots \wedge p_k \rightarrow q$. Это ограничение делает резолюцию эффективно реализуемой, а для хорновских формул полноту сохраняет направленная стратегия SLD-резолюции (глава 16).

Проверка Логика высказываний: нормальные формы и резолюция

1. Почему ДНФ собирается из строк таблицы истинности, где формула истинна, а КНФ — из строк, где она ложна?
2. Почему резолюция работает с КНФ, и почему ДНФ для неё не подходит, хотя обе формы эквивалентны исходной формуле?

ЛОВУШКА Выполнимость — не тавтология

Выполнимость и общезначимость (*validity*) — разные вопросы. Формула x выполнима: при $x = 1$ она истинна. Тавтологией она не является: при $x = 0$ формула ложна. Тавтология $x \vee \bar{x}$ истинна при *всех* присваиваниях. Отрицание выполнимой формулы не обязано быть невыполнимым: $\bar{x}$ выполнима при $x = 0$.

Выбор КНФ как основной формы для SAT — это *инженерный компромисс*. Есть альтернативы: DNF (дизъюнктивная нормальная форма) и схемы (circuit-SAT). DNF-SAT тривиальна: формула выполнима, если хотя бы один терм выполним. Circuit-SAT, наоборот, слишком неструктурирован для систематических алгоритмов.

КНФ занимает промежуточное положение:

- достаточно регулярна для эффективного распространения единичного дизъюнкта (движок CDCL),
- достаточно выразительна для кодирования любой NP-задачи: преобразование Цейтина переводит любую схему в КНФ с линейным ростом (подробнее — ниже).

Пример Выполнимая формула

Формула φ из примера «КНФ-формула» выше выполнима: присваивание $x = 1, y = 1, z = 0$ выполняет все три дизъюнкта, первый — через x , второй — через y , третий — через $\bar{z}$.

Вот как выглядит простейшая невыполнимая формула.

Пример Невыполнимая формула

$\psi = (x) \wedge (\bar{x})$ невыполнима: переменная x не может быть одновременно истинной и ложной.

14.2.1 k-SAT: ограничение длины дизъюнктов

Ограничение длины дизъюнктов порождает семейство задач k-SAT. Сложность задачи резко меняется при переходе от $k = 2$ к $k = 3$.

ОПРЕДЕЛЕНИЕ 14.8 k-SAT

k-SAT — задача SAT, в которой каждый дизъюнкт содержит ровно k литералов.

Для $k = 2$ задача разрешима за полиномиальное время (граф импликаций + компоненты сильной связности). Для $k = 3$ задача NP-полна.

Пример 2-SAT и 3-SAT

Формула $(x \vee y) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z})$ — это 2-SAT: каждый дизъюнкт содержит ровно два литерала.

Формула $(x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee z)$ — это 3-SAT. Добавление третьего литерала в каждый дизъюнкт меняет класс сложности задачи.

14.2.2 Алгоритм для 2-SAT

Полиномиальный алгоритм для 2-SAT использует связь с теорией графов. Каждый дизъюнкт из двух литералов можно рассматривать как импликацию: $(l_1 \vee l_2) \equiv (\bar{l}_1 \rightarrow l_2)$. Построив граф всех таких импликаций, мы сводим вопрос выполнимости к поиску компонент сильной связности (КСС).

Вот как это работает на конкретной формуле.

Пример Граф импликаций 2-SAT

Рассмотрим формулу $\varphi = (x \vee y) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z})$. Переводим дизъюнкты в импликации:

- $(x \vee y) : \bar{x} \rightarrow y, \bar{y} \rightarrow x,$
- $(\bar{x} \vee z) : x \rightarrow z, \bar{z} \rightarrow \bar{x},$
- $(\bar{y} \vee \bar{z}) : y \rightarrow \bar{z}, z \rightarrow \bar{y}.$

Граф содержит 6 вершин $x, \bar{x}, y, \bar{y}, z, \bar{z}$ и 6 рёбер. Переменная и её отрицание не лежат в одной КСС — формула выполнима. Обратный топологический порядок даёт: $x = 1, y = 0, z = 1$.

ЛОВУШКА Дизъюнкт порождает две импликации

Дизъюнкт $(l_1 \vee l_2)$ даёт две импликации: $\bar{l}_1 \rightarrow l_2, \bar{l}_2 \rightarrow l_1$. Вторая — контрапозиция первой: если l_2 ложно, то l_1 обязано быть истинным. Построив в графе только одну из двух, мы потеряем половину рёбер, и компоненты сильной связности изменятся — а вместе с ними и ответ о выполнимости.

Построим ориентированный граф: вершины — все литералы x_i и их отрицания $\bar{x}_i$. Для каждого дизъюнкта $(l_1 \vee l_2)$ добавим два ребра: $\bar{l}_1 \rightarrow l_2, \bar{l}_2 \rightarrow l_1$ (контрапозиции).

УТВЕРЖДЕНИЕ 14.2 Критерий выполнимости 2-SAT

Формула 2-SAT невыполнима тогда и только тогда, когда в графе импликаций некоторая пара $x_i, \bar{x}_i$ принадлежит одной компоненте сильной связности.

Если формула выполнима, выполняющий набор строится так:

1. Находим компоненты сильной связности (алгоритмом Косарайю или Тарьяна) и строим конденсацию графа (DAG компонент).
2. Обрабатываем КСС в **обратном** топологическом порядке — от стоков к истокам.
3. Для каждой ещё не обработанной переменной x_i : если КСС литерала x_i идёт позже в топологическом порядке, чем КСС $\bar{x}_i$, присваиваем $x_i = \text{истина}$. Иначе $x_i = \text{ложь}$.

(Альтернативная формулировка: присваиваем истину всем литералам компоненты и ложь их отрицаниям.)

Набросок доказательства

Докажем корректность построения. Если пара $x_i, \bar{x}_i$ лежит в одной КСС, то $x_i \leftrightarrow \bar{x}_i$ — противоречие.

Если же переменная и её отрицание лежат в разных КСС, компоненты упорядочены топологически. Присваивание по обратному топологическому порядку гарантирует, что при наличии пути из одного литерала в другой первый получает значение не больше второго (ложь < истина), поэтому каждая импликация $\bar{l}_1 \rightarrow l_2$ истинна.

Алгоритм работает за $O(n + m)$, где n — количество переменных, а m — количество дизъюнктов.

Оба дизъюнкта формулы $(x \vee y) \wedge (\bar{x} \vee \bar{y})$ ведут в y : $x \vee y$ даёт импликацию $\bar{x} \rightarrow y$, а $\bar{x} \vee \bar{y}$ даёт импликацию $x \rightarrow \bar{y}$. Контрапозиционные рёбра опущены и на вывод не влияют. Пара $y, \bar{y}$ не лежит в одной КСС, поэтому формула выполнима, и y истинна — это видно на рис. 73.

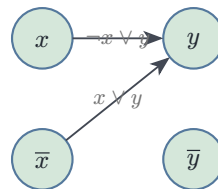


Рис. 73. Граф импликаций для формулы $(x \vee y) \wedge (\bar{x} \vee \bar{y})$.

14.2.3 2-SAT против 3-SAT: структурная причина

Контраст между 2-SAT (разрешима за линейное время алгоритмом на графе импликаций) и 3-SAT (NP-полна) — структурный скачок сложности, ведь добавление третьего литерала к каждому дизъюнкту переводит задачу из класса P в класс NP-полных.

Структурная причина различия: каждый дизъюнкт 2-SAT $(l_1 \vee l_2)$ эквивалентен двум импликациям $\bar{l}_1 \rightarrow l_2, \bar{l}_2 \rightarrow l_1$. Следовательно, вся 2-SAT-формула — это ориентированный граф на $2n$ вершинах (литералах), и выполнимость сводится к проверке отсутствия цикла, содержащего переменную и её отрицание. Эта задача решается алгоритмом Косарайю или Тарьяна за $O(n + m)$.

Для 3-SAT дизъюнкт $(l_1 \vee l_2 \vee l_3)$ не разлагается в пару импликаций — в графе потребовались бы гиперрёбра. Вспомогательные переменные (преобразование Цейтина) здесь не помогают: они лишь переписывают формулу с сохранением выполнимости — результат всё равно содержит 3-дизъюнкты, то есть не становится экземпляром 2-SAT. Сводись всякая 3-КНФ к 2-SAT, SAT решался бы за полиномиальное время — вопреки NP-полноте 3-SAT.

Добавление одного литерала в дизъюнкт разрушает структуру, делающую полиномиальный алгоритм возможным.¹²⁰

Разложение на импликации — мотивировка контраста: оно показывает, почему 2-SAT лёгок, а 3-SAT труден. Сам факт устанавливается сведением произвольной КНФ к 3-КНФ: длинный дизъюнкт разбивают на тройки с помощью вспомогательных переменных, и формула остаётся равновыполнимой исходной.

Пример Сведение длинного дизъюнкта к 3-КНФ

Дизъюнкт из четырёх литералов разбивается одной свежей переменной y :

$$(l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee l_3 \vee l_4).$$

Равновыполнимость: если l_1 или l_2 истинны, берём $y = 0$. Если l_3 или l_4 — берём $y = 1$. Если же все четыре литерала ложны, y не спасёт: первая скобка требует y , вторая — $\bar{y}$. Дизъюнкт из пяти литералов разбивается так же, двумя свежими переменными:

$$(l_1 \vee l_2 \vee y_1) \wedge (\bar{y}_1 \vee l_3 \vee y_2) \wedge (\bar{y}_2 \vee l_4 \vee l_5).$$

Каждый шаг снимает по два литерала исходного дизъюнкта, поэтому формула растёт линейно от длины дизъюнкта, а не экспоненциально.

Странно, что для большинства других пар задач в теории сложности граница не имеет столь же ясного объяснения.

Проверка Граф импликаций 2-SAT

1. Почему дизъюнкт $(l_1 \vee l_2)$ порождает в графе импликаций две стрелки, и почему одной не хватает?

§ 14.3 Кодирование задач в SAT

Теорема Кука говорит нам, что *любая* задача из NP сводится к SAT. Но «сводится» в смысле существования полиномиального сведения, а не в смысле практического

¹²⁰B. Aspvall, M. F. Plass, R. E. Tarjan. «A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas». Information Processing Letters, 1979; контраст с общим случаем: S. A. Cook. «The Complexity of Theorem-Proving Procedures». STOC, 1971 — доказательство NP-полноты SAT.

удобства. Существование сведения и его качество — разные вещи. Плохое сведение порождает КНФ, которую не решит ни один современный решатель. Хорошее — использует структуру задачи и порождает формулу, на которой CDCL работает эффективно. Эффективное кодирование в SAT требует выбирать переменные, отражающие структуру задачи. Механическое переписывание её формального определения в дизъюнкты здесь вредит.

SAT-решатели — мощные инструменты. Но чтобы ими воспользоваться, нужно уметь *кодировать* прикладную задачу в виде КНФ-формулы. Выбор переменных и структуры дизъюнктов радикально влияет на производительность решателя.

Общая стратегия кодирования задачи в SAT:

1. Выбрать булевы переменные, описывающие возможное решение.
2. Записать ограничения задачи в виде дизъюнктов.
3. Запустить SAT-решатель на полученной КНФ.
4. Интерпретировать результат: SAT $\Rightarrow$ декодируем найденную модель, UNSAT $\Rightarrow$ решения не существует.

Рассмотрим несколько классических примеров кодирования.

Пример Преобразование Цейтина

Любую булеву схему можно перевести в равновыполнимую КНФ-формулу с линейным ростом размера. На каждый вентиль заводится одна вспомогательная переменная — значение его выхода — и дизъюнкты связывают эту переменную с входами вентиля. Например, для вентиля $y = \bar{a} \wedge b$ добавляем дизъюнкты

$$(\bar{y} \vee \bar{a}) \wedge (\bar{y} \vee b) \wedge (y \vee a \vee \bar{b}),$$

которые разрешают ровно те комбинации (a, b, y) , где $y = \bar{a} \wedge b$. Модель исходной схемы продолжается до модели формулы, и наоборот, поэтому выполнимость сохраняется.

Пример Раскраска графа в k цветов

Дано: граф $G = (V, E)$ и число k . Вопрос: можно ли раскрасить вершины в k цветов так, чтобы смежные вершины имели разные цвета?

Переменные: $x_{v,c} = 1$, если вершина v имеет цвет c (для $v \in V, c \in \{1, \dots, k\}$).

Ограничения:

- Каждая вершина имеет хотя бы один цвет: $\bigvee_{c=1}^k x_{v,c}$ (длинный дизъюнкт на вершину).
- Ни одна вершина не имеет двух цветов: $\overline{x_{v,c} \vee x_{v,d}}$ для каждой вершины и каждой пары различных цветов $c < d$ (бинарные дизъюнкты).
- Смежные вершины различаются: $\overline{x_{u,c} \vee x_{v,c}}$ для каждого ребра и каждого цвета.

Всего получается $O(|V| \cdot k^2 + |E| \cdot k)$ дизъюнктов.

Раскраска графа — классическая NP-полная задача. Но SAT-кодирование применимо и к более «бытовым» головоломкам.

Пример Судоку

Стандартное судоку 9×9 кодируется в SAT с $9^3 = 729$ переменными: $x_{r,c,d}$ означает, что клетка (r, c) содержит цифру d .

Каждая клетка содержит хотя бы одну цифру — дизъюнкт из девяти литералов:

$$x_{r,c,1} \vee x_{r,c,2} \vee \dots \vee x_{r,c,9}.$$

Остальные ограничения запрещают повторения и записываются бинарными дизъюнктами:

Ограничение	Формула	Количество
клетка содержит не более одной цифры	$\overline{x_{r,c,d}} \vee \overline{x_{r,c,d'}}$ для $d < d'$	$81 \cdot \binom{9}{2} = 2916$
в строке r цифра d не повторяется	$\overline{x_{r,c,d}} \vee \overline{x_{r,c',d}}$ для $c < c'$	$9 \cdot 9 \cdot 36 = 2916$
в столбце c цифра d не повторяется	$\overline{x_{r,c,d}} \vee \overline{x_{r',c,d}}$ для $r < r'$	2916
в блоке 3×3 цифра d не повторяется	попарные запреты для пар клеток внутри блока	2916

Заданные цифры: если в клетке (r, c) стоит цифра d , добавляем единичный дизъюнкт $x_{r,c,d}$.

Суммарно около 12 тысяч дизъюнктов — небольшой размер по меркам современных SAT-решателей. Они решают судоку любой сложности за миллисекунды. SAT-решатель не имеет никакой информации о правилах судоку — ему дана КНФ из 12 тысяч дизъюнктов, в которой строки, столбцы и блоки не названы.

Следующий пример добавляет количественное ограничение: требование «покрыть все рёбра» усиливается до «покрыть не более чем k вершинами». Такие ограничения мощности — отдельный вызов при кодировании в SAT.

Пример Вершинное покрытие

Дано: граф $G = (V, E)$ и число k . Вопрос: существует ли вершинное покрытие размера $\leq k$?

Переменные: $x_v = 1$, если вершина v входит в покрытие.

Ограничения:

- Каждое ребро покрыто: $x_u \vee x_v$ для каждого $\{u, v\} \in E$.
- Ограничение размера: сумма x_v не превышает k . В чистом SAT это кодируется через ограничение мощности (cardinality constraint). Для малых значений, например $k = 2$, можно использовать запреты на тройки: $\overline{x_u} \vee \overline{x_v} \vee \overline{x_w}$ для каждой тройки вершин — «не более двух из трёх». Для произвольного k применяют sequential counter encoding за $O(nk)$ дизъюнктов или encoding через двоичный сумматор: сумма значений x_v сравнивается с k поразрядным переносом.

Ограничение мощности — единственная часть этого кодирования, требующая нелокальных дизъюнктов. Рёберные ограничения локальны (по два литерала на ребро).

От абстрактных графов перейдём к задаче, с которой сталкивается любой университет: составить расписание экзаменов без конфликтов.

Пример Планирование экзаменов

Дано: n экзаменов, m студентов, каждый записан на подмножество экзаменов, и k временных слотов. Вопрос: можно ли составить расписание без конфликтов (студент не сдаёт два экзамена в одном слоте)?

Переменные: $x_{e,t} = 1$, если экзамен e назначен на слот t (для $e \in \{1, \dots, n\}, t \in \{1, \dots, k\}$).

Ограничения:

- Каждый экзамен назначен хотя бы в один слот: $\bigvee_{t=1}^k x_{e,t}$ для каждого экзамена.
- Экзамен не назначен в два слота одновременно: $\overline{x_{e,t_1}} \vee \overline{x_{e,t_2}}$ для каждой пары слотов $t_1 < t_2$.
- Для каждого студента и каждой пары экзаменов (e_1, e_2) , на которые он записан, и каждого слота t : $\overline{x_{e_1,t}} \vee \overline{x_{e_2,t}}$ (студент не может быть в двух местах одновременно).

Эта задача — классический пример того, как NP-полная задача (раскраска графа) возникает в реальной логистике.

Геометрические ограничения тоже переводятся в логические. Задача об N ферзях — пример такого перевода.

Пример N-ферзей как SAT

Задача: расставить N ферзей на шахматной доске $N \times N$ так, чтобы никакие два не атаковали друг друга.

Переменные: $x_{i,j} = 1$, если ферзь стоит в клетке (i, j) .

Ограничения:

- Ровно один ферзь в каждой строке: для каждой строки i ровно одна из переменных $x_{i,1}, \dots, x_{i,N}$ истинна. Кодировается одним «хотя бы один»-дизъюнктом и $\binom{N}{2}$ бинарными «не более одного»-дизъюнктами.
- Не более одного ферзя в каждом столбце: для каждого столбца j и каждой пары строк $i_1 < i_2$: $\overline{x_{i_1,j}} \vee \overline{x_{i_2,j}}$.
- Не более одного ферзя на каждой диагонали: для каждой пары клеток $(i_1, j_1), (i_2, j_2)$, лежащих на одной диагонали ($|i_1 - i_2| = |j_1 - j_2|$), запрещаем одновременное присутствие: $\overline{x_{i_1,j_1}} \vee \overline{x_{i_2,j_2}}$.

При $N = 8$ число дизъюнктов подсчитывается точно. Восемь дизъюнктов «хотя бы один ферзь в строке», по $8 \cdot \binom{8}{2} = 224$ попарных запретов в строках и в столбцах и $2 \cdot (2 \cdot (1 + 3 + 6 + 10 + 15 + 21) + 28) = 280$ запретов на диагоналях обоих направлений — всего $8 + 224 + 224 + 280 = 736$ дизъюнктов. Решатель справляется мгновенно. Так геометрия шахматной доски превращается в набор дизъюнктов.

Искусство кодирования в SAT — в выборе правильных переменных и записи компактных дизъюнктов. Плохие кодировки раздувают число дизъюнктов экспоненциально. Хорошие кодировки наследуют структуру задачи — и на таких формулах CDCL работает особенно быстро.

14.3.1 Принципы эффективного кодирования

У эффективного кодирования четыре принципа.

- **Экономия переменных:** каждая переменная должна нести смысловую нагрузку. Избыточные переменные раздувают пространство поиска экспоненциально.
- **Компактность ограничений:** бинарные дизъюнкты обрабатываются быстрее длинных. Глобальные ограничения вида «не более одного из k » можно заменить группой бинарных дизъюнктов (попарные запреты, при необходимости с вспомогательными переменными) — это часто ускоряет решатель. Сам по себе k -арный дизъюнкт в чистые бинарные не превращается — иначе 3-SAT сводилась бы к 2-SAT (см. выше).
- **Структурная информация:** если задача имеет симметрию или иерархию — это стоит отразить в переменных. Решатели с CDCL извлекают структурные закономерности из повторяющихся паттернов в формуле.
- **Кардинальность:** ограничения вида «не более k из n » кодируются специальными схемами: sequential counter за $O(nk)$ дизъюнктов, cardinality network за $O(n \log^2 n)$ — экспоненциально лучше наивного перебора подмножеств.

Пример Цена наивного кодирования мощности

Ограничение «не более двух из десяти» наивно запрещает каждую тройку переменных: $\binom{10}{3} = 120$ дизъюнктов. Sequential counter обходится $O(10 \cdot 2)$ дизъюнктов — десятки вместо сотен. С ростом числа переменных разрыв стано-

вится катастрофическим: «не более двух из ста» наивно требует $\binom{100}{3} = 161700$ дизъюнктов.



При кодировании задачи в КНФ её внутренняя структура теряется: решатель получает плоский набор дизъюнктов. Он не знает, что это была раскраска графа или sudoku. CDCL затем восстанавливает структурные закономерности через выученные дизъюнкты.

Проверка Кодирование задач в SAT

1. Почему «не более одного» в кодировании раскраски требует $\binom{k}{2}$ попарных дизъюнктов, тогда как «хотя бы один» — лишь одного?
2. Чем хорошее сведение отличается от плохого, если оба формально корректно сводят задачу к SAT?

§ 14.4 От DPLL к CDCL

CDCL — это DPLL плюс умение учиться на конфликтах. Когда поиск заходит в тупик, решатель отступает на шаг назад, как делал DPLL, но с одним отличием. Он анализирует, *почему* произошёл конфликт, извлекает выученный дизъюнкт и добавляет его в формулу. Этот дизъюнкт предотвращает повторение той же конфигурации присваиваний в будущем, отсекая целые подпространства перебора.

Современные SAT-решатели, способные находить модель или доказывать невыполнимость для формул с миллионами переменных, основаны на алгоритме CDCL (Conflict-Driven Clause Learning). Он эволюционировал из алгоритма DPLL (Дэвис–Патнэм–Логеман–Лавленд, 1962).

Базовый DPLL — это поиск с возвратом (backtracking) с двумя оптимизациями:

- *Распространение единичного дизъюнкта* (unit propagation): если в дизъюнкте все литералы, кроме одного, ложны, оставшийся обязан быть истинным.
- *Устранение чистых литералов* (pure literal elimination): если переменная встречается только с одной полярностью, ей можно присвоить выполняющее значение.

Пример Единичное распространение каскадом

Формула $(x) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z) \wedge (\bar{z})$. Единичный дизъюнкт (x) принуждает $x = 1$. Теперь $(\bar{x} \vee y)$ стал единичным: литерал $\bar{x}$ ложен, и y обязан быть истинным. Из $(\bar{y} \vee z)$ следует $z = 1$, после чего единичный $(\bar{z})$ входит в конфликт. Каскад принуждений дошёл до противоречия без единого решения — формула невыполнима.

Пример Устранение чистых литералов

В формуле $(x \vee y) \wedge (\bar{x} \vee z)$ переменная y встречается только положительно, и z — тоже только положительно. Присваивание $y = 1, z = 1$ выполняет оба дизъюнкта при любом значении x : чистая переменная не может навредить ни одному дизъюнкту. Формула выполнима, и перебор по x не нужен.

Оба приёма вместе с ветвлением и возвратом составляют алгоритм DPLL.

Пример DPLL в действии: полный прогон

Рассмотрим формулу

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3).$$

Прогон DPLL с единичным распространением и возвратом:

Ветвь	Решение	Unit propagation	Результат
$x_1 = 1$	$x_1 = 1$	$x_2 = 1$ из $\bar{x}_1 \vee x_2$, затем $x_3 = 0$ из $\bar{x}_2 \vee \bar{x}_3$	конфликт в $\bar{x}_2 \vee x_3$
$x_1 = 0, x_2 = 0$	$x_2 = 0$	$x_3 = 0$ из $x_2 \vee \bar{x}_3$	конфликт в $x_1 \vee x_2 \vee x_3$
$x_1 = 0, x_2 = 1$	$x_2 = 1$	$x_3 = 0$ из $\bar{x}_2 \vee \bar{x}_3$	конфликт в $\bar{x}_2 \vee x_3$

Оба значения x_2 при $x_1 = 0$ приводят к конфликту. Ветвь $x_1 = 1$ тоже конфликтна. Значит, F невыполнима: DPLL исчерпал все возможности и вернул UNSAT.

Прогон демонстрирует три элемента DPLL:

- *Решение* — эвристический выбор значения переменной.
- *Единичное распространение* — принудительное присваивание из дизъюнкта, где все литералы ложны, кроме одного.
- *Возврат* — откат последнего решения при конфликте.

В этом примере возврат хронологический: откатываемся к предыдущему решению.

Поиск с возвратом в DPLL виден целиком на дереве: каждая ветвь — присваивание переменной, unit propagation распространяет вынужденные значения, и когда обе ветви ведут к конфликту, формула невыполнима. Такое дерево показано на рис. 74.

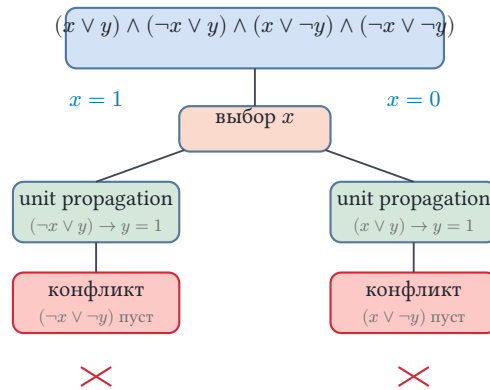


Рис. 74. DPLL-дерево поиска с возвратом.

Примечание

Ядро DPLL уместается в полсотни строк. Формула — это число переменных и список дизъюнктов, где литерал — номер переменной со знаком:

```
// Формула (x1 or x2) and (not x1): дизъюнкты списками литералов.
let clauses = vec![vec![1, 2], vec![-1]];

// Модель: x1 = false (литерал -1 выполнен), x2 = true.
let model = solve(2, &clauses).unwrap();
assert!(!model[0]); // x1 = false
assert!(model[1]);  // x2 = true
```

Внутри solve — три процедуры из прогона выше: решение, единичное распространение, возврат.

CDCL добавляет к этому механизм *обучения на конфликтах*, который радикально ускоряет поиск.

Замечание Сигнальный каскад и граф импликаций

Цепочки причин встречаются и вне вычислений: в сигнальном каскаде клетки ген экспрессировался, потому что сработал рецептор, — это ориентированный граф, где рёбра идут от причины к следствию. Граф импликаций в CDCL — та же структура: ребро $A \rightarrow B$ означает, что B получил значение, потому что дизъюнкт $\bar{A} \vee B$ стал единичным. Конфликт — точка, где две такие цепочки сходятся на одной переменной с противоположными требованиями. Решатель запоминает причину конфликта в виде выученного дизъюнкта.

АЛГОРИТМ CDCL

CDCL расширяет DPLL механизмом обучения на конфликтах. Основные компоненты алгоритма:

- **Распространение единичного дизъюнкта** (Boolean Constraint Propagation, BCP): если в дизъюнкте все литералы, кроме одного, ложны, оставшийся

должен быть истинным. Распространяем принудительные присваивания до фиксированной точки или до конфликта. На практике ВСП занимает подавляющую часть времени работы решателя¹²¹, поэтому его реализация критична для производительности.

- **Принятие решения** (decision): когда распространение исчерпано и конфликта нет, выбираем неприсвоенную переменную и присваиваем ей значение. Эвристика VSIDS (Variable State Independent Decaying Sum) отдаёт приоритет переменным, часто появляющимся в недавних конфликтах.
- **Анализ конфликта** (conflict analysis): когда дизъюнкт становится ложным, анализируем граф импликаций — историю того, какие присваивания привели к конфликту. Из анализа извлекается **выученный дизъюнкт** (learned clause) — новое логическое следствие формулы, запрещающее конфликтующую комбинацию присваиваний. Стандартной техникой является схема первого UIP (Unique Implication Point): UIP — вершина графа импликаций текущего уровня, через которую проходят все пути от последнего решения к конфликту. Первый UIP — ближайшая к конфликту такая вершина.
- **Нехронологический возврат** (non-chronological backtracking): отменяем решения до точки, в которой выученный дизъюнкт становится единичным — это может быть значительно раньше, чем последнее решение.
- **Обучение**: добавляем выученный дизъюнкт в формулу.

Пример Анализ конфликта: первый UIP

Пусть поиск дошёл до решений $x_1 = 1$ (уровень 1) и $x_2 = 0$ (уровень 2), и unit propagation вывел: $x_3 = 1$ из $(\overline{x_1} \vee x_3)$, $x_4 = 1$ из $(\overline{x_3} \vee x_4)$, $x_5 = 1$ из $(x_2 \vee x_5)$. Дизъюнкт $(\overline{x_4} \vee \overline{x_5} \vee x_6)$ стал единичным и дал $x_6 = 1$, после чего $(\overline{x_4} \vee \overline{x_5} \vee \overline{x_6})$ оказался ложным — конфликт.

Граф импликаций: $x_1 \rightarrow x_3 \rightarrow x_4 \rightarrow x_6$, $x_2 \rightarrow x_5 \rightarrow x_6$, а в вершину конфликта $\square$ входят рёбра от x_4 , x_5 и x_6 — всех трёх литералов ложного дизъюнкта. Обе цепочки от решения уровня 2 ($x_2 = 0$) к конфликту проходят через x_5 , а через x_6 — только одна, поэтому первый UIP — x_5 .

Разрез сразу за UIP отделяет причину от следствия. Выученный дизъюнкт — отрицание литералов на стороне причины: $\overline{x_4} \vee \overline{x_5}$. Он запрещает комбинацию $x_4 = 1, x_5 = 1$, породившую конфликт.

Нехронологический возврат: x_4 присвоен на уровне 1, x_5 — на уровне 2, поэтому откатываемся к уровню 1, минуя решение уровня 2. Там выученный дизъюнкт

¹²¹По данным профилирования MiniSat и Glucose, ВСП доминирует в общем времени выполнения — типичные значения 80–90%.

единичен и даёт $x_5 = 0$, дизъюнкт $(x_2 \vee x_5)$ принуждает $x_2 = 1$, и все дизъюнкты выполнены: модель найдена.

Разрез и выученный дизъюнкт из этого примера показаны на рис. 75: две цепочки причин сходятся в $\perp$, а пунктирный разрез сразу за первым UIP разделяет причину и следствие.

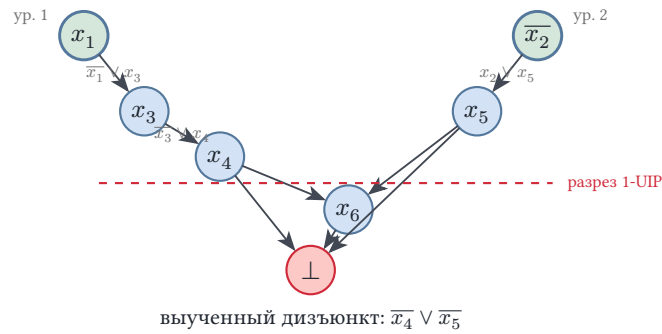


Рис. 75. Граф импликаций при конфликте.

Остаются две инженерные детали, без которых обучение не было бы быстрым.

Watched literals. Вместо того чтобы проверять все литералы дизъюнкта при каждом присваивании, решатель следит ровно за двумя. Пока оба не ложны — дизъюнкт не единичный. Это превращает распространение (BCP) из квадратичной операции в практически линейную по длине формулы. Без watched literals распространение по миллионам дизъюнктов становится практически нереализуемым.

Перезапуски. Периодически решатель отбрасывает текущие присваивания и начинает поиск заново, сохраняя выученные дизъюнкты. Первые решения часто принимаются наспех, без достаточной информации о структуре формулы. Перезапуск с накопленными дизъюнктами даёт VSIDS шанс выбрать лучший порядок переменных. Частота перезапусков — компромисс: слишком частые не дают исследовать ветвь, слишком редкие удерживают поиск в бесперспективной области.

Все описанные приёмы вместе умещаются в несколько тысяч строк кода — так устроено ядро современных решателей.

§ 14.5 Применения SAT-решателей

Современные CDCL-решатели (MiniSat, Glucose, CaDiCaL, Kissat) регулярно решают промышленные примеры с миллионами переменных и десятками миллионов дизъюнктов, несмотря на NP-полноту SAT в худшем случае. Практические промышленные формулы обладают структурой (сообщества переменных, иерархия ограничений, малый treewidth графа дизъюнктов), которую CDCL эксплуатирует через выученные дизъюнкты и эвристику VSIDS.

У случайных формул структуры нет, и картина другая — у случайной k -SAT есть *фазовый переход* по отношению дизъюнктов к переменным (для 3-SAT — $\frac{m}{n} \approx 4.26$).

Ниже порога формула почти наверняка выполнима, выше — почти наверняка нет, а на самом пороге лежит самая трудная для решателя область. Именно такие бесструктурные формулы перебираются экспоненциально долго — эффективность CDCL объясняется структурой формул.

Основные применения SAT:

- **Верификация аппаратуры:** проверка эквивалентности оптимизированной схемы спецификации через митер (см. главу 11). SAT-решатель либо доказывает невыполнимость митера (схемы эквивалентны), либо находит контрпример.
- **Ограниченная проверка моделей** (bounded model checking, BMC): программа разворачивается на k шагов выполнения в КНФ-формулу. Выполнимость означает существование ошибки (нарушения утверждения) в пределах k шагов. Используется в верификации драйверов устройств и критического ПО.
- **Криптоанализ:** шифр с уменьшенным числом раундов переводится в КНФ. SAT-решатель ищет ключ, переводящий известный открытый текст в известный шифротекст. Так были впервые взломаны некоторые варианты SHA-1 и AES с пониженным числом раундов.
- **Управление конфигурациями:** зависимости пакетов Linux-дистрибутива кодируются в КНФ. Пакет p версии v установлен $\rightarrow$ все его зависимости удовлетворены. Решатель находит совместимый набор версий.

Этот феномен — NP-полная задача, систематически решаемая на практике для структурированных входов — превратил SAT из теоретической диковинки в индустриальный инструмент.¹²²

§ 14.6 Локальный поиск: WalkSAT

CDCL — полный (complete) решатель: после конечного числа шагов он возвращает модель либо доказательство невыполнимости, и третьего исхода не бывает. Гарантия сохраняется и на случайных формулах у порога фазового перехода, упомянутого выше, но ожидать ответа приходится экспоненциально долго. Для таких входов выросло второе семейство — неполные (incomplete) решатели. Неполный решатель ищет только модель и доказательств не строит: на выполнимом входе при удаче модель находится быстро, а остановка без модели означает лишь, что лимит ходов и перезапусков исчерпан. Размен — полнота против скорости на выполнимых формулах.

WalkSAT¹²³ работает с присваиванием целиком и не строит дерева поиска. Старт — случайные значения всех переменных. Насколько точка далека от модели,

¹²²S. Malik, L. Zhang. «Boolean Satisfiability: From Theoretical Hardness to Practical Success». Communications of the ACM, 2009; A. Biere et al. (ред.). «Handbook of Satisfiability». 2-е изд., IOS Press, 2021.

¹²³B. Selman, H. Kautz, B. Cohen. «Noise Strategies for Improving Local Search». AAAI, 1994; предшественник — GSAT: B. Selman, H. Levesque, D. Mitchell. «A New Method for Solving Hard Satisfiability Problems». AAAI, 1992.

показывает число нарушенных дизъюнктов. Пока счёт ненулевой, WalkSAT берёт случайный нарушенный дизъюнкт и переворачивает (flip) в нём одну переменную. Кандидатов столько, сколько литералов в дизъюнкте, и ход выбирается из двух стратегий. Жадный ход переворачивает ту переменную, после которой число нарушенных дизъюнктов минимально — поиск спускается по склону этой меры. Шумовой ход подбрасывает монету и переворачивает любую переменную дизъюнкта, даже если счёт от этого вырастет.

Склон ведёт вниз не всегда: на плато все ходы оставляют счёт неизменным, а в локальной яме любой ход его ухудшает. Шум выталкивает и с плато, и из ямы, а перезапуск — забег с нового случайного присваивания — даёт ещё один шанс, когда серия ходов зашла в тупик. Перезапуски уже встречались в CDCL: там они тоже страховка от неудачного старта, только выученные дизъюнкты при этом сохраняются.

Примечание Доля шума

Отношение числа шумовых ходов к общему числу — параметр алгоритма. У порога фазового перехода оптимальная доля ощутимо больше нуля: чистая жадность вязнет на плато, чистый шум превращает поиск в блуждание наугад.

Пример Прогон WalkSAT

Формула из пяти трёхлитеральных дизъюнктов. При трёх переменных возможно $2^3 = 8$ присваиваний, и каждый дизъюнкт запрещает ровно одно из них:

$$\varphi = (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z}) \wedge (x \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y} \vee z).$$

Случайный старт $x = y = z = 1$ нарушает единственный дизъюнкт $C_2 = (\bar{x} \vee \bar{y} \vee \bar{z})$, и счёт равен 1. Пробуем все перевороты в C_2 :

- переворот x даёт $(0, 1, 1)$ и нарушает C_3 ;
- переворот y даёт $(1, 0, 1)$ и нарушает C_4 ;
- переворот z даёт $(1, 1, 0)$ и нарушает C_5 .

Все три хода равноценны, счёт остаётся равным единице, и пусть жадный выбор — при равенстве он произволен — остановится на x . Присваивание $(0, 1, 1)$ нарушает один дизъюнкт $C_3 = (x \vee \bar{y} \vee \bar{z})$, и кандидаты в нём те же три переменные. Переворот x возвращается к $(1, 1, 1)$ с прежним счётом 1, а переворот y или z обнуляет счёт. Жадный ход берёт y , и через два переворота модель найдена: $x = 0, y = 0, z = 1$, истинна ровно одна переменная из трёх.

Первый шаг показателен: счёт не улучшил ни один ход, нарушенный дизъюнкт просто переехал. На больших формулах такие плато тянутся десятки шагов, и вытаскивают оттуда как раз шумовые ходы с перезапусками.

На выполнимых формулах у порога — для случайного 3-SAT это $\frac{m}{n} \approx 4.26$ — локальный поиск обычно находит модель быстрее систематического перебора. Зато

невыполнимость он не докажет никогда: после скольких угодно шагов без модели вывода всё равно не сделать. Отсюда разделение труда: индустриальную формулу, богатую структурой, отдают CDCL, случайную формулу у порога — локальному поиску. Если же формула невыполнима, единственный способ узнать об этом — полный решатель.

§ 14.7 Граф импликаций: от 2-SAT до CDCL

Оба алгоритма главы читают логические ограничения как направленные импликации в графе. Для 2-SAT граф задан формулой целиком, и ответ о выполнимости даёт проверка компонент сильной связности. CDCL строит такой же граф неявно, по ходу присваиваний: единичный дизъюнкт добавляет рёбра, а ложный дизъюнкт становится конфликтом. Анализ конфликта выделяет **разрез** графа импликаций, разделяющий причину конфликта и его следствие. Выученный дизъюнкт — это отрицание литералов на стороне причины: он запрещает комбинацию присваиваний, приведшую к конфликту.¹²⁴

Замечание Два графа под одним именем

Граф импликаций 2-SAT и граф импликаций CDCL — разные объекты, и общее у них только имя. Граф 2-SAT статичен и полон: вершины — все $2n$ литералов, рёбра заданы формулой заранее. Граф CDCL динамичен и частичен: вершины — лишь литералы, уже получившие значения, и рёбра появляются по ходу распространения. Объединяет их одна идея: ребро $A \rightarrow B$ читается как «причина A влечёт следствие B ».

§ 14.8 SAT и NP

SAT — точка отсчёта в теории сложности. Заполненную сетку sudoku проверить легко: ошибка, если она есть, видна за минуту. Найти решение с нуля — труднее: наивный перебор пробует 9^{81} заполнений. Что «найти» действительно труднее «проверить», пока не доказано, — и этот разрыв и есть вопрос $P = NP$. Чтобы говорить о NP-полноте, нужны два понятия (формально — глава 30). Здесь достаточно интуитивных версий.

1. *Класс NP* — задачи, чей ответ «да» можно проверить за полиномиальное время: предъявленному сертификату (например, набору значений переменных) верификатор отвечает быстро.
2. *Полиномиальная сводимость* — преобразование входов одной задачи во входы другой, сохраняющее ответ и выполнимое за полиномиальное время.
3. Задача NP-полна, если она лежит в NP и к ней сводится любая задача из NP.

¹²⁴J. P. Marques-Silva, K. A. Sakallah. «GRASP: A Search Algorithm for Propositional Satisfiability». IEEE Transactions on Computers, 1999.

Задачу из NP, про которую доказано, что она не лежит в P, предъявить нельзя: её существование равносильно $P \neq NP$. Поэтому теория сложности ищет «самую трудную» задачу класса: если решается она, решается всё. Теорема Кука (1971) утверждает, что SAT — NP-полная задача, «самая трудная» в классе NP.

ТЕОРЕМА 14.3 Теорема Кука — формулировка¹²⁵

SAT NP-полна: $SAT \in NP$, и всякая задача из NP полиномиально сводится к SAT.

Набросок доказательства

Докажем в две стороны: сначала покажем, что SAT принадлежит NP, затем — что SAT NP-трудна.

SAT принадлежит NP: сертификат — выполняющий набор переменных, а проверка состоит в подстановке значений в КНФ и проверке всех дизъюнктов, что делается за полиномиальное время.

SAT NP-трудна: нужно показать, что любая задача из NP сводится к SAT. Недетерминированная машина Тьюринга (НДТМ) устроена как обычная, но в каждой конфигурации может выбрать одну из нескольких команд, и вычисление ветвится. Вход принимается, если принимает хотя бы одна ветвь, а угадывание сертификата — это и есть выбор ветви.

Класс NP можно описать и иначе: машина сначала угадывает сертификат, затем работает детерминированно. Эквивалентность этого описания определению через верификатор стандартна и здесь не доказывается. Угадывание — сжатая запись перебора: копии машины, по одной на каждый возможный сертификат, и принимает та копия, чей сертификат оказался верным.

Пусть M — недетерминированная машина Тьюринга, распознающая язык $L \in NP$ за время $p(n)$. Для входа x строится булева формула φ_x в КНФ, выполняемая в точности тогда, когда вход принимается машиной M .

Конструкция использует **таблицу вычисления** размера $p(n) \times p(n)$: ячейка (i, t) кодирует содержимое i -й позиции ленты на шаге t . Переменные: $T_{i,t,a}$ истинна, если ячейка (i, t) содержит символ a , а $H_{i,t,q}$ истинна, если головка в позиции i на шаге t в состоянии q .

Таблица вычисления локальна: содержимое ячейки на следующем шаге определяется тремя ячейками над ней. Окно 2×3 такого фрагмента показано на рис. 76.

¹²⁵S. A. Cook. «The Complexity of Theorem-Proving Procedures». STOC, 1971. Первая доказанная NP-полнота. Конструкция сводит произвольную NP-машину к булевой формуле через таблицу вычисления.

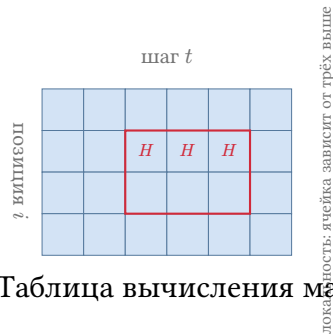


Рис. 76. Таблица вычисления машины Тьюринга.

Набросок доказательства

Построим формулу φ_x из четырёх групп дизъюнктов, кодирующих принимающее вычисление, и проверим её размер.

Первая группа — **начальная конфигурация**: x записан на ленте, головка в начальной позиции, состояние — начальное. Вторая группа — **локальная корректность перехода**: для каждого окна 2×3 в таблице вычисления содержимое ячейки на шаге $t + 1$ определяется тремя ячейками над ней на предыдущем шаге — в соответствии с функцией переходов M . Третья группа — **единственность**: в каждой ячейке — не более одного символа, головка — не более чем в одной позиции, машина — не более чем в одном состоянии. Наконец, четвёртая группа — **принимающее условие**: на последнем шаге M находится в принимающем состоянии.

Каждый дизъюнкт содержит $O(1)$ литералов, общее число дизъюнктов — $O(p(n)^2)$. Формула выполнима тогда и только тогда, когда существует принимающее вычисление M на входе x , а сведение полиномиально — как по времени построения формулы, так и по её размеру.

Из NP-полноты SAT напрямую следует ответ о цене перебора.

СЛЕДСТВИЕ 14.4 Следствие для P vs NP

Если $\text{SAT} \in P$, то $P = NP$. Если SAT требует сверхполиномиального времени, то $P \neq NP$.

Доказательство

Докажем эквивалентность $\text{SAT} \in P$ и $P = NP$ в обе стороны.

Если $\text{SAT} \in P$, то существует полиномиальный алгоритм A для SAT. По теореме Кука любая задача $L \in NP$ полиномиально сводится к SAT функцией f , поэтому композиция $A \circ f$ даёт полиномиальный алгоритм для L . Отсюда $L \in P$ и $P = NP$.

Обратно, если $P = NP$, то SAT, как NP-полная задача, принадлежит P , следовательно, утверждения $P = NP$ и $\text{SAT} \in P$ эквивалентны. Поэтому если SAT требует сверхполиномиального времени, то $P \neq NP$.

«Сверхполиномиальное время» не обязательно означает экспоненту — достаточно, например, $2^{\sqrt{n}}$. Нижняя граница пока не доказана.¹²⁶

Замечание Универсальный представитель

Теорема Кука–Левина устанавливает больше, чем NP-полноту SAT. Конструкция в доказательстве — таблица вычисления недетерминированной машины Тьюринга, закодированная в КНФ — показывает, что задача выполнимости и задача «существует ли принимающее вычисление?» структурно идентичны. Любая NP-задача, от раскраски графа до планирования экзаменов, при сведении к SAT передаёт ему эту структуру: свидетель становится моделью формулы.

Именно эта универсальность делает вопрос « $SAT \in P?$ » эквивалентным вопросу « $P = NP?$ ». Если SAT решается полиномиально, то всякое полиномиально проверяемое утверждение — корректность криптографического протокола, оптимальность расписания, верность математической теоремы с доказательством полиномиальной длины — получает полиномиальный алгоритм распознавания. Если же SAT требует сверхполиномиального времени, то NP-полнота остаётся сертификатом экспоненциального перебора в худшем случае.

Распознать доказательство и найти его — разные умения, и в мире, где $P = NP$, проверка доказательства и поиск доказательства стали бы одной задачей.

Понимание SAT — и алгоритмическое, и теоретическое — до сих пор формирует облик computer science. Статус SAT *двойствен*: NP-полная задача в худшем случае и систематически решаемая на промышленных входах.

NP-полнота — свойство задачи в худшем случае: среди *всех* КНФ-формул есть намеренно сконструированные патологические, на которых решатели работают экспоненциально долго. Пример — КНФ-кодировки принципа Дирихле, на которых resolution требует экспоненциального времени. Масштаб эффекта виден на sudoku: сетку 9×9 щёлкает телефон, а сетку 100×100 в худшем случае не одолеет ни одна машина. Индустриальные же формулы порождены конкретными инженерными задачами — верификацией аппаратуры, планированием, ВМС — и наследуют их структуру.

Три структурных свойства отличают индустриальные формулы от случайных:

1. *Сообщества переменных* (community structure): переменные группируются в кластеры, внутри которых ограничений много, а между кластерами — мало. CDCL учит дизъюнкты, связывающие переменные внутри кластера.
2. *Малая древесная ширина* (treewidth) графа инцидентности (переменные и дизъюнкты): формула близка к древовидной, что позволяет эффективно отсекалть бесперспективные ветви.
3. *Наличие «чёрного хода»* (backdoor): небольшое подмножество переменных, присваивание значений которым сводит формулу к полиномиально разрешимой.

¹²⁶Если SAT принадлежит P, то $P = NP$. Обратное также верно: если $P \neq NP$, то SAT не решается за полиномиальное время. Какое из двух утверждений истинно — открытый вопрос.

CDCL решает SAT на промышленных входах, а не в общем случае. Алгоритм эксплуатирует структурные свойства формул через два механизма:

1. VSIDS — отдаёт приоритет «активным» переменным.
2. Обучение дизъюнктам — накопление «структурного знания» о формуле.¹²⁷

§ 14.9 Итоги главы

SAT сводит вопрос о существовании объекта с нужными свойствами к одной формуле: существует ли набор значений переменных, при котором формула истинна. Эквивалентность двух схем — то же самое: они расходятся ровно тогда, когда формула об их различии выполнима, и проверка через митер сводится к задаче выполнимости. Запись задач в КНФ дала единый формат: каждое ограничение становится дизъюнктом, ограничения соединяются конъюнкцией.

Длина дизъюнкта решает судьбу задачи. Дизъюнкт из двух литералов — это две импликации, и граф импликаций с его сильно связными компонентами даёт ответ за линейное время. Стоит добавить третий литерал — и 3-SAT NP-полна: дизъюнкт становится гиперребром, а теорема Кука–Левина превращает SAT в эталонную NP-полную задачу, к которой сводятся раскраска, sudoku и десятки других.

Вторая половина — про практику. CDCL учится на конфликтах, и выученные дизъюнкты с нехронологическим возвратом направляют поиск. Именно поэтому промышленные формулы с миллионами переменных решаются за секунды, тогда как случайные застревают на пороге фазового перехода. Эффективное кодирование задачи в КНФ — отдельное ремесло: переменные должны отражать структуру задачи, а ограничения — укладываться в форматы, которые решатель обрабатывает быстро.

SAT соединяет схемотехнику (глава 11) с теорией сложности (глава 30): полиномиальный алгоритм для неё означал бы $P = NP$. Резолюция, изученная в этой главе, станет основой следующей: глава 16 превратит её в язык программирования.

Проверка Проверьте себя

1. Почему 2-SAT решается за линейное время, а 3-SAT NP-полна?
2. Что такое сведение, и почему из NP-полноты SAT следует: если SAT в P, то $P = NP$?
3. В чём идея CDCL: откуда берётся выученный дизъюнкт и почему возврат называется нехронологическим?
4. Как закодировать раскраску графа в КНФ, и какие группы дизъюнктов отвечают за «хотя бы один цвет» и «не более одного цвета»?

¹²⁷C. P. Gomes, H. Kautz, A. Sabharwal, B. Selman. «Satisfiability Solvers». Глава 2 в «Handbook of Knowledge Representation». Elsevier, 2008; C. Ansótegui, M. L. Bonet, J. Levy. «On the Structure of Industrial SAT Instances». CP, 2009.

5. Чем индустриальные формулы отличаются от случайных, и почему NP-полная задача решается на первых на практике?
6. Как граф импликаций определяет, выполнима ли 2-SAT-формула?

Что дальше?

Мы остаёмся в логике: следующая глава расширяет SAT теориями. В главе 15 к булевым переменным добавятся арифметика, равенство и массивы, а цикл DPLL(T) соединит булево рассуждение с theory-солверами, чтобы отвечать, выполнима ли формула над этими теориями. SAT решал выполнимость булевых формул. SMT спрашивает то же про формулы с арифметикой и другими теориями — и это станет движком верификации программ в главе 32.

§ 14.10 Упражнения

Базовые понятия SAT.

1. Какие из следующих КНФ-формул выполнимы? Для выполнимых укажите модель.
 - a) $(x \vee y) \wedge (\bar{x} \vee \bar{y})$
 - b) $(x) \wedge (\bar{x})$
 - c) $(x \vee y \vee z) \wedge (\bar{x}) \wedge (\bar{y}) \wedge (\bar{z})$
 - d) $(x \vee y) \wedge (\bar{x} \vee y) \wedge (\bar{y})$
2. Приведите пример невыполнимой КНФ-формулы, содержащей ровно два дизъюнкта. Объясните, почему она невыполнима.

Резолюция.

3. Примените метод резолюции, чтобы доказать: из $p \vee q, \bar{p} \vee r, \bar{q} \vee r$ логически следует r . Нарисуйте DAG резолюционного опровержения.
4. Почему резолюция работает с КНФ, и почему ДНФ для неё не подходит, хотя обе формы эквивалентны исходной формуле?
5. * Чем выполнимость отличается от общезначимости? Приведите пример выполнимой, но не общезначимой формулы и пример невыполнимой, отрицание которой выполнимо.

2-SAT и граф импликаций.

6. Переведите каждый дизъюнкт 2-SAT-формулы

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3)$$

в пару импликаций и постройте граф импликаций. Сколько вершин и рёбер в графе?

7. Решите 2-SAT-формулу с помощью графа импликаций:

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_4) \wedge (\overline{x_4} \vee \overline{x_3}).$$

Найдите все компоненты сильной связности, постройте конденсацию и присвойте значения переменным.

8. * Объясните, почему 2-SAT решается за полиномиальное время, а 3-SAT NP-полна. Что именно добавление третьего литерала в дизъюнкт разрушает в графе импликаций?

Кодирование задач в SAT.

9. Закодируйте задачу 3-раскраски графа-треугольника K_3 в виде КНФ-формулы. Используйте переменные $x_{v,c}$ (вершина v имеет цвет c). Выпишите все дизъюнкты. Выполнима ли формула?
10. Закодируйте условие «ровно две переменные из x_1, x_2, x_3, x_4 истинны» в КНФ. Указание: скомбинируйте ограничения «хотя бы две» и «не более двух». Подсчитайте число дизъюнктов.
11. * Закодируйте задачу поиска гамильтонова цикла в графе с n вершинами как SAT. Опишите переменные и основные группы дизъюнктов. Почему наивное кодирование через перечисление всех перестановок порождает экспоненциально много дизъюнктов и как этого избежать?

DPLL и CDCL.

12. Промоделируйте работу DPLL (с unit propagation) на формуле:

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_3}).$$

Используйте порядок решений: сначала $x_1 = 1$, затем $x_2 = 1$ (если потребуется). Для каждого узла дерева поиска укажите частичное присваивание и результат unit propagation.

13. * Дан частичный вывод CDCL: решение $x_1 = 1$. Решение $x_2 = 0$. Unit propagation из $(\overline{x_1} \vee x_3)$ даёт $x_3 = 1$. Unit propagation из $(x_2 \vee x_4)$ даёт $x_4 = 1$. Конфликт в дизъюнкте $(\overline{x_3} \vee \overline{x_4})$.

Нарисуйте граф импликаций. Примените анализ конфликта: найдите разрез, соответствующий первому UIP, и выпишите выученный дизъюнкт.

SAT и NP.

14. Почему из NP-полноты SAT следует: если SAT решается за полиномиальное время, то $P = NP$?
15. * Докажите, что задача о сумме подмножества (subset sum) NP-полна, сведя к ней 3-SAT. Постройте по 3-КНФ-формуле φ с n переменными и m дизъюнктами множество чисел и целевую сумму T так, что подмножество с суммой T существует тогда и только тогда, когда φ выполнима. Указание: числа записываются

в десятичной системе, и разряды отвечают переменным и дизъюнктам. Каждой переменной x_i отвечают два числа (для x_i и $\overline{x_i}$), каждому дизъюнкту — два числа с цифрами 1 и 2 в его разряде. Целевая сумма T имеет 1 в каждом переменном разряде и 4 в каждом дизъюнктном: сумма в дизъюнктном разряде складывается из единиц выбранных литералов и служебных чисел и равна 4, только если хотя бы один литерал дизъюнкта истинен.

16. * «Доказательство» того, что $\text{SAT} \in P$. Перебираем переменные $x_1, \dots, x_n$ по порядку. Для каждой пробуем оба значения и выбираем то, при котором выполнено больше дизъюнктов (при равенстве — 0). После обработки всех переменных получаем полное присваивание. Если формула была выполнима, алгоритм якобы найдёт выполняющее присваивание. Алгоритм полиномиален: n итераций по m дизъюнктов. Найдите ошибку: почему жадный выбор значения не гарантирует выполнимости, даже если модель существует?

ПРОЕКТ Фазовый переход случайного 3-SAT

Случайные 3-SAT-формулы с n переменными и m дизъюнктами при малом отношении $\frac{m}{n}$ почти всегда выполнимы, при большом — почти всегда нет, а переход происходит в узком окне вблизи $\frac{m}{n} \approx 4.26$. Соберите генератор случайных формул и решатель и проследите на работающем коде, как доля выполнимых и время решения зависят от $\frac{m}{n}$.

① Решатель

Соберите DPLL с unit-распространением и чистыми литералами и добавьте статистику решений, пропагаций и конфликтов. Как расширение продумайте CDCL с анализом конфликтов и изучением дизъюнктов.

② Генератор

Постройте генератор случайных 3-SAT-формул с фиксированным n и регулируемым отношением $\frac{m}{n}$, без дублирующих дизъюнктов, и добавьте вывод в формате DIMACS.

③ Кривая выполнимости

Проследите долю выполнимых формул при росте $\frac{m}{n}$ и сверьте положение перехода с известным $\frac{m}{n} \approx 4.26$.

④ Кривая времени

Измерьте медианное время решения в каждой точке и продумайте, почему самые трудные формулы сосредоточены вблизи порога.

⑤ Структурные формулы

Закодируйте в КНФ задачу со структурой, например sudoku или раскраску графа, и сравните время решения таких формул со случайными той же плотности. Объясните, почему структура делает формулу лёгкой для решателя.

Итог: две кривые фазового перехода с проверкой перехода против известного $\frac{m}{n} \approx 4.26$ и сравнение случайных и структурных формул.

XV

SMT

Инженер проверяет условие $x + y \leq 10$ and $x > 5$: найдётся ли пара целых чисел, которая его выполняет? Ответ есть — например, $x = 6, y = 0$. Но как получить этот ответ механически, не перебирая все пары? Глава 14 научила отвечать на такие вопросы для чисто булевых формул, но здесь переменные x, y — целые, а атом $x + y \leq 10$ — *утверждение об арифметике*.

Закодировать условие в булеву формулу можно: представить числа наборами битов, перевести сложение в булеву схему, а неравенства — в дизъюнкты. Цена велика: булева переменная несёт один бит, и целое число занимает столько же переменных, сколько в нём битов: каждое действие над числами приходится собирать как схему из вентилях. Умножение 64-битных чисел с переносами разрядов — это тысячи вентилях, а перевод схемы в дизъюнкты растягивает её ещё в несколько раз. Один оператор задачи оборачивается десятками тысяч дизъюнктов, и решатель тонет в *битовой пыли*, хотя вопрос был всего о двух простых неравенствах. Дело в способе записи: арифметика переводится в булевы переменные поразрядно, и в такой записи её структура не видна.

Другой путь — дать решателю понимать арифметику напрямую. Тогда структура задачи сохраняется: условие $x + y \leq 10$ остаётся одним атомом, а решатель рассуждает, не перебирая значения. Над целыми числами он выводит:

$$x > 5, x + y \leq 10 \Rightarrow y \leq 4$$

Логика высказываний не видит внутренней структуры атома: для неё $x + y \leq 10$ — просто переменная p , которой надо как-то задать значение. Причина — в словаре языка: в нём нет ни сложения, ни сравнения, ни целых чисел, и атому не на что опереться, поэтому его значение назначается целиком, как у любой булевой переменной.

Логика предикатов из главы 3 умеет говорить об объектах, но допускает любые интерпретации символов. В чистой логике $+$ может означать любую бинарную операцию: в одной интерпретации — «выбрать большее», в другой — «перемножить». Вопрос «найдётся ли пара целых чисел» теряет смысл, потому что ничто не заставляет переменную пробегать именно целые числа. Нужен язык, где смысл символов закреплён: терм имеет тип, сложение интерпретируется как настоящее сложение целых, а массив читается и пишется по индексу, как память.

Многосортная логика даёт для этого точный язык: сорт переменной указывает её предметную область. Запись $x : \text{Int}$ означает, что переменная обозначает целое

число. Запись $a : \text{Аггау}$ — что переменная обозначает массив. Типизированная формула отклоняется ещё до семантики, если сорта не сходятся. Но язык сам по себе не даёт алгоритма: полная логика первого порядка неразрешима (глава 26), и бесконечность арифметики нельзя обойти наивным перебором.

Выход — разделение труда: булева логика и теория работают в одном цикле. Булев решатель отвечает за связки и управляет поиском, а атомы теории — такие, как $x + y \leq 10$, — передаёт процедуре, которая понимает арифметику. Процедура проверяет предложенный набор атомов по своим правилам и, если находит противоречие, возвращает возражение. Булев решатель учитывает возражение и продолжает поиск — раунд за раундом — пока формула не будет решена.

Такой способ рассуждения — булев поиск плюс отдельные процедуры для арифметики и памяти — называется *SMT* (от англ. Satisfiability Modulo Theories, *выполнимость по модулю теорий*). Как устроен такой солвер и почему это работает на практике, хотя полная логика первого порядка неразрешима?

ИСТОРИЯ От разрозненных процедур к общему циклу

К началу 1970-х годов для отдельных теорий существовали собственные процедуры разрешения: для арифметики Пресбургера, для равенства, для линейных неравенств. Каждая процедура работала в своей области и не умела говорить с соседями: формула, смешивающая арифметику с равенством, не поддавалась ни одной из них. В 1979 году Грег Нельсон и Дерек Оппен показали, как соединить процедуры разных теорий в одну согласованную — при условии, что теории делят только равенство.¹²⁸ С этой работы начинается комбинирование теорий — основа современных SMT-солверов.

Второй толчок пришёл со стороны SAT. CDCL — алгоритм обучения на конфликтах из главы 14 — позволил булевым решателям обрабатывать формулы с миллионами переменных. Возникла идея: пусть булев поиск руководит формулой, а процедуры теорий отвечают на его вопросы о наборах атомов. В 2006 году Роберт Нивенхёйс, Альберт Оливерас и Чезаре Тинелли описали этот цикл в абстрактной форме и назвали его DPLL(T).¹²⁹ Название закрепилось, и архитектура стала стандартом: почти все SMT-солверы работают в этом цикле.

Третий шаг — стандартизация. Язык SMT-LIB дал общий формат входных данных, а соревнование SMT-COMP — единый полигон для сравнения солверов.¹³⁰ В 2008 году Леонардо де Моура и Николай Бьёрнер представили солвер Z3¹³¹, который превратил SMT в индустриальный инструмент: верификация, символьное исполнение и синтез программ опираются на него каждый день.

¹²⁸G. Nelson, D. Oppen. «Simplification by Cooperating Decision Procedures». ACM Transactions on Programming Languages and Systems, 1979.

¹²⁹R. Nieuwenhuis, A. Oliveras, C. Tinelli. «Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)». Journal of the ACM, 2006.

¹³⁰S. Ranise, C. Tinelli. «The SMT-LIB Standard». 2004.

¹³¹L. de Moura, N. Bjørner. «Z3: An Efficient SMT Solver». TACAS, 2008.

ОБЗОР ГЛАВЫ

Язык SMT объединяет многосортную логику первого порядка и теории: сорт переменной задаёт её предметную область, а теория сужает класс допустимых интерпретаций. После языка мы перейдём к теориям, важным для практики SMT: разностная логика, равенство с неинтерпретируемыми функциями, линейная арифметика, битовые векторы, теория массивов. У каждой теории своя процедура решения, и разбор этих процедур по очереди объясняет, почему теория разрешима: отрицательный цикл в графе разностей означает невыполнимость, а congruence closure сливает классы равных термов. Затем цикл DPLL(T) покажет, как один булев поиск оркеструет готовые процедуры теорий: они обмениваются кандидатами и возражениями, и один раунд конфликта прослеживается от присваивания до выученного дизъюнкта. Процедура Нельсона–Оппена объяснит, как соединить решатели разных теорий в одну согласованную процедуру, когда теории делят только равенство. В конце главы мы рассмотрим формат SMT-LIB и устройство реальных солверов: запрос читается на входном языке, а ответ солвера предсказывается по правилам теорий.

После этой главы вы сможете:

1. записывать задачи в многосортной логике с теориями арифметики, равенства и массивов;
2. объяснять архитектуру DPLL(T) и роль процедур теории в булевом поиске;
3. применять процедуры для разностной логики и замыкания по равенству;
4. объяснять, почему SMT работает на кванторно-свободных фрагментах;
5. читать запросы в формате SMT-LIB и применять SMT к верификации и символьному исполнению.

§ 15.1 Многосортная логика первого порядка

Глава о логике первого порядка обошлась одним сортом: все термы говорили об одном универсуме, и этого хватало. Формула из вступления таким языком не записывается: она требует, чтобы одна переменная была числом, а другая — массивом, и односортная логика этого различия не видит. В ней терм $\text{read}(x, 0)$ собирается без возражений: подставить массив на место числа — обычная операция, и семантика честно припишет терму значение. Формула получает смысл, которого автор не вкладывал: число читается как массив, а смешение сортов не наказывается. Тип — это запрет, наложенный до семантики: сорта не дают терму собраться, если его части не согласованы, и бессмысленное выражение не существует в языке.

SMT работает с *многосортной* логикой (от англ. many-sorted): у каждого терма есть сорт, и функция принимает аргументы строго определённых сортов. *Сорт* переменной задаёт её предметную область. Запись $x : \text{Int}$ означает, что переменная

обозначает целое число. Запись $a : \text{Array}$ означает, что переменная обозначает массив.

Если функция применяется к аргументу не того сорта, формула отклоняется ещё до рассмотрения семантики. Так и компилятор останавливается на ошибке типов, не запуская программу: вычислять нечего, потому что программа не собрана.

ОПРЕДЕЛЕНИЕ 15.1 Многосортная сигнатура

Многосортная сигнатура $\Sigma = (S, F)$ состоит из множества **сорт**ов S и множества **функциональных символов** F .

Сорта — это типы объектов: булевы значения, целые числа, вещественные числа, массивы. Функциональные символы — это операции и отношения: равенство, сложение, сравнение, чтение и запись.

Каждый функциональный символ f имеет **ранг** — кортеж сортов

$$\text{rank}(f) = (\sigma_1, \dots, \sigma_n, \sigma_{n+1})$$

, показывающий, какие сорта аргументов он принимает и какой сорт возвращает.

Символ без аргументов называется *константой*. Символ, возвращающий булево значение, называется *предикатом*. Всякая сигнатура содержит булев сорт с константами истины и лжи и предикат равенства для каждого сорта.

Пример Сигнатуры

- Арифметика натуральных чисел. Сорта: $S = \{\text{Nat}\}$. Символы: $F = \{0, S, <, +, \times\}$.
- Массивы. Сорта: $S = \{\text{Array}, \text{I}, \text{E}\}$. Символы: $F = \{\text{read}, \text{write}\}$.

Ранги символов:

Символ	Ранг
0	$\text{rank}(0) = (\text{Nat})$
S	$\text{rank}(S) = (\text{Nat}, \text{Nat})$
$<$	$\text{rank}(<) = (\text{Nat}, \text{Nat}, \text{Bool})$
$+$	$\text{rank}(+) = (\text{Nat}, \text{Nat}, \text{Nat})$
$\times$	$\text{rank}(\times) = (\text{Nat}, \text{Nat}, \text{Nat})$
read	$\text{rank}(\text{read}) = (\text{Array}, \text{I}, \text{E})$
write	$\text{rank}(\text{write}) = (\text{Array}, \text{I}, \text{E}, \text{Array})$

Ранги и состав символов отсекают бессмысленные сочетания ещё до семантики.

- Терм с неверным сортом аргумента: $\text{read}(x, i)$. Первый аргумент обязан быть массивом, а здесь стоит переменная индексного сорта.
- Терм с неверным сортом операнда: $\text{true} + x$. Сложение принимает два аргумента натурального сорта, а булевская константа в него не входит.

- Терм с необъявленным символом: $\text{read}(x, 0)$. Константа 0 в сигнатуре массивов не объявлена вовсе, и сорта у неё нет.

Формула с таким термом не получает истинностного значения: её нет в языке.

Сигнатура задаёт словарь: из её символов по рангам собираются выражения, и правила сборки — грамматика языка. Терм — выражение, обозначающее объект некоторого сорта.

ОПРЕДЕЛЕНИЕ 15.2 Терм

Корректно-сортированный терм сорта σ строится по правилам.

- Переменная любого сорта — терм этого сорта.
- Константа любого сорта — терм этого сорта.
- Правило применения: символ с рангом

$$(\sigma_1, \dots, \sigma_n, \sigma)$$

принимает термы сортов $\sigma_1, \dots, \sigma_n$ и даёт терм сорта σ .

Терм, который нельзя собрать этими правилами, называется **некорректно-сортированным**. В языке такого терма нет.

Пример Сборка терма

В сигнатуре массивов терм

$$\text{read}(\text{write}(a, i, v), i)$$

имеет сорт E. Внутренний `write` собирает массив из переменных подходящих сортов. Внешний `read` извлекает из него элемент. Терм описывает чтение только что записанного элемента — ту операцию, ради которой массивы и входят в язык.

Теперь закрепим, что термы означают: интерпретация приписывает сортам и символам их значения.

ОПРЕДЕЛЕНИЕ 15.3 Интерпретация

Интерпретация сигнатуры приписывает сортам, символам и переменным их значения.

- Каждому сорту — непустое множество элементов, **носитель** сорта.
- Каждому символу f — функцию между носителями:

$$f^I : \sigma_1^I \times \dots \times \sigma_n^I \rightarrow \sigma^I$$

.

- Каждой переменной x — значение из носителя её сорта.

Пример Интерпретация целых чисел

В интерпретации для целочисленной арифметики:

- сорт Int — целые числа,
- символ $+$ — сложение,
- символ $\leq$ — нестрогий порядок.

Переменная может получить значение 6, и тогда терм $2x$ обозначит 12. Условие из введения $-x + y \leq 10 \text{ and } x > 5$ — обретает смысл именно здесь: символы интерпретированы, и вопрос «найдётся ли пара» относится к целым числам.

Сорт — город, а носитель — его население. Сорт Int населён целыми числами. Сорт Bool — двумя истинностными значениями. Жители одного города в другом не живут. Функция — дорога между городами: она везёт аргументы из своих сортов и привозит результат в свой. Если дороги между городами нет, перевозка не состоится — формула не собрана.

Носитель сорта обязан быть непустым: пустой сорт лишил бы значения всякий терм этого сорта, а константа такого сорта не имела бы денотата вовсе.

Семантика связок и кванторов в многосортной логике та же, что в односортной из главы 3. Формула собирается из атомов связками: конъюнкция, дизъюнкция, отрицание, импликация. Квантор $\forall x : \sigma$ пробегает по носителю своего сорта. В этой картине формула — терм булева сорта: предикат применяется к аргументам и возвращает истинностное значение, а связки работают с такими термами, как функции. Разница с односортной логикой — только в сортах: значения переменных живут в разных носителях, и смешать сорта в одном терме нельзя.

Одна сигнатура допускает много интерпретаций. Сорт натуральных чисел можно прочесть иначе: как вычеты по модулю 7. Тогда терм $S(6)$ обозначит ноль. Выполнимость формулы зависит от выбора: в чистой логике законна любая интерпретация, в арифметике — только правильные. Какие интерпретации допустимы — решает теория, и к ней переходит следующий раздел.

§ 15.2 Теории первого порядка

Формула $x + x = 2x$ истинна в арифметике. В чистой логике она лишь выполнима: достаточно выбрать операцию $+$, не обладающую нужными свойствами, и равенство нарушится. Одна и та же запись получает разные ответы — изменился класс допустимых интерпретаций, а формула осталась прежней. Вопрос о выполнимости без этого класса не имеет ответа: он зависит от того, кому разрешено свидетельствовать. Теория — это и есть решение о допуске: она фиксирует класс интерпретаций, и только их голоса учитываются. Сужение класса — цена, которую платят за закреп-

лённый смысл: чем уже класс, тем больше истин, но тем сильнее ограничены модели.

ОПРЕДЕЛЕНИЕ 15.4 Теория

Теория первого порядка — пара $\mathcal{T} = (\Sigma, \mathcal{M})$. Здесь Σ — сигнатура. Класс $\mathcal{M}$ — допустимые интерпретации. Класс $\mathcal{M}$ замкнут относительно переназначения переменных: вместе с каждой интерпретацией он содержит любую интерпретацию, отличающуюся от неё только значениями переменных.

Замкнутость означает, что класс не зависит от конкретных имён переменных: переименование переменных формулы не выводит интерпретацию из класса. Интерпретации класса называют $\mathcal{T}$ -интерпретациями.

Без замкнутости теория вела бы себя по-разному для одинаковых по сути формул: переименование переменных не должно менять ответ о выполнимости. Определение теории через класс интерпретаций, а не через множество аксиом — выбор, продиктованный вопросом SMT. Вопрос главы — существует ли модель, а не выводится ли предложение: процедура решения ищет интерпретацию, и класс интерпретаций отвечает на этот вопрос напрямую. Определение через аксиомы потребовало бы перевода: сначала зафиксировать аксиомы, потом доказывать, что модели аксиом — в точности намеченный класс. Класс интерпретаций задаёт и неаксиоматизируемые теории вроде $\text{Th}(\mathbb{N})$: истина в стандартной модели — тоже теория, хотя аксиом у неё нет. Выбор класса как первоосновы делает определение теории достаточно широким, чтобы охватить и такие случаи.

Сужение класса интерпретаций закрепляет смысл символов. В чистой логике символ $+$ может означать любую бинарную операцию: сложение, умножение, выбор минимума. В теории вещественной арифметики символ $+$ обязан быть сложением вещественных чисел. Символ $\leq$ — обычным порядком. Рассуждение ведётся в этой закреплённой математике, и ответы перестают зависеть от произвола интерпретаций.

Пример Теория вещественной арифметики

Теория $\mathcal{T}_{\text{RA}}$ закрепляет смысл символов:

- сорт Real — вещественные числа,
- символ $+$ — сложение,
- символ $\leq$ — нестрогий порядок.

Формула $\mathcal{T}_{\text{RA}}$ -общезначима:

$$x + y = y + x$$

. Сложение вещественных чисел коммутативно. В чистой логике та же формула общезначимой не была бы: ничто не мешает интерпретировать $+$ некоммутативной операцией.

Класс интерпретаций можно задавать и без перечисления — условиями. Теория частичного порядка: сигнатура с одним предикатом $\leq$, аксиомы — рефлексивность, антисимметричность, транзитивность:

$$\begin{aligned}\forall x. x &\leq x, \\ \forall x, y. (x &\leq y \wedge y \leq x) \rightarrow x = y, \\ \forall x, y, z. (x &\leq y \wedge y \leq z) \rightarrow x \leq z.\end{aligned}$$

Модели этих аксиом — в точности частичные порядки: любые множества с отношением, удовлетворяющим трём условиям. Такой способ задания класса интерпретаций называется *аксиоматизацией*, и ниже ему посвящено отдельное определение.

ОПРЕДЕЛЕНИЕ 15.5 $\mathcal{T}$ -выполнимость

Формула α называется $\mathcal{T}$ -выполнимой, если её выполняет некоторая интерпретация теории.

ОПРЕДЕЛЕНИЕ 15.6 $\mathcal{T}$ -общезначимость

Формула α называется $\mathcal{T}$ -общезначимой, если её выполняют все интерпретации теории.

Теория сужает класс допустимых интерпретаций, поэтому $\mathcal{T}$ -общезначимость — это общезначимость по моделям теории.

Совместность конъюнкции литералов — то же, что $\mathcal{T}$ -выполнимость: набор ограничений *совместен*, если существует интерпретация теории, выполняющая все его литералы. Термины «совместный» и «выполнимый» — синонимы. Первый удобен, когда речь о наборах ограничений.

Пример Выполнимость и общезначимость в вещественной арифметике

- $\mathcal{T}_{\text{RA}}$ -выполнима формула:

$$x + y \leq 1$$

. Модель: $x = 0, y = 0$.

- Та же формула не $\mathcal{T}_{\text{RA}}$ -общезначима. Контрмодель: $x = 5, y = 5$.
- $\mathcal{T}_{\text{RA}}$ -невыполнима формула:

$$\forall x, y. x + y \leq 1$$

. Неравенство нарушается при $x = y = 1$.

- $\mathcal{T}_{\text{RA}}$ -общезначима формула:

$$\forall x. (x + y \leq 1.7) \rightarrow (y \leq 1.7 - x)$$

. Над вещественными числами эти два неравенства равносильны.

Вопрос о $\mathcal{T}$ -выполнимости — алгоритмический, и алгоритм, отвечающий на него, называется *процедурой решения*¹³².

ОПРЕДЕЛЕНИЕ 15.7 Процедура решения

Процедура решения (decision procedure) для класса формул — алгоритм, который для любой формулы из класса завершается и отвечает «да» или «нет».

В этой главе процедуры решения отвечают на вопрос о $\mathcal{T}$ -выполнимости кванторно-свободных формул данной теории.

Правильность процедуры решения складывается из трёх свойств.

1. *Корректность*: ответ «да» процедура даёт только на выполнимые формулы.
2. *Полнота*: каждая выполнимая формула получает ответ «да».
3. *Завершимость*: алгоритм останавливается на любом входе.

Корректность и полнота вместе означают, что ответ всегда верен. Завершимость означает, что ответа можно дожидаться.

Ответ на вопрос о выполнимости в каждой теории даёт своя процедура, и сложность каждой сведена в таблицу следующего раздела.

Пример Процедура решения в действии

- Формула $x + y \leq 10 \wedge x > 5$ выполнима в целых числах. Модель: $x = 6, y = 0$.
- Формула $x + y \leq 10 \wedge x > 5 \wedge y > 6$ невыполнима. Целочисленность усиливает неравенства:

$$x > 5 \Rightarrow x \geq 6$$

$$y > 6 \Rightarrow y \geq 7$$

Тогда сумма не меньше 13, и ограничение нарушено.

Процедура решения для целочисленной линейной арифметики обязана выдать «да» в первом случае и «нет» во втором.

15.2.1 Аксиоматизация

Класс $\mathcal{M}$ в определении теории можно задавать двумя способами. Первый — перечислить все интерпретации класса: такой перечень бесконечен и в конечный текст не помещается. Второй — выписать аксиомы: предложения, которым обязана удовлетворять каждая интерпретация из класса.

¹³²D. Kroening, O. Strichman. «Decision Procedures: An Algorithmic Point of View». 2016; A. Bradley, Z. Manna. «The Calculus of Computation: Decision Procedures with Applications to Verification». 2007.

ОПРЕДЕЛЕНИЕ 15.8 Теория, заданная аксиомами

Теория задана аксиомами $\mathcal{A}$, если её интерпретации — в точности модели аксиом. Модели аксиом — интерпретации, выполняющие каждое предложение из $\mathcal{A}$. Класс моделей обозначают $\text{Mod}(\mathcal{A})$.

ОПРЕДЕЛЕНИЕ 15.9 Аксиоматизируемая теория

Теория аксиоматизируема, если существует множество аксиом, задающее её класс интерпретаций.

Такой способ годится для любой теории: аксиомами служат все её истинные предложения — поэтому о произвольной аксиоматизируемости говорят редко. Содержательное понятие — *рекурсивная* аксиоматизируемость: существует алгоритм, перечисляющий аксиомы. Ниже термин «аксиоматизируемая» понимается в рекурсивном смысле.

Проверим введённое понятие на примере, где оно даёт нетривиальный ответ: теория $\text{Th}(\mathbb{N})$ — множество всех предложений, истинных в стандартной модели натуральных чисел. Набор аксиом у $\text{Th}(\mathbb{N})$ известен: все её истинные предложения, только перечислить их алгоритмом нельзя. Вопрос в том, можно ли задать $\text{Th}(\mathbb{N})$ рекурсивно, и ответ — нет, как показывает теорема Гёделя.

Полная теория — это теория, которая для каждого предложения языка решает его судьбу: для любого предложения φ выводится либо оно само, либо его отрицание. Полная и рекурсивно аксиоматизируемая теория разрешима: ответ о предложении φ сводится к ожиданию вывода — либо самого предложения, либо его отрицания. Один из двух выводов гарантирован полнотой, и перебор доказательств рано или поздно его найдёт.

Первая теорема Гёделя о неполноте утверждает: непротиворечивая рекурсивно аксиоматизируемая теория, в которой выразима арифметика, неполна. Отсюда следует, что $\text{Th}(\mathbb{N})$ не рекурсивно аксиоматизируема. Рассуждаем от противного: $\text{Th}(\mathbb{N})$ по определению полна — для каждого предложения она содержит либо его, либо его отрицание, ведь в стандартной модели истинно ровно одно. Если бы $\text{Th}(\mathbb{N})$ была ещё и рекурсивно аксиоматизируемой, она была бы полной рекурсивно аксиоматизируемой теорией с арифметикой — а это противоречит первой теореме Гёделя. Значит, такого набора аксиом нет.

Разрешимая теория всегда рекурсивно аксиоматизируема: процедура решения перечисляет её общезначимые предложения, и этот список можно принять за аксиомы. Поэтому из отсутствия рекурсивной аксиоматизации следует неразрешимость $\text{Th}(\mathbb{N})$. Тот же вывод даёт теорема Тарского о неопределимости истины (глава 26).

Положительные примеры аксиоматизации — обычные алгебраические теории. Теория частичного порядка из начала раздела аксиоматизируема тремя аксиомами: её модели — в точности частичные порядки. Теория групп задаётся тремя аксиомами:

$$\begin{aligned}\forall x, y, z. (x \cdot y) \cdot z &= x \cdot (y \cdot z), \\ \forall x. (x \cdot e &= x \wedge e \cdot x = x), \\ \forall x \exists y. (x \cdot y &= e \wedge y \cdot x = e).\end{aligned}$$

Модели этих аксиом — в точности группы: аксиомы вырезают из всех интерпретаций сигнатуры $\{\cdot, e\}$ ровно намеченный класс.

Отрицательный пример — та самая $\text{Th}(\mathbb{N})$: её аксиомы не перечислимы алгоритмом, хотя теория задана чётко, как истина в стандартной модели. Рекурсивная аксиоматизируемость — рабочее понятие, потому что рекурсивные аксиомы дают алгоритмический доступ к теории: следствия рекурсивно аксиоматизируемой теории полуразрешимы, перебор всех доказательств рано или поздно находит вывод любого следствия. Арифметика Пеано — рекурсивно аксиоматизируемая теория, и к её аксиоматике мы сейчас переходим.

15.2.2 Арифметика Пеано

ОПРЕДЕЛЕНИЕ 15.10 Арифметика Пеано

Арифметика Пеано $\mathcal{T}_{\text{PA}}$ — теория натуральных чисел. Сигнатура — $\{0, S, +, \times, =\}$. Аксиомы:

- Нуль не является образом наследника:

$$\forall x. S(x) \neq 0$$

- Наследник инъективен:

$$\forall x, y. S(x) = S(y) \rightarrow x = y$$

- Сложение задаётся рекурсивно:

$$x + 0 = x, x + S(y) = S(x + y)$$

- Умножение задаётся рекурсивно:

$$x \cdot 0 = 0, x \cdot S(y) = x \cdot y + x$$

- Схема индукции: для каждой формулы $\varphi(x)$ языка аксиома

$$(\varphi(0) \wedge \forall x. (\varphi(x) \rightarrow \varphi(S(x)))) \rightarrow \forall x. \varphi(x)$$

Скажем эти аксиомы обычными словами. Операция S — «прибавить единицу», а нуль — начало отсчёта: из нуля единицей не вернуться, и разные числа имеют

разных наследников. Сложение и умножение заданы рекурсивно: прибавить ноль — ничего не менять, прибавить наследник — сначала прибавить один шаг, и так далее. Умножение — повторное сложение. Схема индукции — это правило доказательства «для всех»: если свойство верно для нуля и переходит с числа на его наследника, то оно верно для всех чисел.

Схема индукции — бесконечное семейство аксиом, по одной на каждую формулу. Схема — чертёж, а не деталь. По одному чертежу штампуют сколько угодно деталей: для каждой формулы φ схема выдаёт свою аксиому, а формул в языке бесконечно много. Конечно аксиоматизируемой $\mathcal{T}_{\text{PA}}$ не является, но рекурсивно аксиоматизируемой — да: формулы языка перечислимы, и аксиомы перечисляются алгоритмом.

Умножение входит в язык и расширяет выразительность. Формула

$$\forall x, y, z. x \cdot (y + z) = x \cdot y + x \cdot z$$

— теорема $\mathcal{T}_{\text{PA}}$: дистрибутивность доказывается индукцией. Предложение

$$S(S(0)) \cdot S(S(0)) = S(S(S(S(0))))$$

— то есть $2 \cdot 2 = 4$ — доказывается вычислением.

Полноты аксиомы Пеано не дают: $\mathcal{T}_{\text{PA}}$ неполна (первая теорема Гёделя о неполноте, глава 26). У теории есть модели, отличные от стандартной, — *нестандартные*.

В нестандартной модели есть элементы, не достижимые из нуля конечным числом применений S . Они не имеют вида $S(S(\dots S(0)\dots))$ ни с каким конечным числом шагов.

Формула «всякое число есть ноль или чей-то наследник»

$$\forall x. (x = 0 \vee \exists y. x = S(y))$$

доказуема в $\mathcal{T}_{\text{PA}}$ индукцией и потому истинна во всех моделях: она не отличает стандартные элементы от нестандартных.

Формула не отличает стандартные элементы: в нестандартной модели всякий элемент, даже не достижимый из нуля, является наследником некоторого элемента — достаточно взять его предшественника по S . У нестандартного элемента предшественник тоже нестандартный, поэтому цепочка уходит в бесконечность, но каждый её шаг в модели существует.

Доказуемая формула истинна во всех моделях теории: что доказано из аксиом, то выполняется в любой модели (это корректность исчисления, входящая в теорему Гёделя о полноте, глава 3). Различие невыразимо формулой первого порядка: свойство «быть достижимым из нуля конечным числом шагов» в языке теории не формализуемо.

Нестандартные модели разбираются в главе 29. Теория $\mathcal{T}_{PA}$ неразрешима: нет алгоритма, отвечающего на вопрос, доказуемо ли предложение (глава 26).

Сравнение арифметики Пеано с вариантом без умножения указывает на источник неразрешимости.

15.2.3 Арифметика Пресбургера

ОПРЕДЕЛЕНИЕ 15.11 Арифметика Пресбургера

Арифметика Пресбургера $\mathcal{T}_{Pres}$ — теория натуральных чисел: арифметика Пеано без умножения. Сигнатура — $\{0, S, +, =\}$. Аксиомы — те же, что у $\mathcal{T}_{PA}$, кроме аксиом умножения. Схема индукции сохраняется.

Вместе с умножением уходит выразительность. Нельзя записать $x \cdot y$: произведение двух переменных — не терм языка. Нельзя закодировать пары и последовательности чисел: нумерация пар обязана расти квадратично.

На квадрате со стороной n :

$$(n + 1)^2$$

пар, все должны получить разные коды.

А все функции, определяемые в арифметике Пресбургера, растут не быстрее линейной — классический результат, который приводится здесь без доказательства. Квадратичный код в такой арифметике не выразим.

Цена выразительности — выигрыш в разрешимости: $\mathcal{T}_{Pres}$ разрешима, и это результат Пресбургера¹³³. Любая процедура решения полной теории требует в худшем случае времени, растущего как двойная экспонента от длины формулы: нижняя оценка Фишера и Рабина¹³⁴. *Двойная экспонента* — это рост вида 2^{2^n} : степень двойки, показатель которой сам есть степень двойки от длины формулы.

Такой рост несравнимо быстрее обычной экспоненты. При $n = 4$:

$$2^n = 2^4 = 16$$

$$2^{2^n} = 2^{2^4} = 2^{16} = 65536$$

.

Нижняя оценка — ограничение скорости, которого не обойти никаким решателем. Как дорога с замером трассы: сколько ни старайся водитель, быстрее разрешённого на каких-то участках не проехать. Оценка Фишера и Рабина говорит: на некоторых входах любой алгоритм обязан потратить время 2^{2^n} — такой работы требует сам ответ, а качество алгоритма здесь ни при чём.

¹³³М. Presburger. «Über die Vollständigkeit eines gewissen Systems der Arithmetik». 1929.

¹³⁴М. J. Fischer, М. O. Rabin. «Super-Exponential Complexity of Presburger Arithmetic». 1974.

Контраст виден на кванторно-свободном фрагменте: он NP-полон — трудный, но не дважды экспоненциальный. Полная теория Пресбургера на порядок дороже. Кванторно-свободный фрагмент $\mathcal{T}_{\text{Pres}}$ NP-полон. Над целыми числами тот же фрагмент называется LIA (линейная целочисленная арифметика, раздел о линейной арифметике ниже) и остаётся NP-полным (глава 30).

Умножение — источник неразрешимости арифметики. Источник не в одних кванторах: кванторно-свободный фрагмент арифметики Пеано, с умножением, тоже неразрешим. Выполнимость кванторно-свободной формулы — это вопрос о значениях её свободных переменных, при которых она истинна. Навешивая на все свободные переменные квантор существования, получаем экзистенциальное предложение арифметики: $\exists x_1, \dots, x_n. \varphi$. А вопрос об истинности экзистенциальных предложений арифметики — это десятая проблема Гильберта, решённая отрицательно Матиясевичем: алгоритма для него нет. Умножение само по себе, без кванторов, уничтожает разрешимость.

15.2.4 Разрешимость и фрагменты

Теории различаются по разрешимости, и это свойство стоит определить отдельно.

ОПРЕДЕЛЕНИЕ 15.12 Разрешимость теории

Теория $\mathcal{T}$ разрешима, если множество её общезначимых предложений разрешимо: существует завершающаяся процедура, отвечающая «да» или «нет» для любого предложения.

То есть разрешимость теории — это алгоритмический вопрос про её общезначимые предложения: умеет ли какой-то алгоритм за конечное время сказать, общезначимо предложение или нет.

Теория $\mathcal{T}_{\text{RA}}$ разрешима: её полная теория допускает *элиминацию кванторов* (результат Тарского, глава 26). Теория $\mathcal{T}_{\text{PA}}$ неразрешима (глава 26). Между ними — $\mathcal{T}_{\text{Pres}}$: разрешима, но с двойной экспоненциальной нижней оценкой. Для кванторно-свободных формул процедуры решения проверяют выполнимость. Связь с общезначимостью: формула общезначима ровно тогда, когда её отрицание невыполнимо.

Процедуры решения для полных теорий либо отсутствуют, либо дороги. SMT-солверы работают с подмножествами формул теорий — *фрагментами*.

ОПРЕДЕЛЕНИЕ 15.13 Фрагмент теории

Фрагмент теории — синтаксически ограниченное подмножество её формул. *Кванторно-свободный фрагмент* — формулы без кванторов: атомы, связки, отрицание.

Кванторы — единственное, что уносит с собой бесконечность: формула без кванторов говорит о конечном числе термов, и её истинность определяется конечными данными, которые можно перебрать. С квантором вопрос становится утверждением

обо всех элементах области, а все элементы бесконечной области не перечислить. Запрет кванторов — это цена, и платит её выразительность: свойства вида «для всех индексов» (экстенциональность массивов), индукция и любые универсальные законы в кванторно-свободный фрагмент не входят. SMT покупает разрешимость именно этой ценой: фрагмент сохраняет формулы, которые возникают как условия в точке программы, и отбрасывает те, что говорят обо всей математике сразу.

Пример Кванторно-свободные формулы

- Формула $x + y \leq 10 \wedge x > 5$ кванторно-свободна: одни атомы и связки.
- Формула $\forall x \exists y. y > x$ кванторно-свободной не является: в ней два квантора.
- Разностная логика — фрагмент линейной арифметики: её формулы — конъюнкции литералов вида $x - y \leq c$.
- Формула $\neg(x + y \leq 10)$ кванторно-свободна: отрицание висит на атоме, а не на формуле с квантором.

Экскурс в разрешимость подводит к выбору SMT: работать с *фрагментами*, а не с полными теориями. Среди полных теорий есть и неразрешимые (арифметика Пеано), и разрешимые, но очень дорогие (арифметика Пресбургера — двойная экспонента). Ни те, ни другие для практики SMT не годятся. Кванторно-свободные фрагменты разрешимы, и их процедуры имеют практический смысл. Что описывает каждая теория и почём обходится ответ, сведено в таблицу следующего раздела. Когда формула смешивает теории, решатели комбинируются в одну процедуру (раздел о комбинации теорий).

Таков язык SMT: теория, выполнимость по модулю теории, процедура решения и фрагмент, — и теперь посмотрим, какие теории действительно решают SMT-солверы и зачем каждая из них нужна.

§ 15.3 Какие теории решает SMT

Проверка простого фрагмента программы требует сразу трёх умений: сравнивать термы, рассуждать об индексах и помнить о массивах. Нельзя собрать одну универсальную процедуру, которая понимала бы всё сразу: каждая теория навязывает свой способ записи ограничений, и противоречие в каждой ищется в своей структуре. Разность двух переменных — это ребро графа, равенство термов — класс эквивалентности, линейное неравенство — полупространство. SMT-солвер — библиотека процедур, а не один алгоритм, и вопрос главы — какая структура стоит за каждой теорией и где в ней прячется противоречие.

Почти все эти процедуры устроены по одному образцу: ограничения переводятся в структуру — граф, классы эквивалентности, многогранник — и противоречие ищется в ней. Структура — *верстак*, на который раскладывают ограничения. Пока они лежат кучей в формуле, противоречие не видно. Разложив их на граф или многогранник, глаз находит его сразу — как мастер видит трещину в балке, когда

деталь положена на стол. У каждой теории свой верстак: у разностной логики граф, у EUF классы эквивалентности, у арифметики многогранник. Дальше мы идём по этим верстакам по очереди.

Разностная логика (DL) — теория ограничений вида $x - y \leq c$: разность двух переменных не превосходит константы. Это простейшая теория раздела, и структура у неё самая наглядная — граф. Такие ограничения возникают в планировании и системах реального времени: сроки, временные метки, привязка событий. Анализ сроков сводит проверку расписания к совместности набора разностных ограничений.

Теория равенства с неинтерпретируемыми функциями ($\mathcal{T}_{\text{EUF}}$) — это равенство и произвольные функции, о которых известно только одно: равные аргументы дают равные значения. Структура здесь — классы эквивалентности термов: противоречие возникает, когда равенства склеивают два терма, объявленных разными. Так доказывают эквивалентность программ: если две функции возвращают равные результаты на равных входах, их можно считать равными, не заглядывая в тела. Компилятор пользуется этим в оптимизации общих подвыражений: доказав равенство двух вхождений подвыражения, он вычисляет его один раз.

Линейная вещественная арифметика (LRA) — ограничения вида $2x + 3y \leq 5$ над вещественными переменными. Она описывает непрерывные величины: время, расстояние, скорость, координаты. Проверка совместности линейных ограничений — задача линейного программирования, одна из самых изученных в вычислительной математике: к ней сводятся геометрические задачи и анализ систем реального времени. Структура — выпуклый многогранник: ограничения вырезают из пространства область допустимых точек, и противоречие означает пустоту этой области.

Линейная целочисленная арифметика (LIA) — те же ограничения над целыми числами. Условия верификации программ населены целыми: счётчики циклов, индексы массивов, размеры буферов. Формула из начала главы, $x + y \leq 10 \wedge x > 5$, — условие в LIA. Разница с вещественным случаем — какие точки области допустимы: целые, а не любые.

Битовые векторы (BV) — арифметика фиксированной разрядности: сложение 32-битных чисел, сдвиги, побитовые операции. Это в точности семантика процессора (глава 11), поэтому BV описывает аппаратуру и низкоуровневый код. Проверка процессорных блоков, протоколов и драйверов переводится в выполнимость формул над битовыми векторами. Структура здесь булева: каждый бит — отдельная переменная, и арифметика разворачивается в схему над ними.

Теория массивов ($\mathcal{T}_{\text{AX}}$) моделирует память: массив читается и пишется по индексу. Чтение по индексу после записи по тому же индексу возвращает записанное значение. Оперативная память программы — это массив, поэтому символьное исполнение и верификация программ описывают память именно так. Утверждение «после записи по адресу чтение по тому же адресу вернёт записанное» — типичный

литерал теории массивов. Противоречие здесь ищется среди аксиом доступа: для каждого чтения из записи инстанцируется правило read-over-write.

Три строки таблицы помечены как NP-полные, и стоит сказать, что за этим стоит. NP-полнота — это статус самых трудных задач класса NP: решение такой задачи можно проверить быстро, а к ней сводится любая задача из NP. NP-полная задача не решается за полиномиальное время, пока $P \neq NP$. На практике её решают перебором, и в худшем случае он экспоненциальный. Здесь достаточно помнить из главы о SAT: NP-полнота описывает худший случай, а реальные формулы часто решаются быстро.

Теория	Что описывает	Кванторно-свободный фрагмент
$\mathcal{T}_{EUF}$	равенство и неинтерпретируемые функции	разрешим, $O(n \log n)$
LRA	линейная арифметика над $\mathbb{R}$	разрешим, полиномиально
LIA	линейная арифметика над $\mathbb{Z}$ (Пресбургер)	NP-полон (глава 30)
DL	разности $x - y \leq c$	разрешим, полиномиально
BV	битовые векторы фиксированной длины	NP-полон (глава 30)
AX	массивы (read/write)	NP-полон (глава 30)
$\mathcal{T}_{PA}$	арифметика Пеано (с умножением)	неразрешим (десятая проблема Гильберта)

Почти каждая теория из таблицы получит собственную процедуру решения. Исключение — арифметика Пеано, отмеченная как неразрешимая. Сложности в таблице — анонс: каждая разрешимая строка будет разобрана в своём разделе (LRA и LIA — в одном), и там станет видно, откуда берётся оценка. Сейчас достаточно понять, что кванторно-свободные фрагменты разрешимы, а их сложность колеблется от полиномиальной до NP-полной.

Осталось договориться, в каком виде процедура теории отвечает на вопрос о выполнимости. Достаточно уметь решать конъюнкции литералов: ветка DPLL-поиска (глава 14) — это частичное присваивание, то есть конъюнкция литералов, уже принятых как истинные. Решатель не строит ветки заранее — каждая ветка порождается поиском, и в каждой ветке theory-солвер проверяет только текущую конъюнкцию. Поэтому вопрос к теории звучит так: совместна ли текущая конъюнкция литералов.

ОПРЕДЕЛЕНИЕ 15.14 Theory-солвер

Theory-солвер ($\mathcal{T}$ -солвер) — процедура, решающая выполнимость **конъюнкций литералов** в теории. На входе — набор литералов, например:

$$x + y \leq 10, f(a) = b$$

. На выходе — совместен набор или нет.

Конъюнкция литералов **совместна**, если она $\mathcal{T}$ -выполнима: существует интерпретация теории, выполняющая все литералы.

ТЕОРЕМА 15.1 Сведение к конъюнкциям литералов

Выполнимость кванторно-свободной формулы в теории разрешима тогда и только тогда, когда разрешима выполнимость конъюнкций литералов в той же теории.

Доказательство

Докажем эквивалентность в обе стороны.

Покажем сначала, что из разрешимости выполнимости конъюнкций литералов следует разрешимость выполнимости кванторно-свободных формул. Кванторно-свободную формулу можно привести к ДНФ (дизъюнктивной нормальной форме) — дизъюнкции конъюнкций литералов, например $(a \wedge b) \vee (c \wedge d)$. Дизъюнкция выполнима тогда и только тогда, когда выполнима хотя бы одна её составляющая: достаточно взять модель одной конъюнкции. Поэтому выполнимость формулы сводится к перебору её конъюнкций и проверке каждой из них.

Обратно: конъюнкция литералов — частный случай кванторно-свободной формулы, поэтому из разрешимости последних следует и разрешимость конъюнкций.

Для каждой из этих теорий есть своя процедура — тот самый theory-солвер из определения выше. Разберём их по очереди, начиная с простейшей — разностной логики.

§ 15.4 Разностная логика

Моменты начала двух событий обозначим переменными x, y . Требование «первое событие не более чем на c минут позже второго» записывается так:

$$x - y \leq c$$

.

Ограничение связывает только разность двух переменных, и такие условия называют *разностными ограничениями*. На числовой оси каждое такое условие — утверждение о расстоянии: одна точка не дальше заданного порога вправо от другой.

Отрицательное c разворачивает смысл. Неравенство

$$x - y \leq -2$$

равносильно $y - x \geq 2$: первое событие на две минуты раньше второго.

Расписания — самый наглядный источник разностных ограничений: сроки, временные метки, привязка событий. Те же условия возникают в планировании, в системах реального времени и в верификации, когда нужно убедиться, что все временные требования выполнимы одновременно. Вопрос всегда один: существует ли набор моментов времени, при котором каждое ограничение выполнено.

Разностная логика — простейшая теория с содержательной процедурой решения.

ОПРЕДЕЛЕНИЕ 15.15 Разностная логика

Разностная логика (DL, от англ. difference logic) — фрагмент линейной арифметики, обозначаемый $\mathcal{T}_{DL}$.

Её формулы — конъюнкции литералов вида $x - y \leq c$: разность двух переменных сравнивается с константой.

Ограничения читаются как над целыми, так и над вещественными числами: DL — фрагмент и целочисленной, и вещественной линейной арифметики.

Присваивание значений переменным — это набор *координат* на оси: у каждой переменной своя точка. Ограничение $x - y \leq c$ превращается в условие на расстояние между точками. Конъюнкция выполнима, когда все точки удаётся расставить по оси так, чтобы каждое условие расстояния выполнилось.

Пример Разностная логика: координаты

Конъюнкция

$$(x - y \leq 2) \wedge (y - z \leq 3) \wedge (x - z \leq 5)$$

выполнима. Расставим точки: $z = 0, y = 3, x = 5$. Проверяем каждое ограничение:

- $x - y = 5 - 3 = 2 \leq 2$ — выполняется.
- $y - z = 3 - 0 = 3 \leq 3$ — выполняется.
- $x - z = 5 - 0 = 5 \leq 5$ — выполняется.

На оси точки расставлены по возрастанию координат. Третье ограничение следует из первых двух: $x - z = (x - y) + (y - z) \leq 2 + 3 = 5$.

Выполнимость любой разностной конъюнкции решается одинаково. Процедура решения для DL — три шага:

1. Нормализация: каждый литерал приводится к виду $u - v \leq c$. Равенство $u - v = c$ распадается на два неравенства:

$$(u - v \leq c) \wedge (v - u \leq -c)$$

- Строгое неравенство над целыми числами заменяется на нестрогое с $c - 1$.
2. Построение графа ограничений: вершины — переменные. Для каждого ограничения

$$u - v \leq c$$

рисуетя ребро с весом c от вычитаемого к уменьшаемому. Ориентация — ход стрелки: из вычитаемого можно попасть в уменьшаемое, сдвинувшись вправо не больше чем на c . Путь по рёбрам превращается в цепочку смещений: каждый шаг добавляет свой сдвиг. Замкнутый путь обязан вернуться в ту же точку, а сумма смещений по кругу равна нулю, — и если она отрицательна, вернуться невозможно. Отрицательный цикл — дорога, которая уводит всё дальше, и конца ей нет.

3. Поиск *отрицательного цикла*: алгоритм Bellman–Ford¹³⁵ находит цикл с отрицательной суммой весов.

Разберём, почему нормализация устроена именно так. Равенство $u - v = c$ означает сразу два условия:

$$(u - v \leq c) \wedge (u - v \geq c)$$

. Второе переписывается как $v - u \leq -c$: умножение на минус единицу меняет знак неравенства. Так равенство распадается на пару нестрогих неравенств того же вида $u - v \leq c$.

Над целыми числами

$$u - v < c \Leftrightarrow u - v \leq c - 1$$

. Причина в том, что разность целых чисел — целое, и ближайшее меньшее число — это $c - 1$.

Над вещественными числами такой замены нет: между константой и любой меньшей константой бесконечно много чисел. Над вещественными числами строгие неравенства обрабатывают иначе: с символьной поправкой ε .

Путь в графе накапливает ограничения. Проследим это по шагам. Выпишем неравенства пути $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$:

$$v_1 - v_0 \leq c_1, v_2 - v_1 \leq c_2, \dots, v_k - v_{k-1} \leq c_k$$

.

Сложим левые и правые части отдельно: $(v_1 - v_0) + (v_2 - v_1) + \dots + (v_k - v_{k-1}) \leq c_1 + \dots + c_k$. Промежуточные переменные входят в сумму с плюсом и с минусом и сокращаются попарно. Остаётся $v_k - v_0 \leq c_1 + \dots + c_k$.

ТЕОРЕМА 15.2 Критерий выполнимости разностной конъюнкции

Конъюнкция разностных ограничений выполнима тогда и только тогда, когда в её графе ограничений нет отрицательного цикла. Если отрицательных циклов нет, кратчайшие расстояния от источника дают выполняющее присваивание.

¹³⁵R. Bellman. «On a Routing Problem». Quarterly of Applied Mathematics, 1958; L. R. Ford Jr. «Network Flow Theory». RAND Corporation, 1956.

Набросок доказательства

Докажем эквивалентность в обе стороны.

Пусть конъюнкция выполнена некоторым присваиванием, а в графе есть цикл с суммой весов $w < 0$. Сложим неравенства вдоль его рёбер, как для пути выше: левые части сокращаются, и получается $0 \leq w$ — противоречие. Значит, у выполнимой конъюнкции отрицательных циклов нет.

Обратно: пусть отрицательных циклов нет. Добавим источник с нулевыми рёбрами во все переменные и возьмём кратчайшие расстояния $d(v)$ от источника. Источник нужен, чтобы расстояние нашлось у каждой переменной: его рёбра достигают всех вершин. Новых отрицательных циклов он не создаёт: в источник не входит ни одно ребро, поэтому цикл через него невозможен. Для каждого ребра $v \rightarrow u$ с весом c кратчайший путь до u не длиннее кратчайшего пути до v , продолженного этим ребром:

$$d(u) \leq d(v) + c$$

то есть $d(u) - d(v) \leq c$ — ровно ограничение, закодированное ребром $v \rightarrow u$. Присваивание $x = d(x)$ выполняет все рёбра, а значит, и все ограничения конъюнкции.

Проверим критерий на конъюнкции, которая невыполнима.

Пример Разностная логика: отрицательный цикл

Рассмотрим конъюнкцию

$$(x - y = 5) \wedge (z - y \geq 2) \wedge (z - x > 2) \wedge (w - x = 2) \wedge (z - w < 0)$$

. Нормализуем каждый литерал к виду $u - v \leq c$:

- $x - y = 5$ распадается на два неравенства:

$$(x - y \leq 5) \wedge (y - x \leq -5)$$

.

- $z - y \geq 2$ переписывается:

$$y - z \leq -2$$

.

- $z - x > 2$ переписывается:

$$x - z \leq -3$$

. Причина — целочисленность: $z - x \geq 3$.

- $w - x = 2$ распадается на два неравенства:

$$(w - x \leq 2) \wedge (x - w \leq -2)$$

- $z - w < 0$ переписывается:

$$z - w \leq -1$$

По правилу построения графа ограничению соответствует ребро с весом c : от вычитаемого к уменьшаемому. Три ребра образуют цикл:

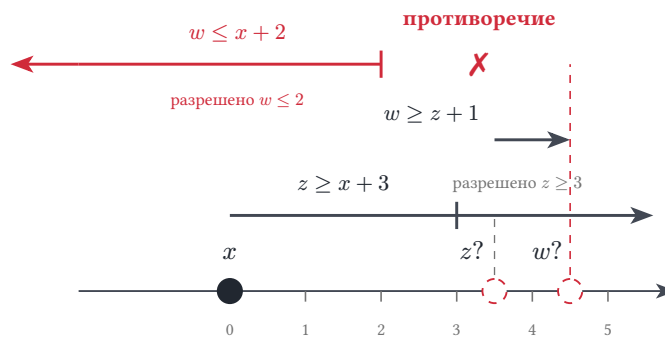
- ребро $x \rightarrow w$ с весом два. Оно построено из $w - x \leq 2$.
- ребро $w \rightarrow z$ с весом минус один. Оно построено из $z - w \leq -1$.
- ребро $z \rightarrow x$ с весом минус три. Оно построено из $x - z \leq -3$.

Сумма весов цикла отрицательна:

$$+2 - 1 - 3 = -2 < 0$$

. Ограничения вдоль цикла складываются. Сумма левых частей обращается в ноль, а сумма правых отрицательна: получается $0 \leq -2$ — противоречие. Конъюнкция невыполнима.

Координатное прочтение того же примера — на рис. 77. Зафиксируем сдвиг начала отсчёта, положив $x = 0$. Из ограничений следует $z \geq 3$, $w \geq z + 1$, $w \geq 4$, а красная граница допускает лишь $w \leq 2$: противоречие. На рисунке взята пробная позиция $z = 3.5$, при которой требуемое $w \geq 4.5$ выходит за красный предел.



$$\text{требуется } w \geq 4.5, \text{ но разрешено } w \leq 2 \Rightarrow -3 - 1 + 2 = -2 < 0$$

Рис. 77. Координатное прочтение примера с отрицательным циклом.

Алгоритм Bellman–Ford находит отрицательный цикл за $O(V \cdot E)$ операций. Здесь V — число переменных. Число E — рёбра после нормализации.

Оценка складывается из раундов расслабления: их ровно V , и в каждом проходят по всем рёбрам. Один раунд распространяет знания о кратчайших расстояниях на один шаг пути: после $V - 1$ раундов до любой переменной доходят все пути, а ещё один раунд нужен как контрольный звонок. Если и после него расстояние уменьшилось, значит, есть цикл с отрицательной суммой — и Bellman–Ford его находит.

Для теории сложности (глава 30) это означает: выполнимость разностной конъюнкции — полиномиальная задача, в отличие от NP-полного кванторно-свободного фрагмента LIA.

Над вещественными числами процедура остаётся полиномиальной. Строгое неравенство не заменяется нестрогим с конкретной константой: между константой и любым меньшим числом бесконечно много значений. Процедура работает с символической бесконечно малой поправкой ε .

Неравенство

$$u - v < c$$

представляется как

$$u - v \leq c - \varepsilon$$

: поправка — формальный символ, меньший любого положительного числа, и алгоритм оперирует им, не выбирая значения.

Модель, выданная такой процедурой, удовлетворяет исходным строгим неравенствам, а оценка сложности не меняется. С поправкой отрицательным считается и цикл с нулевой суммой констант, в котором есть строгое ребро.

Полная арифметика неразрешима (глава 26), а разностный фрагмент решается за полиномиальное время: ограничения превращаются в рёбра графа, и задачу решает алгоритм Bellman–Ford.

Разностная логика — ядро многих SMT-приложений: планирование, системы реального времени, проверка временных свойств. Как процедура всякой теории, DL-процедура работает как theory-солвер. Как SAT-решатель оркеструет такие солверы в цикле DPLL(T), покажет один из следующих разделов.

Проверка Разностная логика

1. Как по ограничению $u - v \leq c$ строится ребро графа ограничений: между какими вершинами и с каким весом?
2. На какие два неравенства распадается равенство $u - v = c$ при нормализации?
3. Почему строгое неравенство над целыми числами заменяется на нестрогое с $c - 1$, а над вещественными такая замена невозможна?

Следующая теория главы — равенство с неинтерпретируемыми функциями. Там стратегия та же: ограничения переводятся в структуру, и противоречие ищется в ней. Роль графа ограничений играют классы эквивалентности термов.

§ 15.5 Равенство с неинтерпретируемыми функциями

Компилятор постоянно доказывает равенство термов. Примеры: совпадение значений функции на равных аргументах, замена повторного вычисления уже посчитанным значением. Тела функций при этом неизвестны, а проверить равенство нужно.

Исключение общих подвыражений (common subexpression elimination) удаляет повторяющийся фрагмент вычисления, полагаясь на то, что равные термы действительно равны. Распространение констант (constant propagation) заменяет переменную её известным значением — тоже по равенству. Дедупликация повторных вычислений сводится к тому же вопросу о совпадении значений термов. Во всех случаях детали вычисления не важны: важно только, равны ли термы.

Решение — отвлечься от тел функций. Функция становится чёрным ящиком, о котором известно единственное правило: равные аргументы дают равные значения. Это правило и есть *конгруэнтность*, а теория, в которой функции ничего больше не умеют, называется теорией равенства с неинтерпретируемыми функциями.

ОПРЕДЕЛЕНИЕ 15.16 Теория EUF

Теория равенства с неинтерпретируемыми функциями $\mathcal{T}_{\text{EUF}}$ имеет сигнатуру с равенством и произвольными функциональными символами.

Её аксиомы — рефлексивность, симметричность и транзитивность равенства, а также **конгруэнтность**: если аргументы равны, то и значения функции равны.

$$x_1 = y_1, \dots, x_n = y_n \Rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$$

.

Функции здесь неинтерпретируемые: про них известно только правило конгруэнтности и ничего больше.

Слово «неинтерпретируемые» означает, что за символом f не закреплено никакой конкретной функции. Одна и та же формула верна в теории $\mathcal{T}_{\text{EUF}}$ независимо от того, какая функция скрывается за символом. Это и делает теорию *корректной абстракцией*: всё, что выведено в ней, остаётся верным для любого конкретного определения f .

Пример Congruence closure

Проверим выполнимость

$$(a = b) \wedge (f(a) = b) \wedge \neg(g(a) = g(f(a)))$$

. Начнём с двух классов: $\{a, b\}$. Второй класс — $\{f(a), b\}$. Оба порождены первыми двумя равенствами.

Классы делят общий элемент b , а равенство транзитивно: равные термы склеиваются в один класс. Поэтому все три терма равны и образуют один класс $\{a, b, f(a)\}$. Теперь конгруэнтность даёт $g(a) = g(f(a))$: равные аргументы дают равные значения. Это прямо противоречит третьему литералу $\neg(g(a) = g(f(a)))$, значит формула невыполнима.

Процесс слияния классов показан на рис. 78: два класса делят общий элемент b и сливаются в один, а конгруэнтность даёт $g(a) = g(f(a))$ — прямое противоречие третьему литералу.

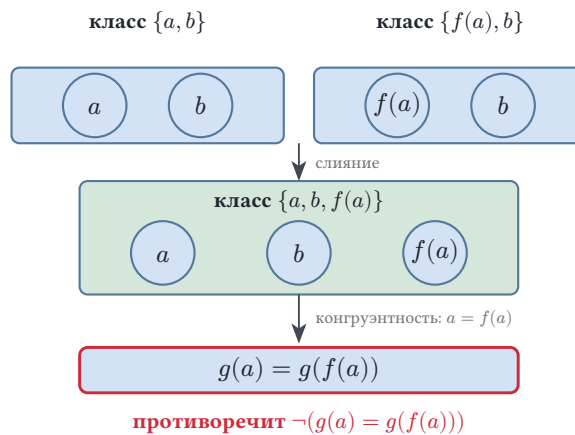


Рис. 78. Слияние классов в congruence closure.

Третий литерал $\neg(g(a) = g(f(a)))$ и создаёт конфликт: без него противоречия не возникает. Формула

$$(a = b) \wedge (f(a) = b)$$

выполнима. Модель простая: $a = b = f(a) = 0$. Функция g в этой точке принимает любое значение. Оба равенства сводятся к $0 = 0$, и никакой отрицательный литерал не заставляет их нарушиться. Так выглядит разница между невыполнимой формулой и её выполнимым подмножеством: конфликт появляется ровно тогда, когда конгруэнтность сталкивается с неравенством.

АЛГОРИТМ Congruence closure

1. Взять все термы формулы как отдельные классы эквивалентности.
2. Для каждого явного равенства $s = t$ слить классы обоих термов.
3. Повторять: если аргументы функции попарно равны, а оба вхождения функции присутствуют в формуле, слить их классы.
4. Если неравенство $s \neq t$ получило оба терма в одном классе — конфликт, формула невыполнима.

УТВЕРЖДЕНИЕ 15.3 Корректность congruence closure

Алгоритм congruence closure останавливается за полиномиальное число шагов. Его ответ верен: конъюнкция литералов невыполнима в $\mathcal{T}_{\text{EUF}}$ тогда и только тогда, когда после слияний термы некоторого неравенства $s \neq t$ оказались в одном классе.

Доказательство

Докажем оба утверждения: остановку с оценкой и совпадение ответа с выполнимостью.

Остановка. Каждое слияние склеивает два разных класса в один и уменьшает их число на единицу. Число классов не превосходит числа термов формулы, поэтому и слияний не больше, чем термов. Между слияниями алгоритм проходит по термам формулы, так что общее число шагов полиномиально.

Невыполнимость при нарушенном неравенстве. Пусть термы s и t попали в один класс. В любой интерпретации, выполняющей конъюнкцию, истинны все её равенства и аксиома конгруэнтности, поэтому термы одного класса получают равные значения. Тогда равенство $s = t$ обязано выполняться — противоречие с литералом $s \neq t$.

Выполнимость в противном случае. Пусть ни одно неравенство не нарушено. Наделим каждый класс собственным значением и интерпретируем функции по классам: значение функции на наборе классов — класс составного терма. Конгруэнтность гарантирует, что результат не зависит от выбора представителей. Все равенства выполнены по построению, а неравенства — потому, что их термы лежат в разных классах.

На практике слияние классов реализуют структурой данных union-find (система непересекающихся множеств): она быстро отвечает, в каком классе лежит терм, и склеивает два класса в один.

У каждого класса есть старший — представитель, как бригадир у артели. Вопрос «в каком классе терм» — это вопрос «чей бригадир у терма»: запрос находит его, идя по цепочке подчинения. Слияние двух классов — сведение двух артелей в одну: бригадир одной становится подчинённым бригадира другой. *Сжатие путей* — разовое переподчинение: после слияния каждый рабочий запоминает бригадира напрямую, и следующий запрос идёт в один шаг. Так запросы и слияния становятся почти мгновенными, и вся работа укладывается в почти линейное время.

Полный алгоритм, который ещё и ищет конгруэнтные пары термов, работает за $O(n \log n)$ (Дауни–Сети–Тарьян) — уточнённая оценка всего процесса.

Кванторно-свободная выполнимость в $\mathcal{T}_{\text{EUF}}$ разрешима за полиномиальное время. Полная теория чистого равенства (без функциональных символов) тоже разрешима, а с неинтерпретируемыми функциями и кванторами она уже неразрешима.

Механизм тот же, что у массивов: функция кодирует пары, а кванторы по парам дают достаточно выразительности для арифметики. Теория наследует её неразрешимость (теорема Чёрча, глава 26). SMT-солверы поэтому работают именно с кванторно-свободным фрагментом EUF: он возникает из верификационных условий и решается быстро.

Есть и другой взгляд на тот же процесс — *сведение Аккермана*. Каждое вхождение функции заменяется свежей переменной, а равенство с исходным термом хранит связь. Конгруэнтностное ограничение — договор между двумя вхождениями: если аргументы равны, то и результаты обязаны быть равны. После такой замены остаётся чистое равенство, которое решает union-find.

Пример Сведение Аккермана в действии

Формула $(f(a) = b) \wedge (a = b) \wedge \neg(f(b) = b)$. Вхождения $f(a)$, $f(b)$ заменяем свежими переменными u , v . Связи $u = f(a)$, $v = f(b)$ хранят исходные термы. Конгруэнтность становится условием $(a = b) \rightarrow (u = v)$. Из $a = b$ и $u = b$ по условию следует $u = v$, а значит $v = b$ — прямое противоречие с $\neg(f(b) = b)$. После замены осталось чистое равенство, и конфликт виден без функций.

Практические солверы работают с общим DAG термов, где совпадающие подтермы представлены один раз, — по сути то же слияние классов.

Проверка Равенство с неинтерпретируемыми функциями

1. Какое единственное правило связывает значения неинтерпретируемой функции со значениями её аргументов?
2. Почему конфликт в congruence closure появляется только тогда, когда конгруэнтность сталкивается с неравенством?
3. Во что превращается конгруэнтность при сведении Аккермана и какая структура данных решает оставшееся чистое равенство?

Теория $\mathcal{T}_{\text{EUF}}$ рассуждает только о равенстве термов. Арифметические утверждения вида $x + y \leq 10$ она не видит. Следующая теория — линейная арифметика — добавляет к равенству сложение и порядок.

§ 15.6 Линейная арифметика

Условие $x + y \leq 10 \wedge x > 5$ из вступления — типичное *верификационное условие*: проверка границ массива, счётчика цикла или свойства целочисленной программы. Такие условия говорят о сумме переменных с числовыми коэффициентами. Это и есть предмет линейной арифметики.

Слово «линейная» означает: терм — сумма переменных, умноженных на константы. Терм $2x$ допустим. Терм $x \times y$ — нет: умножение переменной на переменную выводит терм из класса линейных и радикально меняет сложность задачи.

ОПРЕДЕЛЕНИЕ 15.17 LRA

Линейная вещественная арифметика LRA — теория над вещественными числами.

Символы — сложение, вычитание, сравнение и умножение на константы (для каждой константы своя операция). Термы — суммы вида $c_1x_1 + \dots + c_nx_n + d$: линейные комбинации переменных с константами.

Умножение переменных друг на друга не входит в сигнатуру.

ОПРЕДЕЛЕНИЕ 15.18 LIA

Линейная целочисленная арифметика LIA — та же линейная арифметика, но над целыми числами. Термы — те же суммы $c_1x_1 + \dots + c_nx_n + d$. Сорт `Int` интерпретируется как целые числа. Переменные принимают только целые значения.

Пример Выполнимость в LRA

- Формула над LRA выполнима:

$$x + y \leq 10 \wedge x > 5$$

. Модель: $x = 6, y = 0$.

- Формула $x \times y = 1$ линейной не является: в ней переменные перемножаются, и её выполнимость — задача другого, более трудного класса.

Проверка выполнимости системы линейных неравенств — это вопрос о совместности системы над $\mathbb{R}$, то есть задача линейного программирования о существовании допустимой точки. Сам вопрос о совместности решается за полиномиальное время: метод эллипсоидов Хачияна¹³⁶ и методы внутренней точки Кармаркара¹³⁷ дают полиномиальные алгоритмы.

Геометрия объясняет, почему совместность решается быстро. Система неравенств задаёт выпуклый многогранник, и вопрос один: пуст он или нет. Метод эллипсоидов держит вокруг многогранника эллипсоид и на каждом шаге отсекает от него половину плоскостью нарушенного неравенства. Объём нового эллипсоида убывает в геометрической прогрессии: он умножается на постоянный для данной размерности множитель, меньший единицы. За полиномиальное число шагов либо находится точка, либо доказываемая пустота. Методы внутренней точки прокладывают путь внутри многогранника, минуя вершины: направление на каждом шаге выбирается гладко, без прыжков по рёбрам.

¹³⁶L. G. Khachiyan. «A Polynomial Algorithm in Linear Programming». Soviet Mathematics Doklady, 1979.

¹³⁷N. Karmarkar. «A New Polynomial-Time Algorithm for Linear Programming». Combinatorica, 1984.

При этом классический симплекс-метод в худшем случае экспоненциален: пример Клее–Минти¹³⁸ строит многогранник, на котором симплекс перебирает экспоненциально много вершин. Пример Клее–Минти — слегка скошенный гиперкуб: рёбра чуть наклонены, и симплекс, шагая по рёбрам, блуждает зигзагом по всем 2^n вершинам. В обычном гиперкубе путь от старта к оптимуму короткий, а наклон рёбер превращает его в длинный серпантин. На практике симплекс работает быстро, и SMT-солверы (в том числе инкрементальный симплекс Z3) используют именно симплекс-подобные процедуры. Полиномиальность задачи не противоречит экспоненциальности конкретного алгоритма: для одних входов симплекс мгновенен, для специально построенных — катастрофически медленен.

Целочисленная арифметика LIA труднее вещественной. Задача та же: найти допустимое решение, но теперь координаты обязаны быть целыми.

Пример Граница между LRA и LIA

Над LRA система

$$2x \leq 1, 2x \geq 1$$

выполнима: возьмём $x = 0.5$.

Над LIA она невыполнима. Выражение $2x$ при целочисленном значении переменной всегда чётно, а система требует равенства единице: решения нет. Вещественное решение существует, но целочисленного двойника у него нет: дробная часть области решений исчезает, когда допустимы только целые точки.

Кванторно-свободный фрагмент LIA NP-полон (глава 30). Фрагмент LRA решается за полиномиальное время.

Целые труднее вещественных по одной причине: вещественная задача — поиск точки в выпуклом многограннике, и её решает непрерывный алгоритм. Целочисленная задача — поиск точки с целыми координатами, то есть дискретный перебор кандидатов, среди которых приходится выбирать. Непрерывная геометрия здесь мало помогает: решение лежит в решётке целых точек, а не в континууме.

Механизм NP-трудности — булев выбор. Переменная, ограниченная парой неравенств $0 \leq x \leq 1$, принимает только два значения. Каждая булева формула из главы о SAT выражается целочисленными ограничениями. Кодирование дизъюнкта: сумма литералов не меньше 1 (литерал — переменная, отрицание — дополнение до единицы).

Пример Булева формула как целочисленные ограничения

Переменные $x_1, x_2, x_3 \in \{0, 1\}$ кодируют булевы значения, отрицание — дополнение: $\overline{x_i} = 1 - x_i$. Дизъюнкт $(x_1 \vee \overline{x_2} \vee x_3)$ становится неравенством

¹³⁸V. Klee, G. J. Minty. «How Good is the Simplex Algorithm?». 1972.

$$x_1 + (1 - x_2) + x_3 \geq 1.$$

При $x_1 = 1, x_2 = 0, x_3 = 0$ сумма равна $1 + 1 + 0 = 2 \geq 1$ — дизъюнкт выполнен. При $x_1 = 0, x_2 = 1, x_3 = 0$ сумма равна $0 + 0 + 0 = 0$ — дизъюнкт нарушен. Каждая скобка КНФ превращается в одно неравенство, и выполнимость булевой формулы — это выполнимость системы целочисленных неравенств.

Выполнимость булевых формул NP-полна, а вместе с ней NP-полна и выполнимость в LIA.

Для LIA применяют два семейства методов. *Ветвление и границы* (branch-and-bound) находят вещественное решение, а затем разбивают область ветвями, пока не получают целую точку. *Отсечения* (cutting planes) добавляют к системе неравенств новые ограничения, отсекающие нецелые решения, пока не останется целая вершина.

Отсечение — новый барьер, который режет многогранник, отбрасывая нецелую вершину и не задевая ни одной целой точки. После конечного числа таких резов остаётся многогранник, у которого каждая вершина целая, — и целочисленное решение найдено.

Классическое семейство таких отсечений — отсечения Гомори¹³⁹: они строятся из строк симплекс-таблицы и отрезают дробную часть решения.

Пример Отсечение в действии

Система $x + y \leq 3.5, x, y \geq 0$ над целыми числами. Вещественный решатель выдаст, например, точку $(3.5, 0)$ с дробной координатой. Неравенство $x + y \leq 3$ — отсечение: всякая целая точка области ему удовлетворяет (сумма целых координат не превосходит 3.5, значит, не превосходит 3), а вершина $(3.5, 0)$ отрезана. После добавления отсечения целое решение видно сразу: например, $x = 3, y = 0$.

Завершимость обоих методов — нетривиальная теорема: ветвление исчерпывает конечное число целых точек в ограниченной области, а отсечения Гомори при целочисленной правой части гарантированно сходятся.

Полная теория вещественных чисел с умножением и порядком разрешима (теорема Тарского, глава 26). Для SMT это отвлечённый факт: солверы работают с кванторно-свободной линейной арифметикой, где методы проще и быстрее.

Линейная арифметика предполагает переменные неограниченной точности. Процессор же оперирует числами фиксированной разрядности, и эта арифметика живёт по своим законам. К ней мы и переходим.

¹³⁹R. E. Gomory. «Outline of an Algorithm for Integer Solutions to Linear Programs». Bulletin of the American Mathematical Society, 1958.

§ 15.7 Битовые векторы

Процессор складывает числа фиксированной ширины, и сумма может не поместиться в отведённые биты. Четырёхбитное число 15 плюс единица: результат *переполнился* и обнулится. Пятибитная запись результата не помещается в четыре бита, и в них остаются одни нули. Аппаратура и низкоуровневое программное обеспечение живут именно в этой семантике (глава 11): целые здесь свёрнуты в кольцо по модулю 2^n , а не неограниченные.

ОПРЕДЕЛЕНИЕ 15.19 Битовые векторы

Битовый вектор — последовательность битов фиксированной длины n .

Теория BV — арифметика по модулю 2^n : сложение, вычитание и умножение с переполнением, побитовые операции и знаковые или беззнаковые сравнения, как в процессоре.

Разницу с целой арифметикой показывает пример.

Пример Переполнение и выполнимость

Формула $x + 1 = 0$ над 4-битными векторами выполнима. Возьмём $x = 15$: двоичная запись — четыре единицы. Сложение даёт пятибитную запись, а усечённый до четырёх битов результат — ноль.

Над натуральными числами формула $x + 1 = 0$ невыполнима. Никакое натуральное число её не удовлетворяет.

Над целыми числами единственное решение $x = -1$ лежит вне диапазона четырёхбитного вектора. Дело в ширине, а не в конкретном значении: битовый вектор — величина фиксированной разрядности, и именно это делает BV отдельной теорией.

Формулы над битовыми векторами решают приёмом *bit-blasting* (раздувание в биты): каждый бит вектора становится булевой переменной, а арифметическая операция превращается в булеву схему (глава 11). Полученную схему решает SAT-солвер из главы 14, а кодирование Тсейтина держит размер формулы почти линейным по числу вентиля: каждый вентиль получает свежую переменную и несколько дизъюнктов.

АЛГОРИТМ Bit-blasting

1. Разложить каждый битовый вектор на отдельные булевы переменные по битам.
2. Перевести каждую арифметическую операцию в булеву схему (глава 11).
3. Свести схему к КНФ кодированием Тсейтина (глава 14).
4. Решить полученную SAT-формулу.

Кванторно-свободный фрагмент BV NP-полон (глава 30). NP-полнота относится к формулам, где ширина входит во вход. Bit-blasting разворачивает формулу в SAT-задачу размера, полиномиального от суммарной ширины всех векторов, — отсюда и NP-полнота. Фиксированная ширина задачу не упрощает: полный перебор стоит $2^{n \cdot m}$ шагов, экспоненциально по числу переменных.

Удивляться нечему: решатель ищет значения битов, а такая задача и есть SAT. Для фиксированной ширины область значений битового вектора конечна: 2^n значений, и кодирование в булеву логику ничего не теряет. Масштаб разницы виден по числам. При ширине 4 значений всего 16. При ширине 64 их около 1.8×10^{19} . Полный перебор потому остаётся практичным только для крошечных ширин, а для ширины настоящего регистра он бессмыслен.

Замечание NP-полнота и практика

Кванторно-свободные фрагменты LIA, AX и BV NP-полны, как и сам SAT. Худший случай снова не отражает практику. Инженерная формула наследует структуру породившей её задачи, и эта структура даёт решателю зацепки: выученные дизъюнкты, направления симплекса, эвристики выбора. Например, переменные задачи распадаются на почти независимые сообщества, и поиск работает внутри них (глава 14).

Примечание Bit-blasting против символических процедур

Для битового вектора битовая семантика и есть настоящая семантика: операция по модулю 2^n в точности совпадает с поведением схемы, и перевод в булевы переменные не теряет информации. Для целой или вещественной арифметики тот же приём взрывает формулу: одна переменная несёт 2^n значений, и одно умножение 64-битных чисел раздувается в тысячи вентилях и десятки тысяч дизъюнктов. theory-солверы линейной арифметики рассуждают о неравенствах символически, не раскладывая переменные на биты, и потому работают с неравенствами напрямую, а не перебирают значения.

Битовый вектор моделирует регистр процессора. Память же устроена как набор таких регистров, адресуемых индексом. Следующая — теория массивов — описывает именно эту картину: память как массив битовых векторов.

§ 15.8 Теория массивов

Память компьютера — один большой массив: адресуемые ячейки, в каждой — значение. Операция чтения по адресу возвращает содержимое ячейки, операция записи кладёт в ячейку новое значение. Всякая работа с данными — чтения и записи по адресам, одна за другой.

Инструменты проверки программ строят на этой картине целые рассуждения: *символьное исполнение* в стиле KLEE представляет каждый фрагмент памяти как массив, а кучу — как массив массивов битовых векторов. Верификатор Dafny описывает изменение состояния программы последовательностью записей в такой массив. Чтобы доказать, что две копии данных равны, верификатор сводит вопрос к равенству массивов: содержимое совпадает по каждому адресу, значит, массивы равны.

Логические законы массивов — это законы памяти: правила, по которым чтения и записи складываются в согласованное описание вычисления. Чтобы сделать эти законы точными, нужна теория первого порядка с двумя сортами — индексов и элементов — и двумя функциями чтения и записи.

ОПРЕДЕЛЕНИЕ 15.20 Теория массивов

Теория массивов $\mathcal{T}_{\text{АХ}}$ имеет сигнатуру с тремя сортами — массивы, индексы и элементы — и двумя функциональными символами.

- read — чтение элемента по индексу, с рангом $\text{rank}(\text{read}) = (\text{Array}, \text{I}, \text{E})$.
- write — запись значения по индексу: новый массив, совпадающий со старым всюду, кроме одного индекса, с рангом $\text{rank}(\text{write}) = (\text{Array}, \text{I}, \text{E}, \text{Array})$.

Обе функции тотальны: чтение определено для всякой пары (массив, индекс), запись — для всякой тройки (массив, индекс, элемент). Тотальность избавляет рассуждение от особых случаев: у всякого чтения есть значение, и всякая запись возможна. Массив — это функция из индексов в элементы, а картинка с ячейками — лишь иллюстрация.

Запись не изменяет массив: она возвращает новый массив, согласованный со старым. Старый массив продолжает существовать, и к нему по-прежнему можно обращаться. В формуле $a = \text{write}(a, i, v)$ правая часть — отдельный терм, обозначающий новую версию массива. Старая версия остаётся доступной: две версии совпадают, только если запись не изменила содержимого.

Массив устроен как *записная книжка*, в которой каждая правка делается на новой странице: прежние страницы не зачёркиваются, и к ним можно вернуться в любой момент. Это чисто функциональный взгляд, и у него есть имя — *персистентные структуры данных*: изменение порождает новую версию, а прежние версии остаются доступны. SMT-солвер мыслит именно так: записи добавляют новые версии массива, а прошлое не стирают.

Пример Запись не стирает старый массив

Пусть $b = \text{write}(a, i, 7)$. Новый массив совпадает со старым всюду, кроме одного индекса, где теперь лежит 7. Старый массив продолжает существовать: чтение из него осмысленно и возвращает прежнее значение. Обе переменные стоят в формуле одновременно, и SMT-солвер рассуждает о них обеих.

Поведение чтения и записи задают три аксиомы.

ОПРЕДЕЛЕНИЕ 15.21 Аксиомы теории массивов

Аксиома чтения после записи. Запись по индексу i сразу же читается обратно.

$$\text{read}(\text{write}(a, i, v), i) = v.$$

Аксиома чтения по чужому индексу. Запись по индексу i не меняет значения по другим индексам.

$$i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j).$$

Аксиома экстенциональности. Массивы равны, когда равны по каждому индексу.

$$(\forall i. \text{read}(a, i) = \text{read}(b, i)) \rightarrow a = b.$$

Массивам нужны именно эти аксиомы: без них чтение и запись были бы неинтерпретируемыми функциями, и чтение из свежезаписанного массива могло бы вернуть что угодно. Рассуждение о памяти рассыпалось бы: верификатор не вывел бы $\text{read}(\text{write}(a, i, 7), i) = 7$, потому что аксиомы — единственный источник этого факта. Первые две аксиомы — точная формализация двух свойств памяти: запись видна своему адресу и невидима чужим. Третья аксиома — про равенство, а не про память: она превращает массив в функцию, заданную содержимым, и без неё равенство массивов не следовало бы из равенства содержимого.

Первые две аксиомы называются *read-over-write*. Запись по индексу i читается обратно по тому же индексу, а по остальным индексам чтение возвращает прежние значения. Вместе они описывают согласованность памяти: свежая запись видна своему адресу и не трогает соседние.

Третья аксиома — *экстенциональность* — задаёт равенство массивов как поточечное равенство функций. Два массива равны тогда и только тогда, когда они совпадают по всем индексам. Без этой аксиомы два массива с одинаковым содержимым могли бы оставаться различными, и верификатор не смог бы вывести $a = b$ из равенства содержимого. Экстенциональность — то, что делает массив функцией из индексов в элементы, а не поименованной ячейкой: массив задаётся содержимым, а не именем.

Пример Массивы: аксиомы в действии

Первая аксиома сразу даёт $\text{read}(\text{write}(a, i, 7), i) = 7$: формула общезначима. Вторая покрывает запись по чужому индексу: $i \neq j \rightarrow \text{read}(\text{write}(a, i, 7), j) = \text{read}(a, j)$. Контраст двух записей по одному индексу и по разным индексам:

- $\text{read}(\text{write}(\text{write}(a, i, 5), i, 9), i) = 9$: побеждает последняя запись.
- $i \neq j \rightarrow \text{read}(\text{write}(\text{write}(a, i, 5), j, 9), i) = 5$: записи по разным индексам не мешают друг другу.

Отрицание первой аксиомы невыполнимо: $\text{read}(\text{write}(a, i, 7), i) \neq 7$. Экстенциональность в действии: формула $(\forall k. \text{read}(a, k) = \text{read}(b, k)) \wedge a \neq b$ невыпол-

нима. Она содержит квантор по индексу и потому лежит вне кванторно-свободного фрагмента: запись «для всех k » в нём невозможна, и именно поэтому экстенциональность обрабатывают отдельно. Контраст: формула $\text{read}(a, k) = \text{read}(b, k) \wedge a \neq b$ выполнима: массивы могут расходиться по другим индексам.

Кванторно-свободный фрагмент $\mathcal{T}_{\text{AX}}$ — булевы комбинации равенств и неравенств термов из чтения и записи, без кванторов. Он NP-полон (глава 30). Полная теория массивов (с экстенциональностью) неразрешима. Разница — в кванторах: формула без кванторов говорит о конечном числе адресов, формула с кванторами — сразу обо всех.

Кванторы по индексам дают достаточно выразительности, чтобы закодировать арифметику натуральных чисел. Число n представляется массивом: единицы на первых позициях, нули на остальных. Сравнение выражается квантором по индексу: $n < m$ тогда и только тогда, когда найдётся позиция, где в первом массиве стоит ноль, а во втором — единица. Прибавление единицы — тоже кванторная формула: новый массив отличается от старого ровно в одной позиции.

Это интуитивный набросок: выразимость сложения требует арифметики на индексах, а строгое доказательство неразрешимости кодирует двухсчётчиковую машину либо определяет умножение через последовательности. Как только теория выражает достаточно богатую арифметику (с умножением), она наследует её неразрешимость: арифметическая формула переводится в формулу о массивах с сохранением ответа, а арифметика неразрешима (глава 26). Кванторно-свободный фрагмент таких возможностей лишён: он сохраняет лишь конечные паттерны чтений и записей, и для них решение существует.

Верхняя граница NP следует из сведения к равенству с неинтерпретируемыми функциями. После конечного числа инстанциаций формула превращается в булеву комбинацию литералов равенства: инстанцированные аксиомы добавляют дизъюнкты вида $j = i$ или равенство. Дальше вступает congruence closure, а каждый дизъюнкт требует угадать сторону — это и даёт верхнюю границу NP.

Выполнимость в кванторно-свободном фрагменте решают *ленивой инстанциацией аксиом*: аксиомы подставляют лишь те, что нужны для текущего конфликта, а не все сразу. Процедура сводит задачу к теории равенства с неинтерпретируемыми функциями, для которой алгоритм congruence closure уже разобран выше.

АЛГОРИТМ Ленивая инстанциация аксиом

1. Рассмотреть чтение и запись как неинтерпретируемые функциональные символы и применить к конъюнкции литералов congruence closure.
2. Когда замыкание встречает чтение из write, инстанцировать аксиому read-over-write для пары индексов этого чтения и добавить полученное равенство или дизъюнкт.

3. Повторять, пока не возникнет противоречие (формула невыполнима) или не перестанут выводиться новые равенства (построена выполняющая модель).

Экстенциональность обрабатывают отдельно: разбором случаев по свежему индексу или ленивым добавлением инстанцированных экстенциональных равенств.

Лень здесь необходимость: инстанцировать все аксиомы для всех индексов невозможно, ведь индексов бесконечно много. Инстанцируют ровно то, чего требует текущий конфликт, и ровно в том месте, где он возник. Инстанцированная аксиома для чтения из записи принимает вид дизъюнкта: $j = i$ или равенство результатов чтения.

Дизъюнкт появляется так. Импликация с неравенством в посылке равносильна дизъюнкции:

$$(i \neq j \rightarrow B) \Leftrightarrow (i = j \vee B)$$

. Импликация ложна в единственном случае: посылка истинна, а заключение ложно. Посылка с неравенством истинна ровно тогда, когда равенство ложно, поэтому импликация ложна ровно тогда, когда ложны и равенство, и заключение. Дизъюнкция ложна ровно в этом же случае: ложны оба её члена. Значит, вторая аксиома массивов допускает дизъюнктивную запись: $j = i$ или равенство результатов чтения.

Это та же конфликт-управляемая идея, что и в CDCL из главы 14: решатель учится на противоречии и движется дальше, не трогая то, что ему не мешает. Каждая инстанцированная аксиома — новый литерал, добавленный в конъюнкцию, и цикл замыкания повторится, пока не остановится.

Он останавливается, потому что инстанциация не порождает новых массивов и индексов. Инстанцированная аксиома связывает лишь термы, построенные из массивов и индексов, уже встречавшихся в формуле, а число таких термов конечно. Рано или поздно замыкание либо находит противоречие, либо перестаёт выводить новые равенства.

Вложенные записи сводятся к одной по первой аксиоме: чтение возвращает последнее записанное значение. Инстанцированные аксиомы возвращаются в SAT-часть DPLL(T) как общезначимые в теории дизъюнкты — такие дизъюнкты называются *T-леммами*, и раздел о DPLL(T) разбирает этот механизм.

Массивы — рабочая модель памяти в символьном исполнении. KLEE кодирует память исполняемой программы массивами, и каждая ветвь вычисления проверяется как SMT-формула над массивами. Ограниченная проверка моделей программ с массивами — та же техника: развернуть вычисление на конечное число шагов и спросить SMT-солвер, выполнима ли формула (глава 32). Сама теория массивов проста, а литературы вокруг неё много, и это не случайность.

Замечание Слои вокруг ядра массивов

Кванторно-свободная теория массивов — лишь ядро. В практических задачах индексы — арифметика, элементы — битовые векторы (глава 11), а экстенциональность приходится учитывать вместе с кванторами. Комбинация массивов с теорией индексов, кванторные фрагменты (array property fragment — кванторы по индексам специальной формы) и оптимизация экстенциональности — каждый из этих слоёв породил десятки работ. Везде одна и та же задача: выразительность растёт, а разрешимость надо сохранить.

Массивы редко появляются в формуле в одиночку: индексы — числа, элементы — битовые векторы. Одна формула смешивает несколько теорий, и для неё нужен решатель, понимающий их вместе. Как процедура решения встраивается в общий цикл SAT-поиска, показывает раздел о DPLL(T), а как решатели разных теорий собираются в одну согласованную процедуру — раздел о комбинации теорий (Nelson-Oppen).

§ 15.9 Theory-солверы и DPLL(T)

Теперь у каждой теории есть своя процедура решения. Осталось показать, как один SAT-решатель руководит ими всеми. Theory-солвер отвечает на один вопрос: совместна ли конъюнкция литералов. Формула же — дизъюнкция конъюнкций, и наивный путь — привести её к ДНФ и перебрать все конъюнкции.

Размер ДНФ может расти экспоненциально: число конъюнкций растёт экспоненциально от числа атомов формулы. Например, у формулы $(p_1 \vee q_1) \wedge (p_2 \vee q_2)$ ДНФ состоит из четырёх конъюнкций:

$$(p_1 \wedge p_2) \vee (p_1 \wedge q_2) \vee (q_1 \wedge p_2) \vee (q_1 \wedge q_2)$$

. Каждая скобка удваивает число конъюнкций, и двадцать скобок дают больше миллиона вариантов.

Выход — перебирать конъюнкции неявно, по одной на ветку поиска, и спрашивать теорию только о текущей. SAT-решатель управляет ветками, theory-солвер судит о каждой, и ни один из них не делает чужой работы. Теорема о сведении к конъюнкциям литералов гарантирует полноту такого способа: если формула выполнима, найдётся ветка, которую теория примет. ДНФ-преобразование остаётся лишь доказательством существования: оно показывает, что сведение к конъюнкциям возможно, но строить ДНФ явно невыгодно.

Архитектура DPLL(T) — это SAT-решатель на *булевом скелете* плюс theory-солвер в роли оракула совместности. *Булев скелет* формулы получается заменой каждого атома теории (например, $x + y \leq 10$) свежей булевой переменной.

Дальше цикл повторяет CDCL (глава 14):

1. SAT-решатель ищет модель булева скелета — присваивание булевых переменных.
2. Theory-солвер проверяет конъюнкцию атомов теории, помеченных истинными, — ту ветку, которую нашёл SAT.
3. Если конъюнкция несовместна, theory-солвер возвращает *Т-лемму*: выученный дизъюнкт, отрицающий конфликтную конъюнкцию.
4. SAT-решатель добавляет Т-лемму в формулу и продолжает поиск, как с любым выученным дизъюнктом.

В реальных солверах theory-солвер вызывается и на частичных присваиваниях: он доопределяет атомы, вынужденные теорией (*теория-пропагация*), а Т-лемма строится из объяснения конфликта.

Кандидата-модель и Т-лемму стоит назвать точно. *Кандидат-модель* — это присваивание булевых переменных скелета, которое SAT предлагает на проверку: гипотеза, которой ещё предстоит стать моделью. *Т-лемма* — дизъюнкт, общезначимый в теории $\mathcal{T}$: лемма отрицает конкретную несовместную конъюнкцию атомов и потому истинна в любой интерпретации теории.

Т-лемма — это отказ теории с перечнем несовместных атомов.

Цикл «поиск, конфликт, обучение» — тот же, что в CDCL, но с теорией в роли оракула. Всё, что обсуждалось про CDCL — watched literals, VSIDS, выученные дизъюнкты, — работает и внутри SMT-солвера, потому что Т-лемма приходит в том же формате, что и булев выученный дизъюнкт. DPLL(T) — это CDCL, расширенный на формулы с атомами теории.

Обобщение точнее, чем кажется: в CDCL конфликт — дизъюнкт, ставший ложным при текущем присваивании, в DPLL(T) конфликтом становится любой набор атомов, который теория объявляет несовместным. Выученный дизъюнкт в SAT отрицает булев конфликт, Т-лемма отрицает конфликт теории, и формат у них один. Отличие — в источнике знания: булев дизъюнкт выведен из структуры формулы, Т-лемма — из аксиом теории, и потому она годится для любой формулы с теми же атомами. Обучение работает одинаково: запомнить набор, который не должен повториться.

Цикл обмена между двумя решателями показан на рис. 79: вниз уходит кандидат-модель булева скелета, вверх возвращается Т-лемма.

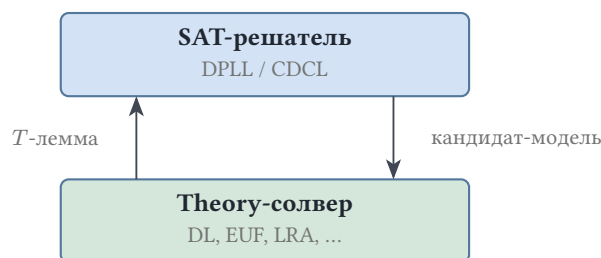


Рис. 79. Цикл обмена между SAT-решателем и theory-солвером.

Кандидат-модель подтверждается только тогда, когда theory-солвер принимает конъюнкцию. Тогда модель булева скелета вместе с моделью теории дают модель всей формулы.

Пример DPLL(T) в действии

Проверим формулу $(x + y \leq 10) \wedge (x > 5) \wedge (y > 5)$ над целочисленной линейной арифметикой. Булев скелет: три атома становятся тремя булевыми переменными:

$$p = (x + y \leq 10), q = (x > 5), r = (y > 5)$$

.

1. SAT-решатель предлагает кандидата-модель: все три переменные истинны, $p = q = r = 1$.
2. Theory-солвер проверяет конъюнкцию $x + y \leq 10 \wedge x > 5 \wedge y > 5$. Над целыми $x \geq 6$: неравенство усиливается. Аналогично $y \geq 6$. Отсюда сумма не меньше 12. Это противоречит $x + y \leq 10$. Конъюнкция несовместна.
3. Конфликтную конъюнкцию образуют все три атома: они помечены истинными, а вместе несовместны. Т-лемма — это отрицание этой конъюнкции, $\neg(p \wedge q \wedge r)$. По закону де Моргана отрицание конъюнкции — дизъюнкция отрицаний: $\neg p \vee \neg q \vee \neg r$. Клауза общезначима в теории $\mathcal{T}_{LIA}$: любая модель теории отвергает хотя бы один из трёх атомов.
4. SAT-решатель добавляет её к скелету: теперь формула $p \wedge q \wedge r \wedge (\neg p \vee \neg q \vee \neg r)$ невыполнима уже на булевом уровне. Ответ — UNSAT.

Контраст даёт выполнимый вариант $(x + y \leq 10) \wedge (x > 5)$. Скелет — $p \wedge q$. SAT предлагает кандидата-модель $p = q = 1$. Theory-солвер проверяет конъюнкцию и находит модель: $x = 6, y = 0$. Конфликта нет, кандидат-модель подтверждена, ответ — SAT.

Примечание Энергичное и ленивое встраивание теорий

Есть два способа соединить SAT и теорию. *Энергичный (eager)* переводит теорию в булеву формулу заранее: для битовых векторов это bit-blasting, когда каждый бит разворачивается в булеву переменную, а всю арифметику решает SAT-солвер. *Ленивый (lazy)* проверяет теорию по требованию, внутри цикла DPLL(T). Именно ленивый подход — стандарт SMT: он тратит работу только на те ветки, которые SAT действительно рассматривает.

Проверка Theory-солверы и DPLL(T)

1. Что такое булев скелет формулы и что подставляется вместо каждого атома теории?
2. Чем Т-лемма отличается от выученного дизъюнкта CDCL по источнику и почему формат у них один?

3. Какой подход — энергичный или ленивый — стал стандартом SMT и на каких ветках поиска он тратит работу?

§ 15.10 Комбинация теорий (Nelson-Oppen)

Одна формула часто смешивает теории: $f(x) = g(y)$ and $x < y$ — равенство с неинтерпретируемыми функциями и арифметику сразу. Один theory-солвер не решит такую формулу: ни EUF-решатель, ни арифметический не понимает половины литералов. Нужно собрать решатели разных теорий в одну согласованную процедуру, и за это отвечает метод Nelson-Oppen.

Запустить оба решателя по очереди нельзя. Даже если формула разбита на две части, каждая в своей сигнатуре, по отдельности они могут быть выполнимы, а общего решения не быть. Метод Nelson-Oppen требует два условия, и каждое устраняет конкретную причину такого расхождения.

Первое условие — *непересекающиеся сигнатуры*: теории делят только равенство $=$. Если обе теории интерпретируют символ $+$, они могут приписывать ему разные смыслы, и соединять их модели нельзя. Поэтому общие подтермы выносятся в переменные, и теории общаются только равенствами и неравенствами об этих переменных. Обмен равенствами достаточен: общие переменные — единственные точки контакта двух частей, и любое влияние одной теории на другую проходит через значения этих переменных. Отношение между двумя значениями исчерпывается альтернативой: они равны или различны, — и, передавая равенства и неравенства, теория сообщает соседке всё, что та вправе знать. Равенство — единственный символ, смысл которого зафиксирован раз и навсегда, поэтому он и становится каналом связи.

Второе условие — *стабильная бесконечность*. Теория $\mathcal{T}$ *стабильно-бесконечна*, если всякая выполнимая кванторно-свободная формула имеет модель, в которой носитель каждого сорта бесконечен. Название читается буквально: бесконечность устойчива в том смысле, что выполнимая формула не теряет модель при переходе к бесконечным носителям. «Стабильно» — потому что свойство сохраняется: если каждая выполнимая формула имеет бесконечную модель, то бесконечность — закон класса, а не случайность отдельных моделей.

Условие нужно, чтобы модели частей можно было склеить в одну модель. Возьмём теорию, все модели которой содержат ровно два элемента некоторого сорта, и вторую теорию, которая требует бесконечного носителя этого сорта. Каждая часть выполнима, но никакая общая интерпретация не годится обеим сразу: одна требует два элемента, другая — бесконечно много. Стабильная бесконечность исключает такой разлад: если обе теории готовы принять бесконечную модель, то есть и модель, на которой их части совпадают по общим переменным.

Бесконечность спасает: при склейке моделей значения общих переменных расставляют попарно различными, и в бесконечном носителе для них всегда найдётся место. Конечная модель такой свободы не даёт: если общих переменных больше, чем элементов носителя, какие-то две переменные получают одно значение, а вторая часть может требовать их различия. Бесконечность — это запас элементов, которого хватает на любой конечный набор общих переменных.

Процедура Nelson-Оппен:

1. *Пурификация*: разбить формулу на литералы по теориям, заменив общие подтермы свежими переменными. Например, $f(y)$ в арифметическом литерале заменяется свежей переменной с равенством.
2. Каждый theory-солвер решает свою чистую часть — формулу в одной сигнатуре.
3. Солверы обмениваются выведенными равенствами и неравенствами об общих переменных.
4. Обмен повторяется, пока не перестанут появляться новые факты.
5. Если новые факты перестали появляться, а теории невыпуклы, перебрать варианты расположения значений общих переменных (*расщепление по случаям*). Противоречие во всех вариантах означает невыполнимость, а наличие совместного варианта позволяет склеить модели.

Свойство, о котором идёт речь, называется *выпуклостью*: теория выпукла, если из выполнимости набора литералов вместе с дизъюнкцией равенств следует выполнимость с одним из этих равенств. EUF выпукла: конгруэнтность выводит равенства по одному. Целочисленная арифметика не выпукла: из двойного неравенства следует альтернатива из двух равенств, но ни одно из них в отдельности не следует. Для выпуклых теорий хватает поочерёдной передачи равенств, для невыпуклых — расщепления по случаям, и имя свойства объясняет, где проходит граница.

ТЕОРЕМА 15.4 Теорема Нельсона–Оппена

Пусть $\mathcal{T}_1$ и $\mathcal{T}_2$ — стабильно-бесконечные теории с непересекающимися сигнатурами, и для каждой разрешима выполнимость конъюнкций литералов. Тогда выполнимость конъюнкций литералов в объединённой теории разрешима: процедурой Нельсона–Оппена с пурификацией и обменом выведенными равенствами и неравенствами об общих переменных, а при невыпуклости теорий — с расщеплением по случаям.

Набросок доказательства

Докажем, что процедура отвечает верно и завершается.

Пусть обмен насытился, и обе части выполнены. Склеим их модели: общим переменным, объявленным равными, сопоставляется один элемент носителя, а различным — разные элементы носителя. Стабильная бесконечность даёт носитель, в котором для всех этих требований есть место. Функции и предикаты каждой теории действуют только на символы своей сигнатуры, и из непересече-

ния сигнатур определения не конфликтуют, поэтому склейка — интерпретация объединённой теории, выполняющая обе части.

Пусть обмен вскрыл конфликт: одна из частей несовместна со своими и полученными литералами. Каждый полученный факт — следствие другой части, поэтому любая модель объединённой конъюнкции выполняла бы и эту несовместную часть. Модели нет, и конъюнкция невыполнима.

Завершимость даёт конечность фактов: общих переменных конечное число, и различных равенств и неравенств о них тоже конечное число, поэтому обмен перестаёт порождать новые факты. Для невыпуклых теорий расщепление перебирает конечное число способов расположить общие переменные по классам равенства, и каждый вариант дальше разбирается как в выпуклом случае.

Пример Nelson-Оппен: разбор примера

Проверим формулу $(x = f(y)) \wedge (x = y + 1) \wedge (f(y) \neq y + 1)$. Работают две теории: равенство с неинтерпретируемыми функциями и целочисленная арифметика. Их сигнатуры не пересекаются, обе стабильно-бесконечны: для EUF бесконечные модели существуют тривиально, а для арифметики моделью служат сами целые числа.

1. Пурификация: подтерм $f(y)$ принадлежит EUF, но встречается и в арифметическом литерале. Вводим свежую переменную x_1 с равенством. EUF-часть:

$$x_1 = f(y), x = x_1$$

. Арифметическая часть:

$$x = y + 1, x_1 \neq y + 1$$

. Общие переменные двух частей — x, y, x_1 .

2. Арифметический решатель обрабатывает свою часть. Разберём шаг подробно. Предположим противное: $x_1 = x$. Тогда в равенстве можно сделать замену и получить $x_1 = y + 1$: прямое противоречие с литералом. Значит, предположение ложно, и $x_1 \neq x$ — следствие арифметической части. EUF-части этот факт неизвестен: её множество равенств совместно. Конфликт вскрывает обмен: арифметическая часть выводит $x_1 \neq x$, а это прямо противоречит EUF-части, поэтому следствие передают EUF-решателю.
3. EUF-решатель получает неравенство, но его часть содержит равенство: прямое противоречие. Формула невыполнима.

Решающий шаг — передача неравенства $x_1 \neq x$: без неё EUF-часть выглядела бы совместной, и процедура не заметила бы конфликта.

Стабильная бесконечность — граница применимости метода. Без неё шаг склейки моделей ломается: каждая часть выполнима, но общего решения нет, и метод такие теории не комбинирует.

§ 15.11 SMT-LIB и солверы

После раздела о теориях у каждой процедуры свой входной формат, и решатели разных теорий говорят на разных языках. Формула, смешивающая арифметику с массивами, требует двух процедур, а передать её каждой в её собственном формате нельзя. Ответ подсказывает архитектура: интерфейс DPLL(T) единообразен — кванторно-свободные формулы над фиксированными сигнатурами теорий, — и этот интерфейс можно сделать языком. Такой язык — *SMT-LIB*: запись в виде Lisp-подобных S-выражений, то есть вложенных списков в круглых скобках. Каждая команда — это список, который начинается с имени команды, а её аргументы — остальные элементы списка. Общность языка — следствие единой архитектуры, а не удобство.

Перед примерами разберём команды.

Команда	Что делает
declare-const	(declare-const x Int) заводит переменную сорта Int
assert	добавляет формулу в задачу: солвер обязан учитывать все добавленные формулы
check-sat	спрашивает, выполняма ли совокупность добавленных формул
get-model	просит выдать модель, если солвер ответил sat
declare-fun	объявляет неинтерпретируемую функцию: (declare-fun f (Int) Int) — функция из целых в целые

Пример SMT-LIB: линейная арифметика

Проверка выполнимости $x + y \leq 10 \wedge x > 5$:

```
(declare-const x Int)
(declare-const y Int)
(assert (<= (+ x y) 10))
(assert (> x 5))
(check-sat)
(get-model)
```

Солвер отвечает sat и выдаёт модель: $x = 6, y = 0$. Ответ sat означает только «решение существует», но какое именно — солвер выбирает сам. К одной и той же задаче он может подойти разными путями и выдать разные модели. Формула get-model просит предъявить одну из них, и любая подойдёт, раз вопрос был о существовании.

Пример SMT-LIB: равенство с неинтерпретируемыми функциями

Проверка выполнимости $f(x) = g(y) \wedge x = y \wedge \neg(f(y) = g(x))$:

```
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(declare-const x Int)
(declare-const y Int)
(assert (= (f x) (g y)))
(assert (= x y))
(assert (not (= (f y) (g x))))
(check-sat)
```

Солвер отвечает `unsat`. Из $x = y$ по конгруэнтности следует:

$$f(x) = f(y), g(y) = g(x)$$

. Цепочка $f(y) = f(x) = g(y) = g(x)$ даёт равенство: оно противоречит отрицанию.

Пример SMT-LIB: невыполнимая система

Добавим к первому примеру ещё одно ограничение:

```
(declare-const x Int)
(declare-const y Int)
(assert (<= (+ x y) 10))
(assert (> x 5))
(assert (> y 5))
(check-sat)
```

Солвер отвечает `unsat`. Над целыми переменные получают усиленные оценки: $x \geq 6$. Аналогично $y \geq 6$. Тогда сумма не меньше 12, и ограничение нарушено. Модели нет, ограничения исключают друг друга.

Солверы — готовая реализация этой архитектуры, и все они устроены одинаково: под капотом у каждого SAT-поиск, theory-солверы и цикл DPLL(T). Различаются происхождением и инженерными деталями:

- Z3 из Microsoft Research, написанный Леонардо де Моурой и Николаем Бьёрнером, — индустриальный стандарт.
- CVC5 — наследник CVC4 и CVC3, развиваемый сообществом.
- Yices — работа SRI International.
- Alt-Ergo, солвер с открытым кодом, широко используют во французском сообществе верификации.

Общий язык SMT-LIB связывает их в одно: одна формула запускается в любом солвере без переписывания.

Эти солверы стали рабочим инструментом автоматического рассуждения.

- *Символьное исполнение* выполняет программу с символьными входными данными вместо конкретных значений: каждый путь кодируется формулой, и SMT-

солвер решает, достижим ли путь. KLEE по умолчанию работает с солвером STP, а Z3 подключается как альтернативный солвер.

- *Верификация программ* (глава 32) сводит условия корректности к SMT-запросам: так работает Dafny.
- *Синтез программ* ищет программу, удовлетворяющую спецификации, перебирая кандидатов и проверяя каждого запросом к солверу (Rosette).
- *Model checking* проверяет свойства системы запросами к её символьной модели, как в главе 33.

Во всех этих задачах одна и та же схема: булево рассуждение из главы 14, теории из этой главы и цикл DPLL(T) под капотом.

§ 15.12 Итоги главы

SMT — это выполнимость по модулю теорий: булево рассуждение из главы 14 работает в одном цикле с языком и семантикой логики первого порядка из главы 3. Многосортная сигнатура дала термам типы, а теория сузила класс допустимых интерпретаций: T -общезначимость — это общезначимость по моделям теории. Полная арифметика неразрешима (глава 26), поэтому практика работает с кванторно-свободными фрагментами, и именно их разрешимость — двигатель SMT.

У каждой теории свой решатель, и каждый устроен как поиск противоречия на своём верстаке — в структуре, куда раскладываются ограничения. Разностная логика ищет отрицательный цикл в графе ограничений, EUF сливает классы равных термов, битовые векторы разворачиваются в схему, а массивы решаются ленивой инстанциацией аксиом. Линейную арифметику решают симплекс и целочисленные методы — ветвление с границами и отсечения. Форма литералов диктует выбор структуры: разностное ограничение делает пару переменных ребром графа, равенство разбивает термы на классы, линейное неравенство вырезает полупространство. Противоречие в теории — невозможность в этой структуре: отрицательный цикл, два терма в одном классе при объявленном различии, пустой многогранник.

Цикл DPLL(T) встраивает любой такой решатель в поиск CDCL: конфликт теории порождает T -лемму, SAT-решатель учит её, и поиск продолжается на булевом скелете. Когда формула смешивает теории, процедура Нельсона–Оппена собирает решатели в одну согласованную процедуру, обмениваясь равенствами и неравенствами об общих переменных.

Формат SMT-LIB дал всем солверам общий язык, а солверы — Z3, CVC5, Yices, Alt-Ergo — превратили архитектуру в рабочий инструмент верификации программ, символьного исполнения и синтеза. Разрешимость кванторно-свободных фрагментов не противоречит NP-полноте многих из них (глава 30).

Проверка Проверьте себя

1. Почему кванторно-свободные фрагменты теорий разрешимы, хотя полная логика первого порядка неразрешима?
2. Как цикл DPLL(T) связывает булев поиск с решателями теорий, и что такое T-лемма?
3. Почему отрицательный цикл в графе разностных ограничений означает невыполнимость, и как его находит процедура решения?
4. Какие процедуры решают кванторно-свободные фрагменты EUF, битовых векторов и массивов, и какая структура ограничений соответствует каждой из них?

Что дальше?

SMT автоматизирует проверку выполнимости по модулю теорий: решатель отвечает, существует ли модель. В главе 16 логика станет языком программирования: программа — множество клауз, а выполнение — поиск доказательства. Вопрос меняется с «существует ли модель» на «какие ответы следуют из теории».

§ 15.13 Упражнения

Базовые понятия SMT.

1. Что такое ранг функционального символа в многосортной логике? Приведите ранг read из теории массивов.
2. Чем теория отличается от чистой логики первого порядка? Что означает $\mathcal{T}$ -общезначимость?
3. Какой результат гарантирует разрешимость арифметики Пресбургера, и какой символ отвечает за неразрешимость арифметики Пеано?

Теории и их решатели.

4. Нормализуйте литералы к виду $u - v \leq c$. Возьмите строгое неравенство $x - y < 3$. Возьмите также нестрогое неравенство $z - x \geq 4$.
5. * Постройте граф для конъюнкции $(x - y \leq 2) \wedge (y - z \leq -1) \wedge (z - x \leq -1)$ и определите, выполнима ли она.
6. Примените congruence closure к $(f(a) = b) \wedge (a = b) \wedge \neg(f(b) = b)$.

Теория массивов.

7. Докажите, что $\text{read}(\text{write}(a, i, 5), i) = 5$ общезначимо в теории массивов. Объясните, почему отрицание этой формулы невыполнимо.
8. Объясните, почему формула $(\forall k. \text{read}(a, k) = \text{read}(b, k)) \wedge a \neq b$ невыполнима в полной теории массивов, но лежит вне кванторно-свободного фрагмента, и как её обрабатывает процедура решения.

DPLL(T) и комбинация теорий.

9. В чём идея DPLL(T)? Какую роль играет theory-солвер, и что возвращает SAT-решателю при конфликте?
10. Проследите цикл DPLL(T) на формуле $(x \geq 0) \wedge (y \geq 0) \wedge (x + y < 0)$. Укажите кандидата-модель, ответ theory-солвера и полученную T-лемму. Почему ответ — UNSAT?
11. Проведите пурификацию и обмен на формуле $(x = f(y)) \wedge (x = y + 1)$. Почему процедура останавливается без противоречия, и как склеиваются модели двух теорий?
12. Почему для битовых векторов применяют bit-blasting, а для линейной арифметики — нет?

SMT-LIB и солверы.

13. Запишите в SMT-LIB проверку выполнимости $f(x) = g(y) \wedge \neg(x = y)$. Какой ответ даст солвер, и почему?
14. Дана система разностных ограничений:

$$z \geq x + 3, w \geq z + 1, w \leq x + 2$$

. Выразите каждое ограничение в форме $u - v \leq c$, постройте взвешенный граф и найдите отрицательный цикл. Какой вывод о выполнимости системы даёт цикл? Что изменится, если последнее ограничение заменить на $w \leq x + 4$?

15. * Почему кванторно-свободная выполнимость в $\mathcal{T}_{\text{EUF}}$ разрешима за полиномиальное время, а общезначимость произвольных формул первого порядка — нет? Свяжите с теоремой Чёрча (неразрешимость проблемы разрешения) из главы 26.
16. * Закодируйте ограничение $x + y = 3$ для беззнаковых 2-битных векторов методом bit-blasting. Пусть биты вектора x — x_1x_0 , биты вектора y — y_1y_0 (в каждом старший бит — первый). Введите битовые переменные, выпишите уравнения сумматора, переведите их в дизъюнкты и приведите выполняющее присваивание.

ПРОЕКТ Мини-SMT: разностный theory-солвер в цикле DPLL(T)

Соберите решатель, который проверяет выполнимость булевых комбинаций разностных ограничений $x - y \leq c$: перебор булевого скелета ведёт SAT-решатель, а совместность каждой ветки проверяет theory-солвер на алгоритме Bellman–Ford. Инструмент воспроизводит цикл главы целиком: кандидат-модель уходит на проверку в теорию, конфликт возвращается T-леммой, и поиск продолжается до ответа sat или unsat.

① Граф ограничений

Нормализуйте литералы к виду $u - v \leq c$ и представляйте конъюнкцию взвешенным ориентированным графом: вершины — переменные, ребро $v \rightarrow u$ с весом c кодирует ограничение. Абсолютная граница $x \leq c$ — это разность $x - 0 \leq c$ против фиктивной нулевой вершины, а $x > c$ — отрица-

ние такой разности. Нулевая вершина служит началом отсчёта и в модели закрепляется в нуле. Загрузите примеры главы: выполняемую конъюнкцию с расстановкой $z = 0, y = 3, x = 5$ и невыполнимую с циклом, сумма весов которого равна -2 . Сумма $a + b \leq c$ к разности не сводится: оставляйте её вне графа и проверяйте по минимальным допустимым значениям концов — расстояниям до нулевой вершины в обращённом графе.

② *Проверка совместности*

Реализуйте Bellman–Ford с фиктивным источником и нулевыми рёбрами: после контрольного раунда расслабления алгоритм либо обнаруживает отрицательный цикл, либо выдаёт кратчайшие расстояния — готовое выполняющее присваивание. Ответ солвера: конъюнкция совместна вместе с моделью или несовместна.

③ *T-лемма из конфликта*

По найденному отрицательному циклу соберите объяснение: список рёбер, сумма весов и дизъюнкт из отрицаний атомов этих рёбер. Убедитесь, что лемма общезначима в теории: при любых значениях переменных хотя бы одно ограничение цикла нарушено, поэтому дизъюнкт истинен.

④ *Цикл DPLL(T)*

Постройте булев скелет формулы и перебирайте его присваивания полным перебором или простым DPLL с возвратом. Каждого кандидата передавайте theory-солверу, отклонённые кандидаты отсекайте T-леммами и продолжайте поиск до sat или unsat. Сверьтесь с примером главы: формула $(x + y \leq 10) \wedge (x > 5) \wedge (y > 5)$ даёт unsat, а T-лемма $\neg p \vee \neg q \vee \neg r$ делает скелет невыполнимым уже на булевом уровне.

⑤ *Замеры*

Сравните три режима: без T-лемм, с T-леммами и с теория-пропагацией, когда солвер проверяет конъюнкцию при добавлении каждого литерала, а не после полного присваивания. Измерьте на случайных формулах из 10–30 ограничений число обращений к теории и полное время решения в каждом режиме.

Итог: решатель, который по булевой комбинации разностных ограничений выдаёт sat с моделью или unsat с сертификатом из отрицательного цикла, а замеры показывают, сколько обращений к теории экономят T-леммы.

XVI

Логическое программирование

В базе данных — пары «родитель — ребёнок»: Том — отец Боба, Боб — отец Элис. Запрос «кто дедушка Элис» не находит ни одной строки: такого факта в базе нет. Ответ придётся вывести из двух имеющихся: дед — это родитель родителя.

Императивный программист написал бы перевод: вложенные циклы по парам, поиск цепочки, соединение результатов. Код целиком занимается переводом с языка отношений на язык инструкций, и такие переводчики есть в любом проекте. Логическое программирование убирает переводчика. Программа — сами утверждения: факты и правило «дед — родитель родителя» в одну строку. Выполнение — поиск доказательства: машина сама собирает цепочку от запроса к фактам. Ей не объясняют, как вычислить ответ, — ей сообщают, что истинно, а вывод она достраивает сама.

В главе 14 о SAT доказательство уже становилось вычислением: резолюция свела вывод к одному правилу, DPLL превратил поиск в процедуру. Здесь этот ход доведён до конца: из метода доказательства вырос язык программирования. Родство, достижимость в графе, списки — всё записывается несколькими строками фактов и правил, и ответы находит машина.

Цена в том, что у вычисления появляется собственная логика. Порядок клауз, рекурсия и поиск вслепую решают, завершится ли программа и что именно она вернёт. Что программа утверждает и что она вычисляет — не одно и то же: программа описывает истину, а выполнение извлекает из неё ответы.

Как машина находит ответы, не получив ни одной инструкции?

ИСТОРИЯ Резолюция как язык

В 1965 году Джон Алан Робинсон предложил унификацию — процедуру, решающую уравнения между термами — и построил на ней принцип резолюции для логики первого порядка. Резолюция была задумана как инструмент автоматического доказательства теорем, но она же породила язык программирования.

В начале 1970-х Ален Колмерау в Марселе работал над программой для обработки естественного языка и искал способ описывать правила рассуждений декларативно, без пошаговых инструкций. Его команда назвала получившуюся систему Prolog — сокращение от французского «programmation en logique», программирование в логике. Параллельно Роберт Ковальский в Эдинбурге сформулировал процедурную интерпретацию хорновских клауз: факты — это

база данных, правила — процедуры, а доказательство цели — вызов процедуры. Эта интерпретация и есть логическое программирование: логика описывает, что истинно, а процедурное чтение объясняет, как это вычисляется.

В 1974 году Ковальский опубликовал статью¹⁴⁰, закрепившую название парадигмы. В 1983 году Дэвид Уоррен описал абстрактную машину, компилирующую Prolog в эффективный код. Prolog стал практическим языком для символьных вычислений, систем ИИ и обработки языков. Из этого периода выросли Datalog — логика без функций и без бэктрекинга, применяемая в базах данных — и системы вида Answer Set Programming. Резолюция, начавшаяся как метод доказательства теорем, обернулась моделью вычислений.

ОБЗОР ГЛАВЫ

Глава строит логическое программирование снизу: от термов — данных программы — через подстановки и унификацию к фактам и правилам. Затем определяется SLD-резолюция — процедура, выполняющая программу как доказательство — и разбирается, как поиск с возвратом находит ответы. Списки показывают, как на этом каркасе живут структуры данных. Рекурсия и терминация объясняют, почему одна программа завершается, а другая закичивается. В конце главы рассматриваются отрицание как провал и cut: два приёма, выходящие за рамки чистой логики, но определяющие практический язык. В результате логическая программа читается как теория: видно, где в выполнении — поиск доказательства, откуда берутся ответы и почему поиск может не завершиться.

После этой главы вы сможете:

1. записывать программы фактами и правилами на языке термов;
2. объяснять работу унификации и SLD-резолюции;
3. понимать, как поиск с возвратом находит ответы и почему он может не завершиться;
4. пользоваться списками и рекурсией в логических программах;
5. различать чистое логическое программирование и приёмы вроде отрицания как провала и cut.

§ 16.1 Термы

Данные логической программы — это *термы*. Терм — это дерево: переменная, константа или функциональный символ, применённый к термам. Запись `parent(alice, bob)` — терм с функциональным символом `parent` и двумя аргументами. `alice` и `bob` — константы. Переменные зарезервированы для того, что

¹⁴⁰R. Kowalski. «Predicate Logic as Programming Language». IFIP Congress, 1974.

программа должна найти: $\text{parent}(\text{alice}, X)$ — вопрос, кто ребёнок *alice*. По соглашению переменные пишутся с заглавной буквы (X, Ys), константы и функторы — со строчной (*alice*, *parent*).

ОПРЕДЕЛЕНИЕ 16.1 Терм

Терм — это либо переменная, либо константа, либо выражение вида $f(t_1, \dots, t_n)$. Здесь f — функциональный символ, а число аргументов неотрицательно. Каждый аргумент t_i — снова терм. Константа — это функциональный символ с $n = 0$: *alice*, *bob*, $[]$.

ОПРЕДЕЛЕНИЕ 16.2 Грунтовый терм

Терм без переменных называется **грунтовым**.

Терм читается рекурсивно: структура — это функтор с аргументами, каждый аргумент — снова терм. Дерево термина $\text{parent}(\text{alice}, \text{bob})$ — корень *parent* с двумя листьями *alice* и *bob*. В отличие от формул логики из глав 1 и 3, терм не содержит связок и кванторов: он лишь называет объект. Утверждение появляется, когда терм ставят в позицию цели: $\text{parent}(\text{alice}, \text{bob})$ как цель означает «истинно, что *alice* — родитель *bob*». Терм и атом — одна и та же синтаксическая сущность: $\text{parent}(\text{alice}, \text{bob})$ вне клаузы называется термом (имя объекта), а в голове или теле клаузы — атомом (утверждение). Различие задаёт позиция: форма у термина и атома одна и та же, поэтому в коде они моделируются одним типом.

На Rust термы моделируются перечислением:

```
/// Терм: переменная, константа или структура.
enum Term {
    Var(usize),           // переменная, номер вместо имени
    Atom(String),         // константа: alice, bob, []
    Struct(String, Vec<Term>), // f(t1, ..., tn)
}
```

Переменные получают номера вместо имён, и тогда подстановки — словари номеров и термов — выглядят как отображения. Переименование при каждом использовании клаузы становится простым сдвигом (зачем клаузы переименовывают — стандартизация-разделение, раздел об SLD-резолуции).

§ 16.2 Подстановки

Доказательство постепенно уточняет, чему равны переменные. Инструмент уточнения — *подстановка*: конечное отображение переменных в термы.

ОПРЕДЕЛЕНИЕ 16.3 Подстановка

Подстановка — это конечное отображение σ переменных в термы. Сама подстановка записывается как $\sigma = \{X_1 \rightarrow t_1, \dots, X_n \rightarrow t_n\}$. Все переменные X_i попарно различны.

ОПРЕДЕЛЕНИЕ 16.4 Применение подстановки

Применение подстановки к терму даёт терм $t\sigma$. В нём каждая переменная X_i заменяется на соответствующий ей терм. Замена одновременная: подстановка применяется ко всем переменным сразу.

Пример Применение подстановки

Подстановка $\sigma = \{X \rightarrow \text{bob}\}$ переводит терм `ancestor(alice, X)` в `ancestor(alice, bob)`.

Подстановка $\tau = \{X \rightarrow \text{carol}, Y \rightarrow \text{alice}\}$ переводит терм `parent(X, Y)` в `parent(carol, alice)`. Замена одновременная: значение переменной подставляется как есть. Подстановка $\rho = \{X \rightarrow \text{bob}, Z \rightarrow X\}$ связывает Z с переменной X . Применение к терму `parent(X, Z)` даёт `parent(bob, X)`. Вхождение X внутри значения Z повторно не заменяется. Поэтому `parent(bob, bob)` не возникает.

```
/// Подстановка: словарь номеров переменных и термов.
type Subst = HashMap<usize, Term>;

/// Применение: рекурсивно заменить связанные переменные.
fn apply(t: &Term, sigma: &Subst) -> Term {
    match t {
        Term::Var(x) => sigma.get(x)
            .map(|v| apply(v, sigma)).unwrap_or_else(|| t.clone()),
        Term::Struct(f, args) => Term::Struct( f.clone(),
            args.iter().map(|a| apply(a, sigma)).collect(), ),
        Term::Atom(_) => t.clone(),
    }
}
```

Код применяет подстановку рекурсивно и доразрешает цепочки до нормальной формы: на примере с $\rho = \{X \rightarrow \text{bob}, Z \rightarrow X\}$ он вернёт `parent(bob, bob)`. Определение заменяет одновременно. В значении Z остаётся свободная переменная X . Реализация доводит применение до конца, что удобно при выдаче ответов. Различие видно только там, где значение переменной само содержит связанную переменную.

Подстановка сама по себе не знает, что истинно. Она лишь соглашение о том, чем заменить переменные. Настоящая работа происходит там, где подстановки строятся из уравнений между термами, — в унификации.

§ 16.3 Унификация

Чтобы применить правило к цели, надо найти подстановку, которая делает голову правила и цель одинаковыми. Процедура, решающая уравнения между термами, называется *унификацией*, а сама такая подстановка — *унификатором*.

ОПРЕДЕЛЕНИЕ 16.5 Унификатор

Унификатор пары термов — подстановка σ , для которой выполнено равенство $s\sigma = t\sigma$. Термы унифицируются, если у них есть унификатор.

ОПРЕДЕЛЕНИЕ 16.6 Наиболее общий унификатор

Унификатор называется **наиболее общим** (МГУ), если всякий другой унификатор получается из него дополнительной подстановкой.

МГУ делает всё, что можно, а остальные унификаторы лишь связывают то, что он оставил свободным.

Алгоритм унификации (Робинсон, 1965) — это разбор уравнений по форме термов:

АЛГОРИТМ Унификация

1. Взять нерешённое уравнение между термами.
2. Если обе части — одна и та же переменная, уравнение решено.
3. Если одна часть — переменная X , а другая — терм. Если X входит в этот терм, унификация невозможна (occurs-check). Иначе связать $X = t$.
4. Если обе части — структуры. Если символы $f = g$ и число аргументов совпадают, уравнение заменяется на уравнения между аргументами. Иначе унификация невозможна.
5. Если части — разные константы, унификация невозможна.

ОПРЕДЕЛЕНИЕ 16.7 Occurs-check

Occurs-check — это проверка того, что переменная не входит в терм, с которым её связывают. Уравнение $X = t$ решается связыванием, только если переменная не входит в терм.

Occurs-check запрещает бесконечные термы. Уравнение $X = f(X)$ не имеет решения. Если бы переменная стала равна $f(X)$, то это значение содержало бы её снова — и так без конца. Термы логического программирования конечны, и проверка вхождения защищает эту конечность.

ТЕОРЕМА 16.1 Унификация

Если два терма унифицируемы, алгоритм унификации находит их наиболее общий унификатор за конечное число шагов.

Набросок доказательства

Докажем три свойства алгоритма унификации по отдельности.

Корректность. Каждый шаг либо решает уравнение, либо сводит его к более простым, не теряя ни одного унификатора.

Конечность. Каждый шаг либо связывает переменную (а переменных в термах конечное число), либо заменяет уравнение на уравнения между аргументами (глубина которых меньше).

Единственность. Два наиболее общих унификатора связаны соотношениями $\tau = \sigma\rho, \sigma = \tau\rho'$. Здесь ρ, ρ' — подстановки, и обе обязаны быть переименованиями. Если бы ρ связывала переменную с нетривиальным термом, то есть дополнительно фиксировала значение, унификатор был бы строго конкретнее. Тогда σ не была бы наиболее общей.

Пример Наиболее общий унификатор

Термы $p(f(X), Y)$ и $p(Z, f(a))$ унифицируются. Разбор уравнения:

1. Обе части — р-арности 2, значит надо унифицировать аргументы.
2. Уравнение $f(X) = Z$. Связать $Z = f(X)$.
3. Уравнение $Y = f(a)$. Связать $Y = f(a)$.

Получилась подстановка $\{Z \rightarrow f(X), Y \rightarrow f(a)\}$, и общий терм — $p(f(X), f(a))$. Она наиболее общая: X остался свободным, и всякий другой унификатор лишь добавит ему значение.

Пример Occurs-check в действии

Уравнение $X = f(X)$ не имеет решения. Без occurs-check подстановку $\{X \rightarrow f(X)\}$ можно было бы построить формально, но применять её бесконечно. Переменная заменяется на $f(X)$, внутри которого снова переменная. Occurs-check отклоняет такое уравнение сразу, на шаге связывания.

Примечание Occurs-check на практике

Большинство Prolog-систем по умолчанию проверку вхождения не выполняют: она требует обхода терма при каждом связывании и заметно замедляет унификацию. Уравнение $X = f(X)$ в такой системе успешно связывает X с $f(X)$ и порождает циклический терм. Любая работа с таким значением — печать, обход, применение подстановки — закликивается. Полная унификация остаётся

в языке отдельным предикатом `unify_with_occurs_check/2` (запись имя/арность задаёт предикат с таким именем и числом аргументов). Реализация следует определению и проверяет вхождение при каждом связывании.

```
// Унификация: sigma дополняется до унификатора.
fn unify(t1: &Term, t2: &Term, sigma: &mut Subst) -> Result<(), UnifyError>
{
    let t1 = apply(t1, sigma);
    let t2 = apply(t2, sigma);
    match (&t1, &t2) {
        (Term::Var(x), Term::Var(y)) if x == y => Ok(()), (Term::Var(x), t)
=> bind(*x, t, sigma),
        (t, Term::Var(x)) => bind(*x, t, sigma), (Term::Atom(a),
Term::Atom(b)) if a == b => Ok(()),
        (Term::Struct(f, xs), Term::Struct(g, ys)) if f == g && xs.len() ==
ys.len() =>
        {
            for (a, b) in xs.iter().zip(ys) {
                unify(a, b, sigma)?;
            }
            Ok(()) }
        _ => Err(UnifyError::Mismatch), }
    }

// Occurs-check: переменная не связывается с термом, содержащим её.
fn bind(x: usize, t: &Term, sigma: &mut Subst) -> Result<(), UnifyError> {
    if t.contains(x) {
        return Err(UnifyError::OccursCheck { var: x, term: t.clone() });
    }
    sigma.insert(x, t.clone());
    Ok(()) }
}
```

Унификация — единственная операция, которая умеет и сравнивать, и связывать. Поэтому же она умеет возвращать значения: переменная связывается, когда её унифицируют с конкретным термом. Из этого свойства вырастает и то, что запрос `?-parent(alice, X)` (знак `?-` — запрос к программе, его переменные экзистенциальны) возвращает ребёнка. Запрос `parent(alice, X)` унифицируется с фактом `parent(alice, bob)`. Унификация связывает `X` с `bob` — это и есть ответ.

Проверка Унификация

1. Унифицируются ли термы `p(X, X)` и `p(a, b)`? Проследите шаги алгоритма.
2. Что связывает унификация термов `f(X, a)` и `f(b, Y)`?
3. Почему подстановка $\{X \rightarrow Y\}$ наиболее общая, а $\{X \rightarrow a\}$ — нет?

§ 16.4 Программы из фактов и правил

Программа — это конечный набор утверждений о том, что истинно. Утверждения бывают двух видов: факты и правила. Вместе они называются *определёнными*

клаузами — знакомыми из главы 14 хорновскими клаузами с ровно одним положительным литералом.

ОПРЕДЕЛЕНИЕ 16.8 Атом

Атом — это атомарная формула вида $p(t_1, \dots, t_n)$. Здесь p — предикатный символ, а каждый аргумент t_i — терм.

ОПРЕДЕЛЕНИЕ 16.9 Определённая клауза

Определённая клауза — выражение вида

$$A \leftarrow B_1, \dots, B_n, \quad n \geq 0$$

где $A, B_1, \dots, B_n$ — атомы. При $n = 0$ клауза — факт (A). При $n > 0$ — правило ($A :- B_1, \dots, B_n$). Все переменные клаузы считаются универсально квантифицированными.

ОПРЕДЕЛЕНИЕ 16.10 Программа

Программа (база знаний) — конечное множество определённых клауз.

ОПРЕДЕЛЕНИЕ 16.11 Цель

Цель (запрос) — конъюнкция атомов $\leftarrow A_1, \dots, A_k$. Её переменные считаются экзистенциально квантифицированными.

Кванторы расставлены по ролям. Клауза — утверждение, и все её переменные связаны универсально: правило $\text{ancestor}(X, Y) :- \text{parent}(X, Z), \text{ancestor}(Z, Y)$ описывает сразу все пары «предок — потомок», а не один пример. Запрос — вопрос о существовании: $?- \text{parent}(\text{alice}, X)$ спрашивает, найдётся ли значение X , при котором утверждение истинно. Машина ищет свидетельство существования, опираясь на утверждения, верные для всех значений.

Одна и та же клауза допускает два прочтения, и логическое программирование живёт на их стыке.

Примечание Декларативное и процедурное чтение

Декларативное чтение: $A \leftarrow B_1, \dots, B_n$ — это утверждение. Если все B_i истинны, то истинна и голова. Процедурное чтение: чтобы доказать голову, докажи подцели $B_1, \dots, B_n$ по порядку. Программа описывает истину, а машина читает описание как процедуру. Факты — это база данных. Правила — «рецепты вывода». Определения машина принимает буквально: достаточно записать $\text{spouse}(X, Y)$ как «у них есть общий ребёнок» — и супругами станут любые двое родителей одного ребёнка. Задуманный смысл в программу не зашит — она отвечает ровно за написанное.

Замечание Магический квадрат

Декларативность означает, что определение свойства не содержит способа его получить. Магический квадрат — таблица 3×3 из чисел от 1 до 9, суммы по строкам, столбцам и диагоналям которой совпадают. Можно записать предикат, который лишь проверяет это свойство, — и машина сама переберёт все перестановки чисел, отыскивая подходящую. Такой программе не нужен план построения: достаточно описания свойства, а перебор машина берёт на себя.

Пример Программа «родство»

Два факта и два правила:

```
parent(alice, bob).
parent(bob, carol).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Запрос `?- ancestor(alice, Y).` спрашивает: для какого Y утверждение `ancestor(alice, Y)` следует из программы? Первое правило отвечает за прямой шаг, второе — за транзитивность. Переменная Z во втором правиле — неизвестный посредник. Он приходится ребёнком X и предком Y . Имя посредника не важно, важно его существование: машина сама найдёт подходящего, если такой есть. Ответы: $Y = \text{bob}$, $Y = \text{carol}$.

```
/// Клауза: голова и тело. У факта тело пустое.
struct Clause {
    head: Term,
    body: Goal,
}

/// Цель: пустая, атом или конъюнкция.
enum Goal {
    True,
    Call(Term),
    Conj(Vec<Goal>),
}

/// Программа: клаузы, индексированные по предикату головы.
struct Database {
    clauses: HashMap<String, usize>, Vec<Clause>,
}
```

Индекс по предикату головы не меняет логики: он лишь ускоряет поиск, подсказывая, какие клаузы вообще могут доказать данную цель.

§ 16.5 SLD-резольюция

Выполнение программы сводится к построению доказательства цели шаг за шагом — так работает *SLD-резольюция*. SLD — это Selective Linear resolution for Definite clauses: резольюция для определённых клауз. «Линейная» — каждый шаг работает с одной целью. «Селективная» — из конъюнкции выбирается одна цель.

Связь с резольвцией из главы 14 видна в форме целей. Цель $\leftarrow A_1, \dots, A_k$ — это клауза $\overline{A_1} \vee \dots \vee \overline{A_k}$ без положительных литералов. В главе о SAT такие дизъюнкты назывались целями. Определённая клауза программы несёт ровно один положительный литерал — голову. SLD-шаг — резольюция этой пары: контрарную пару образуют голова клаузы и выбранная подцель, а резольвента снова оказывается клаузой без положительных литералов. Поиск при любой последовательности шагов остаётся внутри одного класса формул.

Шаги вывода механичны: правило вывода сохраняет истину, и машине не нужно понимать, о чём утверждение. Достаточно распознать форму термов и подставить нужное — следствие получается само. Синтаксис ведёт вывод, не заглядывая в смысл, — поэтому доказательство и может стать вычислением.

ОПРЕДЕЛЕНИЕ 16.12 SLD-шаг

Пусть цель $\leftarrow A_1, \dots, A_k$, а в программе есть клауза $A \leftarrow B_1, \dots, B_m$. **SLD-шаг** состоит из трёх частей:

1. Переменные клаузы переименовываются так, чтобы они не пересекались с переменными цели: это стандартизация-разделение (*standardizing apart*).
2. Пусть θ — наиболее общий унификатор первой подцели и головы клаузы. Если унификатора нет, клауза к шагу неприменима.
3. Цель заменяется на цель

$$\leftarrow (B_1, \dots, B_m, A_2, \dots, A_k)\theta.$$

Здесь A_1 — выбранная цель.

Например, для цели `ancestor(alice, Y)` и переименованной клаузы `ancestor(_1, _2) :- parent(_1, _2)` SLD-шаг даёт цель `parent(alice, Y)`. Выбор левой цели — селекция, принятая в Prolog.

Стандартизация-разделение защищает от ложного склеивания переменных. Если одно и то же правило применяется дважды, его переменные не должны совпасть: иначе шаги вывода связались бы там, где логика этого не требует. Рекурсивное правило `ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y)` применяется к цели `ancestor(alice, Y)`, а затем — к возникшей цели `ancestor(bob, Y)`. Без переименования переменные правила в обоих применениях были бы одними и теми же, и второй шаг вывода был бы вынужден согласовываться с первым там, где логически они независимы.

Конкретно: после первого применения выполнено $X = \text{alice}$. Второе применение унифицирует голову $\text{ancestor}(X, Y)$ с целью $\text{ancestor}(\text{bob}, Y)$. Нужно получить $X = \text{bob}$. Но уже выполнено $X = \text{alice}$, и унификация проваливается. Рекурсивная цепочка обрывается, хотя логически два шага независимы. Переименование даёт каждому применению свежие переменные. Второй шаг выполняет $X = \text{bob}$ независимо от первого.

ОПРЕДЕЛЕНИЕ 16.13 SLD-вывод

SLD-вывод — последовательность SLD-шагов из исходной цели.

ОПРЕДЕЛЕНИЕ 16.14 SLD-опровержение

SLD-вывод, заканчивающийся пустой целью, называется **SLD-опровержением**.

ОПРЕДЕЛЕНИЕ 16.15 Вычисленный ответ

Ограничение итоговой подстановки на переменные исходной цели называется **вычисленным ответом**.

Пустая цель означает успех: все подцели доказаны, и накопленная подстановка — это то, чему должны быть равны переменные запроса.

ТЕОРЕМА 16.2 Корректность SLD

Если цель имеет SLD-опровержение с вычисленным ответом θ , то универсальное замыкание $(A_1 \wedge \dots \wedge A_k)\theta$ — логическое следствие программы. Каждый ответ, который выдаёт машина, корректен.

Набросок доказательства

Докажем индукцией по длине опровержения.

База. Пустая цель следует из программы: она достигнута, когда все подцели доказаны.

Шаг. Новая цель получена применением унификатора θ этого шага. К ней применена клауза $A \leftarrow B_1, \dots, B_m$. Клауза входит в программу, поэтому $A\theta \leftarrow B_1\theta, \dots, B_m\theta$ — тоже следствие (переменные клаузы универсально квантифицированы, значит, любой её инстанс — следствие). Если новая цель $(B_1, \dots, B_m, A_2, \dots, A_k)\theta$ следует из программы, то следует и $A_1\theta, A_2\theta, \dots, A_k\theta$: ведь $A_1\theta = A\theta$ по определению унификатора. Поднимаясь по шагам к исходной цели и накапливая подстановки, получаем следствие для инстанции цели вычисленным ответом.

ТЕОРЕМА 16.3 Полнота SLD для определённых программ

Если экзистенциальное замыкание цели — логическое следствие определённой программы, то существует SLD-опровержение этой цели.

Набросок доказательства

Доказательство полноты опирается на два классических факта, которые мы здесь не доказываем.

Первый факт — теорема Эрбрана. Если цель следует из определённой программы, то она следует из некоторого конечного множества грунтовых инстанций клауз (инстанция — клауза, полученная подстановкой), и это устанавливается резолюцией.

Второй факт — лемма о подъёме. Каждый шаг такого грунтового резолюционного опровержения поднимается до шага SLD, где вместо конкретных значений подставляется более общий унификатор. Собирая поднятые шаги, получаем SLD-опровержение исходной цели.

Полнота утверждает существование опровержения. Конкретный поиск может его и не найти. Поиск в глубину может уйти в бесконечную ветвь и не добраться до существующего опровержения. Это различие — между логикой программы и её исполнением — объясняет, почему поиск рассматривается отдельно от логики.

Пример SLD-вывод запроса

Программа «родство», запрос ?- ancestor(alice, bob). Вывод из двух SLD-шагов:

1. Цель ancestor(alice, bob). Выбираем правило ancestor(X , Y) :- parent(X , Y), переименованное в ancestor($_1$, $_2$) :- parent($_1$, $_2$). Унификация с головой даёт { $_1 \rightarrow$ alice, $_2 \rightarrow$ bob}. Новая цель — parent(alice, bob).
2. Цель parent(alice, bob). Выбираем факт parent(alice, bob). Унификация пуста. Новая цель пуста.
3. Пустая цель — опровержение найдено, ответ: да.

Пример SLD-вывод с рекурсивным правилом

Программа «родство», запрос ?- ancestor(alice, Y). Первый ответ даёт базовое правило. Проследим, как рекурсивное правило приводит ко второму.

1. Цель ancestor(alice, Y). Рекурсивное правило переименовывается в ancestor($_1$, $_2$) :- parent($_1$, $_3$), ancestor($_3$, $_2$). Унификация с головой: { $_1 \rightarrow$ alice, $_2 \rightarrow Y$ }. Новая цель: parent(alice, $_3$), ancestor($_3$, Y).
2. Первая подцель parent(alice, $_3$) унифицируется с фактом parent(alice, bob): { $_3 \rightarrow$ bob}. Осталась цель: ancestor(bob, Y).

3. Базовое правило переименовывается в `ancestor(_4, _5) :- parent(_4, _5)`.
Унификация: $\{_4 \rightarrow \text{bob}, _5 \rightarrow Y\}$. Цель: `parent(bob, Y)`.
4. Факт `parent(bob, carol)` связывает Y с `carol`. Пустая цель — опровержение, второй ответ: $Y = \text{carol}$.

Свежие переменные на каждом применении — это стандартизация-разделение, разобранный выше.

```
// Поиск в глубину: стек отложенных выводов (choice points).
while let Some(ChoicePoint { goals, sigma }) = self.stack.pop() {
    if goals.is_empty() {
        return Some(sigma); // пустая цель: успех
    }
    let (head, rest) = goals.split_first().unwrap();
    if let Goal::Call(term) = apply_goal(head, &sigma) {
        let Some(clauses) = self.db.clauses_for(&term) else {
            continue; // подходящих клауз нет: ветвь обрывается
        };
        // Клаузы в порядке программы: первая должна быть испробована
        // первой, поэтому кладётся на стек последней (перебор в обратном
        // порядке).
        for clause in clauses.iter().rev() {
            let fresh_clause = rename_clause(clause, &mut self.fresh); //
            // свежие переменные
            let mut cand = sigma.clone();
            if unify(&term, &fresh_clause.head, &mut cand).is_ok() {
                self.stack.push(ChoicePoint {
                    goals: prepend(fresh_clause.body, rest.clone()),
                    sigma: cand,
                });
            }
        }
    }
}
```

Запрос `?- ancestor(alice, Y)` с правилами выше возвращает подстановку $\{Y \rightarrow \text{bob}\}$ первым ответом — то есть значения, которые машина накопила, доказывая цель. Реализация возвращает полные подстановки со свежими переменными клауз. Вычисленный ответ — их ограничение на переменные запроса.

Проверка SLD-резолюция

1. Что происходит на SLD-шаге, если выбранная подцель не унифицируется ни с одной головой?
2. Зачем переменные клаузы переименовываются при каждом применении?
3. Чем вычисленный ответ отличается от полной подстановки, накопленной к концу вывода?

§ 16.6 Поиск и бэктрекинг

Когда цели подходят несколько клауз, SLD-резолюция оставляет свободу выбора. Prolog выбирает первую клаузу, а если ветвь не приводит к успеху, возвращается к следующей. Это поиск в глубину по дереву всех возможных выводов — SLD-дереву, как на рис. 80.

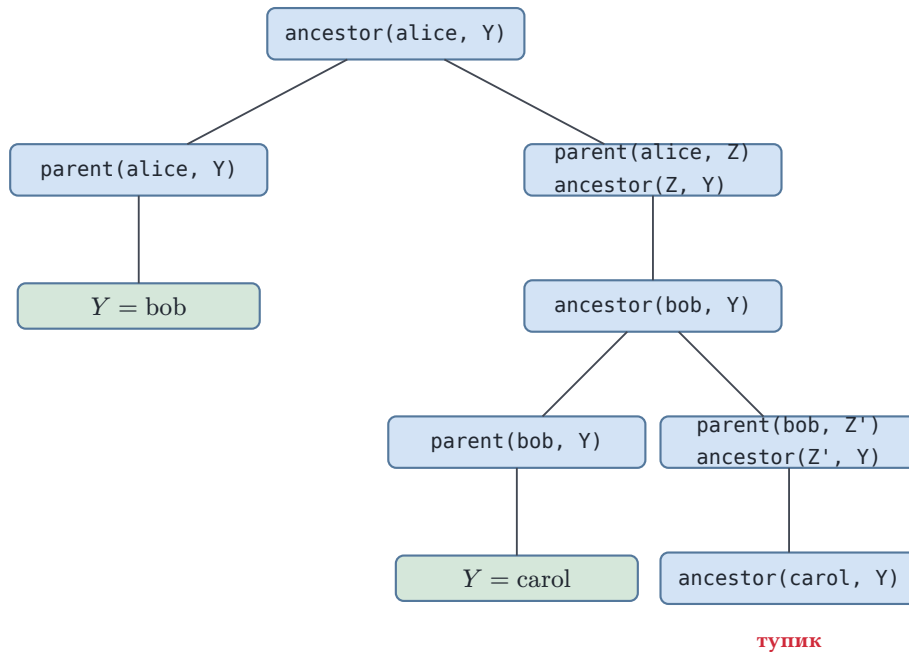


Рис. 80. SLD-дерево запроса ?- ancestor(alice, Y).

Корень дерева — исходная цель. Каждый узел — цель по ходу вывода. Каждая ветвь — применение одной клаузы. Левая ветвь опирается на базовое правило, правая — на рекурсивное, листья с ответами — успех, а лист ancestor(carol, Y) показывает ветвь, где ни одна клауза не подошла. Поиск в глубину идёт по самой левой ветви до конца: если она дала ответ, ответ выдан. Если упёрлась в тупик, поиск возвращается к ближайшей развилке и пробует следующую ветвь. Возврат к отложенному выбору называется *бэктрекингом*. Отложенные выборы лежат на стеке, и глубина стека — это глубина текущей ветви. Размер всего дерева здесь ни при чём.

Примечание Порядок ответов зависит от порядка клауз

Дерево показывает: последовательность ответов определяется порядком клауз в программе. С базовым правилом первым ответы идут $Y = \text{bob}$, $Y = \text{carol}$ — от ближнего предка к дальнему. Если рекурсивное правило поставить первым, поиск в глубину спускается до конца ветви, и порядок меняется. Множество ответов при этом не меняется: меняется только последовательность, в которой их находит машина.

Замечание Почему поиск в глубину?

Поиск в глубину требует памяти, пропорциональной длине текущей ветви: вся история выбора остаётся на стеке. Но поиск в глубину неполон: если бесконечная ветвь стоит раньше, поиск никогда не доберётся до ответа на соседней ветви. Поиск в ширину был бы полон, но хранил бы все незавершённые ветви сразу.

Prolog выбрал глубину: она проще, экономит память и на практике часто достаточно. Ричард О'Киф, автор классической книги о Prolog, выразил это афоризмом: Prolog — эффективный язык, потому что это очень глупый доказыватель теорем. Умный доказыватель взвешивал бы ветви и думал, куда пойти. Глупый идёт по первой и потому успевает. Неполнота оказывается платой за скорость: поиск иногда не доходит до ответа, зато доходит быстро.

Пример Бэктрекинг в действии

В программе «родство» запрос `?- ancestor(bob, Y).` имеет один ответ: `Y = carol`. Базовое правило даёт его сразу: `parent(bob, carol)`. Рекурсивное правило связывает `Z = carol` и сводит задачу к `ancestor(carol, Y)`, а у `carol` детей нет: ни факт `parent(carol, ...)`, ни рекурсивный шаг не дают результата. Ветвь умирает, поиск возвращается к развилке, но других отложенных выборов не осталось. Так выглядит бэктрекинг: неудачная ветвь, возврат к отложенному выбору, конец поиска.

§ 16.7 Списки

Почти все структуры данных в логическом программировании — списки. Список — это либо пустой список `[]`, либо голова с хвостом, который сам является списком. Синтаксический сахар: `[a, b, c]` — это сокращение для `.(a, .(b, .(c, [])))`.

Примечание Списки как термы

Список — соглашение о термах: `[]` — константа, `.` — функциональный символ arity 2. Запись `[X | Xs]` обозначает терм `.(X, Xs)`: голова и хвост. Рекурсивный обход от головы к хвосту делает списки естественным материалом для рекурсивных правил.

Пример Списки: `append`

Два правила склейки списков:

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

Запрос `?- append(Xs, Ys, [a, b]).` спрашивает, на какие части разбивается список `[a, b]`. Ответы:

1. `Xs = [], Ys = [a, b].`
2. `Xs = [a], Ys = [b].`
3. `Xs = [a, b], Ys = [].`

Тот же предикат работает и как конкатенация: `?- append([a], [b, c], Zs).` даёт `Zs = [a, b, c]`. Одна программа покрывает и деление, и склейку: направление вычисления определяет запрос. Предикат не помечает аргументы как вход и выход: роли расставляет запрос. `append(Xs, Ys, Zs)` — это отношение между тремя списками, и неизвестной может оказаться любая позиция.

Рекурсивное правило `append` — классический образец: база (пустой список) и шаг (отщепить голову, склеить хвост). Такой же образец обращает список:

```
reverse([], []).
reverse([X | Xs], R) :- reverse(Xs, Rs), append(Rs, [X], R).
```

Запрос `?- reverse([a, b, c], R).` даёт `R = [c, b, a]`. Каждый шаг рекурсии переносит одну голову из начала в конец.

§ 16.8 Рекурсия и терминация

Рекурсия — основной способ повторения в логическом программировании, но рекурсивное правило должно тратить ресурс. Как только рекурсивный вызов перестаёт уменьшать задачу, поиск закидывается.

Пример Левая рекурсия не завершается

Достижимость в графе допускает две записи:

```
path(X, Y) :- edge(X, Z), path(Z, Y).    % терминальная
path(X, Y) :- path(X, Z), edge(Z, Y).    % закидывается
```

На конечном ациклическом графе первое правило завершается: каждый шаг продвигает стартовую вершину вперёд, а цепь вершин конечна. На графе с циклом не завершается и оно: поиск не запоминает посещённые вершины и может ходить по циклу бесконечно. Во втором правиле рекурсивный вызов стоит первым: ресурс не тратится вовсе, и поиск бесконечно порождает всё более длинные цели. Машина не замечает закидывания: SLD-резолюция продолжает работать, но ни одна ветвь не приводит к ответу.

Рекурсивный вызов называется *хвостовым*, если он — последняя подцель правила: после него не остаётся работы. Такую схему называют *рекурсией по хвосту*. Исполняется она с постоянной памятью: возвращаться в правило после вызова не нужно,

и компилятор превращает вызов в переход. Правило `reverse([X | Xs], R) :- reverse(Xs, Rs), append(Rs, [X], R)`. хвостовым вызовом не обходится: после возврата из `reverse` ещё работает `append`. Накопитель делает то же определение хвостовым:

```
reverse_acc([], Acc, Acc).
reverse_acc([X | Xs], Acc, R) :- reverse_acc(Xs, [X | Acc], R).
```

Запрос `?- reverse_acc([a, b, c], [], R)`. переносит головы в накопитель: после трёх шагов накопитель равен `[c, b, a]`, и база унифицирует его с `R`.

Примечание Ограничение глубины

На практике поиск снабжают пределом глубины или числа шагов. Предел превращает бесконечный поиск в конечный, но с пропусками: ответ, лежащий глубже предела, не будет найден. Это инженерный компромисс: он решает вопрос о времени, оставляя вопрос об истине в стороне.

Другой ответ на зацикливание — табулирование: исполнитель запоминает подцели и их ответы. Повторный вызов подцели не вычисляется заново, а берёт ответы из таблицы. В Datalog, где функциональных символов нет и грунтовых атомов конечное число, табулирование превращает бесконечный поиск в завершающийся. Такая схема известна как SLG-резолюция и реализована, например, в системе XSB.

§ 16.9 * Арифметика: `is/2`

Унификация сравнивает термы по форме и ничего не вычисляет. Запрос `?- X = 2 + 3` связывает `X` с термом `+(2, 3)`, а не с числом 5: плюс здесь — функтор арности 2, и никакая арифметика не выполняется. Вычисление в язык вводит встроенный предикат `is/2`. Цель с `is/2` вычисляет выражение в правом аргументе и унифицирует результат с левым. Запрос `?- X is 2 + 3` даёт `X = 5`. Выражение собирается из чисел, операций `+`, `-`, `*`, `//`, `mod` и переменных, и к моменту вызова все его переменные обязаны быть связаны. Запрос `?- X is Y + 1` со свободной `Y` приводит к ошибке исполнения: значение выражения не определено. Сравнения `<`, `>`, `=`, `=:`, `>=` устроены так же: обе стороны вычисляются, затем сравниваются значения. Арифметика — не отношение: перестановка аргументов меняет смысл цели, направление вычисления фиксировано.

Пример Сумма элементов списка

Сумма считается рекурсией по списку и одним `is` на шаг:

```
sum_list([], 0).
sum_list([X | Xs], S) :- sum_list(Xs, S1), S is S1 + X.
```

Запрос `?- sum_list([1, 2, 3], S)`. спускается к базе `sum_list([], 0)`, а на каждом возврате связывает сумму: $3 = 3 + 0$, $5 = 2 + 3$, $6 = 1 + 5$, итог $S = 6$. Вызов `sum_list(Xs, S1)` здесь не хвостовой: после него остаётся сложение.

§ 16.10 * Отрицание как провал и `cut`

Две черты Prolog определяют практический язык, но выходят за рамки чистой логики: *отрицание как провал* и *cut*.

При отрицании как провале цель `not(P)` считается истинной, если поиск `P` не нашёл ни одного решения. Это не логическое отрицание: из того, что программа не доказывает `P`, не следует, что `P` ложно. Отрицание как провал опирается на *замкнутый мир*: база знаний считается полным описанием истины, поэтому отсутствие доказательства трактуется как ложность.

Пример Отрицание как провал

База: единственный факт `parent(alice, bob)`. Запрос `?- not(parent(alice, carol))` успешен: доказательства нет — значит, по замкнутому миру, ложно. Запрос `?- not(X = 1)` терпит неудачу: `X = 1` унифицируется (связывает переменную), поиск успешен, и отрицание как провал отвергает цель. Поэтому `?- not(not(X = 1))` успешен, но *не связывает X*, тогда как сам `X = 1` связывает: `not(not(P))` — не то же самое, что `P`.

Отрицание как провал рассчитано на связанные цели. Безопасным называют вызов `not(G)`, при котором все переменные `G` к моменту вызова уже получили значения. Поиск перебирает конечный набор вариантов, и провал означает, что ни один не подошёл. Со свободной переменной вызов теряет смысл: `not(X = 1)` из примера выше как раз небезопасен. В правилах отрицательную подцель поэтому ставят после тех, что связывают её переменные.

`Cut !` — оператор, управляющий поиском. Когда цель `!` достигнута, отбрасываются все отложенные выборы, сделанные с момента входа в текущее правило. `Cut` не добавляет логики: он лишь отсекает ветви дерева поиска. Бездумный `cut` может отсечь и правильные ответы. Осознанный `cut` превращает экспоненциальный поиск в линейный.

Пример Cut в действии

Правило для максимума с отсечением:

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

Запрос ?- $\text{max}(5, 3, M)$: первое правило успешно ($5 \geq 3$), `cut` отбрасывает отложенный выбор второго правила, ответ $M = 5$. Без `cut` поиск вернулся бы ко второму правилу и выдал бы ошибочный $M = 3$. Запрос ?- $\text{max}(3, 5, M)$: первое правило терпит неудачу ($3 \geq 5$ ложно), `cut` не достигнут, и второе правило даёт $M = 5$.

Замечание Логика и управление

Отрицание как провал и `cut` — признание того, что чистой декларативности недостаточно. Полный поиск по SLD-дереву решает в принципе, но не в срок. Управление поиском — порядок клауз, `cut`, пределы — решает в срок, но покидает чистую логику. Логическое программирование — это диалог между тем, что истинно, и тем, что быстро находится. В ядре языка, построенном в этой главе, ни `cut`, ни отрицание не нужны: они появляются на следующем уровне, там, где декларативность встречается с эффективностью.

§ 16.11 Итоги главы

Логическое программирование замыкает дугу, начатую резолюцией в главе 14: то же правило, что доказывает теоремы и решает выполнимость, становится здесь языком. Программа — теория, выполнение — поиск доказательства, а ответы — вычисленные подстановки. Термы описывают данные, унификация решает уравнения между ними, а SLD-резолюция ведёт поиск по дереву целей.

Механику мы разобрали до конца. Наиболее общий унификатор единственен с точностью до переименования, а `occurs-check` отбрасывает уравнения вроде $X = f(X)$: без такой проверки подстановка зацикливается. Поиск с возвратом обходит SLD-дерево и выдаёт ответы в последовательности, заданной порядком клауз. Множество ответов от этой последовательности не зависит. Списки и рекурсия живут на том же каркасе.

Практика добавляет к чистой логике управление: порядок клауз, бэктрекинг, `occurs-check`, отрицание как провал и `cut`. Эти механизмы делают язык пригодным для реальных программ, но требуют понимания того, как поиск исполняется. Рекурсия по хвосту исполняется с постоянной памятью: рекурсивный вызов — последняя подцель правила, после него не остаётся работы. Терминация — отдельная забота: рекурсия обязана тратить ресурс, иначе поиск зацикливается. Левая рекурсия зацикливается даже на конечных данных, тогда как правая завершается.

Граница между логикой и её исполнением — главный урок: программа осмысленна как теория, но ведёт себя как алгоритм, поэтому то, что программа утверждает, и то, что она вычисляет, — не одно и то же. Следующая глава (глава 17) уйдёт от логики к структурам, сохранив тот же стиль доказательства.

Проверка Проверьте себя

1. Почему МГУ единствен только с точностью до переименования, тогда как буквально унификаторов несколько?
2. Что защищает occurs-check, и что ломается без него?
3. Почему порядок клауз меняет порядок ответов, тогда как их множество остаётся прежним?
4. В каком смысле SLD-резолуция корректна, а в каком полна? Что теряет поиск в глубину?
5. Почему левая рекурсия заикливается, тогда как правая завершается на конечных ациклических данных?
6. Объясните, почему $\text{not}(P)$ как провал не является логическим отрицанием.

Что дальше?

Логическое программирование завершает логический блок книги. Следующая глава — матроиды (глава 17): единая структура, стоящая за жадными алгоритмами из глав о графах и кодах. Затем — комбинаторика (глава 18): если логика спрашивает, что из чего следует, комбинаторика спрашивает, сколько существует способов.

§ 16.12 Упражнения*Термы и подстановки.*

1. Нарисуйте дерево терма $s(\text{cons}(a, X))$: какие узлы — переменные, какие — константы?
2. Примените по определению (одновременно) подстановку $\sigma = \{X \rightarrow \text{bob}, Y \rightarrow f(X)\}$ к терму $\text{ancestor}(X, Y)$. Объясните, почему в результате вхождение переменной внутри $f(X)$ не заменяется повторно. Чем результат отличается от того, что вернула бы рекурсивная реализация из главы?
3. Какие из термов $\text{parent}(\text{alice}, X)$, $f(g(a))$, $g(X, X)$ грунтовые?

Унификация.

4. Найдите МГУ термов $p(f(X), Y)$ и $p(Z, f(a))$ или объясните, почему их нет.
5. Найдите МГУ термов $q(X, g(Y))$ и $q(g(Z), Z)$.
6. Объясните, почему X и $f(X)$ не унифицируются, и что пошло бы не так без occurs-check.
7. Почему МГУ единствен только с точностью до переименования? Приведите два МГУ одной пары термов.

Программы и SLD.

8. Для программы «родство» выпишите SLD-вывод запроса $?- \text{ancestor}(\text{bob}, \text{carol})$ по шагам.

9. В программе «родство» переставьте правила ancestor местами. В каком порядке пойдут ответы запроса $?- \text{ancestor}(\text{alice}, Y)$? Изменится ли множество ответов?
10. Почему порядок клауз меняет порядок ответов, тогда как их множество остаётся прежним?

Списки.

11. Выпишите все пары (Xs, Ys) , для которых $\text{append}(Xs, Ys, [a, b])$ истинно.
12. Напишите предикат $\text{member}(X, L)$, истинный, когда X входит в список L .
13. Напишите предикат $\text{last}(L, X)$, истинный, когда X — последний элемент списка L .

Терминация и отрицание.

14. Почему $\text{path}(X, Y) :- \text{edge}(X, Z), \text{path}(Z, Y)$ завершается на конечном ациклическом графе, а на графе с циклом может не завершиться? И почему $\text{path}(X, Y) :- \text{path}(X, Z), \text{edge}(Z, Y)$ зацикливается даже на дереве?
15. * Объясните, почему $\text{not}(P)$ как провал не является логическим отрицанием. Приведите программу, где $\text{not}(\text{not}(P))$ отличается от P .

Арифметика.

16. В запросах $?- X = 2 + 3.$ и $?- X \text{ is } 2 + 3.$ переменная X получает разные значения. Выпишите оба значения и объясните, в каком из них участвует вычисление.
17. Проследите вычисление $?- \text{sum_list}([1, 2, 3], S).$ по шагам. Выпишите последовательность целей и порядок связывания S .

ПРОЕКТ Интерпретатор логического программирования: унификация и поиск

Логическая программа — база фактов и правил, а вычисление — доказательство запроса. Унификация сопоставляет термы и выводит подстановку, а возврат обходит дерево поиска и перечисляет все решения. Соберите интерпретатор, который по базе фактов и правил отвечает на запросы.

① Термы и унификация

Представьте терм: атом, переменная, составной терм $f(t_1, \dots, t_n)$, список, и выведите его в форме главы. Реализуйте унификацию: два терма сопоставляются, и выводится наиболее общая подстановка (переменная $\rightarrow$ терм). Атом с атомом — равенство, переменная с термом — связывание, составной с составным — по аргументам. Добавьте проверку на вхождение: $X = f(X)$ отвергается.

② База фактов, правил и поиск

Постройте базу определённых клауз: факт p и правило $p : -q_1, \dots, q_k$, с переименованием переменных правила при каждом применении. Реализуйте SLD-резолюцию: левая цель, унификация с головой, замена телом, рекурсия, а при тупике — возврат к следующей клаузе. Перечислите все решения и

добавьте ограничение глубины, чтобы леворекурсивные правила не зацикливались. Проверьте, что `member(X, [1, 2, 3])` даёт три решения и что предки в базе семейства перечисляются полностью.

③ *Отсечение и отрицание как провал*

Добавьте отсечение `!`: оно срабатывает всегда и запрещает возврат к остальным клаузам текущего предиката. Проверьте на `max(X, Y, X) :- X >= Y, !.` и `max(_, Y, Y) :-` без отсечения вторая клауза выдала бы лишний ответ. Добавьте отрицание как провал `\+`: цель `\+ G` истинна, когда у `G` нет ни одного решения. Проверьте `different(X, Y) :- \+ (X = Y) :-` запрос `different(a, b)` истинен, а `different(a, a)` ложен.

④ *Арифметика и сравнения*

Добавьте арифметику `is/2`: выражение из чисел и операций `+`, `-`, `*`, `/`, `//`, `mod` вычисляется, результат унифицируется с переменной. Добавьте сравнения `==`, `=\=`, `<`, `>`, `=<`, `>=`, которые сравнивают значения двух выражений. Проверьте рекурсивный факториал: правило `fact(N, F) :- N > 0, N1 is N - 1, fact(N1, F1), F is N * F1.` вместе с фактом `fact(0, 1).` даёт `fact(4, F) -> F = 24.`

⑤ *Парсер и REPL*

Реализуйте парсер синтаксиса Prolog: факты, правила `:-`, атомы, переменные, списки `[a | b]`, комментарии. Научите парсер разбирать операторы: арифметику, сравнения, `=`, отсечение `!`, отрицание `\+`. Соберите REPL: цикл читает запросы `?- цель.` и печатает все решения с подстановками.

Итог: интерпретатор, который по базе фактов и правил отвечает на запрос, перечисляя все решения: унификация выдаёт наиболее общую подстановку, отсечение и отрицание как провал управляют поиском, арифметика `is/2` считает, а REPL превращает его в интерактивный инструмент.

XVII

Матроиды

Монеты достоинством 25, 10 и 1, а сумма — 30. Жадный рефлекс подсказывает брать каждый раз самую крупную монету, помещающуюся в остаток. Результат — 25 и пять единиц: шесть монет там, где хватило бы трёх по десять.

Это *жадный алгоритм*: на каждом шаге — наилучший из доступных вариантов, без возврата к сделанному выбору. На размене он ошибся, и таких задач большинство: локально выгодный шаг не обязан складываться в оптимум. Коммивояжёр, каждый раз едущий в ближайший непосещённый город, привозит маршрут в разы длиннее оптимального.

Рядом живут задачи, где та же жадность безошибочна. Минимальное остовное дерево строится рёбрами по возрастанию весов: очередное добавляется, если не создаёт цикла. Интервальное расписание набирает из совместимых встреч первой ту, что раньше кончается. Расписание с дедлайнами сортирует проекты по убыванию ценности и берёт каждый, успевающий к своему сроку. Во всех трёх ответ точен при любых весах, без единого исключения.

Оптимальность каждой можно было бы доказывать по отдельности, но доказательства оказываются одним и тем же рассуждением. Объекты разные — рёбра, встречи, проекты — а ход один: оптимальное решение перестраивается в жадное без потери суммарного веса. За всеми тремя задачами стоит одна структура, частными случаями которой служат и линейно независимые системы векторов, и ациклические наборы рёбер графа. Она называется *матроидом*.

Там, где эта структура есть, оптимальность жадного алгоритма доказывается как теорема. Где её нет, жадность остаётся эвристикой — размен это уже показал.

Какое свойство задачи превращает жадность из эвристики в алгоритм с гарантией?

ИСТОРИЯ Происхождение понятия

Понятие матроида ввёл Хасслер Уитни¹⁴¹. Уитни установил, что линейная независимость векторов и ациклическость множеств рёбер графа удовлетворяют одним и тем же аксиомам. Термин «матроид» образован от слова «матрица» и означает структуру, родственную матрицам. Аксиомы матроида рассматривал также ван дер Варден (1937), однако закрепился термин Уитни.

¹⁴¹H. Whitney. «On the Abstract Properties of Linear Dependence». American Journal of Mathematics, 1935.

ОБЗОР ГЛАВЫ

Сначала формулируется общая задача: дано множество элементов с весами и семейство допустимых подмножеств. Требуется найти допустимое множество максимального веса. Затем разбираются три задачи, в которых жадный алгоритм даёт точное решение, и выделяются два общих свойства их допустимых семейств: наследственность и свойство замены. Семейство с этими свойствами называется матроидом. Приводятся три основных примера матроидов. Формулируется и доказывается теорема Радо–Эдмондса: жадный алгоритм оптимален на матроидах. Контрпример показывает, что без свойства замены жадный алгоритм может быть неоптимален. В конце рассматриваются применения: алгоритм Крускала, расписание с дедлайнами и интервальное расписание.

После этой главы вы сможете:

1. формулировать аксиомы матроида и проверять их для конкретного семейства множеств;
2. приводить примеры графического, линейного и униформного матроидов;
3. формулировать теорему Радо–Эдмондса и объяснять, почему жадный алгоритм на матроиде оптимален;
4. объяснять, почему без свойства замены жадный алгоритм может быть неоптимален;
5. сводить задачу расписания с дедлайнами к задаче о матроиде.

§ 17.1 Задача выбора и жадный алгоритм

Возьмём набор проектов, из которых можно осуществить лишь некоторые: каждый проект приносит выгоду, но проекты ограничены в совместности. Возьмём команду, которую набирают из сотрудников: ценность команды складывается из ценностей участников, а состав ограничен требованиями. Задач такого рода много, и все они имеют одну и ту же форму. Есть конечное множество объектов, каждому объекту приписана ценность, а среди подмножеств выделены *допустимые* (*feasible*). Нужно выбрать допустимое подмножество наибольшей суммарной ценности. Эту общую постановку и рассмотрим.

ОПРЕДЕЛЕНИЕ 17.1 Задача выбора

Дано конечное множество E . Задана весовая функция $w : E \rightarrow \mathbb{R}_{\geq 0}$. Подмножества из семейства $\mathcal{F}$ называются допустимыми. Требуется найти допустимое множество максимального веса.

Примечание Неотрицательные веса

Веса считаются неотрицательными. Если бы среди весов встречались отрицательные, задача во многих случаях вырождалась бы: пустое множество имеет вес нуль, и любой набор с отрицательным суммарным весом был бы заведомо неоптимален.

Естественный способ строить решение такой задачи — жадный алгоритм. Он просматривает объекты по убыванию ценности и включает каждый объект, если допустимость решения при этом сохраняется.

ОПРЕДЕЛЕНИЕ 17.2 Жадный алгоритм

Жадный алгоритм упорядочивает элементы множества E по убыванию веса и последовательно добавляет каждый элемент к текущему множеству, если допустимость при этом сохраняется.

Реализация жадного алгоритма коротка. От задачи требуется лишь проверка независимости — метод `is_independent`, остальное — сортировка и перебор.

```
/// Жадный алгоритм: элементы по убыванию веса,
/// включить, если независимость сохраняется.
pub fn greedy<M: Matroid>(m: &M, weights: &[u32]) -> Vec<u32> {
    let mut order: Vec<u32> = (0..m.n()).collect();
    order.sort_by(|&a, &b| weights[b as usize].cmp(&weights[a as usize]));
    let mut chosen = Vec::new();
    for e in order {
        let mut candidate = chosen.clone();
        candidate.push(e);
        candidate.sort_unstable();
        if m.is_independent(&candidate) {
            chosen = candidate;
        }
    }
    chosen }
```

Трейт `Matroid` — это и есть допустимое семейство: число элементов `n()` и проверка независимости `is_independent`. Вся специфика задачи спрятана в `is_independent`: для лесов это проверка ацикличности, для векторов — линейная независимость.

Примечание Цена оракула независимости

Время работы жадного алгоритма складывается из сортировки и не более $|E|$ обращений к проверке независимости, а цена одного обращения зависит от задачи. Для лесов ацикличность проверяет система непересекающихся множеств — почти линейное время, для двоичных векторов независимость проверяет исключение Гаусса за кубическое время от размерности. Теорема Радо–Эдмондса, доказываемая ниже, гарантирует оптимальность при любой реализации такой проверки, но цену вызова она не контролирует.

Жадный алгоритм прост, но в общем случае не оптимален.

Пример Размен монет

Пусть в обращении есть монеты достоинством 25, 10, 1 цент. Чтобы выдать 30 центов, жадный алгоритм выбирает монету 25, а затем добавляет пять монет по одному центу — всего шесть монет. Между тем 30 центов можно выдать тремя монетами по 10. Жадный алгоритм неоптимален.

С жадными алгоритмами мы уже работали: алгоритм Крускала в главе 9 и алгоритм Хаффмана в главе 13. Их оптимальность доказывалась в каждой главе отдельно. За этими доказательствами стоит одна структура: существуют задачи, где жадный алгоритм оптимален для любых весов, и этот класс допускает аксиоматическую характеристику.

§ 17.2 Три задачи, где жадный алгоритм оптимален

Применим общую постановку к трём задачам. Каждая приходит из своей области: одна — из теории графов, другая — из линейной алгебры, третья — из простейшего счёта. В каждой из них жадный алгоритм даёт оптимальное решение. В конце раздела станет видно, что общего у этих задач.

Начнём с задачи из теории графов.

Пример Максимальный лес

Рассмотрим треугольник с рёбрами $a = (1, 2)$, $b = (2, 3)$, $c = (1, 3)$. Веса рёбер: $w(a) = 5$, $w(b) = 4$, $w(c) = 3$. Допустимыми объявляются ациклические наборы рёбер — леса.

Жадный алгоритм выбирает рёбра a, b . Ребро c отбрасывает: его добавление создаёт цикл. Результат — остов весом $5 + 4 = 9$. Другие остовы треугольника имеют вес 8 или 7, поэтому результат оптимален.

Вторая задача приходит из линейной алгебры.

Пример Базис максимального веса

Рассмотрим три вектора плоскости: $v_1 = (1, 0)$, $v_2 = (0, 1)$, $v_3 = (1, 1)$. Их веса равны 5, 4, 3. Допустимыми объявляются линейно независимые наборы векторов.

Жадный алгоритм выбирает векторы v_1, v_2 . Вектор v_3 отбрасывает: все три вектора линейно зависимы. Результат — базис веса $5 + 4 = 9$. Базисами плоскости являются любые пары данных векторов, и их веса равны 9, 8 и 7, поэтому результат оптимален.

Третья задача — самая простая.

Пример Набор ограниченного размера

Даны четыре элемента с весами 5, 3, 4, 2. Допустимы наборы размера не более двух. Жадный алгоритм выбирает элементы весов 5 и 4. Результат — допустимое множество веса 9. Любой допустимый набор содержит не более двух элементов, а сумма двух самых тяжёлых равна $5 + 4 = 9$, поэтому результат оптимален.

Три задачи задают три правила допустимости: ацикличность, линейную независимость и ограничение на размер. Жадный алгоритм оптимален в каждой из них, и по отдельности это могло бы быть совпадением. Общая причина одна — устройство допустимых семейств, и сравнение этих семейств — следующий шаг.

§ 17.3 Свойства допустимых семейств

Сравним допустимые семейства трёх задач. Окажется, что во всех трёх выполнены одни и те же два свойства.

Первое общее свойство — *наследственность*.

ОПРЕДЕЛЕНИЕ 17.3 Наследственность

Семейство $\mathcal{F}$ называется **наследственным**, если каждое подмножество допустимого множества допустимо, то есть $A \in \mathcal{F} \wedge B \subseteq A \implies B \in \mathcal{F}$.

Наследственность означает, что допустимость не портится при удалении элементов: выбросив часть элементов из допустимого решения, нельзя получить недопустимое. Наследственность объясняет и то, почему жадному алгоритму достаточно одного прохода. Элемент, который нельзя добавить сейчас, не станет допустимым и позже: иначе добавленное множество содержало бы недопустимое подмножество. Свойство легко проверить для всех трёх задач.

Пример Наследственность в трёх задачах

Подмножество леса — лес: удаление рёбер не создаёт цикл. Подмножество линейно независимого набора линейно независимо: удаление векторов не создаёт зависимость. Подмножество набора размера не более k имеет размер не более k .

Второе общее свойство тоньше. Рассмотрим два допустимых множества A, B с $|A| < |B|$. Среди элементов $B \setminus A$ всегда найдётся такой, который можно добавить к меньшему множеству, сохранив допустимость. Это свойство — *свойство замены* (*exchange property*).

Представьте жадный процесс: мы идём по элементам и берём всё, что можно добавить, не нарушая допустимости. Свойство замены — гарантия, что такой процесс не зайдёт в тупик. Пока существует допустимое множество большего размера, текущее

всегда можно пополнить элементом из него. Жадное построение не остановится на локальном максимуме, если есть куда расти.

ОПРЕДЕЛЕНИЕ 17.4 Свойство замены

Семейство $\mathcal{F}$ обладает **свойством замены**, если $A, B \in \mathcal{F} \wedge |A| < |B| \implies \exists x \in B \setminus A : A \cup \{x\} \in \mathcal{F}$.

Проверим свойство замены для трёх задач.

В задаче о лесе оно выполнено. Если $|B| > |A|$, то найдётся ребро из $B \setminus A$, добавление которого к меньшему набору не создаёт цикл. В противном случае каждое ребро из большего набора соединяло бы две вершины одной компоненты связности меньшего леса, и в большем наборе было бы не больше рёбер, чем в меньшем. Это же рассуждение использовано в доказательстве того, что леса образуют матроид.

Пример Свойство замены в задаче о лесе

В треугольнике рассмотрим допустимые множества $A = \{a\}$, $B = \{b, c\}$. Имеем $|A| = 1 < 2 = |B|$. Элемент $b \in B \setminus A$ можно добавить. Множество $\{a, b\}$ ациклично.

Для линейно независимых наборов свойство замены выполнено. Если $|B| > |A|$, то в B найдётся вектор, не лежащий в линейной оболочке (*linear span*) меньшего набора. Иначе набор целиком лежал бы в пространстве размерности $|A| < |B|$, что невозможно для линейно независимого набора.

Пример Замена в линейном матроиде

На векторах плоскости $v_1 = (1, 0)$, $v_2 = (0, 1)$, $v_3 = (1, 1)$ возьмём независимые множества $A = \{v_1\}$, $B = \{v_2, v_3\}$.

Имеем $|A| = 1 < 2 = |B|$. Оба вектора из $B \setminus A$ подходят. Вектор $v_2 = (0, 1)$ не лежит в оболочке v_1 — это ось абсцисс. Не лежит в ней и $v_3 = v_1 + v_2$. Поэтому и $\{v_1, v_2\}$, и $\{v_1, v_3\}$ линейно независимы.

Для наборов размера не более k свойство замены проверяется непосредственно. Если $|A| < |B| \leq k$, то любой элемент из $B \setminus A$ можно добавить.

Итак, в каждом из трёх семейств выполнены наследственность и свойство замены. Именно эти два свойства определяют матроид. Схема трёх задач, ведущих к понятию матроида, приведена на рисунке 81.

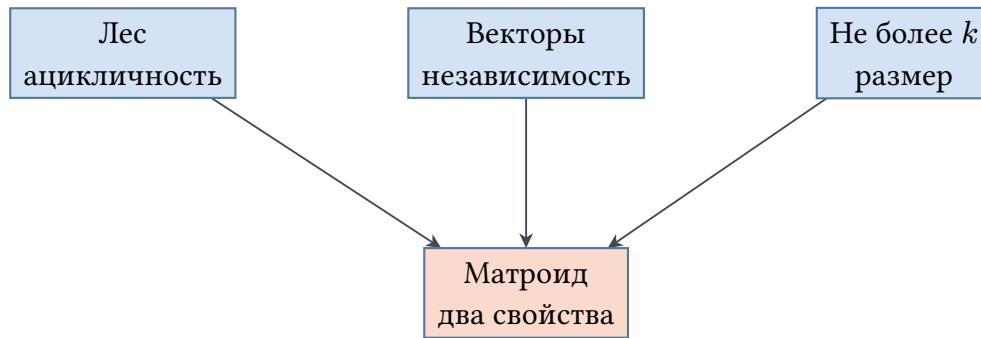


Рис. 81. Три источника — один матроид.

§ 17.4 Определение матроида

Два свойства — наследственность и свойство замены — выделяют класс семейств, для которых жадный алгоритм оптимален. Дадим этому классу имя.

ОПРЕДЕЛЕНИЕ 17.5 Матроид

Пусть E — конечное множество. Пусть $\mathcal{I}$ — семейство его подмножеств. Пара $(E, \mathcal{I})$ называется **матроидом**, если выполнены следующие условия:

1. **Непустота:** семейство $\mathcal{I}$ содержит хотя бы одно множество.
2. **Наследственность:** $A \in \mathcal{I} \wedge B \subseteq A \implies B \in \mathcal{I}$.
3. **Свойство замены:** $A, B \in \mathcal{I} \wedge |A| < |B| \implies \exists x \in B \setminus A : A \cup \{x\} \in \mathcal{I}$.

Множества из $\mathcal{I}$ называются **независимыми множествами**. Элементы множества E называются **элементами матроида**.

Пустое множество всегда независимо, так как из непустоты семейства $\mathcal{I}$ и наследственности следует $\emptyset \in \mathcal{I}$. Свойство замены означает, что любое независимое множество можно увеличивать, добавляя элементы из большего независимого множества, пока их размеры не сравняются. Название «замена» отражает механику из линейной алгебры, где из базиса всегда можно выбросить один вектор и добавить другой, не потеряв свойства быть базисом. В общей формулировке элемент из большего независимого множества подставляется в меньшее, и независимость при этом сохраняется.

Свойство замены не следует из наследственности.

Пример Наследственность без свойства замены

Пусть $E = \{1, 2, 3\}$. Пусть $\mathcal{I} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}\}$. Наследственность выполнена: любое подмножество множества из $\mathcal{I}$ тоже принадлежит семейству.

Свойство замены нарушено. Возьмём множества $A = \{3\}, B = \{1, 2\}$. Выполнено $|A| < |B|$. Однако ни одно из множеств $\{1, 3\}, \{2, 3\}$ не принадлежит семейству. Семейство $\mathcal{I}$ не является матроидом.

С матроидом связаны два стандартных понятия.

ОПРЕДЕЛЕНИЕ 17.6 База

Максимальное по включению независимое множество называется **базой**.

ОПРЕДЕЛЕНИЕ 17.7 Ранг

Размер базы называется **рангом**.

Из свойства замены следует, что все базы матроида имеют одинаковый размер. Доказательство вынесено в упражнение.

Максимальное и наибольшее — не одно и то же. Максимальное множество нельзя увеличить, добавив элемент, а наибольшее содержит больше всего элементов. В произвольной системе независимости максимальное множество может быть меньше наибольшего, но в матроиде они совпадают — это следствие свойства замены.

Пример Максимальное, но не наибольшее

В системе независимости $\mathcal{I} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}\}$ множество $\{3\}$ максимально: ни один элемент нельзя добавить, не покинув семейство.

При этом наибольшее независимое множество — $\{1, 2\}$, и оно больше максимального $\{3\}$. В матроиде такое расхождение невозможно: свойство замены увеличивает любую базу, пока её размер не сравняется с наибольшей.

В линейном матроиде база совпадает с базисом подпространства, натянутого на E . В графическом матроиде базы — остовные леса. Если граф связан, базы — остовные деревья.

Пример Ранг треугольника

В треугольнике с рёбрами a, b, c базы — пары рёбер, поэтому ранг матроида равен 2.

Внутри подмножества $\{a\}$ наибольшее независимое подмножество — оно само, и его размер равен 1. Внутри $\{a, b, c\}$ — любая пара рёбер: три ребра зависимы, а любые два независимы.

Замечание Почему именно независимые множества?

Матроид можно задать не только через независимые множества, но и через базы, через функцию ранга и через оператор замыкания. Определение через независимые множества выбрано потому, что проверка независимости — операция, которую жадный алгоритм выполняет на каждом шаге.

Проверка Определения матроида

1. Почему из непустоты семейства $\mathcal{I}$ и наследственности следует, что $\emptyset \in \mathcal{I}$?
2. Проверьте свойство замены для пары $A = \{1\}, B = \{2, 3\}$ в семействе всех подмножеств множества $E = \{1, 2, 3\}$ размера не более двух.
3. Базы графического матроида связного графа — остовные деревья. Чему равен ранг этого матроида, выраженный через число вершин графа?

§ 17.5 Примеры матроидов

Три семейства из начала главы — это матроиды. Проверим аксиомы для каждого из них.

Начнём с графического матроида.

ОПРЕДЕЛЕНИЕ 17.8 Графический матроид

Пусть G — граф. Пусть E — множество его рёбер. Независимыми объявляются ациклические наборы рёбер.

Независимость рёбер устроена так же, как независимость векторов. Цикл — графовый аналог нуля: пройдя по циклу, возвращаешься в исходную вершину, как линейная комбинация, равная нулю, возвращает в начало координат. Поэтому ацикличность играет в графах ту же роль, что линейная независимость в алгебре.

УТВЕРЖДЕНИЕ 17.1

Семейство лесов графа образует матроид.

Доказательство

Проверим две аксиомы матроида по отдельности.

Наследственность следует из того, что подмножество ациклического набора ациклично.

Свойство замены. Пусть A, B — леса с $|A| < |B|$. Предположим, что ни одно ребро из $B \setminus A$ нельзя добавить без образования цикла. Тогда каждое ребро B соединяет две вершины одной компоненты связности меньшего леса. В каждой компоненте меньшего леса рёбер меньше, чем вершин, откуда $|B| \leq |A|$, что противоречит условию $|A| < |B|$.

Пример Леса треугольника

В треугольнике с рёбрами a, b, c независимы пустое множество, каждое одиночное ребро и каждая пара рёбер. Все три ребра зависимы: они образуют цикл. Базы — остовные деревья, то есть пары рёбер.

Следующий пример — линейный матроид.

ОПРЕДЕЛЕНИЕ 17.9 **Линейный матроид**

Пусть V — векторное пространство, например $\mathbb{R}^n$ или $\text{GF}(2)^n$. Пусть E — множество его векторов. Независимыми объявляются линейно независимые наборы векторов.

УТВЕРЖДЕНИЕ 17.2

Семейство линейно независимых наборов образует матроид.

Доказательство

Проверим две аксиомы матроида.

Наследственность следует из определения линейной независимости.

Свойство замены. Пусть A, B — линейно независимые наборы с $|A| < |B|$. Набор B не может целиком лежать в линейной оболочке меньшего набора, размерность которой равна $|A|$. Следовательно, в большем наборе есть вектор x , не принадлежащий линейной оболочке меньшего, и набор $A \cup \{x\}$ линейно независим.

Над $\text{GF}(2)$ — как в главе 13 — векторы двоичные, и проверка независимости сводится к исключению Гаусса: новый вектор редуцируется относительно уже выбранных, и нулевая строка означает зависимость.

```
/// Линейно независимы ли выбранные двоичные векторы?
fn is_independent(vectors: &[Vec<u8>], set: &[u32]) -> bool {
    let dim = vectors[0].len();
    let mut basis: Vec<Vec<u8>> = Vec::new(); // редуцированный базис
    for &i in set {
        let mut row = vectors[i as usize].clone();
        for b in &basis {
            let pivot = b.iter().position(|&x| x == 1).unwrap();
            if row[pivot] == 1 {
                for c in 0..dim {
                    row[c] ^= b[c];
                }
            }
        }
    }
    if row.iter().all(|&x| x == 0) {
        return false; // вектор --- комбинация остальных
    }
}
```



```

    }
    basis.push(row);
  }
  true
}

```

Три вектора $(1, 0)$, $(0, 1)$, $(1, 1)$ независимы попарно, но все вместе зависимы: третий — сумма первых двух.

Пример Векторы плоскости

Для векторов $v_1 = (1, 0)$, $v_2 = (0, 1)$, $v_3 = (1, 1)$ независимы пустое множество, все одиночные векторы и все пары векторов. Все три вектора зависимы: $v_3 = v_1 + v_2$. Базы — базисы плоскости, то есть любые пары данных векторов.

Наконец, униформный матроид.

ОПРЕДЕЛЕНИЕ 17.10 Униформный матроид $U_{k,n}$

Пусть $|E| = n$. Независимыми объявляются подмножества размера не более k .

УТВЕРЖДЕНИЕ 17.3

Семейство подмножеств размера не более k образует матроид.

Доказательство

Проверим две аксиомы матроида.

Наследственность проверяется непосредственно — подмножество набора размера не более k само имеет размер не более k .

Свойство замены. Пусть A, B — независимые множества с $|A| < |B| \leq k$. Тогда множество $B \setminus A$ не пусто, и добавление любого его элемента не нарушает ограничение размера.

Пример Униформный матроид

В матроиде $U_{2,4}$ независимы все подмножества размера не более двух. Базы — двухэлементные подмножества. Любое трёхэлементное подмножество зависимо. Задача выбора на $U_{2,4}$ — выбор двух самых тяжёлых элементов.

Графический, линейный и униформный матроиды — основные примеры. Существуют и другие матроиды: трансверсальные, матроид расписаний. Один из них — матроид расписаний — рассматривается в разделе о приложениях.

17.5.1 * Матроид Фано

Возьмём в качестве элементов семь ненулевых векторов пространства $GF(2)^3$ и объявим независимыми линейно независимые наборы. Перед нами линейный матроид ранга 3. Занумеруем векторы числами от 1 до 7: $1 = (1, 0, 0)$, $2 = (0, 1, 0)$, $3 = (1, 1, 0)$, $4 = (0, 0, 1)$, $5 = (1, 0, 1)$, $6 = (0, 1, 1)$, $7 = (1, 1, 1)$. Зависимые тройки — векторы с нулевой суммой: 123, 145, 167, 246, 257, 347, 356. Такие тройки называют «прямыми», а всю конструкцию — плоскостью Фано: каждая пара точек лежит ровно на одной прямой. Независимыми оказываются множества из не более двух точек и те тройки, которые не являются прямыми, а любые четыре точки зависимы.

Над полями другой характеристики представления нет. Представление переводит точки 1, 2, 4, не лежащие на одной прямой, в базисные векторы e_1, e_2, e_3 , и каждый вектор-точку можно умножить на ненулевую константу. Точка 3 лежит на прямой с точками 1 и 2, поэтому её вектор пропорционален $e_1 + e_2$, а векторы точек 5 и 6 по тем же причинам пропорциональны $e_1 + e_3$ и $e_2 + e_3$. Определитель векторов $e_1 + e_2, e_1 + e_3, e_2 + e_3$ равен -2 . В характеристике 2 он равен нулю, и тройка 356 зависима, как и положено прямой. При любой другой характеристике определитель отличен от нуля, и прямая оказалась бы независимой тройкой. Матроид Фано представим над полем тогда и только тогда, когда характеристика поля равна двум.

Фано линеен, но не графичен. Любые один и два элемента Фано независимы, поэтому его графическое представление было бы простым графом. Ранг графического матроида простого графа равен числу вершин минус число компонент. Связный граф ранга 3 имеет четыре вершины и не более $\binom{4}{2} = 6$ рёбер, а несвязный граф того же ранга без изолированных вершин несёт ещё меньше рёбер. У Фано элементов семь, так что графического представления не существует. Каждый графический матроид, напротив, линеен: над $GF(2)$ его представляют столбцы матрицы инцидентности, у которой столбец каждого ребра содержит единицы в его концах, и набор столбцов зависим ровно тогда, когда рёбра содержат цикл. Тем самым графические матроиды образуют строгую часть линейных.

§ 17.6 Корректность жадного алгоритма

Теперь у нас есть все ингредиенты: задача выбора, жадный алгоритм и понятие матроида. Сформулируем результат, который их связывает.

Примечание Инвариант жадного алгоритма

Жадный алгоритм поддерживает инвариант: построенное множество остаётся независимым. Анализ корректности основан на сравнении результата алгоритма с оптимальным множеством по позициям. Поддержание инварианта и пошаговое сравнение с оптимумом — приём, применяемый и при верификации программ.

ТЕОРЕМА 17.4 Теорема Радо–Эдмондса

Пусть $(E, \mathcal{I})$ — матроид. Для любой неотрицательной весовой функции w жадный алгоритм находит независимое множество максимального веса.

Доказательство

Сравним жадную базу с оптимальной по позициям и докажем от противного, что жадный вес не меньше.

Все базы матроида имеют одинаковый размер. Если две базы имеют разные размеры, свойство замены увеличивает меньшую из них, что противоречит её максимальной. Обозначим общий размер баз через r . Жадный алгоритм возвращает базу, ведь если к множеству G можно добавить элемент без потери независимости, алгоритм добавил бы его. Поэтому достаточно сравнить результат жадного алгоритма с базой максимального веса.

Упорядочим элементы жадной базы G и оптимальной базы B по убыванию веса: $G = \{g_1, \dots, g_r\}$, $B = \{b_1, \dots, b_r\}$. Докажем, что на каждом шаге выполнено $w(g_i) \geq w(b_i)$.

Предположим, что на некотором шаге выполнено $w(g_i) < w(b_i)$. Множества $G_{i-1} = \{g_1, \dots, g_{i-1}\}$ и $B_i = \{b_1, \dots, b_i\}$ независимы, причём $|B_i| = i > i-1 = |G_{i-1}|$. По свойству замены существует элемент $x \in B_i \setminus G_{i-1}$, для которого множество $G_{i-1} \cup \{x\}$ независимо. Элементы B_i упорядочены по убыванию веса, поэтому $w(x) \geq w(b_i)$, а по предположению $w(b_i) > w(g_i)$, откуда $w(x) > w(g_i)$.

Значит, жадный алгоритм рассмотрел элемент x раньше g_i , и к этому моменту его множество — подмножество G_{i-1} . По наследственности это множество вместе с x независимо, поэтому алгоритм добавил бы x — противоречие.

Итак, $w(G) \geq w(B)$, то есть жадная база не легче базы максимального веса B . Любое независимое множество можно пополнить до базы, не уменьшив вес, поскольку веса неотрицательны, а свойство замены даёт добавляемый элемент, пока множество не станет максимальным. Следовательно, множество G имеет максимальный вес среди всех независимых множеств.

Теорема Радо–Эдмондса относится к целому классу задач. Если допустимые множества задачи образуют матроид, оптимальность жадного алгоритма не требует отдельного доказательства. Обратное утверждение — что вне класса матроидов жадный алгоритм может ошибаться — рассматривается в следующем разделе.

Проверка Корректность жадного алгоритма

1. Почему результат жадного алгоритма на матроиде — всегда база?
2. В каких двух местах доказательства теоремы Радо–Эдмондса работает свойство замены?

3. Почему элемент x из оптимальной базы жадный алгоритм обязан был рассмотреть раньше элемента g_i ?

§ 17.7 Обратное утверждение

Свойство замены — не формальность: именно оно отделяет задачи, где жадный алгоритм оптимален, от задач, где он может ошибаться. Семейство, в котором выполнена только наследственность, называется *системой независимости*.

ОПРЕДЕЛЕНИЕ 17.11 Система независимости

Пусть E — конечное множество. Пусть $\mathcal{I}$ — семейство его подмножеств. Пара $(E, \mathcal{I})$ называется **системой независимости**, если выполнена наследственность, то есть $A \in \mathcal{I} \wedge B \subseteq A \implies B \in \mathcal{I}$.

Матроид — это система независимости, в которой выполнено свойство замены. Покажем, что без свойства замены жадный алгоритм может дать неоптимальный результат.

Пример Контрпример: жадный алгоритм неоптимален

Возьмём семейство из примера о наследственности без свойства замены и обозначим его элементы a, b, c , так что $\mathcal{I} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{b, c\}\}$. Нарушающая пара прежняя: $A = \{a\}$ и $B = \{b, c\}$.

Зададим веса: $w(a) = 5, w(b) = 4, w(c) = 4$.

Жадный алгоритм выбирает a . Затем рассматривает b , но $\{a, b\} \notin \mathcal{I}$. Затем рассматривает c , но $\{a, c\} \notin \mathcal{I}$.

Результат — множество $\{a\}$ веса 5. Оптимальное независимое множество — $\{b, c\}$ веса 8. Жадный алгоритм неоптимален.

Схема контрпримера приведена на рисунке 82. Элемент a несовместим с остальными: жадный алгоритм выбирает его и получает вес 5. Оптимальное множество — набор $\{b, c\}$ весом 8.

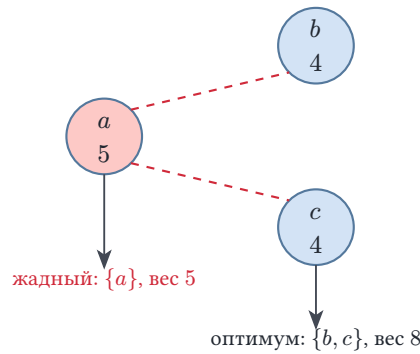


Рис. 82. Контрпример к жадному алгоритму.

Контрпример показывает необходимость свойства замены. Феномен общий: если система независимости не является матроидом, существует весовая функция, на которой жадный алгоритм неоптимален.

Набросок доказательства

Докажем построением весовой функции по нарушающей паре. Пусть множества $X, Y \in \mathcal{I}$ нарушают свойство замены, причём $|X| > |Y|$: ни один элемент из $X \setminus Y$ нельзя добавить к меньшему множеству Y .

Назначим каждому элементу Y вес $|Y| + 2$, каждому элементу $X \setminus Y$ вес $|Y| + 1$, а остальным элементам нулевой вес. Элементы Y тяжелее всех прочих ненулевых, и любой начальный отрезок Y независим по наследственности, поэтому жадный алгоритм заберёт Y целиком. Элементы $X \setminus Y$ к Y не добавляются по нарушенной замене, а элементы нулевого веса вес результата не меняют, так что вес жадного результата равен $|Y|(|Y| + 2)$.

Остаётся сравнить этот вес с весом X . Каждый элемент X весит не меньше $|Y| + 1$, а элементов в X не меньше $|Y| + 1$, поэтому $w(X) \geq (|Y| + 1)^2 = |Y|(|Y| + 2) + 1$. Жадный результат легче оптимального множества X .

§ 17.8 Матроиды в приложениях

Теорема Радо–Эдмондса сводит доказательство оптимальности жадного алгоритма к проверке двух аксиом матроида. Покажем это на известных задачах. Кажется, что перед нами разные алгоритмы: Крускал строит остов, жадный базис перебирает векторы, расписание сортирует работы по выигрышу. На деле это один и тот же жадный алгоритм, запущенный на разных матроидах.

Алгоритм Крускала из главы 9 — жадный алгоритм на графическом матроиде. В главе о графах он строит минимальное остовное дерево. В терминах матроидов он находит максимальный лес. Различие между этими задачами сводится к знаку весов. Раз леса образуют матроид (раздел о примерах матроидов), оптимальность алгоритма Крускала следует из теоремы Радо–Эдмондса.

Пример Крускал как жадный алгоритм

Возьмём тот же треугольник с весами 5, 4, 3, что и в примере о максимальном лесе. Жадный алгоритм берёт рёбра весов 5 и 4 и отбрасывает ребро веса 3: вместе с уже взятыми оно замыкает цикл. Результат — остов весом 9. Оптимальность следует из теоремы Радо–Эдмондса без отдельного рассуждения.

Вторая задача — расписание с дедлайнами. Она интересна тем, что матроид в ней скрыт: задача выглядит как задача об упорядочивании, хотя допустимые решения — выбор множества. Дано n задач. Задача i характеризуется дедлайном и выигрышем: $d_i \in \{1, \dots, n\}$, $p_i \geq 0$. Каждая задача выполняется за единицу времени.

ОПРЕДЕЛЕНИЕ 17.12 Реализуемый набор (feasible set)

Набор задач называется **реализуемым**, если существует порядок выполнения, при котором каждая задача завершается к своему дедлайну.

Требуется найти реализуемый набор максимального суммарного выигрыша. Реализуемость допускает простую характеристику: к моменту t число задач с дедлайном не позже этого момента не должно превышать номер момента.

ЛЕММА 17.5 Критерий реализуемости

Набор X задач реализуем тогда и только тогда, когда для каждого момента выполнено условие:

$$|\{i \in X : d_i \leq t\}| \leq t$$

Доказательство

Докажем критерий в обе стороны.

Прямо. Если набор X реализуем, то в любом допустимом расписании к каждому моменту выполняется не больше задач, чем номер момента. Задачи из X с дедлайном не позже некоторого момента должны быть выполнены к этому моменту. Следовательно, число таких задач не превосходит t .

Обратно. Пусть для каждого момента число задач из X с дедлайном не позже этого момента не превосходит номер момента. Упорядочим задачи из X по возрастанию дедлайнов и выполним их в этом порядке. Задача с дедлайном d встаёт на позицию, не превосходящую числа задач с не более поздним дедлайном. По предположению это число не превосходит d . Значит, задача выполняется не позднее d . Набор X реализуем.

УТВЕРЖДЕНИЕ 17.6

Реализуемые наборы задач образуют матроид.

Доказательство того, что реализуемые наборы удовлетворяют свойству замены, вынесено в упражнение. Этот матроид называют матроидом расписаний.

Пример Расписание с дедлайнами

Даны три задачи. У задачи A дедлайн 1 и выигрыш 50. У задачи B дедлайн 1 и выигрыш 40. У задачи C дедлайн 2 и выигрыш 30.

Реализуемы наборы $\{A, C\}$, $\{B, C\}$. Набор $\{A, B\}$ не реализуем: задачи с дедлайном 1 нельзя выполнить обе.

Жадный алгоритм по выигрышу выбирает A, C . Задачу B отбрасывает. Результат — набор $\{A, C\}$ с выигрышем 80, оптимальный.

Алгоритм Хаффмана из главы 13 — тоже жадный. Однако матроидом он не покрывается: объединение двух символов изменяет само множество элементов, так что семейство допустимых множеств не задано на фиксированном множестве. Поэтому оптимальность Хаффмана доказывается отдельно.

Рассмотрим, наконец, задачу, в которой матроид не возникает, — интервальное расписание.

Пример Интервальное расписание — не матроид

Пусть заданы интервалы времени. Допустимы попарно непересекающиеся наборы интервалов. Допустимые множества образуют систему независимости, но не матроид.

Контрпример к свойству замены — три интервала: $A = [1, 10]$, $B = [1, 2]$, $C = [3, 4]$. Возьмём $X = \{A\}$, $Y = \{B, C\}$. Имеем $|X| = 1 < 2 = |Y|$, и оба множества допустимы.

1. Ни один из интервалов B, C нельзя добавить к меньшему множеству. Каждый из них пересекает A .

По свойству замены хотя бы один элемент из $Y \setminus X$ должен был добавиться к меньшему множеству — и не добавляется.

Свойство замены нарушено. Однако в невзвешенном варианте задачи — когда каждый интервал ценен одинаково — жадный алгоритм по времени окончания даёт оптимальное решение. Его корректность доказывается отдельно (глава 9).

Конкретный набор интервалов приведён на рисунке 83. Длинный $A = [1, 10]$ пересекает $B = [1, 2]$ и $C = [3, 4]$, тогда как их пара допустима.

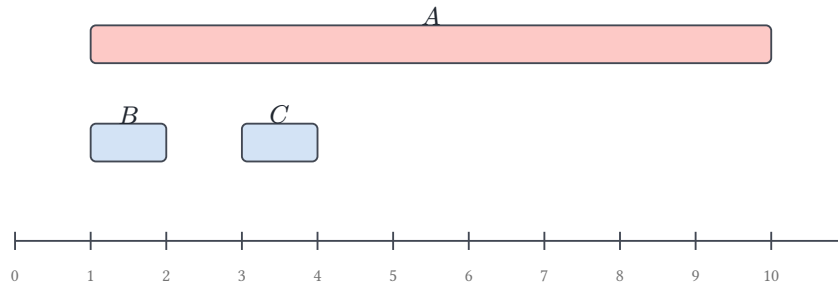


Рис. 83. Интервалы из примера.

Не во всех задачах выбора допустимые множества образуют матроид. Если допустимые множества образуют матроид, корректность жадного алгоритма следует из теоремы Радо–Эдмондса. В противном случае жадный алгоритм либо неоптимален, либо требует отдельного доказательства корректности.

§ 17.9 * Дальше в матроиды

Матроиды — самостоятельная область комбинаторики, и в этой главе затронута лишь её малая часть. Перечислим направления, в которые теория развивается дальше. Каждое заслуживает отдельной главы.

Функция ранга. Ранг $r(A)$ — размер его наибольшего независимого подмножества. Ранг матроида — частный случай этой функции: на всём множестве элементов $r(E)$ равен размеру базы. Ранговая функция субмодулярна: $r(A \cup B) + r(A \cap B) \leq r(A) + r(B)$. Минимизация субмодулярных функций полиномиальна, а максимизация NP-трудна. Многие свойства матроида удобнее формулировать через ранг, чем через независимые множества.

Циклы. Циклом (*circuit*) называется минимальное зависимое множество. В графическом матроиде циклы — циклы графа. В униформном $U_{k,n}$ циклы — подмножества размера $k + 1$. Как и независимые множества, циклы задают матроид: у них свои аксиомы.

Двойственный матроид. Базами двойственного матроида M^* служат дополнения его баз, а ранг M^* равен $|E| - r$, где r — ранг исходного матроида. Для графического матроида получается кографический: независимы множества рёбер, дополнение которых содержит остовное дерево. Базы двойственного матроида называются ко-базами. В связном графе ко-базы — ко-остова, то есть дополнения остовных деревьев до множества всех рёбер.

Пример Базы ко-остова

В треугольнике с рёбрами a, b, c базы графического матроида — пары $\{a, b\}, \{a, c\}, \{b, c\}$. Дополнения этих пар — одиночные рёбра $\{c\}, \{b\}, \{a\}$, и они же — базы двойственного матроида. Ранг исходного матроида равен 2, ранг

двойственного равен $3 - 2 = 1$, и двойственный матроид здесь — униформный $U_{1,3}$.

Каждое одиночное ребро треугольника — ко-остов: остовное дерево состоит из двух рёбер, и оставшееся ребро дополняет его до полного набора.

Трансверсальные матроиды. В двудольном графе независимы подмножества одной доли, которые покрываются паросочетанием. Обобщение теоремы Холла (глава 9) — теорема Радо о независимой системе различных представителей.

Пересечение матроидов. Семейство множеств, независимых сразу в двух матроидах, допускает полиномиальный алгоритм. Пересечение трёх матроидов — NP-трудная задача.

Матроиды и коды. Столбцы проверочной матрицы линейного кода образуют линейный матроид. Двойственность матроидов соответствует переходу к дуальному коду (глава 13).

§ 17.10 Итоги главы

Три задачи из начала главы — остовные деревья, линейно независимые наборы и расписания с дедлайнами — решались одним и тем же жадным алгоритмом, и совпадение оказалось не случайностью. Общая причина — в устройстве допустимых семейств: наследственность и свойство замены выделяют ровно те классы задач, где жадный алгоритм оптимален. Система независимости с этими двумя свойствами называется матроидом, и графические, линейные и униформные матроиды — его естественные примеры.

Теорема Радо–Эдмондса превращает наблюдение в критерий: жадный алгоритм оптимален на матроидах и может ошибаться на системах независимости, которые матроидами не являются. Система независимости с нарушенным свойством замены — контрпример: без свойства замены жадная стратегия уходит в неоптимальное решение. Интервальное расписание напоминает, что граница несимметрична: жадный алгоритм там тоже точен, хотя матроидом задача не является.

Матроид расписаний показывает, что аксиоматика работает и там, где допустимые множества заданы дедлайнами, и связывает главу с алгоритмом Крускала из главы 9. Тот же аппарат протягивается к кодам: столбцы проверочной матрицы линейного кода образуют линейный матроид, а пересечение двух матроидов решается полиномиально, тогда как трёх — уже NP-трудно.

Проверить две аксиомы проще, чем доказывать оптимальность каждого жадного алгоритма заново. Дальше в книге структуры уступят место подсчёту: комбинаторика (глава 18) займётся вопросом о том, сколько существует способов.

Проверка Проверьте себя

1. Сформулируйте аксиомы матроида.
2. Приведите пример системы независимости, не являющейся матроидом.
3. Докажите, что леса графа образуют матроид.
4. Почему жадный алгоритм на матроиде оптимален?
5. Почему интервальное расписание не является матроидом, хотя жадный алгоритм там оптимален?
6. Сформулируйте критерий реализуемости набора задач с дедлайнами.

Что дальше?

Матроиды связаны с материалом первых глав: алгоритм Крускала (глава 9) и коды (глава 13). Следующая глава — комбинаторика (глава 18): если матроиды объясняют, почему жадный алгоритм работает, комбинаторика отвечает, сколько существует способов. Дальнейшая теория матроидов включает функцию ранга, двойственные матроиды и пересечение матроидов.

§ 17.11 Упражнения*Проверка аксиом.*

1. Пусть $E = \{1, 2, 3, 4\}$, независимы подмножества размера не более двух. Является ли это семейство матроидом? Если да, назовите его.
2. Пусть $E = \{1, 2, 3\}$, независимы $\{1, 2\}$, $\{3\}$ и их подмножества. Является ли это семейство матроидом? Проверьте оба свойства.

Жадный алгоритм.

3. В равномерном матроиде $U_{3,5}$ заданы веса 7, 2, 9, 4, 1. Что выберет жадный алгоритм и каков вес результата?
4. Придумайте граф с четырьмя вершинами и пятью рёбрами с весами. Найдите максимальное остовное дерево жадным алгоритмом.

Свойство замены.

5. Докажите, что все базы матроида имеют одинаковый размер.
6. Пусть семейство наследственно и все его базы имеют одинаковый размер. Следует ли из этого свойство замены?

Задачи посложнее.

7. Даны задачи с дедлайнами и выигрышами: $(1, 100)$, $(2, 90)$, $(2, 80)$, $(1, 70)$. Найдите оптимальный набор жадным алгоритмом.

8. Докажите, что реализуемые наборы задач с дедлайнами образуют матроид.
9. * Рассмотрим граф с вершинами $\{1, 2, 3, 4\}$. Его рёбра: $a = (1, 2)$, $b = (2, 3)$, $c = (1, 3)$, $d = (3, 4)$, где a, b, c образуют треугольник, а d — «хвост» при вершине 3. Перечислите все базы графического матроида этого графа и найдите его ранг. Чему равен ранг подмножества $\{a, b, c\}$?
10. Является ли матроидом семейство всех подмножеств размера ровно k при $k \geq 1$? Проверьте наследственность и объясните, что именно ломается.
11. В семействе независимых множеств $\{a\}, \{b\}, \{c\}, \{b, c\}$ заданы веса $w(a) = 3$, $w(b) = w(c) = 2$. Что выберет жадный алгоритм, каков оптимальный набор и почему алгоритм ошибается?

Ранг.

12. Докажите, что в матроиде для любого множества X все максимальные по включению независимые подмножества X имеют одинаковый размер.

ПРОЕКТ Построение контрпримеров для жадного алгоритма

Теорема Радо–Эдмондса утверждает: на матроиде жадный алгоритм точен при любых весах. Обратное утверждение сильнее: если система независимости не является матроидом, то существуют веса, на которых жадный алгоритм гарантированно ошибается. Соберите работающий инструмент, который по системе независимости, заданной оракулом, либо подтверждает, что семейство является матроидом, либо предъявляет пару множеств, нарушающих свойство замены, и веса, на которых жадный алгоритм неоптимален. Проверьте его на матроидах — инструмент должен подтверждать оптимальность жадного алгоритма — и на системах независимости, не являющихся матроидами, где он обязан предъявить конкретную ошибку.

① Оракул и жадный алгоритм

Задайте систему независимости оракулом независимости и соберите жадный алгоритм, который упорядочивает элементы по убыванию веса и добавляет каждый элемент, если независимость сохраняется.

② Нарушение замены и веса

Проверьте наследственность отдельно: без неё семейство не является даже системой независимости. Затем найдите нарушение свойства замены перебором пар независимых множеств A, B с $|A| < |B|$: нарушение — это пара, к меньшему множеству которой не добавляется ни один элемент из $B \setminus A$. По найденной паре постройте веса, на которых жадный алгоритм ошибается, прогоните его на этих весах и проверьте, что результат не оптимален.

③ Коллекция примеров и случайные семейства

Соберите коллекцию примеров: равномерный, графический, линейный и дедлайновый матроиды, где жадный алгоритм должен быть точен, и контрпримеры — семейство $\{a\}, \{b\}, \{c\}, \{b, c\}$ и интервальное расписание, где нарушение есть и жадный алгоритм ошибается на предъявленных весах.

Объясните, почему на одних нарушение есть, а на других нет. Затем прогоните инструмент по случайным наследственным семействам и измерьте, как часто встречается нарушение свойства замены и насколько жадный алгоритм может ошибаться.

④ **Функция ранга и субмодулярность**

Добавьте функцию ранга подмножества: $r(A)$ — размер наибольшего независимого подмножества A . Вычислите её перебором подмножеств A по убыванию размера, до первого независимого. Проверьте субмодулярность $r(A) + r(B) \geq r(A \cup B) + r(A \cap B)$ на всех парах подмножеств. Убедитесь, что на матроидах она выполнена, а на семействе, не являющемся матроидом, найдите конкретную пару, где она нарушается. Объясните, почему рангу матроида субмодулярность обязана выполняться, а у произвольной системы независимости — нет.

⑤ **Двойственность, циклы и трансверсальные матроиды**

Двойственный матроид M^* : множество A независимо в M^* , когда дополнение $E \setminus A$ содержит базис M . Реализуйте это условием на ранге: $r(E \setminus A) = r(E)$. Цикл — минимальное зависимое множество: перечислите все циклы матроида и проверьте, что удаление любого элемента цикла делает множество независимым. Трансверсальный матроид: элементы независимы, когда их можно рассадить по группам так, чтобы разные элементы попали в разные группы. Реализуйте проверку поиском рассадки и продумайте, когда рассадка невозможна: какому подмножеству элементов не хватает групп.

Итог: инструмент, который по системе независимости либо подтверждает, что она является матроидом — и тогда жадный алгоритм точен при любых весах — либо предъявляет конкретные веса, на которых жадный алгоритм ошибается. Это работающее воплощение обратного утверждения теоремы Радо–Эдмондса. В дополнение — конструкции матроидов: двойственный матроид, перечисление циклов и трансверсальный матроид.

XVIII

Комбинаторика

Сколько существует паролей из двух букв и четырёх цифр? Сколькими способами можно раздать колоду из пятидесяти двух карт? Сколько маршрутов соединяет два узла сети? На все эти вопросы комбинаторика отвечает, не открыв ни одного варианта.

Подсчёт кажется простейшей математической операцией: чтобы узнать количество объектов, достаточно их перебрать. Для малых чисел эта интуиция работает безотказно.

Интуиция отказывает, как только варианты перестают помещаться в голове. Количество перестановок колоды из 52 карт — примерно $8 \cdot 10^{67}$ — превышает число звёзд в наблюдаемой Вселенной на сорок пять порядков. Перебрать столько вариантов невозможно ни руками, ни машиной: уже для двух-трёх десятков объектов полный перебор требует времени, превосходящего возраст Вселенной. Это — *комбинаторный взрыв*.

Комбинаторика и есть искусство считать *без* перебора. В её основе лежат несколько элементарных принципов: правила суммы, произведения и дополнения, перестановки и сочетания, биномиальные коэффициенты и принцип Дирихле. Из них собираются замкнутые формулы, рекуррентные соотношения и целые семейства чисел — Каталана, Стирлинга, Белла.

Ремесло складывалось веками, и начало ему положили игроки. Джероламо Кардано в «Liber de ludo aleae» — книге об игре в кости — впервые систематически подсчитал шансы в азартной игре. Блез Паскаль построил треугольник, который до сих пор носит его имя. Готфрид Лейбниц в трактате «De arte combinatoria» назвал эту область искусством и видел в ней общий язык всех наук.

Сегодня это ремесло — рабочий инструмент инженерии. Оценка числа паролей определяет стойкость системы, число путей в графе — надёжность сети, число раскладов — шансы игрока. Без комбинаторных формул не обходится ни анализ алгоритмов, ни теория кодирования. Эти формулы станут фундаментом дискретной вероятности (глава 20) и производящих функций (глава 21).

Чтобы пространство вариантов стало недостижимым для перебора, бесконечность не нужна — достаточно факториала. Формула не перечисляет варианты — она видит их структуру. Как формула разглядывает структуру там, где перебор слепнет?

ИСТОРИЯ От азартных игр до анализа алгоритмов

В 1654 году Блез Паскаль и Пьер Ферма вступили в переписку о задачах азартных игр — в частности, о справедливом разделе ставки в прерванной игре. Эти письма заложили основания комбинаторики и теории вероятностей одновременно. Треугольник, получивший имя Паскаля, был известен задолго до него: Омар Хайям (1048–1131) описал биномиальные коэффициенты в трактате «Трудности арифметики». Китайский математик Ян Хуэй в 1261 году опубликовал изображение треугольника вплоть до шестой степени и указал, что метод восходит к Цзя Сяню (ок. 1050). Вклад Паскаля — «Трактат об арифметическом треугольнике» (1654, опубликован посмертно в 1665): первая систематическая теория конструкции, её тождества и приложения к вероятности.

Якоб Бернулли в «Ars Conjectandi» (1713, посмертно) систематизировал комбинаторные методы: перестановки, сочетания, размещения, числа Бернулли, а также независимо доказал биномиальную теорему для целых показателей. Леонард Эйлер в 1736 году применил комбинаторные методы к кёнигсбергским мостам, положив начало теории графов. Иоганн Петер Густав Лежён Дирихле в 1834 году сформулировал в работе по теории чисел принцип, названный им *Schubfachprinzip* — «принцип ящиков», в англоязычной традиции *pigeonhole principle*. Набор приёмов, рождённый задачами об игре в кости, превратился в систематическую дисциплину с приложениями от анализа алгоритмов до криптографии.

ОБЗОР ГЛАВЫ

Комбинаторика начинается с правил суммы, произведения и дополнения, а рядом с ними стоит принцип Дирихле — способ доказать существование, не предъявляя объекта. Перестановки, размещения и сочетания с повторениями и без ответов на главный вопрос всякого выбора: важен ли порядок? Биномиальные коэффициенты и их тождества свяжут комбинаторику с алгеброй, а формула включений-исключений исправит двойной счёт. Рекуррентные соотношения и метод характеристического уравнения извлекут явные формулы из определений, а числа Каталана, Стирлинга и Белла сведут целые семейства объектов к одной формуле. Двенадцатеричный путь соберёт перестановки, сочетания, разбиения и композиции в одну таблицу: ответ определяется тем, различимы ли объекты, различимы ли ящики и каково ограничение на заполнение. Лемма Бёрнсайда научит считать с точностью до симметрии, теория Рамсея покажет неизбежность порядка в достаточно большой системе, а игра Ним продемонстрирует, как те же приёмы находят выигрышную стратегию.

После этой главы вы сможете:

1. выбирать правило подсчёта — сумму, произведение, дополнение — и применять принцип Дирихле;
2. различать перестановки, размещения и сочетания и отвечать на вопрос, важен ли порядок;
3. доказывать тождества с биномиальными коэффициентами и исправлять двойной счёт формулой включений-исключений;
4. решать линейные рекуррентные соотношения методом характеристического уравнения;
5. считать с точностью до симметрии леммой Бёрнсайда и узнавать числа Каталана, Стирлинга и Белла;

§ 18.1 Правила подсчёта

18.1.1 Правило суммы и произведения

В основе всей комбинаторики лежат два простых на вид, но далеко идущих принципа: правило суммы и правило произведения. Вместе они позволяют разбивать сложные задачи подсчёта на простые независимые шаги.

ОПРЕДЕЛЕНИЕ 18.1 Правило суммы

Если конечные множества A, B не пересекаются, то количество элементов в их объединении равно сумме количеств:

$$A \cap B = \emptyset : |A \cup B| = |A| + |B|.$$

Это правило обобщается на любое конечное число попарно непересекающихся множеств.

Интуиция: если действия взаимоисключающие, общее количество вариантов получается сложением. В столовой три супа и четыре салата, выбрать нужно ровно одно блюдо: либо суп, либо салат. Вариантов $3 + 4 = 7$, и перечислять их незачем. Сложение безопасно ровно потому, что случаи не пересекаются: каждый обед посчитан один раз.

ОПРЕДЕЛЕНИЕ 18.2 Правило произведения

Если выбор состоит из последовательных независимых шагов, причём шаги можно сделать $k_1, k_2, \dots$ способами, то общее количество вариантов равно произведению

$$k_1 \cdot k_2 \cdot \dots \cdot k_m.$$

В терминах множеств:

$$|A_1 \times A_2 \times \dots \times A_m| = |A_1| \cdot |A_2| \cdot \dots \cdot |A_m|.$$

Пример Подсчёт паролей

Пароль состоит из 2 букв латинского алфавита (26 вариантов каждая) и 4 цифр (10 вариантов каждая) в фиксированных позициях. Выбор каждого символа независим, поэтому общее количество: $26^2 \cdot 10^4 = 6,760,000$.

Иногда проще посчитать то, чего *не* нужно, и вычесть его из общего числа. Это даёт третье правило — правило дополнения.

ОПРЕДЕЛЕНИЕ 18.3 Правило дополнения

Если U — универсальное множество (содержащее все рассматриваемые объекты), а $A \subseteq U$, то

$$|A| = |U| - |\bar{A}|.$$

Правило дополнения применяется, когда подсчитать «не- A » проще, чем само A .

Пример Правило дополнения

Сколько 8-битных строк содержат хотя бы одну единицу? Всего 8-битных строк: $2^8 = 256$. Строк без единиц (все нули): ровно одна. Ответ: $256 - 1 = 255$.

18.1.2 Шары и ящики

Многие комбинаторные задачи можно сформулировать как задачу о распределении шаров по ящикам. В зависимости от того, различимы ли шары и различимы ли ящики, ответ получается разным. Следующая таблица даёт все четыре варианта в одной картине:

УТВЕРЖДЕНИЕ 18.1 Четыре ситуации подсчёта шаров по ящикам

Шары	Ящики	Количество способов
Различные	Различные	k^n — каждый из n шаров выбирает один из k ящиков
Неразличимые	Различные	$\binom{n+k-1}{n}$ — метод звёзд и перегородок
Различные	Неразличимые	$S(n, 1) + \dots + S(n, k)$ — числа Стирлинга 2-го рода (раздел ниже)
Неразличимые	Неразличимые	$\sum_{i=1}^k p_i(n)$ — разбиения числа n на не более k частей (раздел ниже)

Отдельного приёма требует случай неразличимых шаров в различных ящиках. Здесь работает метод *звёзд и перегородок* (*stars and bars*).

Доказательство

Докажем построением биекции: каждая раскладка шаров однозначно задаётся выбором позиций перегородок.

Шары становятся n звёздочками, а перегородок нужно $k - 1$: они разделяют звёздочки на группы. Всего позиций $n + k - 1$, из которых $k - 1$ занимают перегородки, что даёт

$$\binom{n+k-1}{k-1} = \binom{n+k-1}{n}.$$

Пример Дюжина бубликов

В булочной четыре сорта бубликов: маковые, кунжутные, луковые и ванильные. Сколькими способами можно набрать дюжину — двенадцать бубликов — в один мешок? Бублики в мешке неразличимы, сорта различимы: это ровно звёзды и перегородки при $n = 12, k = 4$. Прежде чем считать по формуле, посмотрим на малые случаи.

1. Один сорт: дюжина однозначна — способ ровно один.
2. Два сорта: маковых может быть от нуля до двенадцати, остальное — кунжутные: тринадцать способов.
3. Три сорта: зафиксируем, сколько луковых, и выбор остатка снова сводится к двум сортам. Просуммировав по всем вариантам, получаем $13 + 12 + \dots + 1 = 91$.
4. Четыре сорта — формула звёзд и перегородок: $\binom{12+4-1}{4-1} = \binom{15}{3} = 455$.

Не путайте с сочетаниями с повторениями (раздел ниже): там из n типов выбирают k элементов. Формула — $\binom{n+k-1}{k}$: выбор k позиций вместо $k - 1$. Роли n, k меняются местами.

Пример Различные шары, неразличимые ящики

Трое различных студентов заселяются в две одинаковые комнаты ($n = 3, k = 2$), ни одна комната не пустует. Количество способов: $S(3, 2) = 3$. Действительно: либо первый живёт один (а двое других вместе), либо второй один, либо третий один — всего три варианта. Если разрешить пустые комнаты, добавляется ещё $S(3, 1) = 1$ (все трое в одной комнате).

Таблица выше — разрез по различимости шаров и ящиков: её клетки — числа Стирлинга и разбиения. Вопросы «важен ли порядок» и «разрешены ли повторения» дают другой разрез, который организует перестановки, размещения и сочетания. Его полная матрица — ниже в главе.

18.1.3 Принцип Дирихле

ОПРЕДЕЛЕНИЕ 18.4 Принцип Дирихле

Если n объектов размещены по m ящикам и $n > m$, то хотя бы в одном ящике окажется не менее двух объектов.

Обобщённая форма: если n объектов размещены по m ящикам, то найдётся ящик с не менее чем $\lceil \frac{n}{m} \rceil$ объектами. Найдётся и ящик с не более чем $\lfloor \frac{n}{m} \rfloor$ объектами.

На языке функций: если $|A| > |B|$, то не существует инъекции $f : A \rightarrow B$.

По смыслу это теорема существования: принцип утверждает, что объект есть, но не предъявляет его. В Москве более двенадцати миллионов жителей, а волос на голове у человека не более ста пятидесяти тысяч, поэтому двое с одинаковым числом волос найдутся, хотя назвать эту пару невозможно.

Классические примеры — дни рождения (среди 367 человек найдутся двое с общим днём рождения), носки в тёмной комнате (среди 5 носков трёх цветов найдутся два одного цвета) — иллюстрируют принцип, но не его силу. Сила принципа Дирихле — в задачах, где выбор ящиков далеко не тривиален. Рецепт решения таких задач — акт узнавания: спросите себя, кто здесь *голубь*, а что *ящик*.

Пример Замоещение доски домино

Шахматная доска потеряла два противоположных угла. Можно ли замостить её домино? Вырезанные углы одного цвета: светлых клеток осталось 30, тёмных — 32. Каждое домино накрывает ровно одну светлую и одну тёмную клетку. Чтобы закрыть 62 клетки, нужно 31 домино, а светлых клеток только 30: домино — голуби, светлые клетки — ящики. По принципу Дирихле два домино претендуют на одну светлую клетку — замощение невозможно.

Пример Знакомства в компании из шести человек

В любой компании из шести человек найдутся либо трое попарно знакомых, либо трое попарно незнакомых.

Это утверждение — частный случай теоремы Рамсея: рамсеевское число $R(3, 3) = 6$. Доказательство опирается на принцип Дирихле.

Выберем одного человека A . Остальные пятеро делятся на два «ящика»: знакомые A и незнакомые с A . По принципу Дирихле (5 человек, 2 ящика), в одном из ящиков не менее $\lceil \frac{5}{2} \rceil = 3$ человек.

Случай 1: среди знакомых A есть трое. Обозначим их B, C, D . Если среди B, C, D есть хотя бы одна пара знакомых, то эта пара вместе с A образует тройку попарно знакомых. Если же B, C, D попарно незнакомы, то они и есть искомая тройка.

Случай 2: среди незнакомых с A есть трое. Симметричное рассуждение (заменяя «знакомы» на «незнакомы») даёт тройку либо попарно знакомых, либо попарно незнакомых.

Мы не знаем, *какие именно* трое образуют искомую конфигурацию, но знаем, что они *существуют* — в этом и состоит неконструктивный характер принципа Дирихле. Пять — недостаточно: компания из пяти человек может быть устроена так, что ни тройки знакомых, ни тройки незнакомых нет — пятеро по кругу, и каждый знает только соседей.

Пример Сжатие без потерь

Пусть алгоритм сжимает каждый вход из n бит в выход из меньшего числа бит. Длина выхода — $m < n$. Входов и выходов соответственно $2^n, 2^m$. Поэтому $2^n > 2^m$.

Голуби — входы, ящики — выходы: какой-то выход обязан достаться двум разным входам. По такому выходу нельзя восстановить исходный вход. Значит, не существует алгоритма, который сжимает *каждый* вход без потерь.

Проверка Принцип Дирихле

1. Что играет роль голубей и что — ящиков в задаче о замощении доски без двух углов?
2. Сколько объектов гарантированно собирается в одном ящике по обобщённой форме при $n = 100$ и $m = 7$?
3. Почему пятерых человек недостаточно для гарантии тройки попарно знакомых или попарно незнакомых?

§ 18.2 Суммы и замкнутые формы

Комбинаторика постоянно имеет дело с суммами: сумма размеров, сумма вероятностей, сумма сложности. Многие суммы, возникающие в анализе алгоритмов, имеют простые замкнутые формы — выражения без знака суммы, которые можно вычислить за $O(1)$.

УТВЕРЖДЕНИЕ 18.2 Арифметическая прогрессия

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

Доказательство: запишем сумму дважды — в прямом и обратном порядке — и сложим:

$$(1 + n) + (2 + (n - 1)) + \dots + (n + 1) = n \cdot (n + 1),$$

откуда искомая сумма равна $\frac{n(n+1)}{2}$.

УТВЕРЖДЕНИЕ 18.3 Геометрическая прогрессия

Для $r \neq 1$:

$$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}.$$

Для $|r| < 1$ при $n \rightarrow \infty$ бесконечная геометрическая прогрессия сходится:

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1 - r}.$$

Первые две — школьные прогрессии, их замкнутые формы всем знакомы. Две следующие менее привычны: сумма квадратов выводится индукцией, а сумма биномиальных коэффициентов — подстановкой в биномиальную теорему.

УТВЕРЖДЕНИЕ 18.4 Сумма квадратов

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}.$$

Доказывается индукцией по n .

За формулой стоит картинка: единичные кубики, сложенные углом, — слои содержат $n^2, (n-1)^2, \dots$ кубиков, и так до одного. Сосчитать кубики по слоям — и формула получается сама, индукция лишь оформляет подсчёт.

Знание этих формул — часть инструментария каждого разработчика алгоритмов.

- Арифметическая прогрессия: суммарное время двух вложенных циклов `for i in 1..n: for j in i..n` — $O(n^2)$.
- Геометрическая прогрессия: время цикла `while n > 1: n = n // 2` — $O(\log n)$.
- Сумма биномиальных коэффициентов: число всех подмножеств n -элементного множества.

§ 18.3 Перестановки, размещения, сочетания

УТВЕРЖДЕНИЕ 18.5 Матрица комбинаторного выбора

Перестановки, размещения и сочетания организованы вокруг двух независимых параметров выбора: важен ли порядок, и разрешены ли повторения. Четыре комбинации ответов — четыре клетки матрицы 2×2 :

	Повторения запреще- ны	Повторения разреше- ны
Порядок важен	размещения $P(n, k) = \frac{n!}{(n-k)!}$	размещения с повторе- ниями n^k
Порядок не важен	сочетания $\binom{n}{k}$	сочетания с повторения- ми $\binom{n+k-1}{k}$

Перестановки — частный случай размещений при $k = n$. С этой матрицей формулы перестают быть списком для запоминания: каждая формула — подстановка ответов на два вопроса в соответствующую клетку.

18.3.1 Перестановки

ОПРЕДЕЛЕНИЕ 18.5 Перестановка

Упорядочение n различных объектов называется **перестановкой**. Количество перестановок n элементов обозначается

$$n! = 1 \cdot 2 \cdot \dots \cdot n$$

(читается « n факториал»). По соглашению, $0! = 1$.

Факториал растёт очень быстро: $5! = 120$, $10! = 3,628,800$, $20! \approx 2.43 \cdot 10^{18}$. Для приближённой оценки больших факториалов служит формула Стирлинга.

УТВЕРЖДЕНИЕ 18.6 Формула Стирлинга¹⁴²

$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$, $n \rightarrow \infty$. Относительная погрешность стремится к нулю с ростом n .

Набросок доказательства

Вывод формулы Стирлинга опирается на интегральное представление факториала через гамма-функцию и метод Лапласа для оценки интегралов. Для комбинаторных приложений достаточно знать саму формулу — она позволяет оценивать $n!$ с точностью до множителя, близкого к единице, и незаменима при анализе сложности алгоритмов.

Формула проверяется на пальцах: для $n = 20$ погрешность — в полпроцента. Приближение и точное значение: $2.42 \cdot 10^{18}$, $2.43 \cdot 10^{18}$. Для прикидки сложности алгоритма такой точности хватает с запасом.

¹⁴²J. Stirling. «Methodus Differentialis» (Дифференциальный метод). 1730. Первым асимптотику $\log n!$ получил Абрахам де Муавр (Abraham de Moivre, 1730). Стирлинг уточнил константу до $\sqrt{2\pi}$.

Дерево решений позволяет увидеть правило произведения в действии, как показано на рис. 84. Три варианта для первой позиции, два — для второй, один — для третьей. $3 \cdot 2 \cdot 1 = 6$ путей от корня до листьев, и каждый путь даёт ровно одну перестановку.

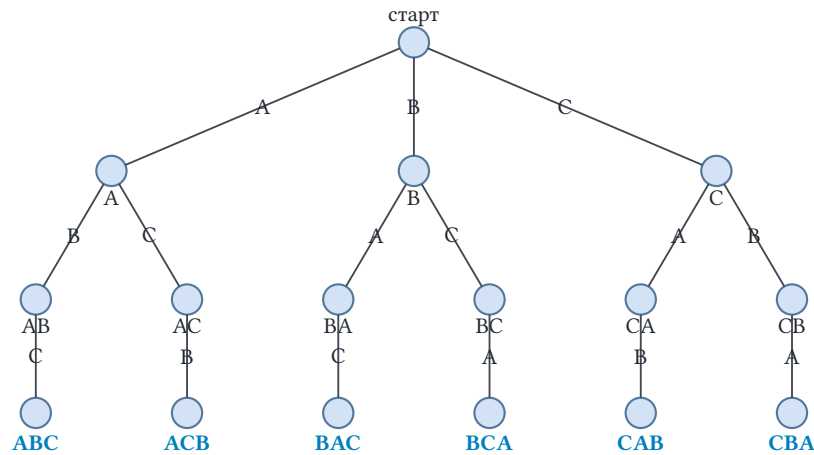


Рис. 84. Дерево решений для перестановок множества $\{A, B, C\}$.

18.3.2 Генерация перестановок

Перечислить все $n!$ перестановок можно в любом порядке, но лексикографический удобен тем, что даёт каждой перестановке естественный номер. Классический алгоритм `next_permutation` выдаёт следующую перестановку за три шага, не перебирая всё заново.

Идея опирается на одно наблюдение. *Убывающий* суффикс перестановки — это последняя расстановка своих элементов: переставить их иначе, чтобы последовательность только выросла, нельзя. Значит, чтобы перейти к следующей перестановке, придётся заменить элемент *перед* самым длинным убывающим суффиксом — пивот.

АЛГОРИТМ Следующая перестановка

1. Найдите самый длинный убывающий суффикс: наибольший индекс i , для которого $p[i..]$ строго убывает.
2. Если $i = 0$, вся перестановка убывает — это последняя, и следующей нет.
3. Иначе пивот — элемент $p[i - 1]$.
4. В убывающем суффиксе найдите наименьший элемент, больший пивота (правый крайний такой).
5. Поменяйте пивот и найденный элемент местами.
6. Переверните суффикс $p[i..]$: убывающий станет возрастающим — наименьшей расстановкой тех же элементов.

УТВЕРЖДЕНИЕ 18.7 Корректность `next_permutation`

Если p не является последней перестановкой, алгоритм выдаёт лексикографически следующую за p .

Набросок доказательства

Пусть $p[i..]$ — самый длинный убывающий суффикс, а $p[i - 1]$ — пивот. Всякая перестановка, большая p , совпадает с p на некотором префиксе и превосходит её в первой точке расхождения. Внутри $p[i..]$ эта точка лежать не может: убывающий суффикс уже максимален. Значит, точка расхождения — на позиции пивота или левее, а наименьший возможный рост даёт замена пивота на наименьший больший элемент. Остальные элементы упорядочиваем минимально — возрастающим суффиксом, то есть переворотом. Получилась наименьшая перестановка среди всех, больших p .

Пример Трассировка: от 2, 4, 1, 3 к 2, 4, 3, 1

1. Убывающий суффикс — один элемент [3], пивот — 1.
2. Наименьший элемент суффикса, больший 1, — это 3.
3. Меняем пивот и 3 местами: 2, 4, 3, 1. Суффикс из одного элемента переворачивать нечего.

Пример Следующая перестановка на Rust

Шаги переносятся в код без потерь — снippet лишь дополняет алгоритм.

```
/// Следующая перестановка в лексикографическом порядке.
/// Возвращает false, если это была последняя (убывающая) перестановка.
fn next_permutation(p: &mut [usize]) -> bool {
    let mut i = p.len() - 1;
    while i > 0 && p[i - 1] >= p[i] {
        i -= 1; // самый длинный убывающий суффикс p[i..]
    }
    if i == 0 {
        return false; // вся последовательность убывает --- это
последняя
    }
    let mut j = p.len() - 1;
    while p[j] <= p[i - 1] {
        j -= 1; // наименьший элемент суффикса, больший пивота
    }
    p.swap(i - 1, j);
    p[i..].reverse(); // суффикс становится возрастающим
    true
}
```

Замечание Другие порядки генерации

Лексикографический порядок — лишь один из многих. Дональд Кнут в четвёртом томе «Искусства программирования» разбирает целый зоопарк алгоритмов генерации перестановок и сочетаний¹⁴³: изменения без повторов (Steinhaus–

¹⁴³D. E. Knuth. The Art of Computer Programming, Vol. 4A (генерация перестановок и сочетаний).

Johnson–Trotter), где соседние перестановки отличаются одной транспозицией, генерацию по цикловой структуре, генерацию сочетаний и другие. Каждый порядок выгоден для своего приложения. Важно, что «перечислить» — это алгоритмическая задача со своими компромиссами.

18.3.3 Размещения и сочетания

При выборе k элементов из n всё решает один вопрос: важен ли порядок выбранных элементов?

ОПРЕДЕЛЕНИЕ 18.6 Размещение (k -permutation)

Выбор k элементов из n различных объектов с учётом порядка называется **размещением**. Количество размещений:

$$P(n, k) = n \cdot (n - 1) \cdot \dots \cdot (n - k + 1) = \frac{n!}{(n - k)!}.$$

За формулой стоит правило произведения: на первое место претендуют n объектов, на второе — $n - 1$, потому что один уже занят, а на последнее — $n - k + 1$.

ОПРЕДЕЛЕНИЕ 18.7 Сочетание

Выбор k элементов из n различных объектов, в котором порядок *не* важен, называется **сочетанием**. Количество сочетаний обозначается

$$\binom{n}{k} = \frac{n!}{k!(n - k)!} = \frac{P(n, k)}{k!}$$

(читается «це из n по k »).

Формула делит размещения на $k!$ не случайно: выбранные k элементов можно упорядочить $k!$ способами, и все эти упорядочения дают одно и то же сочетание.

ЛОВУШКА Размещения и сочетания: порядок решает

Вся разница между размещениями и сочетаниями — в порядке. Если порядок важен, число вариантов — $P(n, k) = \frac{n!}{(n - k)!}$. При $k = n$ это $n!$ перестановок. Если не важен — $\binom{n}{k} = \frac{P(n, k)}{k!}$.

Посчитаем рукопожатия в компании из n человек. Наивный подсчёт даёт n^2 : первого участника пары выбираем n способами, второго — тоже n . Две поправки приводят к формуле $\binom{n}{2}$:

1. Рукопожатие $A-B$ совпадает с $B-A$: каждая настоящая пара посчитана дважды, делим пополам.
2. Рукопожатия с собой не существует: убираем n вырожденных пар.

Получаем $\frac{n^2-n}{2} = \frac{n(n-1)}{2} = \binom{n}{2}$.

Проверка одна: если два выбора различаются только перестановкой — это одно сочетание.

Соглашение $0! = 1$ отражает комбинаторный факт: существует ровно одна перестановка пустого множества. Если бы мы определили факториал только для положительных целых, рекуррента $n! = n \cdot (n-1)!$ сломалась бы. При $n = 1$ не на что опереться. А биномиальная теорема потребовала бы исключения для $k = 0$. То же относится к $\binom{n}{0} = 1$: каждое соглашение кодирует единственность пустой конструкции.

Пример Комитет и пьедестал

Выбрать комитет из 3 человек из 10 кандидатов (порядок не важен): $\binom{10}{3} = 120$.

Выбрать призёров — золото, серебро, бронза — из 10 участников (порядок важен): $P(10, 3) = 10 \cdot 9 \cdot 8 = 720$.

ЛОВУШКА $\binom{n}{k}$ и n^k — разные выборы

n^k считает размещения с повторениями: каждый элемент выбирается независимо, и порядок важен. $\binom{n}{k}$ считает неупорядоченный выбор без повторений — подмножество. Число подмножеств размера 2 из 4 равно $\binom{4}{2} = 6$. Вариант $4^2 = 16$ исключён: выбор $\{1, 2\}$ и $\{2, 1\}$ — одно и то же подмножество.

Примечание Перебор сочетаний на Rust

Формулы дают число сочетаний, но иногда нужны сами подмножества — например, чтобы перебрать все варианты на малых n . Рекурсивный перебор выбирает элементы по возрастанию и порождает каждое k -подмножество ровно один раз:

```
/// Все сочетания по k из [0..n): рекурсивный перебор.
fn combinations(n: usize, k: usize) -> Vec<Vec<usize>> {
    fn go(
        start: usize,
        k: usize,
        cur: &mut Vec<usize>,
        out: &mut Vec<Vec<usize>>,
        n: usize,
    ) {
        if k == 0 { out.push(cur.clone()); return; }
        for x in start..n - k {
            cur.push(x);
            go(x + 1, k - 1, cur, out, n);
            cur.pop();
        }
    }
}
```

```

let mut out = vec![];
go(0, k, &mut vec![], &mut out, n);
out
}

```

18.3.4 Сочетания и перестановки с повторениями

До сих пор мы предполагали, что элементы различны и каждый выбирается не более одного раза. С разрешением повторений картина меняется.

ОПРЕДЕЛЕНИЕ 18.8 Сочетания с повторениями

Выбор k элементов из n типов, в котором повторения разрешены, а порядок не важен, называется **сочетанием с повторениями**. Количество таких сочетаний:

$$\binom{n+k-1}{k}$$

Формула — те же звёзды и перегородки: сочетание записывается как k звёздочек, разделённых $n-1$ перегородкой на n групп по типам, а выбираются позиции звёздочек среди $n+k-1$ мест.

Тот же ответ можно прочесть как *коэффициент степенного ряда*. Число сочетаний с повторениями — это число решений уравнения $x_1 + x_2 + \dots + x_n = k$ в неотрицательных целых, где x_i — сколько раз взят тип i . Каждому типу соответствует ряд $1 + x + x^2 + \dots$: слагаемое x^{x_i} отмечает, что тип i выбран x_i раз. Произведение n таких рядов равно $\frac{1}{(1-x)^n}$, а слагаемое с x^k появляется ровно тогда, когда $x_1 + \dots + x_n = k$, — то есть ровно $\binom{n+k-1}{k}$ раз. Формула доказывается и так: коэффициент при x^k в $\frac{1}{(1-x)^n}$ равен $\binom{n+k-1}{k}$. Это первый набросок метода производящих функций (глава 21): ответ на комбинаторный вопрос прячется в коэффициенте ряда, а не в прямом перечислении.

Сочетания с повторениями игнорируют порядок выбранных элементов. Когда порядок важен, а повторения заданы явно — фиксированное число неразличимых копий каждого типа — мы приходим к другому комбинаторному объекту.

ОПРЕДЕЛЕНИЕ 18.9 Перестановки с повторениями

Упорядочение n объектов с неразличимыми экземплярами называется **перестановкой с повторениями**. Число экземпляров каждого типа обозначается n_i . Типы нумеруются $i = 1, \dots, k$.

Количество таких перестановок — мультиномиальный коэффициент:

$$\frac{n!}{n_1! n_2! \dots n_k!}$$

Деление возникает из учёта неразличимости: перестановок различных объектов было бы $n!$, но перестановки неразличимых экземпляров одного типа между собой не дают новых упорядочений, поэтому каждый тип делит ответ на $n_i!$.

Пример Анаграммы слова БАЛАЛАЙКА

Слово «БАЛАЛАЙКА» состоит из 9 букв: из них А встречается 4 раза, Л — 2. Количество анаграм (различных перестановок букв):

$$\frac{9!}{4! \cdot 2!} = 7,560.$$

Замечание Внезапное совпадение

Балалайка-прима настроена ми-ми-ля. Сколько мелодий из девяти нот можно сыграть, если ля прозвучит четыре раза, ми — два раза, а ре, соль и си — по одному разу?

$$\frac{9!}{4! \cdot 2!} = 7,560.$$

Та же формула, что у анаграмм слова БАЛАЛАЙКА: другой вопрос — тот же подсчёт.

Проверка Перестановки, размещения и сочетания

1. Почему число сочетаний с повторениями из n типов по k элементов равно $\binom{n+k-1}{k}$, и откуда берутся $n - 1$ перегородок?
2. Почему число анаграмм слова БАЛАЛАЙКА равно $\frac{9!}{4! \cdot 2!}$, а полное число перестановок $9!$ здесь не подходит?

§ 18.4 Биномиальные коэффициенты и тождества

Биномиальные коэффициенты $\binom{n}{k}$ скрывают богатую алгебраическую структуру и подчиняются множеству тождеств.

УТВЕРЖДЕНИЕ 18.8 Тождество Паскаля

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

для $1 \leq k \leq n - 1$.

Доказательство

Докажем разбором случаев: включается ли первый элемент.

Чтобы выбрать k элементов из n , либо включаем первый элемент, либо нет. В первом случае остаётся $\binom{n-1}{k-1}$ способов. Во втором — $\binom{n-1}{k}$ способов.

Тождество Паскаля порождает треугольник Паскаля — простейшую комбинаторную структуру, показанную на рис. 85. Каждая строка зеркальна: $\binom{n}{k} = \binom{n}{n-k}$, а каждый внутренний элемент равен сумме двух над ним, $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

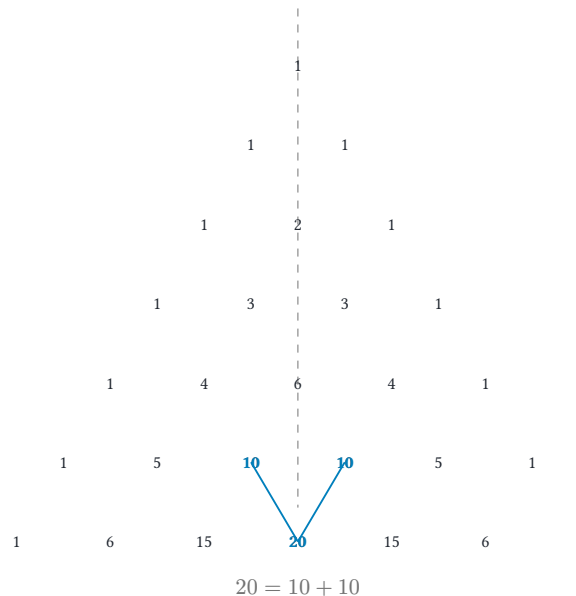


Рис. 85. Треугольник Паскаля: $\binom{n}{k}$, $n = 0, \dots, 6$.

ТЕОРЕМА 18.9 Биномиальная теорема

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k.$$

Биномиальная теорема переводит комбинаторный выбор в алгебраическую операцию.

Набросок доказательства

Докажем комбинаторно, раскрыв произведение в сумму слагаемых.

Раскроем $(x + y)^n = (x + y)(x + y) \cdots (x + y)$ как произведение n одинаковых множителей. Каждое слагаемое получается выбором одной из букв x, y из каждого множителя. Слагаемое $x^{n-k} y^k$: показатель каждой буквы — число множителей, где она выбрана. Количество таких выборов равно $\binom{n}{k}$. Суммирование даёт $(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$.

УТВЕРЖДЕНИЕ 18.10 Основные биномиальные тождества

- **Сумма строки:** $\sum_{k=0}^n \binom{n}{k} = 2^n$ — количество всех подмножеств n -элементного множества.

- **Знакопеременная сумма:** $\sum_{k=0}^n (-1)^k \binom{n}{k} = 0, n \geq 1$. Это подстановка $x = 1, y = -1$ в биномиальную теорему.
- **Тождество «хоккейной клюшки»:** $\sum_{i=k}^n \binom{i}{k} = \binom{n+1}{k+1}$.
- **Свёртка Вандермонда:** $\binom{m+n}{r} = \sum_{k=0}^r \binom{m}{k} \binom{n}{r-k}$. Выбрать r элементов из $m+n$ можно, выбрав k из первых m и $r-k$ из остальных n .
- **Взвешенная сумма:** $\sum_{k=0}^n k \binom{n}{k} = n2^{n-1}$ — средний размер подмножества, умноженный на количество подмножеств.

Набросок доказательства

Докажем каждое тождество по отдельности.

Сумма строки. Подмножества n -элементного множества группируются по размеру: число подмножеств данного размера равно $\binom{n}{k}$, а всего подмножеств — 2^n . Отсюда $\sum_{k=0}^n \binom{n}{k} = 2^n$.

Знакопеременная сумма. Подставим $x = 1, y = -1$ в биномиальную теорему $(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$, получая $0^n = \sum_{k=0}^n (-1)^k \binom{n}{k}$. При $n \geq 1$ левая часть равна нулю.

Свёртка Вандермонда. Выберем r студентов из группы математиков и физиков. Пусть среди выбранных будет k математиков. Тогда математиков можно выбрать $\binom{m}{k}$ способами, а физиков — $\binom{n}{r-k}$ способами. Суммируя по k , получаем $\binom{m+n}{r} = \sum_{k=0}^r \binom{m}{k} \binom{n}{r-k}$.

Тождество «хоккейной клюшки». Число подмножеств размера $k+1$ равно $\binom{n+1}{k+1}$. Сгруппируем эти подмножества по максимальному элементу $i \in \{k, \dots, n\}$: остальные элементы выбираются из $\{0, \dots, i-1\}$, что даёт $\binom{i}{k}$. Суммируя по i , получаем $\sum_{i=k}^n \binom{i}{k} = \binom{n+1}{k+1}$.

Взвешенная сумма. Посчитаем пары «подмножество и отмеченный в нём элемент» двойным подсчётом: метку можно выбрать n способами, а каждый из остальных $n-1$ элементов решает, входить ли в подмножество. Итого $n2^{n-1}$.

Замечание Почему биномиальные коэффициенты возникают везде?

- В алгебре — как коэффициенты разложения $(x+y)^n$.
- В комбинаторике — как количество подмножеств размера k .
- В теории вероятностей — как веса в биномиальном распределении.
- В теории чисел — малая теорема Ферма доказывается через делимость $\binom{p}{k}$ на p .

Это не совпадение. Биномиальный коэффициент — это количество способов выбрать k элементов. Все четыре контекста говорят об одном действии: о выборе подмножества размера k .

- В алгебре — распределение букв x, y по скобкам.
- В теории вероятностей — какие k испытаний завершатся успехом.

- В теории чисел — на какие k позиций поставить копии числа a .

Разные языки — алгебраический, комбинаторный, вероятностный, теоретико-числовой — описывают одну структуру. Когда вы видите $\binom{n}{k}$ в незнакомом контексте, полезно спросить: что здесь выбирается из чего? Ответ почти всегда переводит задачу на язык, в котором она уже решена.

Комбинаторное доказательство работает по принципу «посчитаем одно и то же множество двумя способами». Левая часть тождества — размер объекта, построенного процедурой A . Правая — размер того же объекта, построенного процедурой B . Приравнивание корректно, потому что размер множества не зависит от способа подсчёта.

Разница между алгебраическим и комбинаторным доказательством — это разница между «раскрыть скобки, сократить, получить ответ» и «увидеть, что обе стороны считают один и тот же объект». Комбинаторное доказательство объясняет *почему* тождество верно. Простая проверка такого объяснения не даёт. Например, $\binom{n}{k} = \binom{n}{n-k}$: левая часть — выбрать k элементов для включения в подмножество, правая — выбрать $n - k$ элементов для исключения. И то и другое определяет подмножество размера k , просто разными маршрутами.

§ 18.5 Принцип включений-исключений

Формула включений-исключений (inclusion-exclusion) выглядит как простая поправка на двойной счёт. За знаком $(-1)^{k+1}$ стоит, однако, нетривиальная комбинаторная идея. Принцип включений-исключений — это дискретный аналог формулы для меры объединения, работающий без какой-либо теории меры, на *одних лишь* целых числах. Он систематически исправляет перекрытия, и его приложения простираются от задачи о беспорядках до функции Эйлера и сюръекций. Формула одна — применения радикально различны.

Когда множества пересекаются, правило суммы даёт неверный результат (элементы пересечения считаются дважды). Формула включений-исключений исправляет этот двойной счёт, попеременно прибавляя и вычитая пересечения всё большего числа множеств.

ТЕОРЕМА 18.11 Формула включений-исключений — общий вид

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{k=1}^n (-1)^{k+1} \sum_{1 \leq i_1 < \dots < i_k \leq n} |A_{i_1} \cap \dots \cap A_{i_k}|.$$

Доказательство

Докажем, проверив, что каждый элемент объединения учитывается ровно один раз.

Пусть элемент x принадлежит ровно k из множеств A_i (где $k \geq 1$). В первой сумме $(\sum |A_i|)$ он посчитан $\binom{k}{1}$ раз. Во второй сумме $(\sum |A_i \cap A_j|)$ — $\binom{k}{2}$ раз (вычитается). В третьей сумме — $\binom{k}{3}$ раз (прибавляется). И так далее, причём знак чередуется.

Итого элемент x посчитан

$$\binom{k}{1} - \binom{k}{2} + \binom{k}{3} - \dots + (-1)^{k+1} \binom{k}{k} = - \sum_{i=1}^k (-1)^i \binom{k}{i} = 1 - \sum_{i=0}^k (-1)^i \binom{k}{i} = 1 - (1-1)^k = 1$$

Ровно один раз для любого $k \geq 1$.

Механику формулы удобно увидеть на схеме для трёх множеств на рис. 86. Одиночные области дают +1, двойные пересечения корректируются вычитанием, а тройное пересечение, вычтенное трижды, возвращается. В итоге каждый элемент объединения учитывается ровно один раз благодаря чередованию знаков.

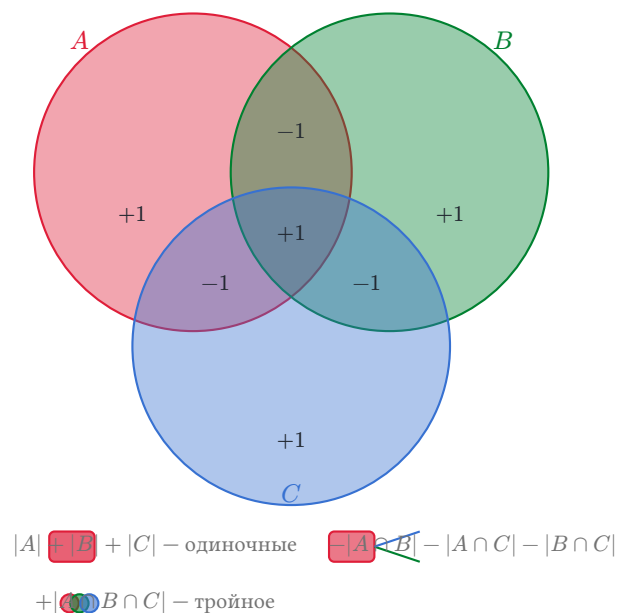


Рис. 86. Формула включений-исключений для трёх множеств.

У формулы включений-исключений множество классических приложений.

Пример Беспорядки

Перестановка n элементов, при которой ни один элемент не остаётся на своём месте, называется **беспорядком** (*derangements*). Количество беспорядков:

$$!n = n! \sum_{i=0}^n \frac{(-1)^i}{i!} \approx \frac{n!}{e}.$$

Вывод: пусть A_i — множество перестановок, в которых элемент i на своём месте. Тогда $|A_i| = (n-1)!$, $|A_i \cap A_j| = (n-2)!$ и так далее. Применяем формулу включений-исключений и получаем результат. Доля беспорядков среди всех перестановок стремится к $\frac{1}{e} \approx 0.3679$, $n \rightarrow \infty$.

Тот же принцип включений-исключений в другом обличье даёт классическую теоретико-числовую функцию.

Пример Функция Эйлера (*Euler's totient function*)

Функция Эйлера $\varphi(n)$ — количество взаимно простых с ним чисел. Подсчёт идёт среди чисел от 1 до n . Если $n = p_1^{e_1} \cdots p_k^{e_k}$ (разложение на простые), то

$$\varphi(n) = n \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right).$$

Вывод: из n чисел вычитаем те, что делятся на каждое p_i . Прибавляем обратно делящиеся на $p_i p_j$, и так далее. Это формула включений-исключений в компактной мультипликативной записи.

Третье классическое приложение — подсчёт сюръекций, отображений «на».

Пример Сюръекции

Количество сюръективных функций из n -элементного множества в k -элементное:

$$k! S(n, k) = \sum_{i=0}^k (-1)^i \binom{k}{i} (k-i)^n.$$

Здесь $S(n, k)$ — числа Стирлинга второго рода (см. ниже). Вывод через включения-исключения: из всех k^n функций вычитаем те, что пропускают конкретный элемент, прибавляем пропускающие два элемента, и так далее.

Замечание Почему знаки в формуле чередуются?

Формула включений-исключений выглядит как технический приём: исправляем двойной счёт, попеременно вычитая и прибавляя пересечения. Знаки сменяются потому, что при каждом добавлении нового множества к пересечению $|B| - |A|$ растёт на единицу, и $(-1)^{|B|-|A|}$ меняет знак. Включения-исключения — это обращение Мёбиуса на булевой решётке подмножеств, где функция Мёбиуса равна $\mu(A, B) = (-1)^{|B|-|A|}$ при $A \subseteq B$. Чередование знаков — прямое следствие этой функции.

Проверка Принцип включений-исключений

1. Сколько слагаемых в формуле для трёх множеств и какие знаки стоят у каждой группы?
2. Во что превращается формула, когда множества попарно не пересекаются?
3. Почему доля беспорядков среди всех перестановок стремится к $\frac{1}{e}$?

§ 18.6 Рекуррентные соотношения

Многие комбинаторные величины определяются *рекуррентным соотношением* — правилом, выражающим следующий член через предыдущие.

ОПРЕДЕЛЕНИЕ 18.10 **Линейное рекуррентное соотношение с постоянными коэффициентами**

Линейным рекуррентным соотношением порядка k с постоянными коэффициентами называется уравнение вида

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k},$$

где $c_1, \dots, c_k$ — константы и $c_k \neq 0$.

Для однозначного определения последовательности задаются начальные условия $a_0, \dots, a_{k-1}$.

Пример Арифметическая и геометрическая прогрессии

- Рекуррента первого порядка $a_n = a_{n-1} + d, a_0 = c$ порождает арифметическую прогрессию. Замкнутая форма: $a_n = c + nd$.
- Рекуррента $a_n = r a_{n-1}, a_0 = c$ — тоже первого порядка, порождает геометрическую прогрессию. Замкнутая форма: $a_n = cr^n$.

В обоих случаях одного начального условия достаточно для однозначного определения всей последовательности.

18.6.1 Решение линейных рекуррент

Для линейных рекуррент с постоянными коэффициентами существует стандартный алгебраический метод решения — метод характеристического уравнения. Пробу $a_n = r^n$ подсказывает сама рекуррента. Подстановка сокращает все степени и оставляет одно уравнение на r — характеристическое. Это ровно та проба, что уже сработала у геометрической прогрессии: там она была ответом, здесь — ключом к уравнению.

УТВЕРЖДЕНИЕ 18.12 Метод характеристического уравнения

Для рекурренты $a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$:

1. Составляем **характеристическое уравнение**: $r^k - c_1 r^{k-1} - \dots - c_k = 0$.
2. Находим его корни $r_1, \dots, r_k$ (возможно, с кратностями).
3. Если все корни различны: общее решение $a_n = \alpha_1 r_1^n + \dots + \alpha_k r_k^n$.
4. Корень кратности m . В решение входят слагаемые $r^n, nr^n, \dots, n^{m-1} r^n$.
5. Коэффициенты α_i определяются подстановкой начальных условий.

Набросок доказательства

Докажем методом характеристического уравнения: будем искать решение в виде $a_n = r^n$.

Подстановка в рекурренту $a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$ даёт

$$r^n = c_1 r^{n-1} + \dots + c_k r^{n-k}.$$

Деление на r^{n-k} приводит к характеристическому уравнению

$$r^k - c_1 r^{k-1} - \dots - c_k = 0,$$

где $r \neq 0$. В силу линейности рекурренты любая линейная комбинация решений также является решением, что и даёт общий вид. Случай кратных корней требует дополнительного обоснования (умножение на n), которое мы опускаем.

Метод характеристического уравнения лучше всего демонстрировать на самом знаменитом рекуррентном соотношении.

Пример Числа Фибоначчи

Рекуррента: $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$.

Характеристическое уравнение: $r^2 - r - 1 = 0$. Корни: $\varphi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2}$ (золотое сечение).

Общее решение: $F_n = \alpha_1 \varphi^n + \alpha_2 \psi^n$. Из начальных условий: $\alpha_1 = \frac{1}{\sqrt{5}}$, $\alpha_2 = -\frac{1}{\sqrt{5}}$.

Формула Бине: $F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$.

В правой части при всех n стоит целое число, несмотря на квадратные корни в записи.

18.6.2 Неоднородные рекуррентные соотношения

Когда в правой части рекурренты есть дополнительная функция $f(n)$, мы имеем дело с *неоднородной* рекуррентой. Решение строится в два этапа.

Для рекурренты $a_n = c_1 a_{n-1} + \dots + c_k a_{n-k} + f(n)$:

1. Находим общее решение $a_n^{(h)}$ однородной части (как если бы $f(n) = 0$).
2. Находим частное решение $a_n^{(p)}$ неоднородного уравнения. Если правая часть — многочлен, умноженный на d^n , угадываем частное решение того же вида с неопределёнными коэффициентами. При пересечении с однородным решением умножаем пробу на n (или на n^m , где m — кратность пересечения).

Случай, когда пробная форма совпала с решением однородной части, — это резонанс: привычный ответ обнуляется, и следующая проба — та же форма, умноженная на n . Механик встречает то же в вынужденных колебаниях на собственной частоте: ответ приходится искать в другом виде.

1. Общее решение: $a_n = a_n^{(h)} + a_n^{(p)}$.

Пример Анализ сортировки слиянием

Время работы сортировки слиянием: $T(n) = 2T(\frac{n}{2}) + n, T(1) = 0$. Это рекуррента вида «разделяй и властвуй» (аргумент делится на 2), поэтому сводим её к линейной подстановкой $n = 2^k$.

Пусть $t_k = T(2^k)$. Тогда $t_k = 2t_{k-1} + 2^k, t_0 = 0$.

Однородное решение: $t_k^{(h)} = A \cdot 2^k$. Частное решение: угадываем $t_k^{(p)} = Bk2^k$. Множитель k нужен, потому что 2^k уже есть в однородном. Подставляем: $Bk2^k = 2 \cdot B(k-1)2^{k-1} + 2^k$. Сокращаем: $Bk = B(k-1) + 1$. Отсюда $B = 1$.

Итак, $t_k = (A + k)2^k$. Начальное условие $t_0 = 0$ даёт $A = 0$. Значит, $t_k = k2^k$. Возвращаясь к $T(n)$: $T(n) = n \log_2 n$.

18.6.3 Основная теорема о рекуррентных оценках

Рекуррентные соотношения — основной инструмент анализа времени работы рекурсивных алгоритмов. Для рекуррент типа «разделяй и властвуй» вида $T(n) = aT(\frac{n}{b}) + f(n)$ существует стандартный инструмент — **основная теорема** (*Master Theorem*). Параметры рекурренты:

1. размер задачи: n ,
2. число подзадач: a ,
3. размер каждой подзадачи: $\frac{n}{b}$,
4. стоимость разделения и комбинирования: $f(n)$.

Теорема сравнивает эту стоимость с функцией $n^{\log_b a}$ — показателем роста числа листьев дерева рекурсии. Выделяются три случая:

Случай 1: стоимость разделения растёт медленнее, чем число листьев: $f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0$. Результат: $T(n) = \Theta(n^{\log_b a})$ — доминируют листья дерева рекурсии.

Случай 2: стоимость разделения *сравнима* с числом листьев: $f(n) = \Theta(n^{\log_b a} \log^k n)$. Результат: $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ — все уровни дерева вносят равный вклад.

Случай 3: стоимость разделения растёт *быстрее*, чем число листьев: $f(n) = \Omega(n^{\log_b a + \varepsilon})$. Дополнительно требуется регулярность: $af(\frac{n}{b}) \leq cf(n)$, $c < 1$. Результат: $T(n) = \Theta(f(n))$ — доминирует корень.

Три случая — три исхода сравнения корня и листьев дерева рекурсии. Стоимость разделения — это корень, вклад листьев — функция $n^{\log_b a}$. Кто растёт быстрее, тот и диктует ответ. Случай 2 — ничейный исход, когда спорщики растут одинаково, и в ответ добавляется лишний логарифм.

Метод характеристического уравнения, изученный в этой главе, — более общий инструмент для линейных рекуррент с постоянными коэффициентами, не сводимых к виду «разделяй и властвуй».¹⁴⁴

§ 18.7 Числа Каталана

Некоторые комбинаторные последовательности возникают так часто, что заслуживают собственных имён и отдельных обозначений. Числа Каталана — возможно, одна из самых часто встречающихся последовательностей в комбинаторике после биномиальных коэффициентов. Вот характерный вход в эту историю.

Сколькими способами можно разрезать выпуклый шестиугольник непересекающимися диагоналями на четыре треугольника? Сосчитайте малые случаи: треугольник — один способ, квадрат — два, пятиугольник — пять. Шестиугольник даёт четырнадцать. Тот же ряд 1, 1, 2, 5, 14 получится для бинарных деревьев с четырьмя внутренними узлами и для правильных скобочных последовательностей (*balanced parentheses*) из четырёх пар — задачи с виду непохожие, а числа одни. Стэнли перечисляет более двухсот комбинаторных интерпретаций чисел Каталана.

ОПРЕДЕЛЕНИЕ 18.11 Числа Каталана

Числа

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

для $n \geq 0$ называются **числами Каталана**. Первые значения:

- $C_0 = 1$
- $C_1 = 1$
- $C_2 = 2$
- $C_3 = 5$

¹⁴⁴Полное доказательство основной теоремы с примерами см. в Т. Н. Cormen, С. Е. Leiserson, R. L. Rivest, С. Stein. «Introduction to Algorithms». 4-е изд., гл. 4.

- $C_4 = 14$
- $C_5 = 42$

Числа Каталана удовлетворяют рекуррентному соотношению:

$$C_0 = 1, C_{n+1} = \sum_{i=0}^n C_i C_{n-i}, n \geq 0.$$

Числа Каталана подсчитывают огромное количество различных комбинаторных объектов:

- Правильные скобочные последовательности из n пар скобок.
- Бинарные деревья с n внутренними узлами.
- Триангуляции выпуклого $(n + 2)$ -угольника непересекающимися диагоналями.
- Пути Дика: пути с шагами $(1, 1), (1, -1)$, не опускающиеся ниже оси абсцисс. Такой путь ведёт из начала координат в точку $(2n, 0)$.
- Расстановка скобок в произведении $n + 1$ матриц: C_n способов (проблема оптимального перемножения цепочки, см. пример ниже).
- Непересекающиеся хорды, соединяющие $2n$ точек на окружности попарно.

$C_3 = 5$. Пять правильных скобочных последовательностей, пять бинарных деревьев, пять триангуляций пятиугольника, пять путей Дика. Ричард Стэнли насчитал 207 комбинаторных интерпретаций последовательности C_n : каждая — способ увидеть одни и те же числа в новой задаче. Одна и та же формула $C_n = \frac{1}{n+1} \binom{2n}{n}$ описывает все эти объекты — подобно тому, как одна и та же кость передней конечности становится крылом летучей мыши, ластом кита и рукой человека. Одна структура — разные выражения. Рекуррента $C_{n+1} = \sum_{i=0}^n C_i C_{n-i}$ перестаёт быть формулой на веру, если увидеть её на хордах.

Набросок доказательства

Докажем построением, разбивая конфигурацию по хорде из фиксированной вершины.

Возьмём $2(n + 1)$ точек на окружности и зафиксируем одну. Проведём из неё хорду так, чтобы по обе стороны осталось чётное число точек: с одной стороны $2i$ точек, с другой — остальные. Соединение точек по каждую сторону — независимая подзадача: слева и справа соответственно C_i, C_{n-i} способов. Правило произведения даёт $C_i C_{n-i}$, а суммирование по всем i — рекурренту из определения.

Числа Каталана возникают в синтаксическом анализе (деревья разбора), анализе алгоритмов (бинарные деревья поиска) и вычислительной геометрии (триангуляции многоугольников)¹⁴⁵.

¹⁴⁵R. P. Stanley. «Catalan Numbers». 2015.

Пример Скобки и матрицы

Произведение четырёх матриц расставляется скобками $C_3 = 5$ способами. Матрицы: $A_1 A_2 A_3 A_4$.

$$((A_1 A_2) A_3) A_4, (A_1 (A_2 A_3)) A_4, A_1 ((A_2 A_3) A_4), A_1 (A_2 (A_3 A_4)), (A_1 A_2) (A_3 A_4).$$

Если размеры матриц различны, разные способы расстановки скобок дают разное количество скалярных умножений. Например, для цепочки $10 \times 30, 30 \times 5, 5 \times 60$: порядок $((A_1 A_2) A_3)$ требует 4500 операций. Порядок $(A_1 (A_2 A_3))$ — 27000 операций. Подсчёт: $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 4500, 30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 27000$. Разница в шесть раз.

Задача о перемножении цепочки матриц (*matrix chain multiplication*) — классический пример динамического программирования: найти способ расстановки скобок, минимизирующий число операций. Пространство поиска — C_{n-1} вариантов. Поэтому полный перебор невозможен уже при $n = 20$. Вариантов там $C_{19} \approx 1.77 \cdot 10^9$.

§ 18.8 Код Прюфера

Числа Каталана считали деревья с точностью до формы. Теперь пометим вершины номерами: деревья с разными пометками — разные, и их уже не перепутаешь. Помеченное дерево допускает сжатие до короткой кодовой записи, и это сжатие обратимо.

Код Прюфера¹⁴⁶ — способ закодировать помеченное дерево с n вершинами последовательностью длины $n - 2$ из чисел $\{1, \dots, n\}$.

Из дерева в код. Повторяем $n - 2$ раза: найти лист (вершину степени 1) с наименьшим номером, записать номер его единственного соседа, удалить лист.

Из кода в дерево. Начинаем с n изолированных вершин. Степень вершины i равна 1 плюс число вхождений i в код. Для каждого элемента кода u слева направо: находим вершину v с наименьшим номером среди вершин степени 1, добавляем ребро $\{u, v\}$, уменьшаем степени u и v на 1. В конце остаются две вершины степени 1 — соединяем их.

Обе процедуры взаимно обратны. Код Прюфера — биекция между помеченными деревьями на n вершинах и последовательностями длины $n - 2$ над алфавитом из n символов.

Пример Код Прюфера дерева на пяти вершинах

Возьмём дерево с рёбрами $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{4, 5\}$.

¹⁴⁶Н. Prüfer. «Neuer Beweis eines Satzes über Permutationen». Archiv der Mathematik und Physik, 1918.

Кодирование. Листья — вершины 2, 3, 5. Берём наименьший лист 2, записываем его соседа 1 и удаляем ребро $\{1, 2\}$. Теперь наименьший лист — 3: записываем 1 и удаляем $\{1, 3\}$. После второго шага листья — вершины 1 и 5. Наименьший из них — 1, его сосед — 4: записываем 4 и удаляем $\{1, 4\}$. Получен код 1, 1, 4. Его длина — $n - 2 = 3$.

Декодирование. Степени вершин: число вхождений 1 в код равно 2, поэтому итоговая степень вершины 1 равна 3. У вершин 2, 3, 5 степень 1, у вершины 4 — 2. Читаем код слева направо. Для первой единицы наименьшая вершина степени 1 — 2: добавляем ребро $\{1, 2\}$. Для второй единицы — 3: добавляем $\{1, 3\}$. Для четвёрки наименьшая вершина степени 1 — 1: добавляем $\{4, 1\}$. В конце степень 1 имеют 4 и 5: соединяем их ребром $\{4, 5\}$. Получились исходные рёбра — кодирование и декодирование взаимно обратны.

18.8.1 Формула Кэли

ТЕОРЕМА 18.13 Формула Кэли¹⁴⁷

Число помеченных деревьев на n вершинах — n^{n-2} .

Доказательство

Докажем построением биекции между деревьями и кодами Прюфера.

По построению, код Прюфера задаёт биекцию между помеченными деревьями на n вершинах и последовательностями длины $n - 2$ из чисел $\{1, \dots, n\}$. Число таких последовательностей — n^{n-2} . Следовательно, помеченных деревьев ровно n^{n-2} .

§ 18.9 Числа Стирлинга и Белла

Разбиения множеств — ещё один класс комбинаторных объектов. Число способов разбить n -элементное множество на непустые блоки дают числа Стирлинга второго рода. Число блоков равно k .

ОПРЕДЕЛЕНИЕ 18.12 Числа Стирлинга второго рода

Количество разбиений n -элементного множества на k непустых непомеченных блоков обозначается $S(n, k)$ и называется **числами Стирлинга второго рода**.

Рекуррентное соотношение:

$$S(n, k) = S(n - 1, k - 1) + kS(n - 1, k),$$

¹⁴⁷ A. Cayley. «A Theorem on Trees». Quarterly Journal of Pure and Applied Mathematics, 1889.

с граничными условиями: $S(0, 0) = 1$, $S(n, 0) = 0$ при $n > 0$, $S(0, k) = 0$ при $k > 0$.

Рекуррента различает судьбу n -го элемента. Он либо образует новый блок — один способ и $S(n-1, k-1)$ вариантов для остальных элементов, либо присоединяется к одному из k существующих блоков — $kS(n-1, k)$ вариантов.

Пример Разбиения 4-элементного множества

Множество $\{1, 2, 3, 4\}$ разбивается на два непустых блока семью способами:

$\{1\}\{2, 3, 4\}$, $\{2\}\{1, 3, 4\}$, $\{3\}\{1, 2, 4\}$, $\{4\}\{1, 2, 3\}$, $\{1, 2\}\{3, 4\}$, $\{1, 3\}\{2, 4\}$, $\{1, 4\}\{2, 3\}$.

Итак, $S(4, 2) = 7$. $S(4, 1) = 1$, $S(4, 3) = 6$, $S(4, 4) = 1$ (в первом случае все элементы в одном блоке). Сумма: $1 + 7 + 6 + 1 = 15 = B_4$, что совпадает с четвёртым числом Белла.

Двойственные числа — Стирлинга первого рода — связаны с циклами перестановок.

ОПРЕДЕЛЕНИЕ 18.13 Числа Стирлинга первого рода

Количество перестановок из n элементов с ровно k циклами обозначается $c(n, k)$ и называется **числами Стирлинга первого рода** (беззнаковыми).

Рекуррентное соотношение:

$$c(n, k) = c(n-1, k-1) + (n-1)c(n-1, k).$$

Рекуррента устроена так же: n -й элемент либо образует собственный цикл, либо вставляется после одного из $n-1$ уже расставленных элементов в одном из k циклов.

Пример Циклы перестановок

- Перестановки трёх элементов с одним циклом: $c(3, 1) = 2$. Это в точности циклические сдвиги (123) , (132) .
- Перестановки с двумя циклами: $c(3, 2) = 3$. Это:
 - $(1)(23)$
 - $(2)(13)$
 - $(3)(12)$
- Перестановки с тремя циклами: $c(3, 3) = 1$ (тождественная).

Сумма $2 + 3 + 1 = 6 = 3!$ — все перестановки трёх элементов.

Наконец, если нас не интересует количество блоков, и мы хотим посчитать все разбиения n -элементного множества, мы получаем числа Белла.

ОПРЕДЕЛЕНИЕ 18.14 Числа Белла

Общее количество всевозможных разбиений n -элементного множества называется **числами Белла** B_n :

$$B_n = \sum_{k=0}^n S(n, k).$$

Первые значения: $B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52$.

Числа Белла — это количество различных отношений эквивалентности на n -элементном множестве (каждое разбиение задаёт отношение эквивалентности, и наоборот).

Пример Разбиения трёхэлементного множества

Все разбиения множества $\{1, 2, 3\}$:

$$\{1, 2, 3\}, \{1\}\{2, 3\}, \{2\}\{1, 3\}, \{3\}\{1, 2\}, \{1\}\{2\}\{3\}.$$

Всего $B_3 = 5$. Первое — один блок (все элементы эквивалентны), последнее — три блока (все элементы попарно неэквивалентны). Пять отношений эквивалентности на трёхэлементном множестве — ровно столько же, сколько разбиений.

Сводка комбинаторных чисел на рис. 9 объединяет все величины, введённые в этой главе: факториалы, биномиальные коэффициенты, числа Стирлинга, Белла и Каталана — с их основными тождествами.

n	$n!$	$\binom{n}{2}$	C_n (Catalan)	$S(n, 3)$ (Stirling)	B_n (Bell)
1	1	0	1	0	1
2	2	1	2	0	2
3	6	3	5	1	5
4	24	6	14	6	15
5	120	10	42	25	52

Таблица 9. Специальные комбинаторные числа при $n = 1, \dots, 5$.

УТВЕРЖДЕНИЕ 18.14 Мультиномиальная теорема

Обобщение биномиальной теоремы на сумму m слагаемых:

$$(x_1 + \dots + x_m)^n = \sum_{k_1 + \dots + k_m = n} \binom{n}{k_1, \dots, k_m} x_1^{k_1} \dots x_m^{k_m},$$

где мультиномиальный коэффициент

$$\binom{n}{k_1, \dots, k_m} = \frac{n!}{k_1! \dots k_m!}.$$

Набросок доказательства

Докажем комбинаторно, раскрыв произведение скобок в сумму слагаемых.

Раскроем $(x_1 + \dots + x_m)^n$ как произведение n одинаковых скобок. Каждое слагаемое получается выбором одного из m переменных из каждой скобки. Пусть переменные выбираются $k_1, \dots, k_m$ раз соответственно. Тогда сумма показателей равна $\sum k_i = n$, а слагаемое имеет вид $x_1^{k_1} \dots x_m^{k_m}$. Число способов получить этот набор показателей равно числу перестановок n объектов, где тип i встречается k_i раз: $\frac{n!}{k_1! \dots k_m!}$.

Пример Перестановки слова МАТЕМАТИКА через мультиномиальный коэффициент

Слово «МАТЕМАТИКА»: 10 букв, из них А — трижды, М и Т — по два, остальные — по одной. Количество перестановок:

$$\binom{10}{3, 2, 2, 1, 1, 1} = \frac{10!}{3! \cdot 2! \cdot 2!} = 151, 200.$$

Это в точности перестановки с повторениями, записанные в форме мультиномиального коэффициента.

§ 18.10 Двенадцатеричный путь

В разделе о шарах и ящиках мы рассмотрели четыре комбинации: шары различимы или нет, ящики различимы или нет. Каждая комбинация давала свою формулу. Но во всех четырёх клетках молчаливо предполагалось, что в ящик можно положить *любое* количество шаров. Это лишь одна из трёх возможностей. Ограничение на заполнение — третий независимый параметр, и он тоже бывает трёх видов: в ящик можно положить любое количество шаров, не более одного, или не менее одного. Четыре клетки превращаются в двенадцать.

Каждый из трёх параметров принимает два или три значения:

- Различимость шаров: различимы, неразличимы.
- Различимость ящиков: различимы, неразличимы.
- Ограничение на заполнение: произвольное, не более одного, не менее одного.

Перемножая мощности множеств значений, получаем полное факторное пространство:

$$2 \times 2 \times 3 = 12.$$

Эта классификация — *двенадцатеричный путь* (*Twelvefold Way*), термин, введённый Джан-Карло Ротой и систематически изложенный Ричардом Стэнли¹⁴⁸. Двенадцать

¹⁴⁸R. P. Stanley. «Enumerative Combinatorics». 2-е изд., 2011.

случаев — не набор разрозненных формул: перестановки, сочетания, числа Стирлинга, разбиения и композиции — это ответы на один и тот же вопрос. Вопрос один: как распределить n шаров по k ящикам. Ответ зависит от значений трёх независимых параметров.

18.10.1 Функциональная формулировка

Различимые шары и различимые ящики — это функции. Разложить n различных шаров по k различным ящикам — то же самое, что задать функцию $f : [n] \rightarrow [k]$. Шар i попадает в ящик $f(i)$. Три ограничения на заполнение приобретают функциональный смысл:

- Произвольное заполнение — все функции.
- Не более одного шара в ящике — инъективные функции.
- не менее одного шара в ящике — сюръективные функции.

Когда шары неразличимы, исчезают метки области определения, и мы считаем наборы прообразов функций. Когда ящики неразличимы, исчезают метки значений. Каждое стирание меток меняет ответ — именно это разнообразие систематизирует таблица в конце раздела.

18.10.2 Шары различимы, ящики различимы

Самый простой блок: каждый шар независимо выбирает ящик.

Произвольное заполнение. Каждый из n шаров — независимый выбор одного из k ящиков. Итого k^n способов.

Пример Письма по ящикам

Три разных письма раскладываются в четыре ящика. У каждого письма четыре варианта, выборы независимы: $4^3 = 64$ способа.

Не более одного шара. Второй шар не может попасть в ящик первого. У шаров соответственно $k, k-1, \dots, k-n+1$ вариантов. Это размещения без повторений:

$$P(k, n) = \frac{k!}{(k-n)!}.$$

Пример Письма по ящикам, не более одного

Те же три письма, но теперь каждый ящик вмещает не более одного. Первое письмо — 4 варианта, второе — 3, третье — 2: $P(4, 3) = 4 \cdot 3 \cdot 2 = 24$.

Не менее одного шара. Каждый ящик обязан получить хотя бы один шар. Сначала разобьём шары на k непустых неразличимых групп: $S(n, k)$ способов. Затем раздадим группам метки ящиков: $k!$ способов. Итого:

$$k!S(n, k).$$

Пример Письма в трёх непустых ящиках

Четыре разных письма по трём ящикам, все ящики непустые. Разбить письма на три группы: $S(4, 3) = 6$ способов. Раздать группам метки ящиков: $3! = 6$ способов. Итого $3! \cdot S(4, 3) = 36$.

18.10.3 Шары неразличимы, ящики различимы

Метки на шарах исчезли, ящики по-прежнему помечены. Распределение — это набор из k чисел $x_1, \dots, x_k$: сколько шаров в каждом ящике. Их сумма равна n : $x_1 + \dots + x_k = n$.

Произвольное заполнение. Число неотрицательных решений считает метод звёзд и перегородок:

$$\binom{n+k-1}{k-1}.$$

Пример Конфеты по коробкам

Пять одинаковых конфет по трём коробкам, пустые коробки разрешены. Пять звёзд и две перегородки: $\binom{7}{2} = 21$ способ.

Не более одного шара. Каждое $x_i \in \{0, 1\}$, и единиц ровно n . Выбираем, какие из k ящиков заполнены:

$$\binom{k}{n}.$$

Пример Мячи по ящикам

Три одинаковых мяча по пяти ящикам, в каждом не более одного. Выбрать три заполненных ящика из пяти: $\binom{5}{3} = 10$.

Не менее одного шара. Каждое $x_i \geq 1$. Положим $y_i = x_i - 1$. Тогда $y_i \geq 0$, $\sum y_i = n - k$. Снова звёзды и перегородки:

$$\binom{n-1}{k-1}.$$

Положительные решения — *композиции* числа n на k частей. Это представления $n = a_1 + \dots + a_k$, $a_i \geq 1$, в которых порядок слагаемых важен.

Пример Шары по непустым ящикам

Пять одинаковых шаров по трём ящикам, все ящики непустые. Композиции числа 5 на 3 части: $\binom{4}{2} = 6$.

Все шесть композиций числа 5 на 3 части получаются перестановками слагаемых: $3 + 1 + 1, 1 + 3 + 1, 1 + 1 + 3, 2 + 2 + 1, 2 + 1 + 2, 1 + 2 + 2$.

Разбиений числа 5 на 3 части всего два: $3 + 1 + 1, 2 + 2 + 1$ (пример выше). Композиция отличается от разбиения тем, что каждая перестановка слагаемых считается отдельной.

18.10.4 Шары различимы, ящики неразличимы

Метки ящиков исчезли, шары остались различимыми. Распределение — разбиение множества шаров на непустые блоки, порядок блоков не важен.

Произвольное заполнение. Блоков может быть сколько угодно, но не больше k :

$$\sum_{i=1}^k S(n, i).$$

Пример Студенты по комнатам

Четыре разных студента расселяются не более чем в три неразличимые комнаты. Разбиения на один, два или три блока: $S(4, 1) + S(4, 2) + S(4, 3) = 1 + 7 + 6 = 14$.

Не более одного шара. Каждый блок — не более одного шара. Если $n \leq k$, способ ровно один: каждому шару — отдельный блок. Если $n > k$, способов нет.

Пример Одиночные ящики

Три разных шара по пяти неразличимым ящикам, в каждом не более одного. Способ ровно один: каждый шар в своём ящике.

Не менее одного шара. Ровно k непустых блоков — числа Стирлинга второго рода $S(n, k)$.

Пример Студенты по командам

Четыре разных студента в три неразличимые команды, все команды непустые. В одну команду попадают двое, в две — по одному. Выбрать пару для команды-двойки: $S(4, 3) = \binom{4}{2} = 6$.

18.10.5 Шары неразличимы, ящики неразличимы

Неразличимы и шары, и ящики. Распределение — способ записать n как сумму целых неотрицательных слагаемых, где порядок слагаемых не важен. Такие представления называются разбиениями числа n . Число разбиений ровно на k частей обозначается $p_k(n)$.

Произвольное заполнение. Частей может быть сколько угодно, но не больше k :

$$\sum_{i=1}^k p_i(n).$$

Пример Шары по неразличимым ящикам

Пять одинаковых шаров по неразличимым ящикам, ящиков не больше трёх. Разбиения числа 5 на одну, две или три части: $p_1(5) + p_2(5) + p_3(5) = 1 + 2 + 2 = 5$.

Не более одного шара. Если $n \leq k$, способ ровно один. Иначе — ноль.

Не менее одного шара. Ровно k частей: $p_k(n)$.

Пример Пять шаров по трём ящикам

Пять одинаковых шаров по трём непустым неразличимым ящикам. Разбиения числа 5 на 3 части: $3 + 1 + 1, 2 + 2 + 1$, всего $p_3(5) = 2$.

Полное число разбиений — без фиксации числа частей — обозначается $p(n) = p_1(n) + \dots + p_n(n)$. Для числа 5 это $5, 4 + 1, 3 + 2, 3 + 1 + 1, 2 + 2 + 1, 2 + 1 + 1 + 1, 1 + 1 + 1 + 1 + 1$ — семь разбиений, и первые значения идут как $p(0) = 1, p(1) = 1, p(2) = 2, p(3) = 3, p(4) = 5, p(5) = 7$.

УТВЕРЖДЕНИЕ 18.15 Рекуррента Эйлера для числа разбиений

Пусть $p(0) = 1$ и $p(n) = 0$ при $n < 0$. Тогда для $n \geq 1$

$$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + p(n-12) + p(n-15) - \dots$$

где аргументы $1, 2, 5, 7, 12, 15, \dots$ — обобщённые пятиугольные числа $g_k = \frac{k(3k-1)}{2}$ для $k = 1, -1, 2, -2, \dots$, а знаки чередуются парами: два плюса, два минуса.

Источник формулы — пентагональная теорема Эйлера: в произведении $\prod_{k \geq 1} (1 - x^k)$ коэффициент при x^n отличен от нуля, только когда n — обобщённое пятиугольное число. Рекуррента проверяется на малых значениях: $p(5) = p(4) + p(3) - p(0) = 5 + 3 - 1 = 7$.

18.10.6 Сводная таблица

Сведём все двенадцать клеток в одну таблицу, показанную на рис. 10.

Шары	Ящики	Ограничение	Количество способов
Различные	Различные	Любое	k^n
Различные	Различные	Не более 1	$P(k, n) = \frac{k!}{(k-n)!}$ — инъективные функции
Различные	Различные	Не менее 1	$k!S(n, k)$ — сюръекции
Неразличимые	Различные	Любое	$\binom{n+k-1}{n}$ — звёзды и перегородки
Неразличимые	Различные	Не более 1	$\binom{k}{n}$ — выбор заполненных ящиков
Неразличимые	Различные	Не менее 1	$\binom{n-1}{k-1}$ — композиции числа n
Различные	Неразличимые	Любое	$\sum_{i=1}^k S(n, i)$ — не более k блоков
Различные	Неразличимые	Не более 1	1 если $n \leq k$, иначе 0
Различные	Неразличимые	Не менее 1	$S(n, k)$ — числа Стирлинга 2-го рода
Неразличимые	Неразличимые	Любое	$\sum_{i=1}^k p_i(n)$ — не более k частей
Неразличимые	Неразличимые	Не более 1	1 если $n \leq k$, иначе 0
Неразличимые	Неразличимые	Не менее 1	$p_k(n)$ — разбиения числа n

Таблица 10. Двенадцатеричный путь: n шаров по k ящикам.

Каждая клетка таблицы — ответ на конкретный вопрос о распределении n шаров по k ящикам. Двенадцать формул выведены одними и теми же приёмами: правилом произведения, звёздами и перегородками, разбиением на блоки. Перестановки, сочетания, числа Стирлинга, разбиения, композиции — всё это клетки одной таблицы $2 \times 2 \times 3$. Структура — в классификации, не в формулах.

Проверка Двенадцатеричный путь

1. По каким трём параметрам различаются клетки двенадцатеричного пути?
2. Какая клетка отвечает инъективным функциям из $[n]$ в $[k]$ и какая формула у неё стоит?
3. В каких клетках появляются числа Стирлинга второго рода и что общего у этих клеток?

18.10.7 Больше двенадцати: ограниченные вместимости

Двенадцать клеток — не предел классификации. Заменяем третий параметр на целое $m \geq 0$ — ограничение на вместимость ящика. Рассматриваем условия: ровно m , не менее m и не более m шаров в каждом ящике. При $m = 1$ условия «не более» и «ровно» совпадают с клетками двенадцатеричного пути. При $m = 0$ условие «не менее» вырождается в «произвольное».

Ровно t шаров в каждом ящике: ограничение на вместимость. Это возможно только при $n = tk$.

- Шары различимы, ящики различимы: разбить n шаров на k групп по t и раздать метки: $\frac{n!}{(t!)^k}$.
- Шары неразличимы, ящики различимы: единственный способ — всем ящикам поровну.
- Шары различимы, ящики неразличимы: разбиения на k блоков размера t : $\frac{n!}{(t!)^k k!}$.
- Шары неразличимы, ящики неразличимы: единственный способ.

Пример Книги по полкам

Шесть разных книг по три на две полки. Выбрать три книги для первой полки, остальные — на вторую: $\binom{6}{3} = 20$. Тот же ответ даёт мультиномиальный коэффициент $\frac{6!}{3!3!} = 20$ — столько же, сколько анаграмм слова из трёх букв А и трёх букв Б.

Не менее t шаров в каждом ящике.

- Шары неразличимы, ящики различимы: кладём по t шаров в каждый ящик заранее. Остаются $n - tk$ шаров на k ящиков без ограничений:

$$\binom{n - tk + k - 1}{k - 1}.$$

- Шары неразличимы, ящики неразличимы: разбиения числа $n - tk$ на не более k частей.
- Для различимых шаров ответы даются включением-исключением: вычитаются распределения, где какой-то ящик получил меньше t шаров.

Пример Шары с минимумом в каждом ящике

Восемь одинаковых шаров по трём ящикам, в каждом не менее двух. Кладём по два шара в каждый ящик (6 шаров), остаются 2 шара на 3 ящика: $\binom{2+3-1}{2} = 6$.

Не более t шаров в каждом ящике.

- Шары неразличимы, ящики различимы: ограниченные звёзды и перегородки считаются включением-исключением — вычитаем распределения, где хотя бы один ящик получил больше t шаров:

$$\sum_{j=0}^k (-1)^j \binom{k}{j} \binom{n - j(m+1) + k - 1}{k - 1}.$$

- Для различимых шаров суммы усложняются. Систематический инструмент — производящие функции (глава 21).

Пример Шары с потолком в каждом ящике

Пять одинаковых шаров по трём ящикам, в каждом не более двух. Всего распределений $\binom{7}{2} = 21$. Вычитаем распределения, где какой-то ящик получил три

и более шаров: $3\binom{4}{2} = 18$ (два ящика сразу получить три шара не могут — не хватит пяти). Остаются $21 - 18 = 3$ шара. Из них собираются тройки $(2, 2, 1)$ во всех порядках.

§ 18.11 Лемма Бёрнсайда и теория Пойа

Многие комбинаторные задачи имеют симметрию. Сколько существует ожерелий из n бусин k цветов, если ожерелья, получающиеся друг из друга поворотом, считаются одинаковыми? Сколько раскрасок граней куба в k цветов с точностью до вращений куба?

Прямой подсчёт даёт k^n — но это считает каждое ожерелье по несколько раз. Ожерелье — это орбита раскраски под действием сдвигов: раскраски одной орбиты не считаются разными. Для непериодического ожерелья в орбите ровно n раскрасок. Нужно систематически учесть симметрию. Для этого потребуется понятие группы — алгебраической структуры, формализующей идею симметрии.

18.11.1 Понятие группы

Группой называется множество с ассоциативной операцией, нейтральным элементом и обратным для каждого элемента, а порядком группы $|G|$ — число её элементов. Определения группы, абелевой группы и порядка элемента находятся в главе 12 (раздел «Группа»). Для подсчёта с точностью до симметрии нужны три семейства групп.

Пример Группы симметрий

- **Циклическая группа $\mathbb{Z}_n$:** числа $\{0, 1, \dots, n-1\}$ со сложением по модулю n . Нейтральный элемент — 0. Обратный элемент: $n-a$ по модулю. Это в точности группа поворотов ожерелья из n бусин.
- **Симметрическая группа S_n :** перестановки с операцией композиции. Порядок группы: $|S_n| = n!$.
- **Группа симметрий квадрата D_4 :** 8 преобразований — 4 поворота и 4 отражения — сохраняющих квадрат. Порядок: $|D_4| = 8$.

18.11.2 Действие группы на множестве

Действие группы на множестве формализует слова «симметрия действует на конфигурации»: повороты ожерелья двигают бусины, вращения куба переставляют грани. Определения действия, орбиты и стабилизатора вместе с доказательством теоремы об орбите и стабилизаторе находятся в главе 12 (раздел «Действие группы на множестве»). Для леммы Бёрнсайда достаточно трёх фактов из этого раздела.

1. Действие сопоставляет элементу $g \in G$ и конфигурации $x \in X$ конфигурацию $g \cdot x$.
2. Орбита $G(x) = \{g \cdot x \mid g \in G\}$ собирает конфигурации, переводимые друг в друга симметриями, и орбиты разбивают X на классы эквивалентности.
3. Неподвижные точки элемента — множество $X^g = \{x \in X \mid g \cdot x = x\}$, а орбита и стабилизатор G_x связаны равенством $|G(x)| \cdot |G_x| = |G|$.

18.11.3 Лемма Бёрнсайда

ТЕОРЕМА 18.16 Лемма Бёрнсайда¹⁴⁹

Количество орбит группы G , действующей на множестве X , равно среднему числу неподвижных точек элементов группы:

$$\text{число орбит} = \frac{1}{|G|} \sum_{g \in G} |X^g|.$$

Набросок доказательства

Докажем двойным подсчётом мощности множества неподвижных пар.

Подсчитаем мощность множества $\{(g, x) \mid g \cdot x = x\}$ двумя способами. С одной стороны: для каждого g вклад равен $|X^g|$. Сумма по всем g даёт левую часть равенства (без деления на $|G|$).

С другой стороны: для каждого элемента x число симметрий g , оставляющих x на месте, равно $|G_x|$. Стабилизатор x имеет размер $|G_x|$ по теореме об орбите и стабилизаторе (глава 12). Суммируя по представителям орбит и группируя по орбитам, получаем $|G|$ раз число орбит. Приравнявая два подсчёта и деля на $|G|$, получаем утверждение.

Пример Ожерелья из трёх бусин двух цветов

$X = \{0, 1\}^3$ — двоичные строки длины 3. Таких строк $2^3 = 8$ (0 — чёрная бусина, 1 — белая). Группа $G = \{0, 1, 2\}$ — циклические сдвиги (поворот на 0, 1 или 2 позиции). $|G| = 3$.

Неподвижные точки:

- Сдвиг на 0 (тождественный): все 8 строк неподвижны.
- Сдвиг на 1: строка неподвижна, только если все бусины одного цвета. Таких строк две: 000, 111.
- Сдвиг на 2: аналогично, 2 строки.

¹⁴⁹W. Burnside. «Theory of Groups of Finite Order». 1897 (2-е изд., 1911, §232). Лемма была известна Коши (1845) и Фробениусу (1887) задолго до книги Бёрнсайда. Бёрнсайд популяризовал её в своём учебнике, и имя закрепилось. В континентальной Европе она чаще известна как лемма Коши–Фробениуса.

Лемма Бёрнсайда: число орбит $= \frac{8+2+2}{3} = 4$.

Действительно: ожерелья — это классы строк с точностью до циклических сдвигов. Классы эквивалентности:

1. $\{000\}$ (все чёрные),
2. $\{111\}$ (все белые),
3. $\{001, 010, 100\}$ (одна белая бусина),
4. $\{011, 101, 110\}$ (две белых).

Всего 4.

Четыре орбиты ожерелий из трёх бусин двух цветов показаны на рис. 87. Каждая связанная группа строк — одна орбита относительно циклических сдвигов.

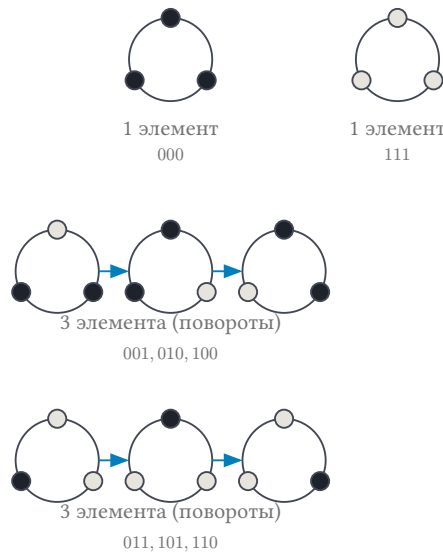


Рис. 87. Четыре орбиты ожерелий из трёх бусин двух цветов относительно циклических сдвигов.

18.11.4 Теорема Пойа

Теорема Пойа обобщает лемму Бёрнсайда: вместо подсчёта неподвижных точек для каждого g она кодирует их количество через цикловую структуру перестановки. Число циклов длины i обозначается c_i . Перестановка g разлагается на такие циклы. Всего циклов: $c(g) = c_1 + c_2 + \dots$. Тогда $|X^g| = k^{c(g)}$: каждый цикл обязан быть одноцветным. Тогда число орбит:

$$\text{число орбит} = \frac{1}{|G|} \sum_{g \in G} k^{c(g)}.$$

Формулу $k^{c(g)}$ сразу видно на ожерельях: нетождественный поворот трёх бусин — один цикл длины три. Все бусины цикла обязаны быть одного цвета: вместо k^3 раскрасок поворот оставляет k — это видно без единой строки перечисления.

Цикловой индекс — это многочлен $P_G(t_1, \dots, t_n) = \frac{1}{|G|} \sum_{g \in G} t_1^{c_1} \dots t_n^{c_n}$. Подстановка $t_i = x_1^i + \dots + x_k^i$ учитывает веса цветов. Для подсчёта числа орбит берётся $t_i = k$.

Примечание Применения леммы Бёрнсайда

Лемма Бёрнсайда и теорема Пойа применяются в:

- Перечислительной комбинаторике (ожерелья, раскраски графов, химические изомеры).
- Теории кодирования (подсчёт кодовых слов с точностью до симметрий алфавита).
- Кристаллографии (подсчёт конфигураций атомов в решётке с точностью до пространственной группы симметрии).
- Комбинаторном дизайне (блок-схемы, латинские квадраты с точностью до изоморфизма).

§ 18.12 * Коды Грея и комбинаторная генерация

В разделе 18.3 мы перечисляли перестановки лексикографически: у каждого объекта — естественный номер. Цена нумерации — размашистые шаги: при переходе $1, 3, 5, 4, 2 \rightarrow 1, 4, 2, 3, 5$ перестраиваются четыре позиции из пяти, а при двоичном счёте подмножеств за маской 0111 идёт 1000 — три элемента ушли, один вошёл, и всё, что поддерживается вдоль перечисления, пересобирается целиком. *Код Грея* — порядок, в котором соседние двоичные слова отличаются ровно в одном бите, так что каждый шаг перечисления меняет подмножество на один элемент.

В главе 11 этот порядок уже работал на схемотехнику: там одновременное переключение нескольких битов порождает ложные состояния, и однобитное соседство его устраняет. Здесь — комбинаторная сторона: код Грея как порядок перечисления, у которого шаг константен и у каждого слова есть номер.

Строится порядок рекурсивно, отражением списка. Список G_1 — слова 0 и 1. Каждая следующая ступень удваивает список: сначала все слова с приписанным слева нулём, затем те же слова в обратном порядке с приписанной слева единицей. Для краткости через G^R обозначаем список, выписанный от последнего слова к первому.

ОПРЕДЕЛЕНИЕ 18.15 Отражённый код Грея

Последовательность списков $G_1, G_2, \dots$ называется **отражённым кодом Грея** (*binary reflected Gray code*), если G_1 состоит из слов 0 и 1, а G_n при $n \geq 2$ получается из G_{n-1} приписыванием слева нуля ко всем словам и затем — списка G_{n-1}^R с приписанной слева единицей.

Для $n = 2$ список даёт слова 00, 01, 11, 10. Для $n = 3$ получается 000, 001, 011, 010, 110, 111, 101, 100. Слово «отражённый» видно прямо в списке:

вторая половина — зеркальный двойник первой. Каждое из 2^n слов встречается ровно один раз: половины различаются старшим битом, а внутри половины повторов нет по предположению индукции.

Вся конструкция нацелена на соседей: два слова, идущие в списке подряд, обязаны различаться минимально. Докажем это свойство — оно и превращает построенный порядок в код Грея.

ТЕОРЕМА 18.17 Однобитное соседство

В отражённом коде Грея G_n любые соседние слова различаются ровно в одном бите. Различаются ровно в одном бите и последнее слово $10\dots 0$ с первым словом $00\dots 0$, поэтому перечисление замыкается в цикл.

Доказательство

Докажем индукцией по n .

База. В G_1 слова 0 и 1 образуют единственную соседнюю пару и различаются единственным битом. Этой же парой замыкается цикл.

Шаг. Пусть для G_{n-1} соседство и цикл уже установлены, первое слово G_{n-1} равно $00\dots 0$, а последнее — $10\dots 0$. Внутри каждой половины G_n приписанный бит постоянен, а порядок слов наследуется от G_{n-1} , в единичной половине — в обратном направлении. Обращение списка сохраняет соседство, поэтому соседи внутри половин различаются ровно в одном бите. На стыке половин встречаются слова $0w$ и $1w$, где w — последнее слово G_{n-1} : обращённый список открывается именно им. Эти два слова различаются одним старшим битом. Замыкает цикл пара из последнего слова G_n , которое равно $100\dots 0 = 10\dots 0$, и первого слова $00\dots 0$ — снова ровно один бит различия.

Список нумерует свои слова: перевод номера в слово называется *анранжированием* (*unranking*), а обратный перевод слова в номер — *ранжированием* (*ranking*). Пусть номер i записан n битами $i_{n-1}\dots i_0$, а $g_{n-1}\dots g_0$ — слово на этом месте, с той же нумерацией битов от младшего. Старший бит переходит без изменений, а каждый следующий равен XOR бита номера с его старшим соседом:

$$g_k = i_k \oplus i_{k+1}, \quad k < n-1.$$

Обратно, номер восстанавливается сверху вниз накопительным XOR: $i_{n-1} = g_{n-1}$ и далее $i_k = g_k \oplus i_{k+1}$. Обе формулы — то же преобразование, которое доказано в главе 11 для перевода между двоичной записью и кодом Грея. Нова здесь интерпретация: двоичная запись i — это номер слова в перечислении.

Пример Анранжирование и ранжирование при $n = 4$

Возьмём номер $i = 0110$: биты $i_3 = 0, i_2 = 1, i_1 = 1, i_0 = 0$. Старший бит сохраняется: $g_3 = 0$. Остальные — XOR с соседом слева: $g_2 = 1 \oplus 0 = 1, g_1 = 1 \oplus 1 = 0,$

$g_0 = 0 \oplus 1 = 1$. Слово с номером 6 (список ведётся с нуля) — это 0101. Обратный ход: $i_3 = g_3 = 0$, затем $i_2 = 1 \oplus 0 = 1$, $i_1 = 0 \oplus 1 = 1$, $i_0 = 1 \oplus 1 = 0$. Номер восстановлен: $i = 0110$.

Отсюда готовый движок перечисления подмножеств. Слово g — маска текущего подмножества n -элементного множества, и переход от номера i к $i + 1$ переключает ровно один бит, номер которого равен $\text{ctz}(i + 1)$ — счёту младших нулей (*count trailing zeros*), числу нулей в младшем конце записи $i + 1$. Прибавление единицы превращает младший блок единиц номера в нули и ставит единицу сразу над блоком. XOR соседних битов гасит все эти изменения, кроме одного переключения на месте новой единицы. Шаг перечисления стоит константное число операций — инкремент, XOR и переключение одного бита маски — причём равномерно на каждом шаге, а не в среднем. Двоичный счётчик тоже меняет биты с амортизированной константой, поскольку каждый следующий разряд переключается вдвое реже предыдущего, но отдельный его шаг способен перестроить всю маску: 0111 $\rightarrow$ 1000. Код Грея снимает и эту неравномерность: подмножество, поддерживаемое как явная структура, уточняется на один элемент и никогда не пересобирается целиком.

То же соседство работает в железе. Угловой энкодер (*rotary encoder*) — диск, разбитый на 2^n секторов и несущий n концентрических дорожек: сектор кодируется словом, и вдоль каждой дорожки стоит свой фотодатчик. Датчики переключаются не одновременно: они разнесены по радиусу, а допуски и вибрация размазывают моменты срабатывания, и на границе секторов каждый датчик короткое время видит свой старый или новый бит. На двоичном диске граница секторов 0111 и 1000 переключает четыре бита, и датчики способны снять любое из 2^4 сочетаний — от 0000 до 1111, то есть ошибиться на половину шкалы. На диске Грея соседние секторы различаются одним битом, поэтому любой промах датчика даёт соседний сектор. Для шарнира робота или наведения телескопа разница между «промахом на сектор» и «перепутанной половиной оборота» — разница между поправкой и аварией.

Практика для этого раздела живёт в проекте главы: его стадия о коде Грея просит собрать отражённую конструкцию, добавить ранжирование с анранжированием и проверить программно, что соседние слова различаются ровно одним битом.

§ 18.13 Начала теории Рамсея

Принцип Дирихле гарантирует: среди $n + 1$ объектов и n ящиков два объекта попадут в один ящик. Теория Рамсея обобщает этот принцип: в любой достаточно большой структуре неизбежно возникает упорядоченная подструктура.

Фрэнк Рамсей (1903–1930), кембриджский математик, доказавший свою теорему в возрасте 25 лет и умерший два года спустя, установил, что для любых s, t существует

число $R(s, t)$. Любой граф с не менее чем $R(s, t)$ вершинами содержит клику размера s либо независимое множество размера t .

Принцип Дирихле, как в начале главы: среди 367 человек найдутся двое с общим днём рождения. Теория Рамсея обобщает это: среди шести человек найдутся либо трое попарно знакомых, либо трое попарно незнакомых — и в этом уже видна структура, выходящая за рамки принципа Дирихле.

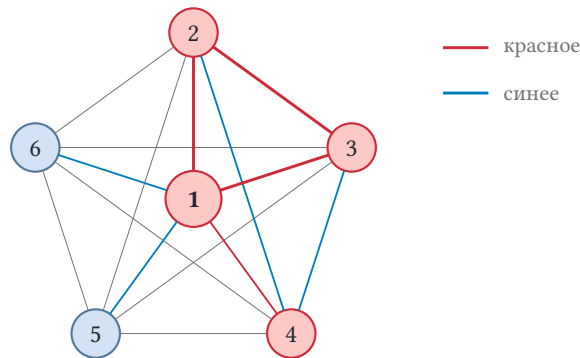
ОПРЕДЕЛЕНИЕ 18.16

Числа Рамсея

Числом Рамсея $R(s, t)$ называется минимальное число вершин, при котором неизбежна клика размера s либо независимое множество размера t (антиклика).

Эквивалентно: $R(s, t)$ — минимальное n , при котором любая раскраска рёбер K_n в два цвета содержит красный K_s либо синий K_t .

Ключ к доказательству $R(3, 3) = 6$ — конфигурация на рис. 88. У вершины 1 минимум три ребра одного цвета, и их концы неизбежно дают монохроматический треугольник.



Из 5 рёбер от вершины 1 минимум 3 одного цвета.

Среди их концов найдётся ребро того же цвета

либо все три ребра — другого цвета.

Рис. 88. Неизбежность порядка: в любой 2-раскраске K_6 найдётся монохроматический K_3 .

Пример Малые числа Рамсея

- $R(3, 3) = 6$: любая компания из шести человек содержит либо троих попарно знакомых, либо троих попарно незнакомых. Доказательство — в разделе о принципе Дирихле.
- $R(4, 4) = 18$ (доказано в 1955 году).
- $R(3, 4) = 9$
- $R(3, 5) = 14$
- $R(3, 6) = 18$
- $R(3, 7) = 23$
- $R(3, 8) = 28$
- $R(3, 9) = 36$

- $R(5, 5)$ неизвестно. Известно лишь, что $43 \leq R(5, 5) \leq 48$.

За полвека вычислений диапазон для $R(5, 5)$ так и не был сведён к точному значению. Пространство перебора астрономически велико: число графов на n вершинах — $2^{\binom{n}{2}}$. Уже при $n = 43$ это число с 272 десятичными цифрами.

Рамсеевская теорема в общем виде: при раскраске r -элементных подмножеств бесконечного множества в k цветов найдётся бесконечное подмножество, все r -подмножества которого одноцветны. Конечная версия: для любых r, k и размера m существует n , такое что при любой k -раскраске r -подмножеств n -элементного множества найдётся m -элементное подмножество, все r -подмножества которого имеют один цвет. Конечная версия выводится из бесконечной компактностным рассуждением. Если бы для всякого n нашлась контрраскраска без одноцветного подмножества нужного размера, из них можно было бы извлечь контрраскраску и для бесконечного множества.

При $r = 1$ это принцип Дирихле. При $r = 2, k = 2$ это утверждение $R(s, t) < \infty$. При больших r — теория Рамсея в полном объёме.

Примечание Рамсеевская теория и computer science

Рамсеевские нижние оценки используются в:

- *Теории сложности*: доказательство нижних оценок размера схем (экстракторы, хэш-функции) использует рамсеевские аргументы.
- *Алгоритмах аппроксимации*: рамсеевские числа показывают, что некоторые приближённые алгоритмы гарантированно находят хорошее решение на достаточно больших экземплярах.
- *Комбинаторной геометрии*: теорема Хелли, теорема Эрдёша–Секереша о выпуклых многоугольниках.¹⁵⁰

§ 18.14 Комбинаторные игры (Ним)

Теория Рамсея утверждает неизбежность порядка. Комбинаторные игры — напротив, поле битвы, где порядок создаётся стратегией.

Игра Ним: кучки камней размерами $n_1, \dots, n_k$. Два игрока ходят по очереди. За один ход игрок выбирает кучку и забирает из неё любое ненулевое количество камней. Проигрывает тот, кто не может сделать ход (все кучки пусты).

Вопрос: для заданной позиции $(n_1, \dots, n_k)$ кто выигрывает при оптимальной игре — первый или второй?

¹⁵⁰R. L. Graham, B. L. Rothschild, J. H. Spencer. «Ramsey Theory». 2-е изд., 1990.

ОПРЕДЕЛЕНИЕ 18.17 Nim-сумма

Nim-суммой позиции называется XOR размеров всех кучек:

$$\text{nim}(n_1, \dots, n_k) = n_1 \oplus n_2 \oplus \dots \oplus n_k.$$

ТЕОРЕМА 18.18 Теорема Бутона¹⁵¹

Позиция в Ниме выигрышна для второго игрока, если её Nim-сумма равна 0. В противном случае выигрывает первый.

Набросок доказательства

Докажем, выявив инвариант: второй игрок поддерживает нулевую Nim-сумму.

Если Nim-сумма равна 0, любой ход делает её ненулевой (изменение равно одной кучки меняет XOR).

Если Nim-сумма не равна 0, существует ход, делающий её нулевой. Для этого найдём старший ненулевой бит Nim-суммы, выберем кучку, в которой этот бит установлен, и заменим её размер на размер xor Nim-сумма . Новое значение всегда меньше исходного (старший бит обнуляется), и XOR становится нулевым.

Терминальная позиция (все кучки пусты) имеет Nim-сумму 0. Таким образом, второй игрок всегда может поддерживать Nim-сумму 0 после своего хода, двигая игру к терминальной позиции, где первый не может сделать ход.

Пример Ним с тремя кучками

Позиция (3, 4, 5). Nim-сумма: $3 \oplus 4 \oplus 5 = (011)_2 \oplus (100)_2 \oplus (101)_2 = (010)_2 = 2$.

Сумма не ноль — выигрывает первый. Ход:

- $4 \oplus 2 = 6 > 4$ (не подходит)
- $3 \oplus 2 = 1 < 3$ (подходит)
- $5 \oplus 2 = 7 > 5$ (не подходит)

Берём кучку 3, оставляем $3 \oplus 2 = 1$ камень. Новая позиция (1, 4, 5) проигрышна для второго: $1 \oplus 4 \oplus 5 = 0$.

Теорема Бутона показывает, как простая алгебраическая операция (XOR) полностью решает, казалось бы, сложную комбинаторную игру. XOR мы впервые встретили в булевой алгебре (глава 10) как логическую операцию. Здесь она даёт ключ к стратегии.

¹⁵¹C. L. Bouton. «Nim, a game with a complete mathematical theory». Annals of Mathematics, 1901. Первый полный анализ impartial-игры. Позже обобщён теоремой Шпрага–Гранди (Sprague–Grundy, 1935–1939).

Примечание Функция Шпрага–Гранди

Ним — не единственная игра с таким свойством. Теорема Шпрага–Гранди (1935–1939) утверждает, что любая impartial-игра (оба игрока имеют одинаковые возможные ходы из каждой позиции) эквивалентна некоторой кучке Нима определённого размера — её **числу Гранди**.

Число Гранди позиции $G(v)$ определяется рекурсивно:

$$G(v) = \text{mex}(\{G(w) \mid w \text{ достижима из } v\}),$$

где mex (*minimum excluded*) — наименьшее неотрицательное целое, не входящее в множество. Игра в целом эквивалентна Ниму с кучкой размера $G(v)$, и XOR чисел Гранди компонент даёт общую оценку.

Это обобщение превращает Nim из одной игры в универсальный метод анализа impartial-игр.

§ 18.15 Итоги главы

Пересчёт колоды из 52 карт сразу показывает, зачем нужна комбинаторика: число вариантов превышает число звёзд в наблюдаемой Вселенной, и перебор здесь бессилен. Вместо перечисления мы собрали аппарат — правила суммы, произведения и дополнения, перестановки и сочетания, включения-исключения и рекурренты — каждый из которых отвечает на вопрос о количестве одним приёмом. Двенадцатеричный путь систематизировал сами вопросы: три параметра — различимы ли шары, различимы ли ящики и каково ограничение на заполнение — дают двенадцать клеток таблицы. Биномиальные коэффициенты и их тождества связали комбинаторику с алгеброй.

Сквозной приём главы — двойной подсчёт: посчитать одно и то же множество двумя способами и приравнять результаты. Лемма Бёрнсайда добавляет к нему подсчёт с точностью до симметрии — деление на $|G|$ — а числа Каталана и Стирлинга показывают, как одна формула собирает разные комбинаторные объекты: скобочные последовательности, триангуляции, пути Дика.

Принцип Дирихле научил нас доказывать существование, не предъявляя искомого объекта, а теория Рамсея и функция Шпрага–Гранди завершают картину: комбинаторные гарантии существуют даже там, где структура кажется произвольной. Рекуррентные соотношения и метод характеристического уравнения превращают подсчёт в алгебру: рекуррента становится алгебраическим уравнением.

Собранный аппарат уходит дальше по курсу. Производящие функции (глава 21) дадут рекуррентам новый инструмент, теория чисел (глава 19) продолжит линию подсчёта в модулярной арифметике, а дискретная вероятность (глава 20) применит те же формулы к случайным исходам.

Проверка Проверьте себя

1. Почему перестановки и сочетания считаются по-разному, и в каком смысле сочетание — это размещение, делённое на $k!$?
2. Что такое принцип Дирихле, и как он доказывает существование, не предъявляя искомого объекта?
3. Когда правило суммы даёт неверный ответ, и как формула включений-исключений исправляет двойной счёт?
4. Как метод характеристического уравнения превращает рекурренту в алгебраическое уравнение?
5. Почему числа Каталана считают и скобочные последовательности, и триангуляции, и пути Дика — одной формулой?
6. Что считает лемма Бёрнсайда, и почему в её формуле фигурирует деление на $|G|$?
7. Как функция Шпрага–Гранди сводит произвольную игру к Ниму?

Что дальше?

Метод характеристического уравнения, изученный в этой главе, работает для линейных рекуррент с постоянными коэффициентами. Для нелинейных рекуррент существует другой инструмент — производящие функции (глава 21). Но непосредственная следующая глава — теория чисел и криптография (глава 19): делимость, модулярная арифметика, RSA и эллиптические кривые. А дискретная вероятность (глава 20) применит формулы этой главы к случайным исходам.

§ 18.16 Упражнения

Правила подсчёта.

1. Покажите, что в любой компании из шести человек найдутся либо трое попарно знакомых, либо трое попарно незнакомых. Почему пяти человек для этого утверждения недостаточно?

Перестановки, размещения и сочетания.

2. Сколькими способами можно выбрать председателя, секретаря и казначея из 20 кандидатов?
3. Сколько различных перестановок букв слова «МАТЕМАТИКА»?
4. Сколькими способами можно рассадить 8 человек за круглым столом? Вращения стола считаются одинаковыми.
5. * Сколько существует перестановок чисел $1, 2, \dots, 10$, в которых числа 1 и 2 не стоят рядом?

Биномиальные коэффициенты и двойной подсчёт.

6. Найдите $\binom{100}{98}$.
7. Докажите комбинаторно: $\sum_{k=0}^n \binom{n}{k} = 2^n$. Что считает левая часть? Что — правая?
8. Докажите комбинаторно: $\sum_{k=0}^n k \binom{n}{k} = n2^{n-1}$. Подсказка: посчитайте двумя способами количество способов выбрать комитет с председателем из n человек.
9. * Докажите тождество Вандермонда комбинаторно: $\binom{m+n}{r} = \sum_{k=0}^r \binom{m}{k} \binom{n}{r-k}$.

Принцип включений-исключений.

10. Найдите количество целых чисел от 1 до 200, которые делятся на 3 или на 7.
11. Найдите количество целых чисел от 1 до 1000000, взаимно простых с 42.
12. Найдите количество сюръекций из множества $\{1, 2, 3, 4, 5, 6\}$ в множество $\{a, b, c, d\}$.
13. * Найдите количество целочисленных неотрицательных решений $x_1 + x_2 + x_3 + x_4 = 20, x_1 \leq 8$.

Рекуррентные соотношения.

14. Решите рекуррентное соотношение $a_n = 7a_{n-1} - 10a_{n-2}, n \geq 2, a_0 = 2, a_1 = 5$.
15. Найдите количество способов замостить полосу $2 \times n$ плитками размера 2×1 (домино). Выведите рекуррентное соотношение и решите его.
16. * Решите неоднородную рекурренту $a_n = 3a_{n-1} - 2a_{n-2} + 2^n, n \geq 2, a_0 = 1, a_1 = 2$.

Специальные числа.

17. Найдите $S(6, 3)$ — число Стирлинга второго рода (количество разбиений 6-элементного множества на 3 непустых блока).
18. ** Найдите количество способов расставить скобки в произведении $n + 1$ матрицы. Выведите рекурренту и проверьте, что ответом служат числа Каталана C_n .

Симметрия, Рамсей и Ним.

19. Сколько различных ожерелий из 6 бусин можно составить из бусин двух цветов? Ожерелья, получающиеся поворотом, считаются одинаковыми. Примените лемму Бёрнсайда.
20. Для позиции в Ниме $(2, 4, 7)$ найдите выигрышный ход первого игрока.
21. * Сколько существует раскрасок граней куба в 3 цвета с точностью до вращений куба? Группа вращений куба содержит 24 элемента: тождественное, 6 вращений на 90° и 270° вокруг осей через середины противоположных граней, 3 — на 180° вокруг тех же осей, 8 — на 120° вокруг осей через противоположные вершины, 6 — на 180° вокруг осей через середины противоположных рёбер.

22. * Покажите, что рамсеевское число $R(3, 4) \leq 9$. Используйте неравенство $R(s, t) \leq R(s-1, t) + R(s, t-1)$ и значения $R(2, t), R(s, 2)$ (при чётных слагаемых — на единицу меньше).
23. * Вычислите коэффициент $[x^7](1+x+x^2+x^3)^4$. Подсказка: коэффициент равен числу целочисленных решений $x_1 + x_2 + x_3 + x_4 = 7, 0 \leq x_i \leq 3$.

ПРОЕКТ Генераторы комбинаторных объектов: перечисление, ранг, де-ранжирование

Комбинаторные объекты — последовательности с точными индексами, а не абстракции: каталог можно перечислять, нумеровать и доставать по номеру. Соберите инструмент, который для перестановок, сочетаний, беспорядков и разбиений числа перечисляет объекты в лексикографическом порядке, присваивает ранг и восстанавливает объект по рангу.

① Перестановки, сочетания и беспорядки

Перестановки перечисляйте классическим `next_permutation`: найдите самый длинный убывающий суффикс, обменяйте опорный элемент (стоящий перед суффиксом) с наименьшим элементом суффикса, который больше него, и переверните суффикс. В лексикографическом порядке получаются все $n!$ перестановок, и следующей после $(2, 4, 1, 3)$ оказывается $(2, 4, 3, 1)$. Ранг и де-ранг ведите по факториальной системе счисления, проверяя $\text{rank}(\text{unrank}(k)) = k$. Сочетания — это k -подмножества из n : перечислите их в лексикографическом порядке и убедитесь, что их $\binom{n}{k}$, сверив $\binom{10}{5} = 252$. Для ранга сочетания на позиции i сбросьте все подмножества, у которых i -й элемент меньше текущего. Беспорядки перечисляйте поиском без неподвижных точек (элемент не стоит на своём месте). Их число считайте субфакториалом $!n = (n-1)(!(n-1) + !(n-2))$, сверяя $!n$: 0, 1, 2, 9, 44, 265 для $n = 1..6$.

② Разбиения числа

Посчитайте $p(n)$ рекуррентой Эйлера из раздела о разбиениях. Перечислите невозрастающие разбиения числа. Сверьте, что перечисление даёт ровно $p(n)$ объектов. Сверьте $p(10) = 42$.

③ Код Грея

Лексикографический порядок меняет сразу много позиций за шаг. Код Грея — порядок, в котором соседние объекты отличаются ровно в одной позиции. Сгенерируйте бинарный отражённый код Грея для n бит: G_n — это G_{n-1} со старшим нулём, за которым идёт G_{n-1} в обратном порядке со старшей единицей. Добавьте ранг и де-ранг и проверьте, что соседние коды отличаются ровно одним битом.

④ Разбиения множества и числа Каталана

Перестановки упорядочивают элементы, сочетания выбирают подмножества, а разбиение множества раскладывает элементы по непустым непомеченным блокам. Сгенерируйте все разбиения множества $\{0, \dots, n-1\}$ и по-

считайте их число Белла $B_n = \sum_k S(n, k)$. Проверьте $B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5$. Числа Каталана C_n считают правильные скобочные последовательности из n пар скобок: сгенерируйте все такие последовательности длины $2n$, добавляя открывающую скобку, пока их меньше n , и закрывающую, пока она не превышает число открывающих. Посчитайте C_n рекуррентно: $C_0 = 1, C_{n+1} = \sum_{i=0}^n C_i \cdot C_{n-i}$. Сверьте с замкнутой формой $C_n = \frac{\binom{2n}{n}}{n+1}$ и первыми значениями 1, 1, 2, 5, 14, 42. Добавьте пути Дика — пути из $(0, 0)$ в $(2n, 0)$ шагами вверх и вниз, не опускающиеся ниже нуля — и сопоставьте их со скобками.

⑤ **Ранг и де-ранг новых семей**

Перечисление — это список, ранг — позиция объекта в списке, а де-ранг восстанавливает объект по номеру. Для разбиений множества ранжируйте строку ограниченного роста $r_0 \dots r_{n-1}$ как число со смешанным основанием: разряд i принимает значения $0.. \max(r_0, \dots, r_{i-1}) + 1$. Для правильных скобочных последовательностей ранжируйте лексикографически, разбивая строку на A и B после первой пары скобок. Проверьте, что ранг и де-ранг взаимно обратны и что перечисление идёт строго по возрастанию ранга.

Итог: инструмент, который для каждого семейства перечисляет объекты, присваивает ранг и восстанавливает объект по рангу. Количество объектов совпадает с формулой $(n!, \binom{n}{k}, !n, p(n))$.

XIX

Теория чисел и криптография

Цезарь сдвигал буквы на три позиции, и его курьеры носили в обе стороны одно число — ключ. Шифрование превращает текст в бессмыслицу, ключ возвращает его обратно. Пока шифруют и расшифровывают двое, логика проста: оба знают один секрет.

Трудность начинается там, где ключ нужно передать. Чтобы защитить канал, нужен секрет, но передать его можно только по каналу, который ещё не защищён. Это круг: веками его разрывали курьерами, личными встречами и физической охраной бумаг. Чем длиннее канал, тем дороже доставка секрета.

Прошлая глава 18 отвечала на вопрос «сколько», не перечисляя: одно правило вместо миллиона шагов. Числа там оставались материалом счёта — факториалы, степени, биномиальные коэффициенты — а сами были в тени. Теперь они выходят из тени: что значит «делится», из каких кирпичей собрано всякое число и почему остатки от деления умеют хранить секрет. Понять устройство числа — значит получить и способ «строить замок», и способ «искать в нём трещину».

Каждый раз, когда браузер показывает замок в адресной строке, две машины, никогда не встречавшиеся, договариваются о секрете. Оплата картой, вход в мессенджер, подтверждение личности — всё это держится на вопросах о числах. Подпись — число, которое умеет составить только владелец ключа, а проверить может каждый. Криптовалюты пошли дальше: там секрет доказывают — и снова арифметикой. Обычно о математике под этими стандартами не думают — до первого взлома.

Идея, обращающая этот круг, называется *криптографией с открытым ключом*. Не нужно договариваться о секрете заранее: каждый участник держит при себе *свой* ключ, а для связи публикует лишь половину — замок, который может закрыть любой, а открыть — только владелец. RSA, протокол Диффи–Хеллмана и эллиптические кривые — три способа построить такой замок, и все три держатся на одной математической предпосылке.

Предпосылка — существование *односторонних функций*: легко вычислить, трудно обратить. Умножить два простых числа может каждый, а разложить на множители их произведение — практически невозможно, если числа занимают тысячи бит. Асимметрия видна уже на малых числах: каждый мгновенно проверит, что 91 — составное, а вот найти его делители 7 и 13 — уже маленький перебор. Чем длиннее число, тем сильнее расходится эта пара: проверка простоты остаётся быстрой, а поиск делителей упирается в перебор, растущий экспоненциально. Возвести число в

степень по модулю — просто. Восстановить показатель по результату — дискретный логарифм — на практике не удаётся. Секретность держится на том, что некоторое вычисление трудно выполнить. Решение инженерное, а его обоснование — математическое: никто до сих пор не доказал, что факторизация и дискретный логарифм действительно трудны. Это предположения, на которых стоит целая индустрия.

Инструменты для строительства — старые, как сама арифметика: делимость, алгоритм Евклида, сравнения по модулю, малая теорема Ферма, китайская теорема об остатках. Каждый из них появился задолго до электричества, а сегодня работает в каждом защищённом соединении. Вопрос «что вообще можно вычислить быстро» уведёт дальше, в теорию сложности (глава 30), а делимость и сравнения ещё понадобятся дискретной вероятности (глава 20). Как из вековых фактов о делимости собрать систему, которую может проверить каждый, а сломать — никому?

ИСТОРИЯ От чисел к ключам

Математика делимости развивалась медленно и без практических приложений. Алгоритм Евклида известен с III века до н.э. Малую теорему Ферма сформулировал Пьер Ферма в 1640 году, первое доказательство опубликовал Эйлер в 1736-м. Китайская теорема об остатках восходит к III веку н.э. (Сунь Цзы). Систематическое западное изложение — «Disquisitiones Arithmeticae» Гаусса¹⁵². Всё это время вопрос «делится ли одно число на другое» оставался абстрактным, без инженерного выхода.

В 1970-е две публикации изменили это положение. Диффи и Хеллман предложили протокол общего ключа в 1976-м, а Ривест, Шамир и Адлеман — RSA в 1977-м. Вековые факты о простых числах и остатках оказалось можно обратить в гарантии секретности. Криптография с открытым ключом стала одним из самых массовых применений чистой математики — от банковских карт до транспортных протоколов интернета.

ОБЗОР ГЛАВЫ

В этой главе мы выясним, что вообще значит «сломать шифр»: принцип Керкгоффса и простейшие атаки — перебор и частотный анализ. Затем поставим фундамент: делимость, алгоритм Евклида и его расширенный вариант, модулярная арифметика с обратными элементами, малая теорема Ферма и построенные на ней тесты простоты, теорема Эйлера и китайская теорема об остатках. На этом фундаменте построим RSA и сразу проверим его атакой: маллобильность и общий модуль покажут, что стойкой математики мало — нужна аккуратная конструкция. Протокол Диффи–Хеллмана добавит вторую одностороннюю функцию — дискретный логарифм — и встретит свои дыры: человека посередине и атаку Полига–Хеллмана. Эллиптические кривые дадут ту же секретность при меньших ключах. В конце мы увидим,

¹⁵²C. F. Gauss. «Disquisitiones Arithmeticae». 1801.

почему вся эта индустрия держится на предположениях о трудности, и что с этими предположениями делает квантовый алгоритм Шора.

После этой главы вы сможете:

1. объяснять принцип Керкгоффса и проверять систему атакой;
2. пользоваться делимостью, алгоритмом Евклида и модулярной арифметикой;
3. применять малую теорему Ферма, теорему Эйлера и китайскую теорему об остатках;
4. проверять простоту больших чисел тестом Миллера–Рабина и объяснять, почему тест Ферма обманываем;
5. строить RSA и протокол Диффи–Хеллмана и находить их уязвимости;
6. объяснять роль односторонних функций и то, что делает с ними квантовый алгоритм Шора.

§ 19.1 Взлом как проверка безопасности

Пока мы не построили ни одной системы, зафиксируем главное правило: безопасность не постулируется, она проверяется атакой. Криптосистема считается защищённой, потому что *попытка* её взломать не удаётся. Поэтому дальше, построив каждую систему, мы сразу же попробуем её сломать.

ОПРЕДЕЛЕНИЕ 19.1 Принцип Керкгоффса

Криптосистема должна оставаться безопасной, даже если злоумышленнику известно всё, кроме ключа. Вся секретность — в ключе. Алгоритм открыт.

Принцип Керкгоффса (Август Керкгоффс, 1883) кажется контринтуитивным: почему бы не спрятать сам алгоритм? Потому что спрятать его невозможно надолго: алгоритмы публикуются, стандарты открыты, а противник может добыть описание системы почти любым способом. Значит, секретным может быть только то, что участники способны *передать* друг другу по защищённому каналу — а это ключ.

ЛОВУШКА Безопасность через неясность

Надеяться, что злоумышленник не знает, как устроена система, — плохая стратегия. Такая безопасность рухнет при первой же утечке описания алгоритма. Правило Керкгоффса переносит секретность с устройства алгоритма на трудность математической задачи: пока труден обратный шаг односторонней функции, секретность держится.

Простейшая атака — *полный перебор* (*brute force*): попробовать все возможные ключи. Шифр Цезаря сдвигает каждую букву на фиксированное число позиций. Ключ — это сдвиг от 1 до 32 (для 33-буквенного русского алфавита).

Пример Взлом шифра Цезаря перебором

Перехвачено слово «ЫЛЧУ». Пробуем все 32 сдвига в обратную сторону. При сдвиге на 3 позиции получаем осмысленное «ШИФР». Перебор тривиален: 32 попытки — и открытый текст найден. Ключей мало, и никакая хитрость не спасает: шифр Цезаря ломается за секунды.

Перебор побеждает, когда ключей мало. Но и длинный ключ не гарантирует безопасность: *структура* сообщения может выдать его без всякого перебора.

Пример Частотный анализ

В русском тексте буква «о» встречается чаще остальных, за ней идут «е», «а», «и». Если шифр простой замены подменяет каждую букву фиксированной буквой, самая частая буква шифротекста, скорее всего, соответствует самой частой букве языка. Сопоставив частоты, злоумышленник восстанавливает замену и расшифровывает текст, не зная ключа. Частотный анализ — исторический пример того, как структура сообщения разрушает шифр, который считался невзламываемым.

Перебор атакует *длину* ключа, частотный анализ — *структуру* сообщения. Современные системы не сдаются ни тому, ни другому, но их уязвимости лежат в более тонких местах: в математике, на которой они построены, и в том, как они используются. Теперь, вооружившись взглядом злоумышленника, построим фундамент.

§ 19.2 Делимость и алгоритм Евклида**ОПРЕДЕЛЕНИЕ 19.2 Наибольший общий делитель**

Наибольшим общим делителем двух целых чисел a, b , не равных одновременно нулю, называется наибольшее целое число d , делящее и a , и b , обозначается $d = \gcd(a, b)$.

Если $\gcd(a, b) = 1$, числа называются *взаимно простыми* (*coprime*): у них нет общего делителя, кроме единицы. Взаимная простота решает, у каких чисел есть обратный элемент по модулю m : ровно у взаимно простых с m .

Как вычислить наибольший общий делитель? Наивный путь — разложить оба числа на простые множители — требует факторизации, а она трудна. Но общий делитель можно найти без разложения, одним лишь делением с остатком.

АЛГОРИТМ Алгоритм Евклида

Основан на простом наблюдении: если $a = b \cdot q + r$, то делители a и b — это делители b и остатка r . Отсюда рекуррентное соотношение:

$$\gcd(a, b) = \gcd(b, a \bmod b), \quad \gcd(a, 0) = a.$$

Итеративно: пока $b \neq 0$, заменяем пару на следующую:

$$(a, b) \rightarrow (b, a \bmod b).$$

Когда $b = 0$, ответ — a .

Доказательство Корректность

Докажем, что наибольший общий делитель не меняется на каждом шаге алгоритма. Общие делители a и b — это общие делители b и остатка $a \bmod b$. Значит, на каждом шаге НОД не меняется, и достаточно вычислить его для последней пары $(d, 0)$. Общие делители d и 0 — делители d , наибольший из них — сам d . Итак, последний ненулевой остаток делит оба исходных числа и делится на любой их общий делитель — это и есть наибольший общий делитель.

Пример Вычисление $\gcd(48, 18)$

Шаг за шагом остаток убывает, и мы доходим до нуля:

$$\gcd(48, 18) = \gcd(18, 12) = \gcd(12, 6) = \gcd(6, 0) = 6.$$

Проверка: 6 делит и 48, и 18. Большого общего делителя нет: $18 = 3 \cdot 6$, $48 = 8 \cdot 6$.

Почему алгоритм быстрый? Потому что остаток быстро убывает: при $a \geq b$ верно

$$a \bmod b < \frac{a}{2}.$$

Доказательство

Докажем разбором двух случаев: $b \leq \frac{a}{2}$ и $b > \frac{a}{2}$. Если $b \leq \frac{a}{2}$, то остаток меньше b и тем более меньше $\frac{a}{2}$. Если же $b > \frac{a}{2}$, то остаток равен $a - b < \frac{a}{2}$. Поэтому каждый второй шаг уменьшает аргумент вдвое, и число шагов — $O(\log a)$.

Наихудший случай дают соседние числа Фибоначчи: $\gcd(F_{n+1}, F_n) = 1$, но на его вычисление уходят все n шагов.

Пример Соседние числа Фибоначчи как наихудший случай

$\gcd(F_8, F_7) = \gcd(21, 13)$. Каждый шаг делит с остатком, и почти все частные равны единице: $21 = 1 \cdot 13 + 8$, $13 = 1 \cdot 8 + 5$, $8 = 1 \cdot 5 + 3$, $5 = 1 \cdot 3 + 2$, $3 = 1 \cdot 2 + 1$, $2 = 2 \cdot 1 + 0$.

Понадобилось шесть делений, и остаток убывает максимально медленно. Для сравнения, пара $(48, 18)$ из примера выше справилась за три шага.

Для модулярной арифметики нам понадобится больше, чем сам делитель: *коэффициенты Безу*.

ОПРЕДЕЛЕНИЕ 19.3 Расширенный алгоритм Евклида

Для a и m с $\gcd(a, m) = 1$ **расширенный алгоритм Евклида** находит целые x и y , такие что

$$ax + my = 1.$$

Коэффициенты находятся теми же шагами, что и в обычном алгоритме: на каждом шаге запоминают, как выражается текущий остаток через исходные a и m .

Для чего нужны коэффициенты Безу? Если $ax + my = 1 \Rightarrow ax \equiv 1 \pmod{m}$, то $x \pmod{m}$ — обратный элемент к a по модулю m . Это первый кирпичик открытых ключей: чтобы «закрыть замок» и «открыть его», нужны взаимно обратные числа по модулю.

§ 19.3 Модулярная арифметика

ОПРЕДЕЛЕНИЕ 19.4 Сравнение по модулю (*congruence*)

Два целых числа a, b **сравнимы по модулю m** , если m делит их разность:

$$a \equiv b \pmod{m} \leftrightarrow m \text{ делит } (a - b).$$

На практике сравнение по модулю — это арифметика остатков: числа, дающие одинаковый остаток при делении на m , попадают в один класс эквивалентности.

Пример Сравнения по модулю 12

Это в точности арифметика на циферблате: 14 часов — это 2 часа пополудни.

$$14 \equiv 2 \pmod{12}, \quad 5 \equiv 17 \pmod{12}, \quad -7 \equiv 5 \pmod{12}.$$

Все числа, кратные 12, сравнимы с 0: $0 \equiv 12 \equiv 24 \equiv -12 \pmod{12}$. Классы вычетов (*residue classes*) по модулю 12: $[0], [1], \dots, [11]$. Здесь $[k] = \{\dots, k - 12, k, k + 12, k + 24, \dots\}$.

Арифметика по модулю m образует *кольцо*: результат любой операции приводится по модулю m . Сложение и умножение работают «как обычно», с одним дополнительным шагом — взятием остатка. Главный вопрос: у всякого ли числа есть *обратный* элемент?

ОПРЕДЕЛЕНИЕ 19.5 Обратный элемент по модулю

Обратным к a по модулю m называется число a^{-1} , для которого

$$a \cdot a^{-1} \equiv 1 \pmod{m}.$$

Обратный элемент существует не всегда: a имеет обратный по модулю m тогда и только тогда, когда $\gcd(a, m) = 1$. Почему? Докажем эквивалентность в обе стороны. Если $ax \equiv 1 \pmod{m}$, то разность $ax - 1$ кратна m . Значит, $ax - 1 = mk$ для некоторого целого k . Любой общий делитель a и m делит 1, значит, a и m взаимно просты. Обратно, при взаимной простоте из равенства $ax + my = 1$ получаем обратный элемент к a . Это число $x \bmod m$.

Пример Обратный элемент по модулю

Найдём $7^{-1} \bmod 26$. Расширенный алгоритм Евклида даёт

$$7 \cdot 15 + 26 \cdot (-4) = 105 - 104 = 1,$$

значит, $7^{-1} \equiv 15 \pmod{26}$. Проверка: $7 \cdot 15 = 105 = 4 \cdot 26 + 1 \equiv 1 \pmod{26}$.

АЛГОРИТМ Быстрое возведение в степень (метод квадратов)

Наивное вычисление $a^e \bmod m$ требует $e - 1$ умножений, а криптографические показатели занимают тысячи бит. Метод квадратов вычисляет $a^e \bmod m$ за $O(\log e)$ умножений по модулю.

1. Разложим показатель по степеням двойки: $e = b_0 + 2b_1 + 4b_2 + \dots$, где каждый бит b_i равен нулю или единице.
2. Вычислим по модулю m последовательные степени $a, a^2, a^4, a^8, \dots$ — каждая следующая получается возведением предыдущей в квадрат.
3. Перемножим по модулю m те степени, у которых соответствующий бит равен единице.

Например, $5^{15} \bmod 23 = 5^8 \cdot 5^4 \cdot 5^2 \cdot 5^1 \bmod 23$, потому что $15 = 8 + 4 + 2 + 1$: три возведения в квадрат и три умножения вместо четырнадцати.

§ 19.4 Малая теорема Ферма и функция Эйлера

Возведение в степень по модулю — будущая односторонняя функция, поэтому важно понять её периодичность. Точную форму этой периодичности для простого модуля даёт малая теорема Ферма.

ТЕОРЕМА 19.1 Малая теорема Ферма

Если p — простое число и a не делится на p , то

$$a^{p-1} \equiv 1 \pmod{p}.$$

Эквивалентная форма — для любого целого a :

$$a^p \equiv a \pmod{p}.$$

Доказательство

Докажем, рассматривая умножение на a как перестановку ненулевых остатков.

Рассмотрим ненулевые остатки $1, 2, \dots, p-1$. Умножение на a переставляет их: числа $a \cdot 1, a \cdot 2, \dots, a \cdot (p-1)$ дают те же остатки в другом порядке. Из $ai \equiv aj \pmod{p}$ сокращение по a даёт $i \equiv j$, потому что обратный a^{-1} существует — его существование обеспечивает взаимная простота $\gcd(a, p) = 1$.

Перемножив обе последовательности, получаем

$$a^{p-1} \cdot (p-1)! \equiv (p-1)! \pmod{p}.$$

Число $(p-1)!$ взаимно просто с p , поэтому на него можно сократить — остаётся $a^{p-1} \equiv 1 \pmod{p}$.

Малая теорема Ферма работает по *простому* модулю, а RSA — по составному $n = pq$. Для составного модуля нужен аналог — теорема Эйлера, а для неё — функция Эйлера.

$\varphi(n)$ считает числа от 1 до n , взаимно простые с n . Для простого p имеем $\varphi(p) = p-1$: все меньшие числа взаимно просты с p . Для $n = pq$ с разными простыми p и q не взаимно просты с n только числа, кратные p (их q), и числа, кратные q (их p). Число pq входит в оба списка. Не взаимно простых всего $q + p - 1$. А взаимно простых — остальные, ровно

$$\varphi(n) = pq - q - p + 1 = (p-1)(q-1).$$

ТЕОРЕМА 19.2 Теорема Эйлера

Если $\gcd(a, n) = 1$, то

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

При простом $n = p$ теорема Эйлера превращается в малую теорему Ферма, так как $\varphi(p) = p-1$. Эйлер обобщил Ферма ровно настолько, насколько нужно для RSA: модуль составной, и показатель степени берётся $\varphi(n)$.

Доказательство

Доказательство повторяет схему малой теоремы Ферма: вместо всех ненулевых остатков берём лишь те, что взаимно просты с n . Таких остатков среди $1, \dots, n$ —

1 ровно $\varphi(n)$. Умножение на a переставляет их: из $ax \equiv ay \pmod{n}$ сокращение законно, потому что у a есть обратный по модулю n . Произведение P всех элементов системы при этой перестановке не меняется:

$$a^{\varphi(n)} \cdot P \equiv P \pmod{n}.$$

Каждый сомножитель P взаимно прост с n , поэтому и произведение взаимно просто с n , и сокращение даёт $a^{\varphi(n)} \equiv 1 \pmod{n}$.

Пример Проверка для $p = 7, a = 3$

Малая теорема Ферма: $3^6 = 729 = 7 \cdot 104 + 1$, поэтому

$$3^6 \equiv 1 \pmod{7}.$$

Период степени по модулю p делит $p - 1$. Здесь $3^3 = 27 \equiv 6 \equiv -1 \pmod{7}$. Поэтому $3^6 \equiv (-1)^2 = 1$.

Степени тройки по модулю 7 идут по кругу: $3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1$, дальше всё повторяется. До шестого шага значения не повторяются, поэтому цикл покрывает все ненулевые остатки: степени одного элемента пробегают всю мультипликативную группу $\mathbb{Z}_7^*$ (точное определение — в разделе 19.8), как на рис. 89. Число шагов до возврата в единицу называют порядком элемента. У тройки порядок максимален и равен $p - 1$, так что малая теорема Ферма для $p = 7$ видна прямо на цикле.

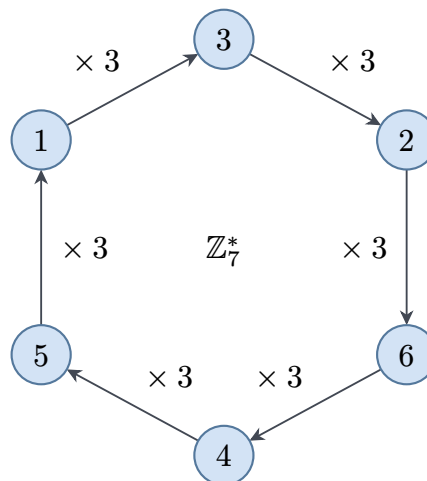


Рис. 89. Цикл степеней тройки по модулю 7.

§ 19.5 Тесты простоты

Генерация ключей RSA начинается со слов «выбрать два больших простых числа», и на этом шаге нужна проверка простоты. Таблиц таких чисел не существует: кандидаты в тысячи бит генерируются случайно и проверяются тестом. Проверить

простоту можно быстро, а разложить число на множители — трудно, поэтому владелец ключа находит простые тестами, а взломщик остаётся с задачей факторизации. Плотность простых чисел делает перебор кандидатов дешёвым: возле числа порядка 2^{1024} простое встречается в среднем каждое 710-е.

Малая теорема Ферма даёт первый тест. Если для некоторого a сравнение $a^{n-1} \equiv 1 \pmod{n}$ нарушено, число n составное: для простого модуля сравнение выполнялось бы при любом a , не делящемся на n . Проверка одного основания — это одно быстрое возведение в степень. Если же сравнение выполнено, простота ещё не доказана: такое сравнение дают и составные числа.

ОПРЕДЕЛЕНИЕ 19.6 Псевдопростое число

Составное число n , взаимно простое с a , называется **псевдопростым** по основанию a , если

$$a^{n-1} \equiv 1 \pmod{n}.$$

Классический пример — $341 = 11 \cdot 31$: число составное, но $2^{340} \equiv 1 \pmod{341}$, и тест Ферма по основанию 2 его пропускает. Доказательство этого сравнения — в упражнении, а здесь важен вывод: одно основание ничего не гарантирует, и у каждого фиксированного основания псевдопростых чисел бесконечно много. Хуже того, существуют числа, обманывающие тест Ферма по всем основаниям, взаимно простым с ними, — числа Кармайкла (*Carmichael numbers*). Наименьшее из них — $561 = 3 \cdot 11 \cdot 17$. Для числа Кармайкла тест Ферма неотличим от настоящей проверки простоты, поэтому тест нуждается в усилении.

Усиление опирается на квадратные корни из единицы. По простому модулю сравнение $x^2 \equiv 1 \pmod{p}$ имеет только решения $x \equiv \pm 1$, а по составному модулю появляются корни, отличные от ± 1 . Тест Миллера–Рабина проверяет именно их: если квадрат числа равен единице, а само число не равно ± 1 , модуль составной.

АЛГОРИТМ Тест Миллера–Рабина

Дано нечётное n . Запишем $n - 1 = 2^s d$, где d нечётно, и выберем случайное основание a от 2 до $n - 2$.

1. Вычисляем $x = a^d \pmod{n}$. Если $x = 1$ или $x = n - 1$, раунд пройден.
2. Иначе $s - 1$ раз возводим x в квадрат по модулю n . Появление $n - 1$ проходит раунд. Появление 1 без предварительного $n - 1$ обнаруживает нетривиальный корень из единицы и доказывает, что n составное.
3. Если после $s - 1$ возведений значение $n - 1$ так и не появилось, n составное.

Тест односторонний: вердикт «составное» всегда верен, а ошибиться может только вердикт «простое». По оценке Монье–Рабина у составного нечётного n обманывают тест не более четверти оснований из диапазона $1, \dots, n - 1$, поэтому k независимых раундов дают вероятность ошибки не более 4^{-k} , и 40 раундов

означают 2^{-80} . Этот тест находит простые модули в реальной генерации ключей и реализуется в проекте главы.

Тест Миллера–Рабина ловит и то, что прошло тест Ферма. Для числа 341 запись $340 = 2^2 \cdot 85$ даёт $x = 2^{85} \equiv 32 \pmod{341}$, и 32 не равно ни 1, ни 340. Одно возведение в квадрат даёт $32^2 = 1024 \equiv 1 \pmod{341}$: квадрат числа, отличного от ± 1 , оказался единицей, значит, найден нетривиальный корень из единицы и 341 составное.

Проверка Тесты простоты

1. Почему тест Ферма нельзя спасти выбором удачного основания, и какую роль играют числа Кармайкла?
2. Какому свойству квадратных корней из единицы обязан тест Миллера–Рабина и почему по составному модулю такие корни существуют?
3. Что означает односторонность теста Миллера–Рабина и откуда берётся оценка 4^{-k} для k раундов?

§ 19.6 Китайская теорема об остатках

Малая теорема Ферма говорит о степенях по одному модулю. Китайская теорема об остатках решает обратную задачу: как собрать решение из решений по *нескольким* взаимно простым модулям.

ТЕОРЕМА 19.3 Китайская теорема об остатках

Пусть $m_1, \dots, m_k$ — попарно взаимно простые числа. Тогда для любых остатков $a_1, \dots, a_k$ система сравнений

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

имеет решение, и оно единственно по модулю произведения $m_1 m_2 \cdots m_k$.

Доказательство

Докажем построением: соберём решение из чисел-заготовок, а затем отдельно покажем единственность.

Положим $M = m_1 m_2 \cdots m_k$ и $M_i = \frac{M}{m_i}$. Число M_i делится на все модули, кроме m_i , и взаимно просто с m_i , поэтому у M_i есть обратный по модулю m_i . Заготовка $e_i = M_i \cdot M_i^{-1}$ даёт остаток 1 по модулю m_i и остаток 0 по всем остальным мо-

дулям. Тогда сумма $x = \sum a_i \cdot e_i$ даёт по модулю m_i ровно a_i : вклад остальных заготовок обнуляется, а свой даёт $a_i \cdot 1$. Решение существует.

Покажем единственность. Если x и y — два решения, то разность $z = x - y$ делится на каждый модуль m_i . Для пары модулей m_i, m_j равенство Безу $um_i + vm_j = 1$, умноженное на z , даёт $z = um_i z + vm_j z$. Первое слагаемое делится на произведение $m_i m_j$, потому что m_j делит z , а второе — потому что m_i делит z . Значит, $m_i m_j$ делит z , и индукция по числу модулей распространяет делимость на всё произведение M . Поэтому $x \equiv y \pmod{M}$ и решение единственно по модулю M .

CRT — рабочий инструмент модулярной арифметики. Она позволяет разбить вычисления по большому модулю на независимые вычисления по меньшим взаимно простым модулям. В RSA это используется для ускорения дешифрования. Вместо $m = c^d \bmod n$ (где $n = pq$) вычисляют остатки

$$m_p = c^d \bmod p, m_q = c^d \bmod q,$$

а затем собирают m по CRT.

Пример Система из двух сравнений

Решим

$$\begin{cases} x \equiv 2 \pmod{3} \\ x \equiv 3 \pmod{5} \end{cases}.$$

Перебираем числа, дающие 2 по модулю 3: 2, 5, 8, Первое, дающее 3 по модулю 5, — это 8: $8 \bmod 5 = 3$. Значит, $x \equiv 8 \pmod{15}$: решения — 8, 23, 38, ...

Тот же ответ даёт конструкция из доказательства теоремы. Положим $M_1 = 5$: обратным к нему по модулю 3 служит 2. Проверка: $5 \cdot 2 = 10 \equiv 1 \pmod{3}$. Для $M_2 = 3$ обратным по модулю 5 служит тоже 2. Взвешенная сумма: $x = 2 \cdot 5 \cdot 2 + 3 \cdot 3 \cdot 2 = 38 \equiv 8 \pmod{15}$ — то же решение, что и перебором.

§ 19.7 Шифрование с открытым ключом (RSA)

RSA (Rivest–Shamir–Adleman, 1977) — первый широко известный и практически применяемый алгоритм асимметричного шифрования. Его стойкость — в предположительной трудности факторизации произведения двух больших простых чисел. Вся математика уже на столе: алгоритм Евклида, теорема Эйлера и CRT.

УТВЕРЖДЕНИЕ 19.4 Генерация ключей RSA

1. Выбрать два больших простых числа p, q .

2. Вычислить модуль и функцию Эйлера: $n = pq$, $\varphi(n) = (p-1)(q-1)$.
3. Выбрать открытый показатель e , взаимно простой с $\varphi(n)$ (обычно $e = 65537$).
4. Вычислить закрытый показатель $d = e^{-1} \bmod \varphi(n)$ расширенным алгоритмом Евклида.

Открытый ключ: (n, e) . **Закрытый ключ:** (n, d) . Шифрование и дешифрование — возведение в степень по модулю:

$$c = m^e \bmod n, \quad m = c^d \bmod n.$$

Открытый показатель e обычно мал — стандарт берёт $e = 65537$ — а закрытый d того же размера, что $\varphi(n)$, и огромен. Зашифровать быстро может каждый, а расшифровать — только владелец большого d . После генерации пары ключей простые числа p и q больше не нужны: в открытом ключе от них остаётся лишь произведение $n = pq$. Сами p и q можно уничтожить.

Примечание Почему $e = 65537$

Число $65537 = 2^{16} + 1$ простое, а его двоичная запись 10000000000000001_2 состоит из двух единиц. Метод квадратов возводит в такую степень шестнадцатью возведениями в квадрат и одним умножением, поэтому шифрование и проверка подписи дешёвы. Простота e делает проверку взаимной простоты с $\varphi(n)$ лёгкой: достаточно убедиться, что $\varphi(n)$ не кратен e . Совсем малый показатель опасен при неудачном использовании: без паддинга одно сообщение, зашифрованное показателем $e = 3$ для трёх получателей, восстанавливается по китайской теореме об остатках — это атака Хостада. Стандартный ответ — показатель 65537 вместе с обязательным паддингом.

Почему дешифрование возвращает исходное сообщение? Проверим, что $c^d \equiv m \pmod{n}$.

Набросок доказательства

Докажем разбором случаев: сначала случай $\gcd(m, n) = 1$, затем m , делящееся на p или q . Поскольку d выбран как $e^{-1} \bmod \varphi(n)$, для некоторого целого k верно

$$ed = 1 + k\varphi(n).$$

Тогда

$$c^d \equiv (m^e)^d = m^{ed} = m \cdot (m^{\varphi(n)})^k.$$

По теореме Эйлера: $\gcd(m, n) = 1 \Rightarrow m^{\varphi(n)} \equiv 1 \pmod{n}$. Отсюда $c^d \equiv m \pmod{n}$. Если $\gcd(m, n) \neq 1$, то m делится на p или q .

Если $p \mid m$, то обе части сравнения $m^{ed} \equiv m \pmod{p}$ равны 0 по модулю p . Если же p не делит m , то $m^{p-1} \equiv 1 \pmod{p}$. $ed - 1$ кратно $p - 1$, поэтому $m^{ed} \equiv m \pmod{p}$.

То же рассуждение по модулю q , и китайская теорема об остатках собирает сравнение по модулю n . В любом случае $c^d \equiv m \pmod{n}$.

Пример RSA на малых числах

Выберем $p = 5, q = 11$. Тогда $n = 55, \varphi(n) = 4 \cdot 10 = 40$. Возьмём открытый показатель $e = 3$: он взаимно прост с 40. Закрытый показатель: $d = 3^{-1} \pmod{40} = 27$. Проверка: $3 \cdot 27 = 81 = 2 \cdot 40 + 1$.

Сообщение $m = 7$. Шифрование: $c = 7^3 = 343 \equiv 13 \pmod{55}$. Для дешифрования возводим в квадрат: $13^2 \equiv 4, 13^4 \equiv 16, 13^8 \equiv 36, 13^{16} \equiv 31 \pmod{55}$. Так как $27 = 16 + 8 + 2 + 1$,

$$13^{27} \equiv 31 \cdot 36 \cdot 4 \cdot 13 \equiv 7 \pmod{55},$$

— исходное сообщение восстановлено.

Злоумышленник знает только $(n, e) = (55, 3)$. Чтобы найти d , ему нужно разложить 55 на множители. На этом примере — перебором, при настоящих размерах ключа такой перебор неосуществим.

Что именно делает RSA «односторонней функцией»? Открытый ключ (n, e) позволяет зашифровать — это возведение в степень, выполнимое за доли секунды. Обратный шаг — восстановить d — требует знания $\varphi(n) = (p-1)(q-1)$. А оно эквивалентно разложению $n = pq$. Без разложения вычислить d по открытому ключу предположительно требует экспоненциального времени, а при модуле n размером 2048+ бит задача неосуществима на практике.

Сообщение m обязано быть меньше модуля n : остаток $m^e \pmod{n}$ лежит в пределах от 0 до $n-1$. Восстановить по нему число, большее n , нельзя. Поэтому реальные системы режут сообщение на блоки по длине модуля — при 2048-битном n это блоки по 2048 бит — и шифруют каждый блок отдельно.

Квантовый алгоритм Шора факторизует n за полиномиальное время — именно поэтому развивается постквантовая криптография (см. главу 30).

Примечание Криптография на решётках

Кандидаты на замену RSA и Диффи–Хеллмана, стандартизованные NIST в 2024 году, строятся на решётках — сетках точек в $\mathbb{R}^n$, порождённых целочисленными комбинациями базисных векторов. Стойкость схем ML-KEM и ML-DSA опирается на трудность поиска кратчайшего вектора решётки и связанной задачи обучения с ошибками (*learning with errors*). Ни факторизация, ни дискретный логарифм в этих задачах не участвуют, поэтому алгоритм Шора к ним неприменим. Слово «решётка» здесь означает другую структуру, чем решётки порядка из главы 8, — это два разных объекта с одним именем.

Но даже при стойкой факторизации «голый» RSA опасно использовать — и сейчас мы увидим почему.

19.7.1 Подпись RSA

Та же ключевая пара годится и для *подписи*: то же возведение в степень, но закрытым ключом.

$$s = m^d \bmod n.$$

Проверка — возведение открытым ключом: подпись принимается, если $s^e \equiv m \pmod{n}$. Подписать сообщение может только владелец d , а проверить — любой, у кого есть открытый ключ (n, e) . Секретность и аутентичность — две стороны одной математики: подделать подпись без d так же трудно, как разложить n .

19.7.2 Маллобильность RSA

RSA мультипликативен: произведение шифротекстов — шифротекст произведения сообщений. Это свойство называется *маллобильностью* (malleability) — и это атака, потому что злоумышленник может управлять значениями чужих сообщений.

Пример Маллобильность

Перехватив два шифротекста $c_1 = m_1^e, c_2 = m_2^e$, злоумышленник вычисляет

$$c_1 c_2 \equiv (m_1 m_2)^e \pmod{n}$$

— шифротекст сообщения $m_1 m_2$ — не зная ключа. Такой злоумышленник не читает сообщения, но может подменять их значения. Например, если m_1 — сумма перевода, умножение шифротекста на r^e незаметно меняет её на $m_1 r$. Поэтому «голый» RSA не используют как шифр: стандарты добавляют случайность (padding OAEP), разрушающую мультипликативную структуру.

Примечание Подпись тоже мультипликативна

Подписи RSA мультипликативны так же, как шифротексты: из подписей s_1 на m_1 и s_2 на m_2 перемножением получается корректная подпись на сообщении $m_1 m_2$. Проверка $(s_1 s_2)^e \equiv m_1 m_2 \pmod{n}$ подтверждает корректность подделки, хотя владелец ключа сообщение $m_1 m_2$ не подписывал. Стандартизованная схема RSA-PSS разрушает структуру: подписывается хеш сообщения с паддингом, зависящим от случайных данных.

19.7.3 Атака с общим модулем

Иногда один модуль n используется несколькими пользователями с разными показателями e — по ошибке, экономя на генерации ключей — и это фатально.

Пример Общий модуль

Пусть два пользователя используют один модуль n с разными показателями e_1 и e_2 , причём эти показатели взаимно просты. Злоумышленник видит $c_1 = m^{e_1}$, $c_2 = m^{e_2}$. Расширенный алгоритм Евклида даёт целые u, v с $e_1 u + e_2 v = 1$. Тогда

$$c_1^u c_2^v \equiv m^{e_1 u + e_2 v} = m \pmod{n}$$

— сообщение раскрыто без знания закрытых ключей. Если коэффициент u или v отрицателен, соответствующая степень вычисляется через обратный элемент: $c^{-k} = (c^{-1})^k \pmod{n}$. А он существует, только когда $\gcd(c, n) = 1$. Иначе атака так не проходит. У каждого пользователя должен быть свой модуль n : общий модуль — классическая ошибка распределения ключей.

Замечание Другие атаки на RSA

- **Винер (1990):** малый закрытый показатель $d < \frac{n^{\frac{1}{4}}}{3}$ восстанавливается из открытой пары (n, e) непрерывными дробями (*continued fractions*).
- **Тайминг-атака:** время дешифрования зависит от битов d . Измерения времени многих дешифрований выдают секрет.
- **Квантовый алгоритм Шора:** факторизует n за полиномиальное время.

Защита — большие ключи, аккуратные показатели и стандартизованные схемы с padding.

Маллобильность и тайминг-атаки реально ломали системы в истории криптографии. Поэтому правило простое: использовать стандарт. «Голый» RSA для этого не подходит.

Проверка RSA и атаки на него

1. Почему после генерации ключей простые p и q можно уничтожить и что при этом сохраняет владелец?
2. Почему открытый показатель выбирают малым, а закрытый обязан быть большим?
3. Как злоумышленник меняет значение зашифрованного перевода, не зная ключа, и что его останавливает в реальной системе?

§ 19.8 Протокол Диффи–Хеллмана

RSA опирается на (предположительную) трудность факторизации. Но в криптографии есть и другая односторонняя функция: возведение в степень в конечной циклической группе.

Ненулевые остатки $1, \dots, p-1$ образуют по умножению группу $\mathbb{Z}_p^*$. (Группа — множество с операцией: умножение остатков замкнуто, у каждого элемента есть обратный.) Группа как структура с операцией и обратными элементами определена в главе 12, а лемма Бёрнсайда применяет группы к перечислению в главе 18.

ОПРЕДЕЛЕНИЕ 19.7 Образующий

Образующий — такой элемент g , что его степени $g^0, g^1, \dots, g^{p-2}$ пробегает все ненулевые остатки.

Пример Образующий $\mathbb{Z}_7^*$

Элемент 3 — образующий: его степени по модулю 7 равны

$$3^0 = 1, 3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5$$

и пробегает все ненулевые остатки $1, \dots, 6$.

Поэтому для любого $h \neq 0$ найдётся показатель x с $h = g^x$. На этом держится корректность определения ниже.

ОПРЕДЕЛЕНИЕ 19.8 Дискретный логарифм

Даны простое число p , образующий g группы $\mathbb{Z}_p^*$ и элемент $h = g^x \bmod p$. Задача **дискретного логарифмирования** (DLOG) — найти x .

Прямой шаг — возвести g в степень x — выполняется методом квадратов за $O(\log x)$ умножений по модулю. Обратный шаг — по h восстановить x — для больших p (≥ 2048 бит) предположительно труден. Как и факторизация, он не имеет известного полиномиального алгоритма на классическом компьютере. Квантовый алгоритм Шора (тот же, что факторизует n) решает и эту задачу за полиномиальное время.

Протокол Диффи–Хеллмана (Diffie, Hellman, 1976) использует DLOG для установления общего секретного ключа по открытому каналу связи. Название «обмен ключами» обманчиво: по каналу не передаётся ни одного ключа. Алиса и Боб создают секрет порознь: каждый берёт общий открытый материал, добавляет свой секретный ингредиент — и получает то же число, что и собеседник. Ключ нигде не путешествует, поэтому перехватить его в дороге нечего.

1. Алиса выбирает случайное a и вычисляет $A = g^a \bmod p$. Отправляет A Бобу.
2. Боб выбирает случайное b и вычисляет $B = g^b \bmod p$. Отправляет B Алисе.
3. Алиса вычисляет $s = B^a = g^{ba} \bmod p$.
4. Боб вычисляет $s = A^b = g^{ab} \bmod p$.

Оба получили одно и то же число $s = g^{ab} \bmod p$ — общий секрет. Перехватчик видит p, g, A, B , но чтобы найти s , ему нужно решить DLOG. Достаточно найти $a = \log_g A$ или $b = \log_g B$.

Замечание Протокол на красках

Всю схему можно представить на кухне: общий жёлтый g , у Алисы красная a , у Боба — синяя b . Смешать две краски легко, а разделить смесь на исходные — практически невозможно, и именно эта асимметрия делает протокол безопасным. Порядок смешивания не важен: жёлтый с красным, а потом с синим — тот же цвет, что жёлтый с синим, а потом с красным, — поэтому $g^{ab} = g^{ba}$. Перехватчик видит жёлтый и обе смеси, но разделить их на исходные краски он не умеет.

Пример ДН с малыми числами

Возьмём $p = 23$, $g = 5$. Число 5 — образующий $\mathbb{Z}_{23}^*$. Алиса выбирает $a = 6$. Боб выбирает $b = 15$.

$$A = 5^6 \bmod 23 = 8, \quad B = 5^{15} \bmod 23 = 19.$$

Общий секрет:

$$s = 19^6 \bmod 23 = 2 = 8^{15} \bmod 23.$$

Перехватчик знает ($p = 23$, $g = 5$, $A = 8$, $B = 19$). Ему нужно решить $5^a \equiv 8 \pmod{23}$ или $5^b \equiv 19 \pmod{23}$, чтобы найти $s = 2$.

Казалось бы, протокол решает задачу: общий секрет получен, никто его не перехватил. Но в протоколе есть дыра, и её причина — в самом протоколе. Математика здесь ни при чём.

19.8.1 Атака «человек посередине»

ОПРЕДЕЛЕНИЕ 19.9 Атака «человек посередине»

«Человек посередине» (man-in-the-middle, MITM) — атака, при которой злоумышленник вклинивается между двумя участниками и ведёт два отдельных протокола: один с каждым. Оба участника думают, что общаются напрямую, а каждый общается со злоумышленником.

Пример MITM против Диффи–Хеллмана

Голый протокол ДН не аутентифицирует собеседника — Алиса не знает, что A дошёл именно до Боба — и этим пользуется Ева — сцена атаки на рис. 90.

1. Алиса отправляет Бобу $A = g^a$.
2. Ева перехватывает и от имени Алисы отправляет Бобу своё $A' = g^x$.
3. Боб отправляет Алисе $B = g^b$.
4. Ева перехватывает и от имени Боба отправляет Алисе своё $B' = g^y$.
5. Алиса и Ева получают общий секрет $s_1 = g^{ay}$.
6. Боб и Ева — секрет $s_2 = g^{xb}$.

7. Ева расшифровывает сообщение Алисы секретом s_1 , читает его и зашифровывает для Боба секретом s_2 .

Оба участника не подозревают о подмене и считают канал защищённым. Защита — *аутентификация*: подписи, сертификаты, заранее согласованные публичные ключи.



два секрета: с Алисой и с Бобом

Рис. 90. Атака «человек посередине» на Диффи–Хеллмана.

MITM показывает принципиальное различие между «сломать математику» и «сломать протокол». Математика DH безупречна. Уязвим протокол, не проверяющий собеседника. Поэтому в реальных системах DH всегда работает вместе с аутентификацией.

Следующая атака — уже на математику. Протокол здесь ни при чём.

19.8.2 Атака Полига–Хеллмана

Атака Полига–Хеллмана уменьшает группу: она сводит дискретный логарифм к логарифмам в подгруппах малых порядков.

АЛГОРИТМ Атака Полига–Хеллмана

Пусть порядок группы $n = p - 1$ раскладывается на простые степени:

$$n = q_1^{a_1} \cdots q_k^{a_k}.$$

Для каждого простого делителя q^a порядка:

1. вычисляем $g' = g^{\frac{n}{q^a}}$ и $h' = h^{\frac{n}{q^a}}$, проецируя задачу на подгруппу порядка q^a .
2. элемент g' имеет порядок ровно q^a , и из $h' = (g')^x$ перебор по элементам подгруппы даёт остаток $x \bmod q^a$.

Остатки, собранные по всем простым делителям $q_i^{a_i}$, дают полный ответ x по китайской теореме об остатках.

Когда все q_i малы (порядок гладкий), полное время — полиномиальное, и DLOG падает.

Пример Полига–Хеллмана на $p = 29$

Берём $p = 29$, $g = 2$ — образующий, его порядок 28. Ищем x из $2^x \equiv 14 \pmod{29}$. Настоящий ответ — 13. Порядок $28 = 2^2 \cdot 7$ — гладкий, поэтому разбираем его по частям.

Остаток по модулю 4. Проецируем на подгруппу порядка 4: $g_4 = 2^{\frac{28}{4}} = 2^7 = 12$, $h_4 = 14^7 = 12$. Логарифм 12 по основанию 12 равен 1: $x \equiv 1 \pmod{4}$.

Остаток по модулю 7. Проецируем на подгруппу порядка 7: $g_7 = 2^{\frac{28}{7}} = 2^4 = 16$, $h_7 = 14^4 = 20$. Перебираем степени 16: $16^0 = 1$, $16^1 = 16$, $16^2 = 24$, $16^3 = 7$, $16^4 = 25$, $16^5 = 23$, $16^6 = 20$ — совпадение на шестой степени. Значит, $x \equiv 6 \pmod{7}$.

Сборка по CRT. Ищем $x \equiv 1 \pmod{4}$, $x \equiv 6 \pmod{7}$. Числа, дающие 1 по модулю 4, — 1, 5, 9, 13, Первое, дающее 6 по модулю 7, — это 13. Итак, $x = 13$. Вместо перебора всех 28 показателей хватило нескольких проверок в маленьких подгруппах — так гладкий порядок убивает DLOG.

Замечание Почему порядок группы решает

Безопасность DLOG зависит не только от размера p , но и от структуры порядка $p - 1$. Стандартный выбор — простое p , для которого $p - 1 = 2q$ с большим простым q (безопасное простое). Тогда подгруппы Полига–Хеллмана тривиальны. У таких p атака Полига–Хеллмана не даёт выигрыша, и единственный путь — решать DLOG в полной группе.

§ 19.9 Эллиптические кривые и криптография

Группа $\mathbb{Z}_p^*$ — не единственная группа, в которой DLOG предположительно труден. Эллиптические кривые над конечными полями дают альтернативную группу — и при той же стойкости требуют *меньших* размеров ключа.

ОПРЕДЕЛЕНИЕ 19.10 Эллиптическая кривая над $\text{GF}(p)$

Для простого $p > 3$ **эллиптическая кривая** задаётся уравнением

$$E : y^2 = x^3 + ax + b, \quad (a, b \in \text{GF}(p)),$$

где дискриминант

$$\Delta = 4a^3 + 27b^2 \neq 0 \pmod{p}$$

— условие отсутствия кратных корней.

Точки кривой — все пары $(x, y) \in \text{GF}(p)^2$, удовлетворяющие уравнению. К ним добавляется специальная точка $\mathcal{O}$ («точка на бесконечности»). Множество точек образует абелеву группу относительно операции сложения, где $\mathcal{O}$ служит нейтральным элементом.

Сложение точек определяется геометрически. Через две точки P и Q на кривой проводим прямую (если $P = Q$ — касательную). Она пересекает кривую в третьей

точке R . Отражаем R относительно оси x и получаем $P + Q$. За геометрией стоят алгебраические формулы.

Для $P \neq Q$ и $x_P \neq x_Q$:

$$\begin{aligned}\lambda &= \frac{y_Q - y_P}{x_Q - x_P} \bmod p, \\ x_R &= \lambda^2 - x_P - x_Q, \\ y_R &= \lambda(x_P - x_R) - y_P.\end{aligned}$$

Для удвоения $P = Q$:

$$\begin{aligned}\lambda &= \frac{3x_P^2 + a}{2y_P} \bmod p, \\ x_R &= \lambda^2 - 2x_P, \\ y_R &= \lambda(x_P - x_R) - y_P.\end{aligned}$$

Пример Кривая над GF(17)

Рассмотрим кривую $E : y^2 = x^3 + 2x + 3$ над GF(17). Дискриминант: $4 \cdot 2^3 + 27 \cdot 3^2 = 32 + 243 = 275 \equiv 3 \pmod{17}$, не ноль — кривая неособая.

Точки найдём перебором. Для $x = 2$ правая часть $2^3 + 2 \cdot 2 + 3 = 15 \equiv 7^2 \equiv 10^2 \pmod{17}$ — квадрат. Поэтому получаем две точки $(2, 7), (2, 10)$. Продолжая, собираем ординаты для всех x , где правая часть — квадрат по модулю 17:

x	2	3	5	8	9	11	12	13	14	15	16
y	7	6	6	2	6	8	2	4	2	5	0
$-y$	10	11	11	15	11	9	15	13	15	12	—

Таблица 11. Точки кривой $y^2 = x^3 + 2x + 3$ над GF(17).

Для остальных x правая часть квадратом не является, и точек там нет. Точки идут парами $(x, y), (x, -y)$ — зеркалами относительно оси x . При $y = 0$ пара вырождается в одну точку. Всего $10 \cdot 2 + 1 = 21$ аффинная точка. Вместе с точкой на бесконечности группа имеет порядок 22.

Проверим групповой закон: сложим $P = (2, 7), Q = (3, 6)$.

$$\begin{aligned}\lambda &= \frac{6 - 7}{3 - 2} = -1 \equiv 16 \pmod{17}, \\ x_R &= \lambda^2 - x_P - x_Q = 16^2 - 2 - 3 = 251 \equiv 13 \pmod{17}, \\ y_R &= \lambda(x_P - x_R) - y_P = 16 \cdot (2 - 13) - 7 = -183 \equiv 4 \pmod{17}.\end{aligned}$$

Итак, $P + Q = (13, 4)$ — снова точка из таблицы: сумма не вышла за пределы кривой.

Задача дискретного логарифмирования на эллиптической кривой (ECDLP): даны точки P и $Q = kP$ (скалярное умножение k раз), найти k . Для правильно выбранной кривой эта задача предположительно трудна — и при той же стойкости, что и 3072-битное RSA, достаточно 256-битной эллиптической кривой.

Протокол Диффи–Хеллмана на эллиптической кривой (ECDH):

1. Алиса и Боб договариваются о кривой E над полем $\text{GF}(p)$ и базовой точке G большого простого порядка.
2. Алиса выбирает a , вычисляет $A = aG$ (скалярное умножение), отправляет A Бобу.
3. Боб выбирает b , вычисляет $B = bG$, отправляет B Алисе.
4. Общий секрет: $aB = abG = bA$.

Безопасность ECDH опирается на ECDLP: найти a по G и aG вычислительно трудно. Преимущество перед классическим DH: ключи короче (256 бит ECC эквивалентны 3072 битам RSA/DH), операции быстрее, расход энергии меньше. Поэтому ECC используется в мобильных устройствах, блокчейнах (Bitcoin использует кривую secp256k1) и протоколах с ограниченными ресурсами.

Как и в классическом DH, безопасность ECDH зависит не только от размера поля, но и от порядка группы точек.

Пример Выбор кривой

По теореме Хассе порядок группы точек кривой над $\text{GF}(p)$ близок к p :

$$|n - (p + 1)| \leq 2\sqrt{p}.$$

Если n гладкое, Полига–Хеллмана побеждает и здесь: ECDLP сводится к подгруппам малых порядков, как DLOG в $\mathbb{Z}_p^*$. Поэтому используют кривые с простым порядком или с малым кофактором: у secp256k1 — кривой Bitcoin — порядок — большое простое число, и атака Полига–Хеллмана неприменима.

Замечание Атака неверной кривой

Злоумышленник может подсунуть точку Q , лежащую на другой, более слабой кривой. Если протокол не проверяет принадлежность точки кривой, секрет утекает по кусочкам. Защита — явная проверка, что Q лежит на согласованной кривой.

Эллиптические кривые повторяют историю DH: математика стойка, но реализация обязана проверять параметры. Правило, вынесенное из всех атак этой главы, одно: секретность живёт в математике и в строгости её применения — а не в надежде, что противник не догадается.

§ 19.10 Итоги главы

Делимость и остатки — древняя арифметика, но именно на ней стоит современная криптография: алгоритм Евклида находит наибольший общий делитель, расширенный алгоритм — обратные элементы, а китайская теорема об остатках собирает решения систем сравнений. Малая теорема Ферма и её обобщение — теорема Эйлера — задают периодичность степеней по модулю, и именно эта периодичность делает возможным RSA. Простые множители для ключей RSA находит вероятностный тест Миллера–Рабина, построенный на той же малой теореме Ферма. Всё это — вековые факты, которые сегодня работают в каждом защищённом соединении.

Три криптосистемы с открытым ключом — RSA, Диффи–Хеллмана и эллиптические кривые — построены на одной предпосылке: существует вычисление, лёгкое в одну сторону и трудное в обратную. Умножить два простых числа может каждый, а разложить их произведение — практически невозможно. Возвести число в степень по модулю просто, а восстановить показатель (дискретный логарифм) — на практике нет. Секретность в открытом ключе остаётся в одних руках, и хранить общий секрет заранее не нужно.

Каждую конструкцию мы проверили атакой, и каждая атака уточнила требования: подписи против маллобильности, аутентификация против человека посередине, большой простой порядок группы против Полига–Хеллмана, проверка принадлежности точки — против атаки неверной кривой. Принцип Керкгоффса остаётся рамкой всей главы: секретность живёт в ключе при открытом алгоритме, и безопасность проверяется атакой, а не верой в недогадливость противника.

Алгебраический факт превращается в гарантию секретности, пока остаются верными предположения о трудности факторизации и дискретного логарифма, — а их никто не доказал. Проверка этих предположений — вопрос теории сложности (глава 30), вероятностные методы (глава 20) придадут формулам главы новое применение. Квантовый алгоритм Шора факторизует n за полиномиальное время, и поэтому он ломает RSA: это риск для всей криптографии с открытым ключом, а ответом служат большие ключи и постквантовые схемы. Алгоритм Гровера сокращает перебор ключа с 2^k до $2^{\frac{k}{2}}$ шагов — поэтому AES теряет половину длины ключа, а не весь ключ¹⁵³.

Проверка Проверьте себя

1. Почему секретность должна жить в ключе, и что утверждает принцип Керкгоффса?
2. Как алгоритм Евклида находит gcd без факторизации, и что дают коэффициенты Безу?
3. Почему обратный элемент по модулю существует ровно для взаимно простых чисел?

¹⁵³L. K. Grover. «A fast quantum mechanical algorithm for database search». STOC, 1996.

4. Как малая теорема Ферма переходит в теорему Эйлера при переходе от простого модуля к составному?
5. Почему дешифрование RSA возвращает исходное сообщение, и где используется $\varphi(n)$?
6. Чем маллобильность и атака с общим модулем опасны для «голого» RSA?
7. Как Ева перехватывает обмен Диффи–Хеллмана, и почему аутентификация закрывает дыру?
8. Как атака Полига–Хеллмана сводит DLOG к подгруппам малых порядков?
9. Почему порядок группы точек кривой должен быть большим простым числом?
10. Как тесты простоты дают владельцу ключа RSA преимущество перед взломщиком, и почему тест Миллера–Рабина не обманывается числами Кармайкла?

Что дальше?

Криптография с открытым ключом держится на предположительной трудности факторизации и дискретного логарифма — но никто не доказал, что они действительно трудны. Вопрос «можно ли быстро разложить число» — часть более общего вопроса о сложности вычислений, которым занимается глава 30. А прежде чем уйти в теорию сложности, применим формулы этой главы к случайным исходам: дискретная вероятность (глава 20) использует делимость, сравнения и те же одно-сторонние функции в анализе алгоритмов.

§ 19.11 Упражнения

Делимость и алгоритм Евклида.

1. Найдите $\gcd(2024, 1024)$ с помощью алгоритма Евклида. Выпишите все промежуточные шаги.
2. Найдите $\gcd(70, 99)$, $\gcd(68, 100)$. Какие из этих пар взаимно простые?
3. * Пусть F_n — n -е число Фибоначчи, где $F_0 = 0$, $F_1 = 1$. Сколько шагов делает алгоритм Евклида на паре (F_{n+1}, F_n) и почему это наихудший случай?

Модулярная арифметика.

4. Найдите обратные элементы по модулю: $7^{-1} \bmod 26$, $3^{-1} \bmod 11$, $10^{-1} \bmod 17$.
5. Решите систему сравнений:

$$\begin{cases} x \equiv 1 \pmod{4} \\ x \equiv 2 \pmod{5} \\ x \equiv 3 \pmod{7} \end{cases}$$

6. * Докажите, что a имеет обратный элемент по модулю m тогда и только тогда, когда $\gcd(a, m) = 1$.

RSA и атаки.

7. В RSA с $p = 61, q = 53$ вычислите n и $\varphi(n)$. Выберите $e = 17$. Закрытая экспонента: $d = e^{-1} \bmod \varphi(n)$. Зашифруйте сообщение $m = 65$ и расшифруйте результат.
8. Объясните, почему атака с общим модулем требует $\gcd(e_1, e_2) = 1$, и что произойдёт, если у показателей есть общий делитель.
9. * Опишите, как злоумышленник использует маллобильность RSA, чтобы подменить банковский перевод m на $m' = m \cdot r$ (без знания закрытого ключа). Что защищает от этого в реальных системах?

Протокол Диффи–Хеллмана и атаки.

10. При $p = 23, g = 5$ Алиса выбрала $a = 4$, Боб — $b = 9$. Вычислите A, B и общий секрет s .
11. Опишите атаку «человек посередине» на протокол DH: какие значения видит Ева, какие секреты она получает и почему Алиса и Боб не замечают подмены. Какая мера защиты закрывает дыру?
12. * Вычислите дискретный логарифм x из $2^x \equiv 14 \pmod{29}$ атакой Полига–Хеллмана. Выпишите все шаги: подгруппы порядка 4 и 7, сборка по CRT.
13. * Почему для Диффи–Хеллмана выбирают безопасное простое p с $p - 1 = 2q$? Что даёт атака Полига–Хеллмана, если $p - 1$ имеет много малых простых делителей?

Эллиптические кривые.

14. На кривой $y^2 = x^3 + x + 1$ над $\text{GF}(5)$ выпишите формулу удвоения. Проверьте, что $2(0, 1) = (4, 2)$.
15. Объясните, почему порядок группы точек эллиптической кривой должен быть большим простым числом, и как атака Полига–Хеллмана ломает ECDH при гладком порядке.
16. * Алгоритм Шора решает и факторизацию, и дискретный логарифм за полиномиальное время. Почему криптография с открытым ключом уязвима к квантовому компьютеру, а симметричное шифрование (типа AES) — лишь теряет половину длины ключа?
17. * Число $341 = 11 \cdot 31$ составное. Тем не менее $2^{340} \equiv 1 \pmod{341}$. Докажите это, сведя проверку к модулям 11 и 31 с помощью китайской теоремы об остатках и малой теоремы Ферма. Объясните, почему отсюда следует, что проверка малой теоремой Ферма не доказывает простоту числа.

ПРОЕКТ Измерение трудоёмкости факторизации и дискретного логарифма

Измерьте асимметрию односторонней функции: проверить, что 91 составное, может каждый, а найти его делители 7 и 13 — уже перебор. Чем длиннее число, тем сильнее расходятся эти две операции, и вся стойкость держится на том, что факторизация и дискретный логарифм трудны. Но насколько трудны? Определите собственным кодом тот размер ключа, на котором методы факторизации перестают укладываться в секунды, — и тем самым измерьте саму асимметрию.

① Ядро длинной арифметики

Соберите ядро длинной арифметики: быстрое возведение в степень, расширенный алгоритм Евклида, проверку простоты Миллером–Рабином. Это быстрая сторона односторонней функции.

② Методы факторизации

Реализуйте методы факторизации: пробное деление, метод Ферма ($n = a^2 - b^2$), ρ -метод Полларда. Прогоните их по модулям растущего размера и запишите кривую времени.

③ Методы дискретного логарифмирования

Реализуйте перебор, baby-step giant-step с $\sqrt{p}$ шагами и атаку Полига–Хеллмана. Продумайте, почему на гладком $p - 1$ атака завершается быстро, и как время меняется на безопасном простом $p = 2q + 1$.

④ Измерение времени

Генерируйте ключи растущего размера, запускайте наилучший метод и стройте кривую времени в логарифмической шкале. Экстраполируйте её к большим размерам ключа и продумайте, где обратное преобразование перестаёт быть практичным.

⑤ Уязвимости «голого» RSA

Подмените сообщение маллобильностью — умножением шифротекста на r^e , — а затем восстановите сообщение при общем модуле расширенным алгоритмом Евклида. Объясните, почему стойкой математики мало и нужна аккуратная конструкция.

Итог: измеренная зависимость «размер ключа — время факторизации» с точкой перелома, где обратное преобразование перестаёт быть практичным, и демонстрация атак на «голый» RSA — подмены сообщения и атаки с общим модулем.

ХХ

Дискретная вероятность

Случайность кажется противоположностью знания. Интуиция отвечает на вопрос о случайном процессе коротко: ничего определённого. Бросок монеты непредсказуем, следующий бросок — тоже, и никакой закономерности тут не выловить.

Парадокс дней рождения говорит об обратном. 23 человека в комнате — и вероятность совпадения дней рождения превышает половину. Интуиция ожидала 183. Систематическая ошибка интуиции выдаёт главное: в случайности *есть* структура — нужно только правильное средство измерения.

Случайность нужна и криптографии. В предыдущей главе 19 ключи выбирались наугад, а простые числа — из случайной выборки. Стойкость RSA держалась на предположении, что противник не сумеет найти секретный показатель. Там случайностью пользовались, но не измеряли её. А стойкость шифра — это вопрос о вероятности: насколько мал шанс, что ключ окажется слабым или что атака удастся. Вероятность — мера неопределённости, и без неё не сформулировать даже, что значит «шифр защищён».

Дискретная вероятность и есть эта мера. Она не отменяет случайность — она даёт язык, на котором случайность можно измерять и сравнивать. Вероятностная модель собирается из пространства исходов, меры на событиях, условных вероятностей и независимости. Этот язык один для всех: броска монеты, колоды карт, лотереи, медицинского теста и рандомизированного алгоритма.

Подсчитать количество комбинаций — полдела. Второй вопрос — насколько та или иная комбинация *вероятна*. При броске двух кубиков сумма 7 выпадает в шесть раз чаще суммы 2 — хотя обе считаются в одном и том же пространстве исходов. Комбинаторика (глава 18) дала инструменты для подсчёта. История теории вероятностей началась в 1654 году с переписки Паскаля и Ферма о разделе ставок в прерванной игре.

Вероятность измеряет событие числом от 0 до 1, и для конечного числа равновероятных исходов всё начинается с простого принципа: она равна доле благоприятных. Но простота обманчива: вся работа прячется в выбор пространства исходов и в подсчёт благоприятных.

Рандомизированный алгоритм делает случайный выбор и отвечает «с вероятностью», а не «всегда». Так устроены быстрая сортировка со случайным опорным элементом, проверка умножения матриц Фривалда и оценки методом Монте-Карло. Рандомизация не отменяет худший случай — она делает его статистически

пренебрежимым. Без вероятностного языка такие алгоритмы нельзя ни описать, ни проанализировать: утверждение о времени работы превращается в утверждение о вероятности.

Вероятность — точная форма знания: исход неизвестен, но мера исхода известна. Именно поэтому одни и те же формулы описывают и спор о разделе ставок XVII века, и стойкость современного шифра. Как из незнания исхода вырастает точное знание о мере исхода?

ИСТОРИЯ Аксиоматика Колмогорова

Сто лет после Лапласа теория вероятностей работала без строгих оснований. Само понятие «равновозможных исходов» содержало порочный круг: равно-возможными назывались исходы, имеющие равные вероятности. Парадокс Бертрана (1889) показал, что наивный подход даёт разные ответы на один и тот же вопрос в зависимости от выбора пространства исходов. В 1933 году Андрей Колмогоров поставил теорию на аксиоматическую основу: вероятностная мера — частный случай меры, а аддитивность — аксиома. Определение вероятностного пространства из этой главы — прямое следствие этой аксиоматики. Полную формулировку аксиом см. в звёздном разделе «Аксиомы Колмогорова» в конце главы.

ОБЗОР ГЛАВЫ

В этой главе мы соберём вероятностное пространство для дискретного эксперимента — исходы, события, меру — и научимся вычислять вероятности комбинаторным подсчётом. Условная вероятность и независимость покажут, как новая информация сужает пространство исходов, а формула Байеса объяснит, как пересчитать априорную оценку в апостериорную. Случайные величины с матожиданием и дисперсией дадут числа, которыми описывается исход. Стандартные распределения — Бернулли, биномиальное и геометрическое — свяжут элементарное испытание с его классическими вопросами, а линейность матожидания станет рабочим приёмом анализа алгоритмов. Неравенства концентрации — Маркова, Чебышёва, Чернова — скажут, как часто случайная величина уходит далеко от среднего. Центральная предельная теорема покажет, какую устойчивую форму приобретает распределение суммы с ростом числа слагаемых. Парадокс дней рождения объяснит, почему один и тот же подсчёт предсказывает совпадения в комнате и ускоряет поиск коллизий хеш-функций, а вероятностный метод покажет, как из положительной вероятности следует существование. В конце язык главы выйдет за рамки подсчёта: случайные графы, марковские цепи и энтропия опишут структуры, которые вероятность порождает и измеряет.

После этой главы вы сможете:

1. строить вероятностное пространство для дискретного эксперимента и вычислять вероятности комбинаторным подсчётом;
2. применять условную вероятность, независимость и формулу Байеса;
3. работать со случайными величинами, матожиданием и дисперсией;
4. пользоваться неравенствами Маркова, Чебышёва и Чернова;
5. формулировать центральную предельную теорему и оценивать по ней хвосты сумм независимых слагаемых;
6. объяснять парадокс дней рождения и применять вероятностный метод.

§ 20.1 Вероятностное пространство

Формализация начинается с трёх объектов: множество исходов, множество событий и функция, приписывающая событиям числа.

ОПРЕДЕЛЕНИЕ 20.1 Дискретное вероятностное пространство

Дискретным вероятностным пространством называется пара (Ω, P) , где:

- Ω — конечное или счётное множество **исходов** (sample space).
- $P : \mathcal{P}(\Omega) \rightarrow [0, 1]$ — **вероятностная мера**, удовлетворяющая условиям:
 1. $P(\emptyset) = 0$ и $P(\Omega) = 1$.
 2. для несовместных событий $A \cap B = \emptyset$ выполняется **конечная аддитивность**:

$$P(A \cup B) = P(A) + P(B).$$

В этой главе мы работаем с конечными пространствами, где конечной аддитивности достаточно. Для счётного Ω потребовалась бы ещё и счётная аддитивность.

Для любого исхода его вероятность неотрицательна, и вероятности всех исходов в сумме дают единицу:

$$P(\{\omega\}) \geq 0, \sum_{\omega \in \Omega} P(\{\omega\}) = 1.$$

ОПРЕДЕЛЕНИЕ 20.2 Событие

Событием называется любое подмножество $A \subseteq \Omega$.

Пример Бросание игральной кости

При бросании игральной кости $\Omega = \{1, 2, 3, 4, 5, 6\}$. Событие «выпало чётное число» — подмножество $\{2, 4, 6\}$.

Вероятность события равна сумме вероятностей составляющих его исходов:

$$P(A) = \sum_{\omega \in A} P(\{\omega\}).$$

В случае конечного Ω с *равновероятными* исходами ($P(\{\omega\}) = \frac{1}{|\Omega|}$ для всех ω):

$$P(A) = \frac{|A|}{|\Omega|}.$$

Это формула Лапласа (1812) — простейший частный случай. Вся комбинаторика обслуживает именно эту формулу: подсчёт величин $|A|$, $|\Omega|$.



«Доля вероятности» — плеоназм: вероятность и есть доля.

Примечание Четырёхшаговый метод

Большинство задач дискретной вероятности решаются по одной схеме — четыре шага, которые полезно держать в голове явно:

1. **Шаг 1.** Определите пространство исходов Ω . Что считается элементарным исходом?
2. **Шаг 2.** Определите событие $A \subseteq \Omega$, вероятность которого нужно найти.
3. **Шаг 3.** Назначьте вероятности исходам. Равновероятны ли они? Если нет — какие вероятности у каждого?
4. **Шаг 4.** Вычислите

$$P(A) = \sum_{\omega \in A} P(\{\omega\}),$$

в равновероятном случае $P(A) = \frac{|A|}{|\Omega|}$.

Шаг 1 — самый частый источник ошибок: неверное пространство исходов обесценивает все остальные шаги.

Примечание Равновероятность — свойство модели, а не задачи

Утверждение «все исходы равновероятны» — это *допущение*, которое мы делаем при построении вероятностной модели, а не факт о мире.

Бросок двух кубиков: исходы (i, j) равновероятны. Вероятность каждого исхода — $\frac{1}{36}$. Тогда вероятность суммы 7 равна $\frac{1}{6}$. Если бы мы ошибочно считали равновероятными суммы $\{2, 3, \dots, 12\}$, вероятность каждой была бы $\frac{1}{11}$.

Выбор пространства исходов определяет ответ. Поэтому проверяем его раньше, чем начинаем считать.

Пример Бросок двух кубиков

Пространство исходов: $\Omega = \{(i, j) \mid 1 \leq i, j \leq 6\}$. Его размер: $|\Omega| = 36$.

Событие A = «сумма 7». Множество исходов: $A = \{(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)\}$. Число исходов: $|A| = 6$. $P(A) = \frac{6}{36} = \frac{1}{6}$.

Событие B = «сумма 2». Множество исходов: $B = \{(1, 1)\}$. Число исходов: $|B| = 1$. $P(B) = \frac{1}{36}$.

Событие C = «сумма 7 или 2». Вероятность: $P(C) = P(A) + P(B) = \frac{7}{36}$ (аддитивность).

Пример Колода карт

Ω = все 5-карточные комбинации из 52 карт. Число комбинаций: $|\Omega| = \binom{52}{5} = 2598960$.

Событие F = «флеш-рояль» (5 карт одной масти от 10 до туза). Число таких комбинаций: $|F| = 4$ (по одной на масть). $P(F) = \frac{4}{2598960} \approx 0.0000015$.

Событие P = «пара» (ровно две карты одного достоинства). Комбинаторный подсчёт даёт $|P| = 1098240$. $P(P) = \frac{1098240}{2598960} \approx 0.423$.

Там, где числа становятся по-настоящему большими, формула Лапласа остужает азарт. В лотерее «Русское лото» из 90 шаров вытаскивают 15, и число возможных наборов — $\binom{90}{15} \approx 4.6 \cdot 10^{16}$, около 46 квадриллионов. Джекпот выигрывается с вероятностью примерно 1 к 46 квадриллионам. Даже если билет купит каждый из восьми миллиардов жителей планеты, шанс, что выиграет хоть кто-то, — около одной шестимиллионной. Посчитавший эти шансы не станет в очередь за билетом.

§ 20.2 Условная вероятность и независимость

Новая информация меняет оценку вероятности. Если мы узнали, что событие B произошло, пространство исходов сужается до него. Вероятности пересчитываются относительно суженного пространства.

ОПРЕДЕЛЕНИЕ 20.3 Условная вероятность

Условной вероятностью называется отношение вероятности пересечения события и условия к вероятности условия.

$$P(A | B) = \frac{P(A \cap B)}{P(B)}.$$

Формула определена при $P(B) > 0$.

Условная вероятность — та же вероятность, пересчитанная относительно суженного пространства исходов B . Суженное пространство — это B , и его суммарная вероятность должна снова равняться единице. Раньше вероятность всего суженного

пространства была $P(B)$. Чтобы вернуть ему нормировку, каждую вероятность внутри делят на $P(B)$. Отсюда и формула: $P(A | B) = \frac{P(A \cap B)}{P(B)}$.

ЛОВУШКА Условная вероятность $\neq$ пересечение

$P(A | B)$, $P(A \cap B)$ — разные величины. Условная вероятность нормирует пересечение на вероятность условия: $P(A | B) = \frac{P(A \cap B)}{P(B)}$. Пересечение не превосходит $P(A)$. Условная же вероятность $P(A | B)$ может быть и больше — это служит признаком зависимости. Неверно и равенство $P(A | B) = P(B | A)$: формула Байеса связывает эти величины, но значения у них разные.

Пример Бросок кубика

$\Omega = \{1, \dots, 6\}$. Событие A = «выпала шестёрка». Его вероятность: $P(A) = \frac{1}{6}$. Событие B = «выпало чётное число». Его вероятность: $P(B) = \frac{1}{2}$.

Если мы знаем, что выпало чётное, вероятность шестёрки:

$$P(A | B) = \frac{P(A \cap B)}{P(B)} = \frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{3}.$$

ОПРЕДЕЛЕНИЕ 20.4 Независимость

События A и B называются **независимыми**, если

$$P(A \cap B) = P(A) \cdot P(B).$$

Эквивалентное условие: $P(A | B) = P(A)$. Знание B не меняет оценку A .

Независимость не означает «отсутствие причинной связи» — это чисто математическое свойство вероятностной меры. Два бросания монеты независимы потому, что вероятностное пространство определено так, что исходы разных бросаний перемножаются:

$$\Omega = \Omega_1 \times \Omega_2, \quad P((\omega_1, \omega_2)) = P_1(\omega_1) \cdot P_2(\omega_2).$$

Визуально это — дерево вероятностей, где каждое ребро несёт вероятность соответствующего исхода, а вероятности вдоль пути перемножаются. Для смещённой монеты ($P(H) = 0.6$, $P(T) = 0.4$) дерево выглядит так:

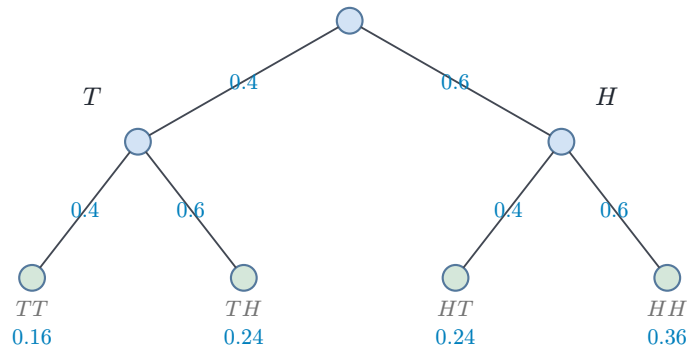


Рис. 91. Дерево вероятностей для двукратного бросания монеты.

Замечание Почему независимость и непересечение — не одно и то же?

Если A и B несовместны ($A \cap B = \emptyset$), то $P(A \cap B) = 0$. Если они оба имеют ненулевую вероятность, то $P(A) \cdot P(B) > 0$, и они не могут быть независимы. Несовместность — свойство структуры событий (как множеств). Независимость — свойство вероятностной меры (как чисел). Одно и то же событие может быть независимым при одной мере и зависимым при другой.

ЛОВУШКА Попарная независимость $\neq$ независимость в совокупности

Из попарной независимости всех пар событий не следует независимость событий в совокупности. Попарная независимость требует $P(A_i \cap A_j) = P(A_i) \cdot P(A_j)$ для каждой пары, совокупная — того же для любого подмножества.

Контрпример — три события при подбрасывании двух монет:

1. A — «первая монета — орёл».
2. B — «вторая монета — орёл».
3. C — «монеты совпали».

Пары независимы: $P(A \cap B) = P(A \cap C) = P(B \cap C) = \frac{1}{4}$. В каждой паре произведение вероятностей тоже даёт $\frac{1}{4}$. Но $P(A \cap B \cap C) = \frac{1}{4} \neq P(A) \cdot P(B) \cdot P(C) = \frac{1}{8}$. Третье событие не добавляет новых исходов сверх пары (A, B) . Пересечение $A \cap B \cap C$ — единственный исход: обе монеты выпали орлом. Его вероятность не совпадает с произведением вероятностей: $\frac{1}{4} \neq \frac{1}{8}$.

Пример Парадокс Монти Холла

Вы на шоу, и перед вами три двери. За одной дверью стоит автомобиль, за двумя другими — козы. Вы выбираете дверь (скажем, дверь 1). Ведущий Монти Холл, зная, что за дверями, *обязан* открыть дверь, за которой коза, и никогда не открывает дверь с автомобилем. Он открывает одну из оставшихся дверей с козой (скажем, дверь 3) — и предлагает: остаётесь при своём выборе или переключаетесь на дверь 2?

Ведущий не выбирает дверь наугад. Он всегда открывает заведомо проигрышную дверь, независимо от того, угадали вы или нет.

Интуитивный ответ: осталось две двери, шансы 50:50 — без разницы. **Вероятностный ответ:** переключение даёт вероятность выигрыша $\frac{2}{3}$.

Пространство исходов — три равновероятных расположения автомобиля (A, B, C) . Без потери общности считаем, что вы выбрали дверь A .

1. Автомобиль за A : ведущий открывает одну из двух других дверей. Переключение — проигрыш.
2. Автомобиль за B : ведущий обязан открыть единственную оставшуюся дверь. Переключение — выигрыш.
3. Автомобиль за C : ведущий обязан открыть единственную оставшуюся дверь. Переключение — выигрыш.

Переключение выигрывает в 2 из 3 случаев. Действие ведущего несёт информацию — и условная вероятность пересчитывается. Полезная картинка — концентрация: изначальная вероятность $\frac{2}{3}$, что автомобиль не за выбранной дверью, после хода ведущего целиком собирается за единственной неоткрытой дверью.

Полезно представить крайний случай: 100 дверей вместо 3. Вы выбираете одну. Ведущий открывает 98 дверей с козами — и предлагает переключиться на единственную оставшуюся. Интуиция немедленно подсказывает: ваш первоначальный выбор имел шанс $\frac{1}{100}$. Оставшаяся дверь — $\frac{99}{100}$. Для трёх дверей логика та же, только числа меньше — и интуиция даёт сбой.

§ 20.3 Формула Байеса

Условная вероятность связывает пару $P(A | B), P(B | A)$. Формула Байеса — способ пересчитать «причинную» вероятность в «диагностическую».

ТЕОРЕМА 20.1 Формула Байеса

$$P(H | E) = \frac{P(E | H) \cdot P(H)}{P(E)}.$$

- $P(H)$ — априорная вероятность гипотезы.
- $P(E | H)$ — правдоподобие (likelihood): вероятность свидетельства при условии гипотезы.
- $P(H | E)$ — апостериорная вероятность гипотезы после получения свидетельства.

Доказательство

Докажем, дважды применив определение условной вероятности.

По определению условной вероятности:

$$P(H | E) = \frac{P(H \cap E)}{P(E)}.$$

Снова по определению:

$$P(E | H) = \frac{P(E \cap H)}{P(H)},$$

откуда $P(H \cap E) = P(E | H) \cdot P(H)$. Подстановка в первое равенство даёт формулу Байеса:

$$P(H | E) = \frac{P(E | H) \cdot P(H)}{P(E)}.$$

УТВЕРЖДЕНИЕ 20.2 Закон полной вероятности

Если $H_1, \dots, H_k$ образуют разбиение пространства, то

$$P(E) = \sum_{i=1}^k P(E | H_i) \cdot P(H_i).$$

Набросок доказательства

Докажем, разбив событие E на непересекающиеся части по гипотезам и применив аддитивность.

Гипотезы $H_1, \dots, H_k$ образуют разбиение пространства: они попарно несовместны и в объединении дают всё множество исходов. Событие распадается на непересекающиеся части $E \cap H_i$ — по одной на каждую гипотезу.

По аддитивности:

$$P(E) = \sum_{i=1}^k P(E \cap H_i).$$

По определению условной вероятности каждое слагаемое — произведение:

$$P(E \cap H_i) = P(E | H_i) \cdot P(H_i).$$

Отсюда и следует закон полной вероятности.

Пример Медицинский тест

Болезнь встречается у 1% населения: $P(D) = 0.01$. Тест даёт положительный результат с вероятностью 0.95 для больного: $P(+ | D) = 0.95$. Тест даёт ложноположительный результат с вероятностью 0.05 для здорового: $P(+ | \neg D) = 0.05$.

Насколько вероятно, что пациент с положительным тестом действительно болен?

Полная вероятность положительного результата:

$$P(+) = P(+ | D)P(D) + P(+ | \neg D)P(\neg D) = 0.95 \cdot 0.01 + 0.05 \cdot 0.99 = 0.059.$$

Апостериорная вероятность болезни по формуле Байеса:

$$P(D | +) = \frac{0.95 \cdot 0.01}{0.059} \approx 0.16.$$

При положительном тесте вероятность болезни — всего 16%. Интуиция подсказывает «тест точен на 95%, значит, вероятность болезни 95%» — ошибка base rate fallacy: мы игнорируем, что сама болезнь редка.

Тот же результат становится интуитивно ясным, если перейти от вероятностей к частотам. Рассмотрим 1000 случайных людей:

1. В среднем 10 действительно больны: 9–10 из них получают положительный тест (чувствительность 95%).
2. Из 990 здоровых около 50 получают ложноположительный результат (специфичность — доля здоровых с отрицательным тестом, здесь она 95%, значит доля ложноположительных результатов = $1 - 0.95 = 5\%$).
3. Итого около 60 положительных тестов, из которых лишь 10 — от реально больных.

Отсюда $\frac{10}{60} = P(D | +) \approx \frac{1}{6}$. Формат частот (natural frequencies) превращает формулу Байеса из абстрактной дроби в наглядный подсчёт.

Связь причины и следствия удобно нарисовать байесовской сетью: направленным графом, где стрелка — причинное влияние, а число на ней — условная вероятность. На рис. 92 узел «Грипп» влияет на «Кашель» и «Температуру».

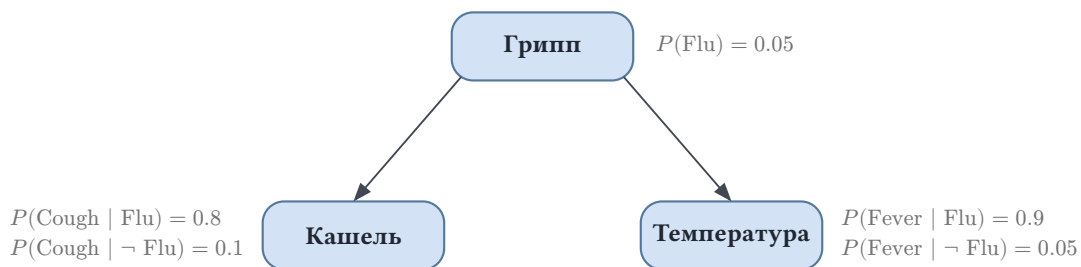


Рис. 92. Байесовская сеть: грипп влияет на кашель и температуру.

Замечание Байес как рисунок

Формулу Байеса удобно каждый раз рисовать.

Представьте пространство всех возможностей квадратом площади 1, а гипотезу H — полосой внутри него. Ширина полосы — $P(H)$. Свидетельство E вырезает

из квадрата фигуру. Внутри полосы гипотезы она занимает долю $P(E | H)$. Вне полосы её доля — $P(E | \neg H)$. Апостериорная вероятность $P(H | E)$ — доля, которую полоса гипотезы занимает в этой суженной фигуре.

Рисунок сам подсказывает формулу — и напоминает, что в знаменателе стоит полная вероятность свидетельства.

Проверка Условная вероятность и формула Байеса

1. Чем отличаются правдоподобие и апостериорная вероятность: $P(E | H)$, $P(H | E)$?
2. Как формула полной вероятности выражает вероятность свидетельства через разбиение на гипотезы $H_1, \dots, H_k$?

§ 20.4 Случайные величины

Числовой исход эксперимента — *случайная величина*. Это функция из пространства исходов в $\mathbb{R}$, и она несёт с собой распределение вероятностей.

ОПРЕДЕЛЕНИЕ 20.5 Случайная величина

Случайной величиной (random variable) на вероятностном пространстве называется функция $X : \Omega \rightarrow \mathbb{R}$.

ОПРЕДЕЛЕНИЕ 20.6 Распределение

Распределением случайной величины называется набор вероятностей $P(X = x)$ по всем возможным значениям.

После эксперимента значение $X(\omega)$ уже известно — это обычное число. Случайные величины независимы, если для всех значений x, y

$$P(X = x, Y = y) = P(X = x)P(Y = y).$$

Пример Сумма двух кубиков

X = сумма очков на двух кубиках. Распределение симметрично: максимум — на семёрке, к краям вероятности убывают.

x	$P(X = x)$
2	$\frac{1}{36}$
3	$\frac{2}{36}$
4	$\frac{3}{36}$
5	$\frac{4}{36}$

x	$P(X = x)$
6	$\frac{5}{36}$
7	$\frac{6}{36}$
8	$\frac{5}{36}$
9	$\frac{4}{36}$
10	$\frac{3}{36}$
11	$\frac{2}{36}$
12	$\frac{1}{36}$

Распределение — полная картина, но для сравнения величин из неё удобно извлекать несколько чисел. Первое из них — *матожидание*.

ОПРЕДЕЛЕНИЕ 20.7 Матожидание

Матожиданием (expected value) случайной величины X называется

$$E[X] = \sum_x x \cdot P(X = x).$$

Матожидание сводит распределение к одной точке. Два распределения с одинаковым матожиданием могут различаться тем, насколько тесно значения сосредоточены вокруг неё. Тесноту этого разброса измеряет *дисперсия*.

ОПРЕДЕЛЕНИЕ 20.8 Дисперсия

Дисперсией случайной величины X называется

$$\text{Var}[X] = E[(X - E[X])^2].$$

Дисперсия измеряет разброс значений относительно среднего. Среднее абсолютное отклонение $E[|X - E[X]|]$ не менее разумно, но квадрат алгебраически удобнее: у функции x^2 нет излома в нуле, как у модуля. Для суммы независимых бросаний кубика дисперсии складываются, тогда как модули — нет. Эта аддитивность нужна, чтобы вычислять дисперсию суммы независимых величин. Само же неравенство Чебышёва — это неравенство Маркова, применённое к квадрату отклонения. Для вычислений удобна эквивалентная форма:

$$\text{Var}[X] = E[X^2] - E[X]^2.$$

Дисперсия измеряется в квадратах исходных единиц: для очков на кубике — в квадратных очках. Чтобы вернуть разброс к тем же единицам, берут квадратный корень.

ОПРЕДЕЛЕНИЕ 20.9 Стандартное отклонение

Стандартным отклонением случайной величины X называется

$$\sigma[X] = \sqrt{\text{Var}[X]}.$$

Матожидание — взвешенное среднее возможных значений, где веса — вероятности. Для равновероятных исходов матожидание — обычное среднее арифметическое.

Пример Матожидание суммы на кубике

X — значение на одном честном кубике. $P(X = i) = \frac{1}{6}, i = 1, \dots, 6$.

1. $E[X] = \frac{1+2+3+4+5+6}{6} = 3.5$
2. $E[X^2] = \frac{1+4+9+16+25+36}{6} = 15.17$
3. $\text{Var}[X] = 15.17 - 3.5^2 = 2.92$

§ 20.5 Линейность матожидания

Факт, на который мы будем опираться постоянно: матожидание суммы равно сумме матожиданий — всегда, без каких-либо условий о независимости.

ТЕОРЕМА 20.3 **Линейность матожидания**

Для любых случайных величин $X_1, \dots, X_n$ и произвольных констант:

$$E[a_1 X_1 + \dots + a_n X_n] = a_1 E[X_1] + \dots + a_n E[X_n].$$

Никаких предположений о независимости не требуется.

Доказательство

Докажем линейность через определение матожидания и перегруппировку сумм.

По определению матожидание — сумма по всем исходам:

$$E[X] = \sum_{\omega \in \Omega} X(\omega) P(\{\omega\}).$$

Это та же формула $\sum_x x \cdot P(X = x)$: сгруппируем исходы по значению величины. Ведь $P(X = x) = \sum_{\omega: X(\omega)=x} P(\{\omega\})$.

Раскроем сумму двух величин и перегруппируем слагаемые:

$$E[X + Y] = \sum_{\omega} (X(\omega) + Y(\omega)) P(\{\omega\}) = \sum_{\omega} X(\omega) P(\{\omega\}) + \sum_{\omega} Y(\omega) P(\{\omega\}) = E[X] + E[Y].$$

Линейность — структурное свойство суммы, а не вероятностный факт: сумма сумм раскладывается независимо от того, как связаны слагаемые.

Константа выносится за знак суммы аналогично:

$$E[aX] = \sum_{\omega} aX(\omega)P(\{\omega\}) = a \sum_{\omega} X(\omega)P(\{\omega\}) = aE[X].$$

ОПРЕДЕЛЕНИЕ 20.10 Индикатор события

Индикатором события называется случайная величина 1_A , равная 1, если событие произошло, и 0 иначе.

Матожидание индикатора равно вероятности события:

$$E[1_A] = P(A).$$

Индикаторы — простейшие случайные величины и одновременно самый эффективный инструмент: разложив сложную величину в сумму индикаторов, мы вычисляем матожидание без возни с совместным распределением.

Пример Среднее число неподвижных точек перестановки

Случайная перестановка n элементов. Сколько элементов в среднем останутся на своих местах?

Пусть $X_i = 1$, если соответствующий элемент на месте, иначе 0. $P(X_i = 1) = \frac{1}{n}$ (элемент равновероятно попадает в любую позицию). Общее число неподвижных точек: $X = X_1 + \dots + X_n$.

$$E[X] = E[X_1] + \dots + E[X_n] = n \cdot \frac{1}{n} = 1.$$

В среднем ровно один элемент остаётся на месте, независимо от n . Прямой подсчёт через распределение числа неподвижных точек потребовал бы формулы включений-исключений.

Пример Ожидаемое число сравнений в быстрой сортировке

Быстрая сортировка (quicksort) на случайном входе. Пусть X — общее число сравнений. Занумеруем элементы по возрастанию: $y_1 < y_2 < \dots < y_n$.

Индикатор $X_{ij} = 1$, если соответствующая пара сравнивается в процессе сортировки. Два элемента сравниваются, если один из них выбран как опорный (pivot) раньше, чем любой элемент между ними. $P(X_{ij} = 1) = \frac{2}{j-i+1}$.

$$E[X] = \sum_{i < j} E[X_{ij}] = \sum_{i < j} \frac{2}{j-i+1} \approx 2n \ln n.$$

Линейность матожидания превращает сложный анализ в подсчёт сумм.

Замечание Почему линейность — структурное свойство

Формула $E[X + Y] = E[X] + E[Y]$ воспринимается обычно как трюк: разложи сложную величину в сумму индикаторов — и матожидание вычислено. Кажется, что фокус в удачном выборе индикаторов. Но это впечатление обманчиво.

Линейность матожидания не имеет отношения ни к индикаторам, ни к независимости. Математическое ожидание — это сумма по пространству элементарных исходов.

Сумма сумм раскладывается независимо от того, как слагаемые связаны между собой:

$$\sum_{\omega} (X(\omega) + Y(\omega)) = \sum_{\omega} X(\omega) + \sum_{\omega} Y(\omega).$$

Никакого условия на совместное распределение здесь не требуется — потому что на уровне определения матожидания случайные величины ещё не стали «случайными». Это просто функции на одном и том же пространстве, и сумма линейна по построению.

Индикаторы не создают линейность — они её эксплуатируют. Линейность была там всегда: это структурное свойство суммы, а не вероятностный факт. Именно поэтому формула работает для любых случайных величин, а индикаторы — лишь самое эффективное её применение. Независимость нужна для мультипликативных утверждений — $E[XY] = E[X] \cdot E[Y]$ — и это уже другой, существенно вероятностный факт.

§ 20.6 Основные распределения

Элементарный опыт здесь один: испытание, оканчивающееся успехом с вероятностью p . Вопросов о нём три классических: каков исход одного испытания, сколько успехов наберётся за фиксированное число испытаний, сколько испытаний придётся ждать первого успеха. Каждому вопросу отвечает собственное распределение, и для каждого матожидание с дисперсией вычисляются до конца.

ОПРЕДЕЛЕНИЕ 20.11 Распределение Бернулли

Распределением Бернулли с параметром p называется распределение случайной величины, равной 1 с вероятностью p и 0 с вероятностью $1 - p$.

Величина с распределением Бернулли — это индикатор события «испытание удалось» из предыдущего раздела. По определениям

$$E[X] = p, \text{Var}[X] = p(1 - p).$$

Действительно, значения 0 и 1 не меняются при возведении в квадрат, поэтому $E[X^2] = E[X] = p$. Дисперсия максимальна при $p = \frac{1}{2}$: предсказать честную монету труднее всего.

ОПРЕДЕЛЕНИЕ 20.12 Биномиальное распределение

Биномиальным распределением с параметрами n и p называется распределение числа успехов в n независимых испытаниях с вероятностью успеха p в каждом.

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, k = 0, 1, \dots, n.$$

Биномиальная величина раскладывается в сумму индикаторов: $X = X_1 + \dots + X_n$, где X_i указывает успех i -го испытания. Линейность матожидания даёт $E[X] = np$ без единого обращения к биномиальным коэффициентам. Слагаемые независимы, поэтому складываются и дисперсии: $\text{Var}[X] = np(1-p)$. Хвосты именно таких сумм ограничивает неравенство Чернова из звёздного раздела ниже.

ОПРЕДЕЛЕНИЕ 20.13 Геометрическое распределение

Геометрическим распределением с параметром p называется распределение номера первого успеха в серии независимых испытаний с вероятностью успеха p в каждом.

$$P(X = k) = (1-p)^{k-1} p, k = 1, 2, \dots$$

Матожидание геометрической величины находится самоподобием процесса. Обозначим $\mu = E[X]$ и выделим первое испытание. С вероятностью p оно успешно, и вся величина равна 1. С вероятностью $1-p$ оно неудачно: одно испытание потрачено, и процесс начинается заново, так что осталось в среднем μ . Отсюда

$$\mu = p + (1-p)(1 + \mu) = 1 + (1-p)\mu,$$

и $\mu = \frac{1}{p}$. Тот же разбор первого испытания даёт $E[X^2] = \frac{2-p}{p^2}$, откуда $\text{Var}[X] = \frac{1-p}{p^2}$.

Проверка Случайные величины и распределения

1. Почему дисперсия величины с распределением Бернулли максимальна при $p = \frac{1}{2}$?
2. Какая теорема даёт $E[X] = np$ для биномиальной величины без подсчёта $P(X = k)$?
3. Сколько в среднем испытаний придётся ждать первого успеха при $p = 0.01$?

§ 20.7 Неравенства концентрации

Линейность матожидания отвечает на вопрос «где в среднем». Но рандомизированному алгоритму нужно знать и то, как часто величина уходит далеко от среднего. Время работы должно быть небольшим в среднем и почти всегда укладываться в разумный предел. Оценки на вероятность больших отклонений от среднего называются *неравенствами концентрации*. Все они следуют одной схеме: хвост распределения ограничивается через моменты.

ТЕОРЕМА 20.4 Неравенство Маркова

Для неотрицательной случайной величины и любого $a > 0$:

$$P(X \geq a) \leq \frac{E[X]}{a}.$$

Доказательство

Докажем, оценив X снизу порогом, помноженным на индикатор события $X \geq a$.

Индикатор события $X \geq a$ равен 1 на исходах, где событие наступило, и 0 иначе. Поточечно выполнено $X \geq 0 \Rightarrow a \cdot 1_{X \geq a} \leq X$. При $X \geq a$ левая часть равна порогу. При $X < a$ слева ноль, а величина неотрицательна. Возьмём матожидание обеих частей. Матожидание индикатора — вероятность события:

$$a \cdot P(X \geq a) \leq E[X].$$

Разделив на $a > 0$, получаем неравенство Маркова.

Самый наглядный смысл неравенства: доля значений, превосходящих k -кратное среднее, ограничена. Точная оценка: $P(X \geq k\mu) \leq \frac{1}{k}$.

Пример Большие значения редки

Пусть даны неотрицательные числа со средним μ . Обозначим их $x_1, \dots, x_n$. Пусть X — случайное число, выбранное равномерно из этого набора. Тогда $E[X] = \mu$, и по неравенству Маркова:

$$P(X \geq k\mu) \leq \frac{\mu}{k\mu} = \frac{1}{k}.$$

Число значений, превосходящих среднее в k раз, ограничено. Оно не превосходит $\frac{n}{k}$. Для $k = 10$ это звучит так: не более десятой части значений больше десяти средних — и для этого не нужно знать ничего, кроме среднего.

Пример Оценка времени работы

Пусть X — время работы рандомизированного алгоритма. Матожидание: $E[X] = 100$ миллисекунд. Вероятность, что алгоритм проработает дольше секунды:

$$P(X \geq 1000) \leq \frac{100}{1000} = \frac{1}{10}.$$

Не зная распределения, мы уже знаем: хуже секунды — не более чем в одном запуске из десяти. Оценка слабая, но бесплатная: от распределения нужен только средний показатель.

Неравенство Маркова знает лишь среднее. Если известно больше — разброс — оценку можно усилить: применить Маркова к квадрату отклонения от среднего.

ТЕОРЕМА 20.5 Неравенство Чебышёва

Для случайной величины с конечным матожиданием $\mu = E[X]$ и дисперсией выполнено неравенство Чебышёва. Для любого $a > 0$:

$$P(|X - \mu| \geq a) \leq \frac{\text{Var}[X]}{a^2}.$$

Доказательство

Докажем, сведя неравенство Чебышёва к неравенству Маркова, применённому к квадрату отклонения $Y = (X - \mu)^2$.

Матожидание этой величины — дисперсия: $E[Y] = \text{Var}[X]$. События совпадают: $|X - \mu| \geq a \Leftrightarrow Y \geq a^2$. По Маркову:

$$P(|X - \mu| \geq a) = P(Y \geq a^2) \leq \frac{E[Y]}{a^2} = \frac{\text{Var}[X]}{a^2}.$$

СЛЕДСТВИЕ 20.6 Сигма-правило

Обозначим $\sigma = \sigma[X]$ — стандартное отклонение. Полагая $a = k\sigma$, получаем:

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}.$$

За пределами k стандартных отклонений вероятность ограничена. Она не превышает $\frac{1}{k^2}$. Для трёх стандартных отклонений — не более $\frac{1}{9}$. Для пяти — не более $\frac{1}{25}$.

Пример Среднее многих бросков кубика

Бросаем честный кубик n раз. Значения бросков — $X_1, \dots, X_n$. Сумма: $S_n = X_1 + \dots + X_n$. Для одного броска $E[X_i] = 3.5$, $\text{Var}[X_i] = \frac{35}{12} \approx 2.92$ (пример выше). Для суммы $E[S_n] = 3.5n$. Дисперсия тоже складывается, но здесь независимость существенна (в отличие от линейности математического ожидания): $\text{Var}[S_n] = n \cdot \frac{35}{12}$.

Броски кубика независимы, поэтому дисперсия суммы — сумма дисперсий.

Какова вероятность, что среднее отклонится от математического ожидания больше чем на ε ? События равносильны: $|\frac{S_n}{n} - 3.5| \geq \varepsilon \Leftrightarrow |S_n - 3.5n| \geq n\varepsilon$. По Чебышёву:

$$P\left(\left|\frac{S_n}{n} - 3.5\right| \geq \varepsilon\right) \leq \frac{35 \frac{n}{12}}{n^2 \varepsilon^2} = \frac{35}{12n\varepsilon^2}.$$

При $n \rightarrow \infty$ правая часть стремится к нулю. В этом и состоит слабый закон больших чисел в зародыше: среднее многих независимых одинаковых бросков сходится по вероятности к математическому ожиданию.

Проверка Неравенства концентрации

1. В каком смысле неравенство Чебышёва — это неравенство Маркова, применённое к квадрату отклонения?
2. Почему неравенство Маркова требует неотрицательности случайной величины, а неравенство Чебышёва — нет?

§ 20.8 * Неравенство Чернова

Марков и Чебышёв дают хвосты, убывающие как $\frac{1}{a}$, $\frac{1}{a^2}$ — полиномиально. Для больших отклонений этого мало: хочется хвостов, убывающих экспоненциально. Экспоненциальный спад даёт неравенство Чернова — ценой условия: оно работает для сумм независимых индикаторов. А это именно тот случай, который нужен анализу алгоритмов: рандомизированный алгоритм почти всегда складывает независимые случайные биты.

ТЕОРЕМА 20.7 Неравенство Чернова

Пусть $X_1, \dots, X_n$ — независимые случайные величины со значениями 0 и 1. Для каждой: $P(X_i = 1) = p_i$. Положим $X = X_1 + \dots + X_n$, $\mu = E[X] = \sum_{i=1}^n p_i$. Для любого $\delta > 0$:

$$P(X \geq (1 + \delta)\mu) \leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu.$$

СЛЕДСТВИЕ 20.8 Упрощённая форма

При $0 < \delta \leq 1$ из оценки Чернова следует

$$P(X \geq (1 + \delta)\mu) \leq e^{-\mu \frac{\delta^2}{3}}.$$

Доказательство

Докажем методом экспоненциальных моментов: перейдём к экспоненте, применим неравенство Маркова, оценим произведение моментов и выберем параметр t оптимально.

Зафиксируем $t > 0$. Значение выберем позже. Перейдём от X к экспоненте. События равносильны: $X \geq (1 + \delta)\mu \Leftrightarrow e^{tX} \geq e^{t(1+\delta)\mu}$. По неравенству Маркова, применённому к неотрицательной величине e^{tX} :

$$P(X \geq (1 + \delta)\mu) \leq \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}}.$$

Матожидание экспоненты суммы факторизуется по независимости:

$$E[e^{tX}] = \prod_{i=1}^n E[e^{tX_i}].$$

Для каждого индикатора $E[e^{tX_i}] = 1 + p_i(e^t - 1) \leq e^{p_i(e^t - 1)}$. Последнее неравенство — стандартное: $1 + x \leq e^x$. Значит,

$$E[e^{tX}] \leq e^{\mu(e^t - 1)}.$$

Собираем всё вместе:

$$P(X \geq (1 + \delta)\mu) \leq e^{\mu(e^t - 1) - t(1+\delta)\mu}.$$

Показатель минимален при $t = \ln(1 + \delta)$:

$$P(X \geq (1 + \delta)\mu) \leq e^{\mu(\delta - (1+\delta)\ln(1+\delta))} = \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu.$$

Упрощённая форма следует из стандартной оценки $\delta - (1 + \delta)\ln(1 + \delta) \leq -\frac{\delta^2}{3}, 0 < \delta \leq 1$.

Пример Сколько орлов можно реально увидеть

Подбрасываем честную монету тысячу раз: $n = 1000, \mu = 500$. Какова вероятность, что орлов окажется хотя бы 600, то есть $X \geq 1.2\mu$? Здесь $\delta = 0.2$, и по упрощённой форме Чернова:

$$P(X \geq 600) \leq e^{-500 \cdot \frac{0.2^2}{3}} = e^{-6.67} \approx 0.0013.$$

Сравним с Чебышёвым: $\text{Var}[X] = 250, a = 100$. Тогда $P(X \geq 600) \leq \frac{250}{10} = 0.025$. Отклонение на 20% от среднего при тысяче бросков — по Чернову, событие с вероятностью порядка 10^{-3} . По Чебышёву — лишь 2.5%. Такова разница между полиномиальным и экспоненциальным хвостом.

Пример Перегрузка ящиков

Бросаем n шаров в m ящиков независимо и равномерно. Пусть $X_i = 1$, если соответствующий шар попал в выделенный ящик. Тогда $X = X_1 + \dots + X_n$ — число шаров в этом ящике. Матожидание: $\mu = E[X] = \frac{n}{m}$. Вероятность, что в ящике окажется вдвое больше шаров, чем в среднем ($\delta = 1$):

$$P(X \geq 2\mu) \leq e^{-\frac{\mu}{3}}.$$

При средней нагрузке 30 шаров на ящик перегрузка вдвое имеет вероятность $e^{-10} \approx 4.5 \cdot 10^{-5}$. Это редкое событие. А если ящиков мало и μ мало, оценка слабеет — честно: в маленькой таблице перегрузки неизбежны.

Соберём все три оценки в одну таблицу.

Неравенство	Хвост	Требования	Когда применять
Маркова	$\frac{E[X]}{a}$	$X \geq 0$	известно только среднее
Чебышёва	$\frac{\text{Var}[X]}{a^2}$	конечная дисперсия	известны среднее и разброс
Чернова	$e^{-\mu \frac{\delta^2}{3}}$	сумма независимых индикаторов	нужен экспоненциальный хвост

Неравенства концентрации — обратная сторона линейности матожидания. Линейность говорит, где находится среднее. Концентрация — насколько редко величина от него уходит. Вместе они составляют основной инструмент анализа рандомизированных алгоритмов.

§ 20.9 Центральная предельная теорема

Неравенства концентрации ограничивают хвосты сверху, и форма распределения в них выпадает: в неравенства входят только среднее и разброс. Подбросим честную монету $n = 100$ раз и спросим, как часто число орлов отклонится от полусотни не меньше чем на десять. Сигма-правило ограничивает это событие четвертью, а точный подсчёт даёт 0.057: граница завышена более чем вчетверо, и внутри Чебышёва её не улучшить. Природа хвостов у сумм иная: форма их распределения подчиняется точному закону.

Закон виден на гистограммах: сто бросков монеты рисуют холм с вершиной у 50, сто бросков кубика — холм с вершиной у 350, и форма у обоих одна — «колокол».

Говорить о предельной форме самой суммы, однако, бессмысленно: её центр $n\mu$ уходит вправо, разброс растёт как $\sigma\sqrt{n}$, и с ростом n сумма расплывается. Стабильна только стандартизованная величина — сумма минус её центр, делённая на собственный разброс. Именно у неё форма перестаёт зависеть от n , и эту устойчивую форму надо назвать.

Все величины главы до сих пор были дискретными: вероятность — масса на значениях, а «колокол» — непрерывная кривая, где вероятность промежутка равна площади под кривой.

ОПРЕДЕЛЕНИЕ 20.14 Стандартное нормальное распределение

Стандартным нормальным распределением называется распределение случайной величины Z с плотностью $\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. Вероятность попадания в промежуток равна площади под этой кривой: $P(a \leq Z \leq b) = \int_a^b \varphi(x) dx$.

Накопленную площадь слева от точки обозначим $\Phi(z) = P(Z \leq z)$. Значения Φ сведены в таблицы, и несколько пригодятся сразу: $\Phi(0) = 0.5$, $\Phi(1) \approx 0.8413$, $\Phi(2) \approx 0.9772$, $\Phi(3) \approx 0.9987$.

ТЕОРЕМА 20.9 Центральная предельная теорема

Пусть $X_1, X_2, \dots$ — независимые одинаково распределённые случайные величины с матожиданием μ и дисперсией $\sigma^2 > 0$. Обозначим их частичные суммы $S_n = X_1 + \dots + X_n$. Тогда при $n \rightarrow \infty$ для любого $z \in \mathbb{R}$

$$P\left(\frac{S_n - n\mu}{\sigma\sqrt{n}} \leq z\right) \rightarrow \Phi(z).$$

Доказательство остаётся за рамками курса: классический путь проходит через характеристические функции — аппарат, опирающийся на комплексный анализ.

В формулировке в числителе стоит отклонение суммы от её центра $n\mu$, в знаменателе — собственный разброс: дисперсии независимых слагаемых складываются, поэтому разброс растёт как $\sigma\sqrt{n}$. Теорема утверждает: у отклонения, измеренного в этих естественных единицах, предельная форма одна и та же для всех исходных распределений. Отсюда практическое чтение: типичные отклонения суммы от центра имеют порядок $\sigma\sqrt{n}$, а вероятность ухода на несколько таких масштабов даётся площадью колокола. Для ухода за три масштаба в обе стороны колокол даёт $2(1 - \Phi(3)) \approx 0.0027$, тогда как сигма-правило гарантировало лишь границу $\frac{1}{9}$.

Пример Сто бросков монеты

Пусть X — число орлов при $n = 100$ бросках честной монеты, биномиальная величина со средним $E[X] = 50$, дисперсией $\text{Var}[X] = 25$ и отклонением $\sigma[X] = 5$. Сравним её форму с колоколом на двух числах — в центре и в хвосте.

В центре точный подсчёт даёт $P(X = 50) = \frac{\binom{100}{50}}{2^{100}} \approx 0.0796$. Точка k на непрерывной шкале занимает отрезок длины один, в единицах $z = \frac{k-50}{5}$ — длину $\frac{1}{5}$, и колокол отводит ему массу $\frac{\varphi(z)}{5}$. Для $k = 50$ получается $z = 0$ и масса $\frac{\varphi(0)}{5} = \frac{1}{5\sqrt{2\pi}} \approx 0.0798$ — совпадение до третьего знака.

В хвосте точный подсчёт даёт $P(X \geq 60) = \sum_{k=60}^{100} \frac{\binom{100}{k}}{2^{100}} \approx 0.0284$. Значение $X = 60$ занимает на непрерывной шкале отрезок от 59.5 до 60.5, его масса считается площадью над этим отрезком, и потому хвост начинается с $z = \frac{59.5-50}{5} = 1.9$:

$$P(X \geq 60) \approx 1 - \Phi(1.9) \approx 0.0287.$$

Приближение работает и в центре, и в хвосте, хотя из свойств монеты взяты только среднее и разброс.

Примечание Неравенство Берри–Эссина

Скорость приближения к колоколу измеряет неравенство Берри–Эссина. Если $\rho = E[|X_1 - \mu|^3]$ — третий абсолютный момент слагаемого, то наибольшее по z отклонение вероятности $P(Z_n \leq z)$ от $\Phi(z)$ ограничено:

$$\sup_{z \in \mathbb{R}} |P(Z_n \leq z) - \Phi(z)| \leq C \cdot \frac{\rho}{\sigma^3 \sqrt{n}},$$

где C — абсолютная константа, а известные оценки дают $C \leq 0.56$. Для честной монеты $\rho = \sigma^3 = \frac{1}{8}$, и оценка превращается в $\frac{C}{\sqrt{n}}$: при $n = 100$ это 0.056, а реальное наибольшее отклонение — около 0.04.

Нормальное распределение — аттрактор: сумма многих независимых одинаково распределённых вкладов притягивается к колоколу, каким бы ни был отдельный вклад. Этим объясняется, почему погрешность измерения, складывающаяся из множества мелких независимых причин, имеет распределение, близкое к нормальному. Конечность дисперсии при этом существенна: у распределений с тяжёлыми хвостами суммы притягиваются к другим предельным формам.

С законом больших чисел, зародыш которого появился в разделе о концентрации, теорема связана напрямую. Закон утверждает: среднее $\frac{S_n}{n}$ сходится по вероятности к μ . Из формулы для Z_n следует $\frac{S_n}{n} = \mu + \sigma \frac{Z_n}{\sqrt{n}}$: отклонения среднего от предела имеют масштаб $\frac{\sigma}{\sqrt{n}}$. Закон отвечает, куда сходится среднее, а центральная предельная теорема — с каким масштабом и в какой форме оно колеблется вокруг предела.

Проверка Центральная предельная теорема

1. Почему предельную форму ищут у стандартизированной суммы, а не у самой S_n ?

2. Монета подбрасывается $n = 100$ раз: оцените по центральной предельной теореме вероятность увидеть не меньше 60 орлов и сравните её с границей Чебышёва.
3. Что измеряет неравенство Берри–Эссина и от каких свойств слагаемого зависит его правая часть?

§ 20.10 Парадокс дней рождения

Пример Вероятность совпадения дней рождения

В комнате n человек. Какова вероятность, что хотя бы у двоих день рождения в один день?

Проще сначала посчитать вероятность дополнительного события — что у всех дни рождения *разные*. Для $n \leq 365$:

$$P(\text{все разные}) = \frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \dots \cdot \frac{365 - n + 1}{365}.$$

Для 23 человек: $P(\text{все разные}) \approx 0.493$. $P(\text{хотя бы одно совпадение}) \approx 0.507$.

С 23 людьми в комнате вероятность совпадения превышает 50% — хотя интуитивно кажется, что нужно около $\frac{365}{2} \approx 183$ человек.

Парадокс дней рождения — конфликт между интуицией и вычислением: логического противоречия в нём нет. Интуиция сравнивает каждого человека с *одним* конкретным (собой) — это линейный рост числа пар $(n - 1)$. Вычисление учитывает *все* пары — а их $\binom{n}{2}$. Для 23 человек: $\binom{23}{2} = 253$ пары, каждая — отдельный шанс на совпадение. Пар квадратично много — поэтому и вероятность растёт быстрее, чем ожидает интуиция.

Прикидка по парам предсказывает и сам порог. Ожидаемое число совпадающих пар равно $\frac{\binom{n}{2}}{365}$. Совпадение становится вероятным, когда эта величина достигает порядка единицы. Отсюда порог: $n \approx \sqrt{2 \cdot 365} \approx 27$. Точная формула: $P \approx 1 - e^{-\frac{\binom{n}{2}}{365}}$. Для 23 человек: $1 - e^{-\frac{253}{365}} \approx 0.50$ — рядом с точным значением 0.507. Сводку вероятности совпадения при разных n даёт таблица ниже.

n	$P(\text{совпадение})$
10	11.7%
20	41.1%
23	50.7%
30	70.6%
40	89.1%
50	97.0%
60	99.4%
70	99.92%

Таблица 12. Вероятность совпадения дней рождения при n человеках.

Эта же математика лежит в основе атаки «дней рождения» на хеш-функции. Чтобы найти коллизию, хватает примерно $2^{\frac{m}{2}}$ значений — корень из размера кодомона.

Примечание Монте-Карло оценка вероятности совпадения

Точную формулу из примера выше можно проверить экспериментом. Метод Монте-Карло разыгрывает дни рождения многих групп по m человек и считает долю групп, где совпадение есть. Генератор случайных чисел — линейный конгруэнтный (LCG) с параметрами из книги Кнута, поэтому внешние зависимости не нужны.

```

/// Парадокс дней рождения методом Монте-Карло
/// (без внешних зависимостей, LCG).
fn birthday(m: usize, trials: usize) -> f64 {
    let mut x: u64 = 1;
    let mut hits = 0;
    for _ in 0..trials {
        let mut seen = [false; 365];
        let mut dup = false;
        for _ in 0..m {
            x = x
                .wrapping_mul(6364136223846793005)
                .wrapping_add(1442695040888963407);
            let day = (x % 365) as usize;
            if seen[day] { dup = true; }
            seen[day] = true;
        }
        if dup { hits += 1; }
    }
    hits as f64 / trials as f64
}

```

Для группы из 23 человек функция возвращает около 0.51 при большом числе опытов. Это согласуется с точным значением $P(\text{совпадение}) \approx 0.507$.

Пример Задача о шляпах (*derangement problem*)

n человек сдают шляпы в гардероб. Шляпы возвращаются случайно. Какова вероятность, что *никто* не получит свою шляпу?

Пусть A_i — событие, что соответствующий человек получил свою шляпу. Нас интересует $P(\cup A_i)$ — вероятность, что хотя бы один получил свою. Тогда $1 - P(\cup A_i)$ — что никто не получил.

По формуле включений-исключений (глава 18):

$$P(\cup A_i) = \sum_i P(A_i) - \sum_{i < j} P(A_i \cap A_j) + \dots + (-1)^{n-1} P(A_1 \cap \dots \cap A_n).$$

$P(A_i) = \frac{1}{n}$ (шляпа с этим номером с равной вероятностью попадает к любому).
 $P(A_i \cap A_j) = \frac{1}{n(n-1)}$ (две конкретные шляпы к своим владельцам). $P(A_{i_1} \cap \dots \cap A_{i_k}) = \frac{(n-k)!}{n!}$.

$$P(\cup A_i) = n \cdot \frac{1}{n} - \binom{n}{2} \cdot \frac{1}{n(n-1)} + \dots = 1 - \frac{1}{2!} + \frac{1}{3!} - \dots + (-1)^{n-1} \frac{1}{n!}.$$

Искомая вероятность (никто не получил свою):

$$P(\text{никто}) = 1 - P(\cup A_i) = \frac{1}{2!} - \frac{1}{3!} + \dots + (-1)^n \frac{1}{n!}.$$

Это частичная сумма ряда Тейлора для экспоненты при $x = -1$. Напомним: $e^{-1} = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!}$. Первые два члена ($k = 0, 1$) дают ноль. Поэтому разложение для вероятности «никто» начинается с члена $\frac{1}{2!}$. При $n \rightarrow \infty$:

$$P(\text{никто}) \rightarrow \frac{1}{e} \approx 0.368.$$

Интуиция может подсказывать, что с ростом n шанс, что никто не получит свою шляпу, стремится к 0 или к 1. Вычисление показывает: он стабилизируется около 37%, не завися от числа участников (при $n \geq 5$ приближение уже отличное). Формула включений-исключений (комбинаторика) и предел $\frac{1}{e}$ (анализ) встречаются в вероятностной задаче — три области математики в одном сюжете.

Замечание Дрейф генов и случайные блуждания

Возьмём популяцию снежных барсов в горной долине. Десять особей, у каждой по две копии гена (диплоидная популяция). Версий у гена две. В каждом поколении аллели передаются потомкам — и частоты версий меняются. Особей мало, и наследование — случайная выборка из родительского генофонда, а не следствие того, что одна версия «лучше».

Это дрейф генов — изменение частот аллелей, вызванное случайной выборкой при образовании следующего поколения. В отличие от естественного отбора, дрейф не связан с приспособленностью. В реальной популяции оба процесса действуют одновременно. С математической стороны — случайное блуждание на конечном отрезке с поглощающими границами. Состояние системы — число копий одного аллеля, от 0 до $2N$ (по две копии гена на особь). На каждом шаге

состояние сдвигается влево или вправо случайно. Матожидание следующего состояния равно текущему, но разброс накапливается. Рано или поздно блуждание упирается в границу — и процесс останавливается.

Через много поколений одна версия гена исчезает, другая фиксируется навсегда. В среднем частота каждой версии оставалась постоянной всё это время — но среднее не говорит всей правды: дисперсия росла, пока один из аллелей не достиг границы. Та же математическая структура описывает распространение инноваций в социальной сети, случайные блуждания по графу и поведение рандомизированных алгоритмов на длинных временах.

§ 20.11 Применения в алгоритмах

Дискретная вероятность — рабочий инструмент анализа алгоритмов. Рандомизированный алгоритм — это алгоритм, делающий случайный выбор на некоторых шагах. Его корректность и время работы описываются вероятностными утверждениями.

Быстрая сортировка (Hoare, 1961) с детерминированным выбором первого элемента даёт $O(n^2)$ на уже отсортированном массиве — худший случай, который на практике встречается чаще, чем хотелось бы. Случайный выбор опорного элемента превращает худший случай в статистическую аномалию: ожидаемое время $O(n \log n)$ на любом входе.

Алгоритм Фривалда (Freivalds, 1977) решает задачу, которая выглядит не имеющей быстрого решения: проверить $AB = C$, не вычисляя матричное произведение. Выберем случайный вектор x : каждая координата независимо равна 0 или 1 с вероятностью $\frac{1}{2}$. Проверим равенство $A(Bx) = Cx$. Если $AB = C$, проверка проходит всегда. Если $AB \neq C$, ошибка возможна, и её вероятность досчитывается до конца. Пусть $D = AB - C \neq 0$. В матрице D есть ненулевой элемент, скажем $d_{ij} \neq 0$. Зафиксируем все координаты вектора x , кроме x_j . Уравнение $Dx = 0$ в строке i превращается в линейное уравнение с одной неизвестной:

$$d_{ij}x_j + \sum_{k \neq j} d_{ik}x_k = 0.$$

Все прочие слагаемые известны, а коэффициент d_{ij} отличен от нуля, поэтому уравнение допускает ровно одно значение x_j из двух возможных. Условная вероятность ошибки при любых фиксированных остальных координатах не превышает $\frac{1}{2}$. По закону полной вероятности безусловная вероятность $P(Dx = 0)$ есть усреднение этих условных вероятностей по значениям остальных координат, поэтому и она не превышает $\frac{1}{2}$. После k независимых повторов с новыми векторами она падает экспоненциально: до $\frac{1}{2^k}$.

Метод Монте-Карло оценивает величины, которые трудно вычислить точно, случайной выборкой. Оценка числа π бросанием иголки (Бюффон, 1777) — исторически первый пример. Современные применения: оценка интегралов в высокой размерности, байесовский вывод (MCMC), физическое моделирование.

Проверка Применения в алгоритмах

1. Почему проверка $A(Bx) = Cx$ для случайного x дешевле вычисления произведения AB ?
2. Что меняет случайный выбор опорного элемента в быстрой сортировке: худший случай или его вероятность?
3. Как алгоритм Фривалда делает вероятность ошибки сколь угодно малой?

§ 20.12 Вероятностный метод

Вероятностный метод — техника доказательства существования, использующая вероятность как инструмент чистого существования. Чтобы доказать, что объект с заданным свойством *существует*, достаточно показать, что вероятность случайно выбрать такой объект положительна. Если $P(\text{объект обладает свойством}) > 0$, то объект с этим свойством обязан существовать — иначе вероятность была бы нулевой.

Схема доказательства вероятностным методом:

1. Построить вероятностное пространство на множестве интересующих нас объектов.
2. Показать, что интересующее нас свойство выполняется с ненулевой (обычно — близкой к 1) вероятностью.
3. Сделать вывод: объект с этим свойством существует.

Пример Нижняя граница числа Рамсея $R(k, k)$

Теорема: $R(k, k) > 2^{\frac{k}{2}}$, $k \geq 3$. То есть существует граф на $n = \lfloor 2^{\frac{k}{2}} \rfloor$ вершинах без клики и без независимого множества такого же размера.

Доказательство. Рассмотрим случайный граф на n вершинах. Рёбер всего $\binom{n}{2}$. Каждое ребро включается независимо с вероятностью $\frac{1}{2}$. Это модель Эрдёша-Реньи $G(n, \frac{1}{2})$.

Зафиксируем подмножество из k вершин. Обозначим его K . Для клики должны присутствовать все $\binom{k}{2}$ рёбер. Вероятность этого: $P(K \text{ --- клика}) = \left(\frac{1}{2}\right)^{\binom{k}{2}}$. Аналогично, $P(K \text{ --- независимое множество}) = \left(\frac{1}{2}\right)^{\binom{k}{2}}$.

Всего таких подмножеств $\binom{n}{k}$. По union bound (неравенство Буля):

$$P(\text{есть клика или независимое множество размера } k) \leq 2 \cdot \binom{n}{k} \cdot \left(\frac{1}{2}\right)^{\binom{k}{2}}.$$

При $n = \lfloor 2^{\frac{k}{2}} \rfloor$ правая часть строго меньше 1. Проверка: всего наборов $\binom{n}{k} \leq \frac{n^k}{k!}$. Конкретный набор образует клику с вероятностью $2^{-\binom{k}{2}}$ и независимое множество — тоже. По объединению:

$$P(\text{есть клика или независимое множество}) \leq 2 \cdot \binom{n}{k} \cdot 2^{-\binom{k}{2}} \leq 2 \cdot \frac{2^{\frac{k^2}{2}}}{k!} \cdot 2^{-\frac{k(k-1)}{2}} = \frac{2^{\frac{k}{2}+1}}{k!}.$$

Это меньше единицы при $k \geq 3$. Например, при трёх вершинах: $\frac{2^{2.5}}{6} \approx 0.94$. Следовательно, $P(\text{нет ни клики, ни независимого множества размера } k) > 0$ — такой граф существует.

Мы доказали существование графа с определённым свойством, *не построив* его явно. Это суть вероятностного метода: неконструктивное доказательство существования.

Вероятностный метод, изобретённый Полом Эрдёшем в 1940-х годах, открыл в комбинаторике целое направление: доказательства существования без явного построения. Многие комбинаторные результаты (глава 18) — в частности, нижние оценки чисел Рамсея и существование графов с высоким хроматическим числом и обхватом — доказываются вероятностным методом.

§ 20.13 Случайные графы

Вероятностный метод естественно приводит к модели случайного графа. Модель Эрдёша–Реньи $G(n, p)$: рёбра включаются независимо.

ОПРЕДЕЛЕНИЕ 20.15 Модель $G(n, p)$

Случайный граф $G(n, p)$ — вероятностное пространство на всех графах с заданным числом вершин. Каждое ребро присутствует с вероятностью p , независимо от остальных. Вероятность конкретного графа с m рёбрами:

$$P(G) = p^m (1-p)^{\binom{n}{2}-m}.$$

При $p = \frac{1}{2}$ все графы равновероятны. Каждый конкретный граф имеет вероятность $\left(\frac{1}{2}\right)^{\binom{n}{2}}$.

Пример Малый случай $G(3, \frac{1}{2})$

У трёх вершин рёбер три, поэтому графов $2^3 = 8$. Каждый конкретный граф — от пустого до полного — имеет вероятность $\frac{1}{8}$.

Связны из них четыре: три «пути» по два ребра и треугольник. Так модель проверяется ручным перечислением ещё до формул.

Случайные графы демонстрируют *фазовые переходы*: при увеличении p свойства графа появляются скачком. Например, при $p \approx \frac{\log n}{n}$ граф почти наверняка становится связным. При $p \approx \frac{1}{n}$ появляется гигантская компонента. Её размер — порядка n .

Пример Порог связности

Если $p < (1 - \varepsilon) \frac{\log n}{n}$, то почти наверняка есть изолированная вершина. Граф при этом не связан. Если $p > (1 + \varepsilon) \frac{\log n}{n}$, то граф почти наверняка связан.

Здесь «почти наверняка» означает «с вероятностью, стремящейся к 1 при $n \rightarrow \infty$ ».

Порог $\frac{\log n}{n}$ объясняется изолированными вершинами — главным препятствием связности: их ожидаемое число равно $n(1 - p)^{n-1} \approx ne^{-pn}$. Оно стремится к нулю, когда $pn \geq \log n$, — отсюда логарифм в пороге.

Примечание Почти наверняка — это не «кроме конечного числа случаев»

В теории вероятностей «почти наверняка» означает «с вероятностью 1». Исключения — исходы, где событие не случилось — образуют множество вероятности нуль. Такое множество не обязано быть конечным. Например, точка, выбранная равномерно на $[0, 1]$, почти наверняка иррациональна — но рациональных точек счётно много, просто вероятность каждой из них нулевая. «Вероятность нуль» не значит «невозможно»: все-орлы в бесконечной серии — возможный исход с вероятностью нуль.

В этом разделе у словосочетания другой, более слабый смысл: «с вероятностью, стремящейся к 1 при $n \rightarrow \infty$ ». Это свойство последовательности случайных графов, а не одного эксперимента. Для каждого конечного n вероятность может отличаться от 1 — важно лишь, что она к 1 стремится.

Случайные графы соединяют комбинаторику (подсчёт графов, глава 18), теорию графов (свойства графов, глава 9) и вероятность. Многие свойства, трудные для проверки в детерминированном графе, для случайного графа выполняются с вероятностью, близкой к 1 — что даёт и доказательства существования, и понимание «типичной» структуры графа.

§ 20.14 * Марковские цепи

До сих пор мы смотрели на вероятностные события как на однократные эксперименты — бросок кубика, раздача карт, случайный граф. Но многие процессы разворачиваются во времени: каждый следующий шаг зависит от того, где мы находимся сейчас. Такие процессы описываются цепями Маркова. Тасовка колоды, распад радиоактивного ядра, завтрашняя погода и следующее слово в тексте — явления из разных миров, а машина у них одна.

Цепь Маркова — это система с конечным числом состояний. Из каждого состояния система переходит в другое (или остаётся в том же) с заданной вероятностью. Все переходы из одного состояния в сумме дают единицу — матрица переходов P не теряет массы и полностью описывает цепь.

Марковское свойство — главный приём, делающий модель обозримой: следующий шаг зависит только от текущего состояния, а не от того, как система в него попала. Вся предыстория сжата в настоящее. Это не всегда реалистично, но часто оказывается достаточным — и делает модель вычислительно обозримой.

Если цепь работает долго, она часто приходит к равновесию — *стационарному распределению* π . Оно удовлетворяет уравнению $\pi P = \pi$. Равновесие не зависит от начального состояния: система «забывает», с чего начинала. Независимость равновесия от начального состояния — условие на цепь: она должна быть неприводимой и непериодической. Если цепь распадается на два замкнутых класса, стационарное распределение зависит от того, где система начинала, а периодичность не даёт распределению сойтись. Погодный пример — как раз неприводимая и непериодическая цепь.

Простейший пример — модель погоды с двумя состояниями: солнечно и дождливо. Обозначим их S, R . Матрица переходов:

$$P = \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix}$$

Солнечный день с вероятностью 0.9 сменяется солнечным же. Дождливый с вероятностью 0.5 остаётся дождливым.

Стационарное распределение $\pi = (\pi_S, \pi_R)$. Оно удовлетворяет $\pi P = \pi$, $\pi_S + \pi_R = 1$:

$$\begin{cases} 0.9\pi_S + 0.5\pi_R = \pi_S \\ 0.1\pi_S + 0.5\pi_R = \pi_R \end{cases}$$

Из первого уравнения: $0.5\pi_R = 0.1\pi_S \Rightarrow \pi_S = 5\pi_R$. Из нормировки и первого уравнения: $\pi = (\frac{5}{6}, \frac{1}{6})$. Пять дней из шести — солнечные, независимо от того, с какой погоды мы стартовали.

Схема этой цепи — на рис. 93: два состояния S, R , а числа на стрелках — вероятности переходов.

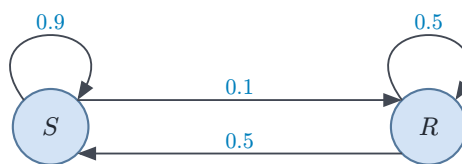


Рис. 93. Марковская цепь с двумя состояниями S, R .

ИСТОРИЯ Первая марковская цепь: «Евгений Онегин»

Цепи ввёл Андрей Марков, отвечая на спор о законе больших чисел: обязательна ли независимость событий для регулярности? Он взял первые 20 000 букв пушкинского «Евгения Онегина» и насчитал 43% гласных и 57% согласных. Если бы буквы были независимы, доля пар из двух гласных составила бы $0.43 \cdot 0.43 \approx 0.18$. Реально она около 0.06. Буквы зависимы — и всё же длинная цепь сходится к тем же 43% гласных. Марков показал: регулярность длинной цепи сохраняется и при зависимости букв — следующий символ достаточно описывать лишь текущим. Цепи, которые он построил, носят его имя.

Цепи Маркова — рабочий инструмент, а не только теория.

Алгоритм PageRank (Брин и Пейдж, 1998) моделирует случайного пользователя, блуждающего по веб-страницам. С вероятностью 0.85 он переходит по случайной ссылке с текущей страницы. С вероятностью 0.15 — «телепортируется» на любую страницу в сети. Матрица переходов между миллиардами страниц разрежена, но её стационарное распределение ($\pi = \pi P$) можно найти итерациями — и это распределение и есть рейтинг важности страниц.

Стоящая за этим мысль стара как библиотечная карточка: чем больше отметок о выдаче у книги, тем нужнее она читателям. Ссылка на страницу — такая же отметка-рекомендация. Чем больше ссылок страница раздаёт, тем меньше веса в каждой из них.

Та же математическая структура — цепь Маркова на гигантском графе — обслуживает поисковые системы. Методы Монте-Карло с марковскими цепями (MCMC) оценивают распределения в пространствах астрономической размерности, от байесовского вывода до физического моделирования.

Марковская цепь соединяет темы этой главы в одну конструкцию:

1. вероятностное пространство — состояния и переходы.
2. условная вероятность — каждый шаг.
3. линейная алгебра — матрица P и её собственный вектор.
4. предельное поведение — стационарное распределение как неподвижная точка.

Тот же объект смотрит на нас с разных сторон — верный признак того, что он заслуживает внимания.

§ 20.15 * Энтропийный метод в комбинаторике

Вероятностное пространство даёт нам распределения. Энтропия измеряет их информационное содержание — и это измерение оказывается комбинаторным инструментом. Энтропия связывает вероятность с комбинаторикой: ограничение на информацию превращается в ограничение на размер.

Для дискретной случайной величины с распределением $p_i = P(X = x_i)$ *энтропия Шеннона* определяется как

$$H(X) = - \sum_i p_i \log p_i,$$

где логарифм берётся по основанию 2 (тогда энтропия измеряется в битах). Энтропия всегда неотрицательна. Она равна нулю, когда один из исходов имеет вероятность 1 — неопределённости нет. Максимум $H(X) = \log n$ достигается на равномерном распределении вероятностей. В этом случае $p_i = \frac{1}{n}$, и неопределённость максимальна.

Идея энтропийного метода: чтобы ограничить размер комбинаторной структуры, строят случайный объект внутри неё и ограничивают его энтропию. Если каждый исход кодируется в среднем $H(X)$ битами, то

$$|\Omega| \geq 2^{H(X)}.$$

Рассмотрим граф на n вершинах. Пусть X — случайная вершина, выбранная равномерно. Энтропия выбора соседа вершины не превосходит логарифма её степени: $H(X) \leq \log d(X)$. Усреднение по всем вершинам даёт связь с числом рёбер. Из выпуклости логарифма среднее от $\log d(X)$ не превосходит логарифма средней степени. Средняя степень равна $2\frac{m}{n}$. Так энтропия локального выбора ограничивает глобальное число рёбер m . Точные формулировки даёт лемма Ширера.

Более тонкий инструмент — лемма Ширера (Shearer's lemma): энтропия совместного распределения не превосходит суммы энтропий проекций, делённых на кратность покрытия. Она обобщает субаддитивность $H(X, Y) \leq H(X) + H(Y)$ и приводит к точным оценкам в задачах о числе независимых множеств, раскрасок и совершенных паросочетаний в графах.

Вероятностное пространство задаёт меру. Энтропия измеряет, сколько информации несёт эта мера. Ограничение на информацию оборачивается ограничением на размер — и сугубо комбинаторная задача получает вероятностное решение. Темы этой главы — распределения, условная вероятность, случайные графы — при взгляде через энтропию складываются в единую картину: мера и информация суть два аспекта одной структуры.

§ 20.16 * Аксиомы Колмогорова

Определение дискретного вероятностного пространства из начала главы — рабочая версия аксиом Колмогорова. Она достаточна для конечного и счётного Ω : событие — любое подмножество, аддитивность — конечная для конечного пространства и счётная для счётного. Но два вопроса определение оставляет открытыми. Первый: какие множества вообще заслуживают имени *события*? Второй: почему вероятности складываются конечными порциями, а не счётными? Аксиоматика Колмогорова

отвечает на оба — и заодно объясняет, почему дискретное определение работает именно для счётных пространств.

События образуют структуру: семейство подмножеств, замкнутое относительно операций, которые к событиям применяются. Дополнение события — снова событие: «не выпала шестёрка». Объединение событий — тоже событие: «выпала шестёрка или семёрка». Замкнутость и есть определение *сигма-алгебры*.

ОПРЕДЕЛЕНИЕ 20.16 Сигма-алгебра событий

Сигма-алгеброй на множестве Ω называется семейство его подмножеств, удовлетворяющее трём свойствам:

1. Содержит пустое множество: $\emptyset \in \mathcal{F}$.
2. Замкнутость относительно дополнения: $A \in \mathcal{F} \Rightarrow \bar{A} \in \mathcal{F}$.
3. Замкнутость относительно счётных объединений: $A_1, A_2, \dots \in \mathcal{F} \Rightarrow \bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$.

Для конечного и счётного пространства сигма-алгеброй служит всё семейство подмножеств $\mathcal{P}(\Omega)$. Оно замкнуто относительно дополнений и объединений автоматически — поэтому в дискретной главе про сигма-алгебры можно было не думать. Понадобятся они на несчётных пространствах вроде отрезка $[0, 1]$: там не всякое подмножество может получить вероятность.

ОПРЕДЕЛЕНИЕ 20.17 Аксиомы Колмогорова

Вероятностным пространством в смысле Колмогорова называется тройка $(\Omega, \mathcal{F}, P)$:

- $\mathcal{F}$ — сигма-алгебра событий.
- $P : \mathcal{F} \rightarrow [0, \infty)$ — вероятностная мера, удовлетворяющая трём аксиомам:
 1. **Неотрицательность**: $P(A) \geq 0$ для любого события.
 2. **Нормировка**: $P(\Omega) = 1$.
 3. **Счётная аддитивность**: для попарно несовместных событий $A_1, A_2, \dots \in \mathcal{F}$:

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i).$$

Счётная аддитивность включает конечную как частный случай: добавим к конечному набору пустые множества, и лишние члены ничего не изменят. Определение дискретного пространства из начала главы — частный случай колмогоровского: Ω счётно, а сигма-алгебра состоит из всех подмножеств. Счётная аддитивность отличает колмогоровскую вероятность от наивной: бесконечные суммы разрешены, и отсюда следуют ограничения, которых не видно в конечном случае.

УТВЕРЖДЕНИЕ 20.10 Первые следствия аксиом

Из аксиом Колмогорова следует всё, чем мы пользовались в главе:

1. $P(\emptyset) = 0$.
2. $P(\overline{A}) = 1 - P(A)$.
3. **монотонность:** $A \subseteq B \Rightarrow P(A) \leq P(B)$.
4. $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
5. **неравенство Буля:** $P(\bigcup_{i=1}^n A_i) \leq \sum_{i=1}^n P(A_i)$.

Набросок доказательства

Докажем каждое следствие по отдельности.

Пустое множество: применим счётную аддитивность к последовательности $\emptyset, \emptyset, \dots$. Получим уравнение $P(\emptyset) = P(\emptyset) + P(\emptyset) + \dots$. Оно выполняется только при $P(\emptyset) = 0$. **Формула дополнения:** $\Omega = A \cup \overline{A}$, слагаемые несовместны. Тогда $1 = P(A) + P(\overline{A})$.

Монотонность: $B = A \cup (B \setminus A)$. По аддитивности $P(B) = P(A) + P(B \setminus A)$, а второе слагаемое неотрицательно. Отсюда $P(B) \geq P(A)$.

Формула объединения: $A \cup B = A \cup (B \setminus A)$. $B \setminus A = B \setminus (A \cap B)$. Дальше — аддитивность. **Неравенство Буля:** индукция по n с опорой на монотонность.

Определение из начала главы требовало $P(\emptyset) = 0$. Здесь то же условие появляется как следствие аксиом.

Пример Почему нельзя выбрать натуральное число наугад

Попробуем построить равномерное распределение на $\Omega = \mathbb{N}$. Каждое натуральное число должно иметь одну и ту же вероятность $P(\{n\}) = c$. Из нормировки $P(\mathbb{N}) = 1$, а из счётной аддитивности:

$$P(\mathbb{N}) = \sum_{n=0}^{\infty} P(\{n\}) = \sum_{n=0}^{\infty} c.$$

Ряд сходится только при $c = 0$ — но тогда сумма равна нулю, а не единице. Равномерного распределения на $\mathbb{N}$ не существует. С конечной аддитивностью противоречие не возникает: конечные порции дают ноль, а сумма по всему бесконечному семейству просто не определена. Именно счётная аддитивность превращает невозможность «выбрать натуральное число наугад» из неудобства в теорему.

Пример Вероятность попасть в одну точку

На отрезке $[0, 1]$ зададим вероятность как длину. Для промежутка $[a, b] \subseteq [0, 1]$ вероятность равна его длине:

$$P([a, b]) = b - a.$$

Неотрицательность и нормировка выполнены по построению, счётная аддитивность — свойство длины для непересекающихся промежутков. Получается честное вероятностное пространство — но несчётное, далёкое от дискретных.

Чему равна вероятность попасть ровно в точку 0.5? Точка — промежуток нулевой длины: $P(\{0.5\}) = 0$. Вероятность попасть в рациональное число тоже нулевая: рациональных чисел счётно много, а каждое по отдельности имеет вероятность нуль. И всё же $P([0, 1]) = 1$. Интуиция протестует: «если каждая точка невозможна, как вообще что-нибудь выпадает?» Противоречия нет: складывать разрешено лишь счётные семейства, а отрезок — несчётное объединение точек.

«Длина» из примера выше — это *мера Лебега*: единственная счётно-аддитивная функция, которая задаёт длину промежутков и не меняется при сдвиге. Определена она на измеримых подмножествах $\mathbb{R}$ — тех самых, что мы обсуждали в разделе о сигма-алгебрах.

ОПРЕДЕЛЕНИЕ 20.18 Мера Лебега

Мерой Лебега называется функция μ с тремя свойствами:

1. **Согласованность с длиной:** $\mu([a, b]) = b - a$ для любого промежутка.
2. **Инвариантность к сдвигу:** $\mu(A + x) = \mu(A)$ для любого сдвига.
3. **Счётная аддитивность:** для попарно не пересекающихся измеримых $A_1, A_2, \dots$ выполнено

$$\mu\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mu(A_i).$$

Существование и единственность такой функции — нетривиальная теорема теории меры. На отрезке $[0, 1]$ мера Лебега и есть вероятность-длина из примера выше: полная мера отрезка равна 1. Здесь важнее следствие.

УТВЕРЖДЕНИЕ 20.11 Счётные множества имеют меру нуль

Если множество $A = \{a_1, a_2, \dots\}$ счётно, то его мера равна нулю.

Доказательство

Докажем, разбив множество на отдельные точки и применив счётную аддитивность.

Точка — промежуток нулевой длины, поэтому мера каждой точки a_i равна нулю. По счётной аддитивности:

$$\mu(A) = \sum_{i=1}^{\infty} \mu(\{a_i\}) = 0.$$

Из этой простой теоремы следуют три факта о случайном вещественном числе из $[0, 1]$. Все три опираются на одну схему: счётное множество имеет меру нуль. Рациональных чисел счётно много — случайное число почти наверняка (с вероятностью 1) иррационально. Алгебраических чисел (корней многочленов с целыми коэффициентами) тоже счётно много — случайное число почти наверняка трансцендентно. Наконец, вычислимых вещественных чисел счётно много: каждое задаётся конечной программой, а программ счётно число (та же диагональная картина, что в главе 25). Поэтому случайное вещественное число почти наверняка *невычислимо*.

Пример Случайное число невычислимо

Выберем точку на $[0, 1]$ равномерно. Она почти наверняка иррациональна: рациональных чисел счётно много. Она почти наверняка трансцендентна: алгебраических чисел счётно много. И она почти наверняка невычислима: вычислимых чисел счётно много.

Ирония в том, что предъявить такой пример мы не можем: предъявить вещественное число — значит указать алгоритм, выдающий его десятичные знаки, то есть вычислить его, а вычислимых чисел — лишь счётное множество. Существование невычислимых чисел не нуждается в явной конструкции: несчётность вещественных против счётности алгоритмов (глава 7 и глава 25) уже гарантирует его. Вычислимые числа при этом *плотны* в $[0, 1]$: между любыми двумя точками лежит вычислимое число, а рациональные подходят к каждой точке сколь угодно близко. Плотность и мера нуль не противоречат друг другу: одна говорит, где множество лежит, другая — какую долю оно занимает. Поэтому «почти наверняка невычислимо» не отменяет плотности: доля вычислимых чисел — ноль, хотя в каждой окрестности их бесконечно много.

Обратное к теореме неверно: мера нуль не означает счётность. Канторово множество — классический пример: из $[0, 1]$ последовательно выкидывают средние трети, и то, что остаётся, несчётно, но имеет меру нуль. Числа канторова множества — те, в троичной записи которых нет единицы. Их столько же, сколько двоичных последовательностей, а суммарная длина выкинутых третей равна единице.

Замечание Почему не всякое подмножество — событие

На $[0, 1]$ с вероятностью-длиной нельзя приписать вероятность каждому подмножеству. Существуют *неизмеримые* множества: *измеримым* называется множество, которому можно приписать вероятность, не ломая ни одной аксиомы. Первый пример неизмеримого множества построил Витали (1905), используя аксиому выбора. Конструкция с полным доказательством — в главе 22.

Поэтому события и обязаны быть сигма-алгеброй: множества, на которых мера не уместается, исключаются заранее. В дискретной главе этой проблемы нет — на счётном Ω все подмножества измеримы.

Три случая из этого раздела удобно свести в одну таблицу.

Пространство	Аддитивность	События	Пример
Конечное	конечная	все подмножества	монета, кубик
Счётное	счётная	все подмножества	выбор из $\mathbb{N}$
Несчётное	счётная	сигма-алгебра	отрезок $[0, 1]$

Пример Парадокс Бертрانا

История из начала главы: у вопроса «случайная хорда длиннее стороны вписанного равностороннего треугольника?» есть три естественных ответа — $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$. «Случайная хорда» определяется по-разному: двумя случайными концами на окружности, случайной точкой внутри круга как серединой хорды, случайной точкой на случайном радиусе. Наивная геометрическая интуиция видит парадокс: один вопрос, три ответа. Аксиоматика снимает напряжение: три ответа — три разных *вероятностных пространств*. Аксиомы не решают, какое пространство «правильное», — они обязывают указать, о каком пространстве идёт речь. Парадокс превращается в напоминание: сначала пространство, потом вероятности.

Аксиомы Колмогорова — минимальный перечень того, что обязана уметь любая разумная вероятностная мера. Дискретное определение из начала главы — их частный случай для счётных пространств. За пределами дискретных пространств та же тройка $(\Omega, \mathcal{F}, P)$ становится фундаментом теории меры и интеграла — основы современной теории вероятностей.

§ 20.17 Итоги главы

Глава началась с вопроса, как из незнания исхода вырастает точное знание о мере исхода, и ответ сложился из двух ходов. Сначала эксперимент описывается вероятностным пространством — парой из множества исходов и меры на событиях. В равновероятном случае задача сводится к комбинаторному подсчёту $P(A) = |A|/|\Omega|$ (глава 18). Затем новая информация пересчитывает меру: условная вероятность сужает пространство исходов, а формула Байеса переводит априорную оценку гипотезы в апостериорную. Положительный тест на редкую болезнь поднимает вероятность болезни куда меньше, чем ожидает интуиция.

Случайные величины переводят исходы в числа: матожидание указывает центр распределения, дисперсия — его разброс. Для распределений Бернулли, биномиального и геометрического обе характеристики выражаются через параметры явно.

Биномиальная величина приносит в среднем np успехов, а геометрическая ждёт первого успеха в среднем $\frac{1}{p}$ испытаний. Линейность матожидания разбирает сложную величину на индикаторы без каких-либо условий независимости, а неравенства концентрации — от Маркова до экспоненциальных хвостов Чернова — отвечают, как часто величина уходит далеко от среднего. На этой связке держится анализ рандомизированных алгоритмов: ошибка Фривалда падает как $\frac{1}{2^k}$, ожидаемое время быстрой сортировки остаётся $O(n \log n)$ на любом входе.

Тот же язык описывает структуры, а не только числа. Парадокс дней рождения — следствие квадратичного числа пар, и тот же подсчёт ускоряет поиск коллизий в хеш-функциях. Вероятностный метод доказывает существование объекта без явного построения: достаточно положительной вероятности случайно выбрать его. Случайные графы показывают, как свойства возникают скачком на пороге (глава 9), а цепь Маркова сохраняет регулярность при зависимых шагах: равновесие описывает стационарное распределение. Энтропия замыкает картину со стороны информации: ограничение на информационное содержание распределения оборачивается ограничением на размер структуры.

Проверка Проверьте себя

1. Почему $P(A | B)$, $P(A \cap B)$ — разные величины?
2. Что даёт линейность матожидания и почему для неё не нужна независимость?
3. Почему парадокс дней рождения ускоряет поиск коллизий в хеш-функциях?
4. Как формула Байеса пересчитывает априорную вероятность в апостериорную?
5. Чем независимость отличается от несовместности событий?
6. Почему из положительной вероятности свойства следует существование объекта с этим свойством?
7. Почему для дисперсии суммы нужна независимость, а для матожидания — нет?

Что дальше?

Комбинаторика научила нас считать, вероятность — взвешивать. Теперь мы сделаем следующий шаг: научимся упаковывать целые последовательности в формальные степенные ряды и извлекать из них явные формулы. Вероятностная производящая функция кодирует распределение случайной величины в одном ряде. В главе 21 мы изучим производящие функции, связывающие дискретную математику с анализом.

§ 20.18 Упражнения

Вероятностное пространство.

1. Бросаем два кубика. Какова вероятность, что сумма равна 8? Что сумма не меньше 10?
2. Из стандартной колоды в 52 карты вытягивают одну карту. Какова вероятность, что это туз или карта пиковой масти?
3. В урне 3 красных и 5 синих шаров. Вынимаем два шара без возвращения. Какова вероятность, что оба красные? Что хотя бы один красный?
4. * Случайный граф $G(4, \frac{1}{2})$: граф на 4 вершинах, каждое из 6 рёбер включается независимо. Какова вероятность, что граф связан?

Условная вероятность и независимость.

5. Бросаем три симметричные монеты. Событие A — «выпало ровно два орла». Событие B — «первая монета — орёл». Найдите $P(A)$, $P(B)$, $P(A \cap B)$, $P(A | B)$. Независимы ли A , B ?
6. В семье двое детей. Известно, что один из них — мальчик. Какова вероятность, что оба — мальчики? Считаем рождения мальчика и девочки равновероятными и независимыми.
7. В урне 4 белых и 6 чёрных шаров. Вынимаем два шара с возвращением. Какова вероятность, что они разного цвета?
8. * Алиса, Боб и Чарли бросают симметричную монету по очереди: Алиса, Боб, Чарли, Алиса и так далее. Выигрывает тот, у кого первого выпадет орёл. Найдите вероятность выигрыша для каждого.

Формула Байеса.

9. Тест на редкое заболевание даёт положительный результат с вероятностями 0.99, 0.02: для больного и здорового соответственно. Болезнь встречается у 0.1% населения. Найдите вероятность, что человек с положительным тестом действительно болен.
10. * В задаче Монти Холла: три двери, за одной — автомобиль, за двумя — козы. Вы выбрали дверь, ведущий открыл дверь с козой и предлагает переключиться. Почему переключение выигрывает в $\frac{2}{3}$ случаев, а не в половине?

Случайные величины и линейность матожидания.

11. Случайная величина принимает значения 0, 1, 2. Вероятности: 0.2, 0.5, 0.3. Обозначим её через X . Найдите $E[X]$, $\text{Var}[X]$.
12. Бросаем 10 игральные кости. Найдите ожидаемое значение суммы выпавших очков.
13. Случайная перестановка колоды из 52 карт. Найдите ожидаемое число карт, оказавшихся на той же позиции, что и в исходной колоде.

14. * Используя линейность математического ожидания, найдите ожидаемое число пустых ящиков при бросании m шаров в n ящиков (каждый шар равновероятно попадает в любой ящик).

Неравенства концентрации.

15. Неотрицательная случайная величина имеет математическое ожидание $E[X] = 20$. Что можно гарантировать про $P(X \geq 60)$?
16. Случайная величина X имеет математическое ожидание 10 и дисперсию 9. Оцените $P(|X - 10| \geq 6)$.
17. Бросаем кубик n раз. Найдите такое n , при котором неравенство Чебышёва гарантирует

$$P\left(\left|\frac{S_n}{n} - 3.5\right| \geq 0.1\right) \leq 0.05.$$

18. * Подбрасываем честную монету $n = 100$ раз. Оцените $P(X \geq 75)$ неравенством Чебышёва и неравенством Чернова. Какая оценка точнее и почему?

Парадокс дней рождения и вероятностный метод.

19. В комнате 30 человек. Какова вероятность, что хотя бы у двоих дни рождения в один день? Вычислите точно и сравните с приближением $1 - e^{-\frac{\binom{30}{2}}{365}}$.
20. * В задаче о шляпах: n человек сдают шляпы, шляпы возвращаются случайно. Для пяти человек вычислите вероятность, что ровно k человек получат свои шляпы. Переберите значения $k = 0, 1, 2, 3, 4, 5$. Убедитесь, что сумма равна 1.
21. * Докажите вероятностным методом: существует граф на $n = \lfloor 2^{\frac{k}{2}} \rfloor$ вершинах без клики и без независимого множества такого же размера.
22. * Бросаем игральную кость до первого выпадения шестёрки. Найдите ожидаемое число бросков. Подсказка: рассмотрите первый бросок и выразите $E[N]$ через само себя.

ПРОЕКТ Поиск коллизий в усечённых хеш-функциях

Парадокс дней рождения ускоряет поиск коллизий в хеш-функциях. Наивный перебор требовал бы 2^m попыток. С учётом парадокса хватает примерно $2^{\frac{m}{2}}$ — корня из размера кодомона. Проверьте это предсказание предъявлением: обрежьте хеш FNV-1a до m бит, наивным поиском найдите пару разных строк с одинаковым хешем и покажите, что игрушечный дедупликатор, считающий файлы равными по хешу, принимает их за равные. Сверьте эксперимент с формулой дня рождения и свяжите результат с криптографией.

① Усечённый хеш

Напишите ручную хеш FNV-1a (64 бита) и обрежьте его до m младших бит, где m от 8 до 40.

② Наивный поиск коллизии

Храните увиденные хеши и считайте вставки до первого повтора. Сверьте кривую числа вставок от m с ориентиром $2^{\frac{m}{2}}$. Проверьте, что она не растёт как 2^m .

③ Сверка с формулой дня рождения

Сравните вероятность коллизии как функцию числа попыток с измеренной долей коллизий по множеству прогонов. Теоретическая кривая должна совпасть с измеренной.

④ Предъявление коллизии

Найдите две осмысленно разные строки с одинаковым m -битным хешем. Игрушечный дедупликатор, считающий файлы равными по хешу, должен принять их за равные.

⑤ Связь с криптографией

Свяжите результат с криптографией (19). Объясните, почему для полного 256-битного хеша наивному поиску потребовалось бы 2^{128} попыток.

Итог: пара разных строк с одним усечённым хешем, принятая дедупликатором за равные, вместе с измерением закона $\sqrt{N}$ и объяснением, на чём держится стойкость криптографического хеша.

XXI

Производящие функции

На стол падают две игральные кости. Сколькими способами можно получить сумму 7? Можно выписать все 36 исходов и посчитать: $(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)$ — шесть способов. Для суммы 9 — четыре способа. Для каждой суммы — отдельный подсчёт.

В предыдущей главе 20 те же кости отвечали на другой вопрос: «с какой вероятностью», а не «сколькими способами». Для суммы 7 вероятность равна $\frac{6}{36}$. Для суммы 9 — $\frac{4}{36}$. Вероятность — отношение двух подсчётов: благоприятных исходов ко всем. Счёт стоит под вероятностью, как фундамент под домом. Комбинаторика (глава 18) дала для такого счёта правила суммы и произведения, формулы сочетаний и перестановок.

Дьёрдь Поля сравнивал такой подход с горстью мелких предметов, которые приходится перебирать по одному. Производящая функция (*generating function*) — это «мешок», в который можно упаковать всю последовательность сразу. Одно умножение многочленов — и ответ готов для *всех* сумм одновременно:

$$(x + x^2 + x^3 + x^4 + x^5 + x^6)^2 = x^2 + 2x^3 + 3x^4 + \dots + 6x^7 + \dots + x^{12}.$$

Коэффициент при x^n — количество способов получить сумму n . Ответы для всех сумм — коэффициенты от x^2 до x^{12} — лежат в одном многочлене. Две кости — 36 исходов, перебор ещё терпим. Три кости — 216 исходов, и выписывать их по одному уже утомительно. А многочлен возводится в куб тем же умножением.

Конструкцию легко проверить одной подстановкой: при $x = 1$ многочлен одной кости обращается в 6. Квадрат этой шестёрки — $6 \cdot 6 = 36$ — равен сумме всех коэффициентов, то есть числу всех исходов. Подстановка суммирует коэффициенты: в точке $x = 1$ любая степень x^n обращается в единицу.

Это и есть идея производящей функции: упаковать всю последовательность в один формальный ряд. В x не подставляют значений и о сходимости не спрашивают: смысл ряда исчерпывается его коэффициентами.

Упаковка окупается сразу: операции над последовательностями становятся алгебраическими операциями над рядами, и рекуррентное соотношение превращается в уравнение. Вместо бесконечной цепочки равенств — одно алгебраическое выражение.

Инструмент не ограничен игральными костями. Ожидаемое число сравнений быстрой сортировки — рекуррента, которую производящие функции помогают замкнуть. Распределения дискретных случайных величин кодируются производящими функциями, и свёртка распределений становится произведением функций. Формальные ряды подводят и к бесконечному: работа с целыми последовательностями — шаг к вопросам о том, как устроены числа (глава 22).

Сама по себе упаковка — только новая запись. Сила появляется, когда ряд начинают комбинировать: складывать, умножать, делить. Тогда структура последовательности — её рекуррента, её свёртка — становится алгебраическим объектом. Как мешок с последовательностью превращается в объект, с которым работает алгебра?

ОБЗОР ГЛАВЫ

В этой главе мы определим обычную производящую функцию и увидим, как сдвиг, свёртка и частичные суммы получают простые алгебраические выражения. Рекуррентное соотношение превратится в уравнение, а разложение на простейшие дроби и извлечение коэффициента дадут замкнутую формулу — от чисел Фибоначчи до Каталана. Для помеченных объектов понадобится экспоненциальная производящая функция, где в свёртке восстанавливается биномиальный коэффициент. Приложения посчитают композиции числа, выведут среднее число сравнений быстрой сортировки и опишут вероятностные распределения через производящие функции. Факультативная ступень в конце подведёт нас к комбинаторным видам: конструкции переводятся в уравнения, а особенности функции определяют асимптотику коэффициентов. Задача о двух костях, с которой мы начали, окажется простейшим случаем того же приёма.

После этой главы вы сможете:

1. строить производящую функцию по последовательности и извлекать коэффициенты;
2. переводить сдвиг, свёртку и частичные суммы в алгебраические операции;
3. решать рекуррентные соотношения уравнениями на производящие функции;
4. пользоваться экспоненциальными производящими функциями для помеченных объектов;
5. применять производящие функции к комбинаторике, анализу алгоритмов и распределениям.

§ 21.1 Обычные производящие функции

Вернёмся к игральным костям из открытия. Одну кость мы закодировали многочленом $D(x) = x + x^2 + x^3 + x^4 + x^5 + x^6$: показатель степени — сумма на грани, коэффициент — количество способов её получить. Две кости — произведение $D(x)^2$. Коэффициент при x^n даёт ответ для суммы n . Для суммы 7: $[x^7]D(x)^2 = 6$.

ИСТОРИЯ От формальных рядов к аналитической комбинаторике

Леонард Эйлер в «*Introductio in analysin infinitorum*»¹⁵⁴ первым применил степенные ряды как инструмент комбинаторного анализа, в частности для подсчёта разбиений чисел. Карл Бойер назвал этот двухтомник величайшим учебником математики нового времени. Он свободно манипулировал бесконечными произведениями и суммами, не заботясь о сходимости. Для Эйлера ряд был формальным алгебраическим объектом — с ним обращались как с многочленом, и это работало. Строгое обоснование — теория сходимости Коши, затем Вейерштрасса — пришло позже, в XIX веке, и лишь подтвердило корректность результатов Эйлера задним числом. Так формальные ряды стали инструментом комбинаторного анализа.

Пьер-Симон Лаплас в «*Théorie analytique des probabilités*»¹⁵⁵ систематически использовал производящие функции для вероятностных распределений. Он показал, что свёртка распределений соответствует произведению их производящих функций — факт, который мы теперь используем для анализа алгоритмов. Лаплас превратил эйлеровский трюк в систематический метод.

В XX веке метод распался на две ветви: алгебраическую (формальные степенные ряды, комбинаторные виды Андре Жуайяля, 1981) и аналитическую (асимптотика коэффициентов через особенности в комплексной плоскости). Филипп Флажолет и Роберт Седжвик в «*Analytic Combinatorics*»¹⁵⁶ объединили эти ветви в единую теорию: расположение и тип особенностей производящей функции определяют асимптотический рост её коэффициентов. Метод производящих функций прошёл путь от формального трюка Эйлера до универсального инструмента анализа алгоритмов.

21.1.1 Определение ОПФ

Идея производящей функции восходит к Лапласу и Эйлеру: вместо того чтобы работать с последовательностью (a_n) напрямую, мы упаковываем её в коэффициенты степенного ряда. Сдвиг, свёртка и суммирование получают на этом языке простые алгебраические выражения.

ОПРЕДЕЛЕНИЕ 21.1 Обычная производящая функция

Формальный степенной ряд

$$A(x) = \sum_{n=0}^{\infty} a_n x^n$$

¹⁵⁴L. Euler. «*Introductio in analysin infinitorum*». 1748.

¹⁵⁵P.-S. Laplace. «*Théorie analytique des probabilités*». 1812.

¹⁵⁶P. Flajolet, R. Sedgewick. «*Analytic Combinatorics*». 2009.

называется **обычной производящей функцией** (ОПФ) последовательности $(a_n)_{n=0}^{\infty}$.

Коэффициент при x^n извлекается обозначением:

$$[x^n]A(x) = a_n.$$

Производящая функция собирает всю бесконечную последовательность в один объект. Символ x — только позиция для счёта: раскрытие ряда в точности повторяет комбинаторную конструкцию. Вопросы сходимости ряда не имеют значения. Коэффициент при x^n — ответ для размера n . Так вместо бесконечного списка $a_0, a_1, a_2, \dots$ мы получаем один символ $A(x)$, который отвечает разом на все вопросы о числе объектов размера n .

Примечание Почему формальные ряды?

Почему формальные степенные ряды, а не функции комплексного переменного? Альтернатива — работать со сходящимися рядами и использовать аппарат анализа. Но тогда последовательность $n!$ оказалась бы вне игры: её обычная производящая функция $\sum n!x^n$ имеет нулевой радиус сходимости. Между тем та же последовательность совершенно законна в формальном смысле (а её экспоненциальная производящая функция, ЭПФ, $\frac{1}{1-x}$ сходится при $|x| < 1$).

Формальный подход разделяет две задачи: алгебраическую работу с коэффициентами и аналитический вопрос о сходимости. Первая интересует нас в этой главе. Вторая возвращается позже, на этапе получения асимптотик, но уже как инструмент, а не как ограничение.

21.1.2 Операции над производящими функциями

Сила ОПФ — в том, что у каждой операции над последовательностью находится знакомый алгебраический двойник. Секрет этого двойника — простое тождество $x^a \cdot x^b = x^{a+b}$: показатели складываются. Выбор одного слагаемого из каждого множителя даёт один член произведения, а его показатель — сумму показателей. Поэтому умножение многочленов считает комбинации: каждый способ выбрать по одному слагаемому из каждого множителя даёт отдельный вклад в коэффициент.

УТВЕРЖДЕНИЕ 21.1 Операции над ОПФ

- **Сложение:** $A(x) + B(x) = \sum (a_n + b_n)x^n$ — суммирование последовательностей поэлементное.
- **Сдвиг вправо:** умножение на x вставляет ноль в начало последовательности.

$$xA(x) = \sum a_n x^{n+1} = \sum a_{n-1} x^n.$$

- **Сдвиг влево:** деление на x (после удаления константы) сдвигает последовательность влево.

$$\frac{A(x) - a_0}{x} = \sum a_{n+1} x^n.$$

• **Свёртка:**

$$A(x)B(x) = \sum_{n=0}^{\infty} c_n x^n$$

$$c_n = \sum_{i=0}^n a_i b_{n-i},$$

произведение функций соответствует свёртке последовательностей.

• **Частичные суммы:**

$$\frac{A(x)}{1-x} = \sum_{n=0}^{\infty} s_n x^n$$

$$s_n = \sum_{i=0}^n a_i,$$

деление на $(1-x)$ даёт последовательность частичных сумм.

Набросок доказательства

Докажем каждое правило по отдельности.

Сложение следует непосредственно из определения суммы рядов.

Сдвиг вправо: умножение на x увеличивает степень каждого члена на 1 — последовательность сдвигается, в позицию 0 встает 0.

Сдвиг влево: $\frac{A(x)-a_0}{x} = \sum_{n=1}^{\infty} a_n x^{n-1} = \sum_{n=0}^{\infty} a_{n+1} x^n.$

Свёртка: произведение рядов $\sum a_i x^i \cdot \sum b_j x^j = \sum_{i,j} a_i b_j x^{i+j}$. Группируем слагаемые с одинаковым $i+j=n$, и коэффициент при x^n равен $\sum_{i=0}^n a_i b_{n-i}$.

Частичные суммы: $A(x) \cdot (1-x)^{-1} = (\sum a_n x^n)(\sum x^m) = \sum_{n,m} a_n x^{n+m}$. Коэффициент при x^k равен $\sum_{n=0}^k a_n$.

Сдвиги — это основной инструмент для перевода рекуррентных соотношений в уравнения. Свёртка открывает комбинаторные задачи, где объект составляется из двух независимых частей. Геометрически это особенно наглядно: рис. 94 показывает, как каждая диагональная полоса суммирует вклады в один коэффициент c_n .

		a_0	a_1	a_2	a_3	a_4	
		2	1	3	0	1	
b_0	1	2	1	3	0	1	c_0
b_1	3	6	3	9	0	3	c_1
b_2	2	4	2	6	0	2	c_2
b_3	0	0	0	0	0	0	c_3
							c_4
							c_5
							c_6
							c_7

$$c_n = \sum_{i+j=n} a_i b_j$$

Рис. 94. Свёртка последовательностей.

Пример Свёртка двух рядов из единиц

Возьмём $A(x) = B(x) = \frac{1}{1-x}$: обе последовательности состоят из одних единиц. По правилу свёртки коэффициент произведения:

$$c_n = \sum_{k=0}^n a_k b_{n-k} = \sum_{k=0}^n 1 \cdot 1 = n + 1.$$

С другой стороны, $A(x)B(x) = \frac{1}{(1-x)^2}$. Коэффициент $[x^n] \frac{1}{(1-x)^2} = n + 1$ — это следует из дифференцирования геометрической прогрессии. Ответы совпали.

Число $n + 1$ — это количество пар $(k, n - k)$. Первая координата пробегает все значения от 0 до n , вторая определяется однозначно. Это первый пример того, как произведение рядов считает комбинаторные пары.

21.1.3 Базовые ряды

Некоторые ОПФ встречаются настолько часто, что их полезно знать наизусть. Большинство из них выводятся из формулы суммы геометрической прогрессии:

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}.$$

Ряд $(1+x)^k$ при целом $k \geq 0$ — биномиальная теорема (конечная сумма). При отрицательном целом или нецелом k это бесконечный ряд с обобщёнными биномиальными коэффициентами.

УТВЕРЖДЕНИЕ 21.2 Стандартные ОПФ

Последовательность a_n	ОПФ $A(x)$
$a_n = 1$	$\frac{1}{1-x}$
$a_n = n$	$\frac{x}{(1-x)^2}$
$a_n = \binom{k}{n}$	$(1+x)^k$
$a_n = \binom{n+k-1}{n}$	$\frac{1}{(1-x)^k}$

Последовательность a_n	ОПФ $A(x)$
$a_n = c^n$	$\frac{1}{1-cx}$

Набросок доказательства

Докажем каждую строку таблицы по отдельности.

Геометрическая прогрессия $\frac{1}{1-x} = \sum x^n$ даёт первую строку таблицы: $a_n = 1$.

Дифференцирование: $\frac{d}{dx} \frac{1}{1-x} = \frac{1}{(1-x)^2} = \sum nx^{n-1}$. Отсюда $\frac{x}{(1-x)^2} = \sum nx^n$ — последовательность $a_n = n$.

Повторное дифференцирование даёт $\frac{1}{(1-x)^k} = \sum \binom{n+k-1}{n} x^n$.

Бином Ньютона: $(1+x)^k = \sum_{n=0}^k \binom{k}{n} x^n$.

Замена $x \rightarrow cx$ в геометрической прогрессии даёт:

$$\frac{1}{1-cx} = \sum c^n x^n.$$

Коэффициенты ряда $\frac{1}{(1-x)^k}$ — это сочетания с повторениями, а сама функция порождается дифференцированием геометрической прогрессии. Ряд $(1+x)^k$ распадается на два режима. При целом $k \geq 0$ — конечный многочлен (бином Ньютона), при остальных k — бесконечный ряд с обобщёнными биномиальными коэффициентами.

21.1.4 Метод простейших дробей (*partial fraction decomposition*)

Когда рекуррента превратилась в рациональную функцию $A(x) = \frac{P(x)}{Q(x)}$, остаётся извлечь коэффициент $[x^n]A(x)$. Для этого нужно разложить $A(x)$ на простейшие дроби — сумму слагаемых вида $\frac{c}{(1-\rho x)^m}$.

АЛГОРИТМ Разложение рациональной ОПФ на простейшие дроби

1. Разложить знаменатель $Q(x)$ на линейные множители: $Q(x) = \prod (1 - \rho_i x)^{m_i}$.
2. Записать общий вид разложения: $A(x) = \sum_i \sum_{j=1}^{m_i} \frac{A_{ij}}{(1-\rho_i x)^j}$.
3. Найти коэффициенты A_{ij} методом неопределённых коэффициентов или подстановкой частных значений x .
4. Извлечь $[x^n]$ из каждой простейшей дроби: $[x^n] \frac{1}{(1-\rho x)^m} = \binom{n+m-1}{n} \rho^n$.

Пример Разложение на простейшие дроби

Пусть $A(x) = \frac{x}{1-x-x^2}$ (производящая функция чисел Фибоначчи из примера ниже). Разложим знаменатель: $1-x-x^2 = (1-\varphi x)(1-\psi x)$. Здесь $\varphi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2}$.

Общий вид разложения:

$$\frac{x}{(1-\varphi x)(1-\psi x)} = \frac{A}{1-\varphi x} + \frac{B}{1-\psi x}.$$

Умножаем обе части на знаменатель и приравниваем коэффициенты:

$$x = A(1-\psi x) + B(1-\varphi x).$$

Подстановка $x = 0$ даёт $0 = A + B$. Отсюда $B = -A$. Подстановка $x = \frac{1}{\varphi}$ даёт

$$\frac{1}{\varphi} = A\left(1 - \frac{\psi}{\varphi}\right) = A\left(\frac{\varphi - \psi}{\varphi}\right),$$

откуда $A = \frac{1}{\varphi - \psi} = \frac{1}{\sqrt{5}}$.

Итого:

$$A(x) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\varphi x} - \frac{1}{1-\psi x}\right).$$

Коэффициент: $[x^n]A(x) = \frac{1}{\sqrt{5}}(\varphi^n - \psi^n)$ — формула Бине.

В первом примере корни знаменателя иррациональны, и подстановка частных значений вела к выражениям с $\sqrt{5}$. Разберём ещё один случай — с целыми корнями: он проще, и на нём удобно проследить, откуда берётся типичная ошибка в коэффициентах.

Пример Разложение функции $\frac{1}{(1-x)(1-2x)}$

Разложим на простейшие дроби функцию $A(x) = \frac{1}{(1-x)(1-2x)}$. Знаменатель уже разложен: $\rho_1 = 1, \rho_2 = 2$.

Общий вид:

$$\frac{1}{(1-x)(1-2x)} = \frac{C}{1-x} + \frac{D}{1-2x}.$$

Умножаем на знаменатель:

$$1 = C(1-2x) + D(1-x).$$

Подстановка $x = 1$ даёт $1 = -C$. Отсюда $C = -1$. Подстановка $x = \frac{1}{2}$ даёт $1 = \frac{D}{2}$. Отсюда $D = 2$.

Итого:

$$\frac{1}{(1-x)(1-2x)} = \frac{2}{1-2x} - \frac{1}{1-x}.$$

Разворачиваем каждую дробь в геометрический ряд:

$$\frac{1}{1-2x} = \sum_{n=0}^{\infty} 2^n x^n$$

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n.$$

Собираем коэффициент:

$$[x^n]A(x) = 2 \cdot 2^n - 1 = 2^{n+1} - 1.$$

Проверка: при $n = 0, 1, 2, 3$ получаем 1, 3, 7, 15. Это $2^{n+1} - 1$.

ЛОВУШКА Сдвиг и потерянный множитель

Две ошибки одного рода: коэффициент читается по сходству, а не извлекается правилом. Пара из таблицы базовых рядов, $\frac{1}{(1-x)^2} = \sum_{n=0}^{\infty} (n+1)x^n$ и $\frac{x}{(1-x)^2} = \sum_{n=0}^{\infty} nx^n$, различается сдвигом. Умножение на x сдвигает каждый коэффициент на одну позицию вправо. Читать обе функции как $\sum nx^n$ — значит потерять разницу между n и $n+1$. В разложении выше числитель 2 при $\frac{2}{1-2x}$ находится подстановкой частных значений x , и его потеря даёт $2^n - 1$ вместо $2^{n+1} - 1$. Обе ошибки ловит проверка на малом n : коэффициенты 0 и 1 при $n = 0$ различают сдвинутые ряды, а расхождение 3 против 1 при $n = 1$ ловит потерянный множитель.

Если знаменатель $Q(x)$ содержит множитель $(1-\rho x)^m$ с $m > 1$, в разложении появляется несколько слагаемых. Общий вид: $\frac{A_1}{1-\rho x} + \frac{A_2}{(1-\rho x)^2} + \dots + \frac{A_m}{(1-\rho x)^m}$. Коэффициенты находятся из общей формулы: $[x^n] \frac{1}{(1-\rho x)^m} = \binom{n+m-1}{n} \rho^n = \binom{n+m-1}{m-1} \rho^n$. Пример рекурренты с кратными корнями разобран в следующем разделе.

Проверка Операции над ОПФ и базовые ряды

1. Что делает с последовательностью умножение её ОПФ на x , и зачем этот сдвиг при переводе рекурренты в уравнение?
2. Как из геометрической прогрессии получается ОПФ последовательности $a_n = n$?
3. Чем она отличается от ОПФ последовательности $a_n = n + 1$?

21.1.5 Решение рекуррентных соотношений через ОПФ

Метод ОПФ превращает рекурренту в явную формулу за семь шагов.

Идея — умножить рекурренту на x^n , просуммировать по всем n и получить уравнение на производящую функцию $A(x)$.

АЛГОРИТМ Метод ОПФ

1. Записать рекуррентное соотношение для всех $n \geq k$, где k — порядок рекурренты.
2. Умножить обе части на x^n и просуммировать по всем $n \geq k$.
3. Выразить получившиеся суммы через $A(x)$ с помощью операций сдвига.
4. Решить алгебраическое (или дифференциальное) уравнение относительно $A(x)$.
5. Разложить $A(x)$ на простейшие дроби (для рациональных ОПФ).
6. Разложить каждую дробь в степенной ряд.
7. Прочитать коэффициент: $a_n = [x^n]A(x)$.

Ниже — четыре примера, от линейных рекуррент к нелинейным. На практике ряды удобно писать один под другим, выравнивая одинаковые степени x : сдвиг выглядит как сдвиг строки, сложение — как сложение столбцов. Сравнение позиций показывает, какое слагаемое рекурренты во что сдвигается.

Пример Числа Фибоначчи через ОПФ

Рекуррента:

$$F_n = F_{n-1} + F_{n-2}, \quad (n \geq 2), \quad F_0 = 0, \quad F_1 = 1.$$

Пусть $F(x) = \sum_{n=0}^{\infty} F_n x^n$. Умножаем рекурренту на x^n и суммируем по $n \geq 2$:

$$\sum_{n=2}^{\infty} F_n x^n = \sum_{n=2}^{\infty} F_{n-1} x^n + \sum_{n=2}^{\infty} F_{n-2} x^n.$$

Левая часть: $F(x) - F_0 - F_1 x = F(x) - x$. Первое слагаемое правой части: $x \sum_{n=2}^{\infty} F_{n-1} x^{n-1} = x(F(x) - F_0) = xF(x)$. Второе слагаемое: $x^2 \sum_{n=2}^{\infty} F_{n-2} x^{n-2} = x^2 F(x)$.

Уравнение:

$$F(x) - x = xF(x) + x^2 F(x)$$

$$F(x) = \frac{x}{1 - x - x^2}.$$

Дробь $\frac{x}{1-x-x^2}$ ничем не напоминает числа Фибоначчи — и всё же это тот самый мешок Поля: разложение в ряд возвращает их все. В этом и состоит сила упаковки: последовательность спрятана в алгебраическое выражение, с которым можно работать, не глядя на неё.

Разложение этого выражения на простейшие дроби и извлечение коэффициента уже проделаны в примере «Разложение на простейшие дроби» выше. Результат — формула Бине:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}.$$

Примечание ОПФ против характеристического уравнения

Сопоставим этот вывод с методом характеристического уравнения из главы 18. Тот метод требует:

1. записать характеристический многочлен,
2. найти его корни,
3. записать общее решение как линейную комбинацию,
4. подставить начальные условия и решить систему.

Каждый шаг требует отдельного обоснования: почему решение ищется в виде λ^n ? Почему общее решение — линейная комбинация?

Метод ОПФ отвечает на эти вопросы автоматически:

1. умножаем рекурренту на x^n и суммируем — получаем алгебраическое уравнение без угадывания вида решения.
2. знаменатель $1 - x - x^2$ — это характеристический многочлен с заменой $\lambda \rightarrow \frac{1}{x}$.
3. разложение на простейшие дроби автоматически даёт коэффициенты — система уравнений решается алгебраически.

Метод характеристического уравнения — частный случай метода ОПФ для линейных рекуррент. Но ОПФ работает и для нелинейных рекуррент (Каталан), где характеристическое уравнение бессильно.

Числа Фибоначчи иллюстрируют линейный случай с простыми корнями. Следующий пример показывает, что происходит при кратных корнях.

Пример Рекуррента с кратным корнем

$$a_n = 4a_{n-1} - 4a_{n-2}, \quad (n \geq 2), \quad a_0 = 1, \quad a_1 = 4.$$

Пусть $A(x) = \sum_{n=0}^{\infty} a_n x^n$. Умножаем рекурренту на x^n и суммируем по $n \geq 2$:

$$\sum_{n=2}^{\infty} a_n x^n = 4 \sum_{n=2}^{\infty} a_{n-1} x^n - 4 \sum_{n=2}^{\infty} a_{n-2} x^n.$$

Левая часть: $A(x) - a_0 - a_1 x = A(x) - 1 - 4x$. Первое слагаемое правой части: $4x(A(x) - a_0) = 4xA(x) - 4x$. Второе слагаемое: $-4x^2 A(x)$.

Уравнение:

$$A(x) - 1 - 4x = 4xA(x) - 4x - 4x^2 A(x),$$

$$A(x)(1 - 4x + 4x^2) = 1$$

$$A(x) = \frac{1}{1 - 4x + 4x^2} = \frac{1}{(1 - 2x)^2}.$$

Знаменатель имеет кратный корень $\rho = 2$. Кратность корня — $m = 2$. По формуле для кратного корня:

$$\begin{aligned} a_n &= [x^n] \frac{1}{(1 - 2x)^2} \\ &= \binom{n+1}{n} 2^n \\ &= (n+1)2^n. \end{aligned}$$

Проверка: $a_0 = 1 \cdot 2^0 = 1$, $a_1 = 2 \cdot 2^1 = 4$, $a_2 = 3 \cdot 2^2 = 12$. Рекуррента: $4 \cdot 4 - 4 \cdot 1 = 12$.

Кратный корень — лишь одно из усложнений в рекуррентах: правая часть может быть знакомой функцией от n вместо нуля.

Пример Неоднородная рекуррента

$$a_n = a_{n-1} + n, \quad (n \geq 1), \quad a_0 = 1.$$

Эта рекуррента неоднородна: справа стоит n , а не константа. Метод ОПФ справляется и с этим случаем.

Умножаем на x^n и суммируем по $n \geq 1$:

$$\sum_{n=1}^{\infty} a_n x^n = \sum_{n=1}^{\infty} a_{n-1} x^n + \sum_{n=1}^{\infty} n x^n.$$

Левая часть: $A(x) - a_0 = A(x) - 1$. Первое слагаемое правой части: $xA(x)$. Второе слагаемое — известный ряд: $\sum_{n=1}^{\infty} n x^n = \frac{x}{(1-x)^2}$ (см. таблицу базовых рядов).

Уравнение:

$$A(x) - 1 = xA(x) + \frac{x}{(1-x)^2},$$

$$A(x)(1-x) = 1 + \frac{x}{(1-x)^2}$$

$$A(x) = \frac{1}{1-x} + \frac{x}{(1-x)^3}.$$

Извлекаем коэффициент. Первое слагаемое: $[x^n] \frac{1}{1-x} = 1$. Второе слагаемое: $\frac{x}{(1-x)^3} = x \cdot \sum_{k=0}^{\infty} \binom{k+2}{2} x^k$. Отсюда

$$[x^n] \frac{x}{(1-x)^3} = [x^{n-1}] \frac{1}{(1-x)^3} = \binom{n+1}{2} = \frac{n(n+1)}{2}.$$

Ответ: $a_n = 1 + \frac{n(n+1)}{2}$. Проверка: $a_0 = 1, a_1 = 2, a_2 = 4, a_3 = 7$. Рекуррента: $a_1 = a_0 + 1 = 2, a_2 = a_1 + 2 = 4, a_3 = a_2 + 3 = 7$.

Три предыдущих примера — линейные рекурренты. Метод ОПФ не ограничен линейным случаем. Следующий пример — числа Каталана — показывает, как тот же подход работает для нелинейной рекурренты.

В главе 18 мы привели формулу $C_n = \frac{1}{n+1} \binom{2n}{n}$ без вывода. Теперь получим её методом ОПФ.

Пример Числа Каталана через ОПФ

Рекуррента:

$$C_0 = 1, \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i}, \quad (n \geq 0).$$

Правая часть — свёртка последовательности (C_n) с собой, то есть коэффициент $[x^n]C(x)^2$.

Откуда эта рекуррента геометрически? Каталаново число C_n считает бинарные деревья. У дерева n внутренних узлов. Удалим корень: дерево распадается на левое и правое поддеревья с $i, n-1-i$ узлами. Левое поддерево можно выбрать C_i способами. Правое — C_{n-1-i} способами. Суммируя по всем i , получаем в точности рекурренту.

Разложение проиллюстрировано на рис. 95. Удаление корня разбивает дерево C_4 на два независимых поддерева, давая в точности свёрточную сумму.

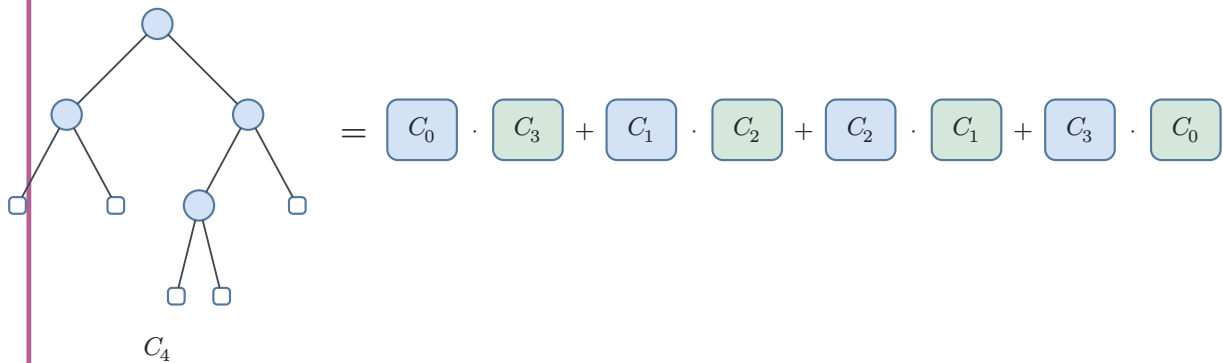


Рис. 95. Дерево C_4 и его разложение на пары $C_i C_{3-i}$

Пусть $C(x) = \sum_{n=0}^{\infty} C_n x^n$.

$$\begin{aligned}
C(x) &= C_0 + \sum_{n=0}^{\infty} C_{n+1} x^{n+1} \\
&= 1 + x \sum_{n=0}^{\infty} \left(\sum_{i=0}^n C_i C_{n-i} \right) x^n \\
&= 1 + x C(x)^2.
\end{aligned}$$

Квадратное уравнение: $x C(x)^2 - C(x) + 1 = 0$. Решение:

$$C(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

Второй корень $\frac{1 + \sqrt{1 - 4x}}{2x}$ при $x \rightarrow 0$ расходится и формальным степенным рядом не является. Остаётся корень со значением $C(0) = C_0 = 1$.

Раскладываем $\sqrt{1 - 4x}$ через обобщённый бином Ньютона:

$$\begin{aligned}
\sqrt{1 - 4x} &= (1 - 4x)^{\frac{1}{2}} \\
&= \sum_{n=0}^{\infty} \binom{\frac{1}{2}}{n} (-4x)^n.
\end{aligned}$$

Подставляем в $C(x)$:

$$C(x) = \frac{1}{2x} \left(1 - \sum_{n=0}^{\infty} \binom{\frac{1}{2}}{n} (-4x)^n \right).$$

При $n = 0$ первое слагаемое суммы равно $\binom{\frac{1}{2}}{0} (-4x)^0 = 1$ и сокращается с единицей. Остаётся:

$$\begin{aligned}
C(x) &= -\frac{1}{2x} \sum_{n=1}^{\infty} \binom{\frac{1}{2}}{n} (-4x)^n \\
&= \sum_{n=1}^{\infty} \left(-\frac{1}{2} \binom{\frac{1}{2}}{n} (-4)^n \right) x^{n-1}.
\end{aligned}$$

Обобщённый биномиальный коэффициент:

$$\begin{aligned}
\binom{\frac{1}{2}}{n} &= \frac{\left(\frac{1}{2}\right)\left(\frac{1}{2} - 1\right) \cdots \left(\frac{1}{2} - n + 1\right)}{n!} \\
&= \frac{(-1)^{n-1} \cdot 1 \cdot 3 \cdot 5 \cdots (2n - 3)}{2^n n!}.
\end{aligned}$$

Упрощаем коэффициент при x^{n-1} . Положим $m = n - 1$. Подставляя $\binom{\frac{1}{2}}{m+1}$ из формулы выше, получаем

$$C_m = (1 \cdot 3 \cdot 5 \cdots (2m - 1)) \cdot \frac{2^m}{(m + 1)m!}.$$

Произведение нечётных чисел равно $\frac{(2m)!}{2^m m!}$. Степени двойки сокращаются, и остаётся $C_m = \frac{1}{m+1} \binom{2m}{m}$.

Итого: $C_n = \frac{1}{n+1} \binom{2n}{n}$ — замкнутая форма для чисел Каталана.

Любое линейное рекуррентное соотношение с постоянными коэффициентами даёт рациональную ОПФ — отношение двух многочленов.

$A(x) = \frac{P(x)}{Q(x)}$, где знаменатель $Q(x) = 1 - c_1 x - \dots - c_k x^k$ — это характеристический многочлен, записанный в обратном порядке. Разложение этой дроби на простейшие и разложение каждой простейшей дроби в степенной ряд даёт явную формулу для a_n (как в примере с числами Фибоначчи выше).

Нелинейные рекурренты (например, $C_{n+1} = \sum C_i C_{n-i}$ для чисел Каталана) приводят к алгебраическим уравнениям на производящую функцию. В правой части возникает $A(x)^2$ — свёртка последовательности с собой. Решение такого уравнения даёт $A(x)$, содержащую квадратный корень — признак алгебраической, но не рациональной функции. Коэффициенты извлекаются через обобщённый бином Ньютона.

В обоих случаях метод ОПФ работает систематически: рекуррента $\rightarrow$ уравнение на $A(x)$ $\rightarrow$ решение $\rightarrow$ разложение в ряд $\rightarrow$ коэффициент. Вся техника держится на одном соответствии: умножение рядов $A(x)B(x) = \sum c_n x^n$, $c_n = \sum a_i b_{n-i}$ соответствует свёртке последовательностей.¹⁵⁷

Идея упаковки последовательности в коэффициенты ряда может показаться трюком. Почему алгебраические манипуляции с бесконечными суммами дают правильные комбинаторные ответы?

Потому что операции над рядами — сложение, умножение, дифференцирование — определены через коэффициенты, без обращения к пределу (см. note «Почему формальные ряды?»). Утверждение $A(x) = \frac{1}{1-x}$ означает равенство коэффициентов. Для всех индексов n выполнено $[x^n]A(x) = 1$.

Анализ возвращается на этапе асимптотики, но уже как инструмент, а не как ограничение: особенности функции в комплексной плоскости определяют рост коэффициентов. Формальное и аналитическое не противоречат друг другу — они работают на *разных уровнях* одной конструкции.

§ 21.2 Экспоненциальные производящие функции

ОПФ удобна для неразмеченных объектов (сочетания, разбиения, композиции): умножение рядов даёт свёртку. Для помеченных объектов нужна другая конструкция — экспоненциальная производящая функция.

Биномиальный коэффициент $\binom{n}{i}$ возникает при сборке помеченного объекта из частей. В обычном умножении рядов он теряется. В экспоненциальном — восста-

¹⁵⁷Н. Wilf. «generatingfunctionology». 2006. Главы 1–2.

навливаются. Переход $x^n \rightarrow \frac{x^n}{n!}$ — выбор представления, в котором комбинаторная структура помеченных объектов становится *видимой*. Множитель $\frac{1}{n!}$ в знаменателе делает ЭПФ удобной для последовательностей, содержащих факториальный рост или описывающих помеченные структуры.

ОПРЕДЕЛЕНИЕ 21.2 Экспоненциальная производящая функция

Формальный степенной ряд

$$E(x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}$$

называется **экспоненциальной производящей функцией** (ЭПФ) последовательности $(a_n)_{n=0}^{\infty}$.

Пример ЭПФ последовательности единиц

Возьмём последовательность из одних единиц: $a_n = 1$ для всех n . Подставляем в определение ЭПФ:

$$\begin{aligned} E(x) &= \sum_{n=0}^{\infty} 1 \cdot \frac{x^n}{n!} \\ &= \sum_{n=0}^{\infty} \frac{x^n}{n!} \\ &= e^x. \end{aligned}$$

Экспонента — ЭПФ последовательности единиц.

Коэффициент ЭПФ извлекается с факториалом: $a_n = n![x^n]E(x)$. Для e^x имеем $[x^n]e^x = \frac{1}{n!}$. Отсюда $a_n = n! \cdot \frac{1}{n!} = 1$ — мы вернулись к исходной последовательности.

Сравним с ОПФ: та же последовательность единиц имеет ОПФ $\frac{1}{1-x}$. Одна последовательность — два представления: $\frac{1}{1-x}$, e^x . Разница — только в выборе слагаемых: x^n , $\frac{x^n}{n!}$.

Операции над ЭПФ отражают комбинаторные операции над помеченными объектами.

УТВЕРЖДЕНИЕ 21.3 Операции над ЭПФ

- **Сдвиг влево:** $\frac{d}{dx}E(x) = \sum_{n=0}^{\infty} a_{n+1} \frac{x^n}{n!}$ — дифференцирование сдвигает последовательность на шаг влево.
- **Биномиальная свёртка:** произведение ЭПФ соответствует свёртке с биномиальными коэффициентами:

$$A(x)B(x) = \sum_{n=0}^{\infty} c_n \frac{x^n}{n!}$$

$$c_n = \sum_{i=0}^n \binom{n}{i} a_i b_{n-i}.$$

Выбираем i элементов из n для первой структуры, остальные $n - i$ — для второй.

Набросок доказательства

Докажем каждое правило по отдельности.

Сдвиг влево: $\frac{d}{dx} \sum a_n \frac{x^n}{n!} = \sum a_n n \frac{x^{n-1}}{n!} = \sum a_{n+1} \frac{x^n}{n!}.$

Биномиальная свёртка: $A(x)B(x) = \left(\sum a_i \frac{x^i}{i!} \right) \left(\sum b_j \frac{x^j}{j!} \right) = \sum_{i,j} a_i b_j \frac{x^{i+j}}{i!j!}.$ Коэффициент при $\frac{x^n}{n!}$ равен

$$\sum_{i=0}^n a_i b_{n-i} \frac{n!}{i!(n-i)!} = \sum_{i=0}^n \binom{n}{i} a_i b_{n-i}.$$

Биномиальный коэффициент естественно появляется из отношения $\frac{n!}{i!(n-i)!}.$

Зная, как операции над ЭПФ отражают комбинаторные конструкции, зафиксируем несколько базовых рядов: из них складываются более сложные структуры.

УТВЕРЖДЕНИЕ 21.4 Стандартные ЭПФ

Последовательность a_n	ЭПФ $E(x)$
$a_n = 1$	e^x
$a_n = n!$	$\frac{1}{1-x}$
$a_n = c^n$	e^{cx}
$a_n = S(n, k)$ (Стирлинга 2-го рода)	$\frac{(e^x - 1)^k}{k!}$
$a_n = B_n$ (числа Белла)	$e^{e^x - 1}$

Пример Проверка строки таблицы: $S(n, 2)$

Возьмём строку таблицы при $k = 2$. ЭПФ чисел Стирлинга второго рода равна $\frac{(e^x - 1)^2}{2}.$

Разворачиваем: $\frac{(e^x - 1)^2}{2} = \frac{e^{2x} - 2e^x + 1}{2}.$ Коэффициент при $\frac{x^n}{n!}$ равен $2^{n-1} - 1.$

Это ровно число способов разбить n элементов на два непустых блока: каждый элемент независимо выбирает блок, минус два вырожденных случая (все в одном блоке), и пополам из-за неразличимости блоков.

ЭПФ для чисел Белла — нетривиальный случай.

Примечание Числа Белла и конструкция e^{e^x-1}

ЭПФ для чисел Белла — $B(x) = e^{e^x-1}$. Она кодирует всю бесконечную последовательность чисел Белла в одном выражении. Число B_n — это количество разбиений n -элементного множества.

Откуда эта формула? Разбиение множества — это набор непустых непересекающихся блоков, порядок блоков не важен. Структура «непустое множество» существует ровно при $n \geq 1$ и на каждом таком носителе единственна. Поэтому её ЭПФ есть $e^x - 1$: вычитание единицы убирает пустую структуру при $n = 0$.

ЭПФ неупорядоченного набора структур есть $e^{F(x)}$. Здесь $F(x)$ — ЭПФ одной такой структуры. Набор из k компонент (порядок не важен) имеет ЭПФ $\frac{F(x)^k}{k!}$. Деление на $k!$ убирает порядок компонент.

Суммируем по всем $k \geq 0$. Получаем $\sum_{k=0}^{\infty} \frac{F(x)^k}{k!} = e^{F(x)}$. Подстановка даёт $B(x) = e^{e^x-1}$.¹⁵⁸

Пример Перестановки без неподвижных точек через ЭПФ

Перестановку n элементов без неподвижных точек называют *беспорядком*, а их число — субфакториалом D_n . $D_0 = 1, D_1 = 0, D_2 = 1, D_3 = 2$. Число беспорядков удовлетворяет рекурренте

$$D_n = nD_{n-1} + (-1)^n, \quad (n \geq 1), \quad D_0 = 1.$$

Получим замкнутую формулу методом ЭПФ. Обозначим $D(x) = \sum_{n=0}^{\infty} D_n \frac{x^n}{n!}$. Умножаем рекурренту на $\frac{x^n}{n!}$ и суммируем по $n \geq 1$:

$$\sum_{n=1}^{\infty} D_n \frac{x^n}{n!} = \sum_{n=1}^{\infty} nD_{n-1} \frac{x^n}{n!} + \sum_{n=1}^{\infty} (-1)^n \frac{x^n}{n!}.$$

Левая часть: $D(x) - D_0 = D(x) - 1$. В первом слагаемом правой части $\frac{n}{n!} = \frac{1}{(n-1)!}$, поэтому

$$\sum_{n=1}^{\infty} nD_{n-1} \frac{x^n}{n!} = x \sum_{n=1}^{\infty} D_{n-1} \frac{x^{n-1}}{(n-1)!} = xD(x).$$

Второе слагаемое: $\sum_{n=1}^{\infty} (-1)^n \frac{x^n}{n!} = e^{-x} - 1$.

Уравнение:

$$D(x) - 1 = xD(x) + e^{-x} - 1,$$

откуда

$$D(x) = \frac{e^{-x}}{1-x}.$$

¹⁵⁸Wilf, гл. 3.

Извлекаем коэффициент: $e^{-x} = \sum_{k=0}^{\infty} (-1)^k \frac{x^k}{k!}$ и $\frac{1}{1-x} = \sum_{m=0}^{\infty} x^m$. Свёртка даёт

$$[x^n] \frac{e^{-x}}{1-x} = \sum_{k=0}^n \frac{(-1)^k}{k!},$$

а значит $D_n = n! \sum_{k=0}^n \frac{(-1)^k}{k!}$. Проверка: $D_3 = 6(1 - 1 + \frac{1}{2} - \frac{1}{6}) = 2$ — ровно две перестановки из трёх без неподвижных точек.

Рекуррента $D_n = nD_{n-1} + (-1)^n$ выражается в ЭПФ одним сдвигом на x . Это и есть то, о чём говорит различие ОПФ и ЭПФ: последовательность с факториальным ростом ($D_n \sim \frac{n!}{e}$) удобнее нести через $\frac{x^n}{n!}$, где $n!$ входит в знаменатель, а не в коэффициенты.

Когда использовать ОПФ, а когда — ЭПФ?

Примечание ОПФ или ЭПФ?

Выбор между обычной и экспоненциальной производящей функцией диктуется природой объекта:

- **ОПФ** для неразмеченных объектов, где порядок элементов в конструкции не важен (сочетания, разбиения на слагаемые).
- **ЭПФ** для размеченных объектов, где каждый элемент несёт уникальную метку (перестановки, помеченные деревья, разбиения множества).

Практический тест: если в комбинаторной формуле появляется биномиальный коэффициент $\binom{n}{k}$ при сборке объекта из частей — скорее всего, нужна ЭПФ.¹⁵⁹

Проверка Экспоненциальные производящие функции

1. Почему произведение ЭПФ даёт свёртку с биномиальными коэффициентами, а не обычную свёртку?
2. Как из ЭПФ $E(x)$ восстановить член последовательности a_n ?
3. Почему рекуррента для беспорядков с множителем n свернулась в уравнение одним сдвигом?

§ 21.3 Применения производящих функций

Производящие функции возникают в нескольких контекстах:

- точные формулы для помеченных структур,
- анализ алгоритмов,
- вероятностные распределения.

¹⁵⁹Теория комбинаторных видов (combinatorial species, Joyal 1981) даёт словарь для автоматического перевода комбинаторных конструкций в уравнения на ЭПФ. См. F. Bergeron, G. Labelle, P. Leroux. «Combinatorial Species and Tree-like Structures». 1998.

Размен денег — самый бытовой пример того же приёма. Сколькими способами можно разменять n рублей монетами по 1, 2 и 5 рублей? Обозначим достоинство монеты через d . Каждая такая монета вносит вклад x^d . Выбор m таких монет даёт слагаемое x^{md} . Суммируя по всем m , получаем множитель $\frac{1}{1-x^d}$ — геометрический ряд. Вся задача о размене — произведение трёх множителей:

$$\frac{1}{1-x} \cdot \frac{1}{1-x^2} \cdot \frac{1}{1-x^5},$$

и коэффициент при x^n — число способов разменять ровно n рублей.

Пример Размен шести рублей

Ищем коэффициент $[x^6] \frac{1}{(1-x)(1-x^2)(1-x^5)}$.

Переберём способы набрать степень 6: $6 = 5 + 1 = 2 + 2 + 2 = 2 + 2 + 1 + 1 = 2 + 1 + 1 + 1 + 1 = 1 + 1 + 1 + 1 + 1 + 1$. Каждому разложению отвечает ровно один способ выбрать слагаемые из трёх геометрических рядов.

Ответ: $[x^6] = 5$ — пять способов разменять шесть рублей.

ОПРЕДЕЛЕНИЕ 21.3 Композиция числа

Композицией числа n называется представление n в виде упорядоченной суммы положительных целых слагаемых.

Пример Подсчёт композиций числа

Например, композиции числа 3:

- 3
- $2 + 1$
- $1 + 2$
- $1 + 1 + 1$

— всего 4.

Сколько композиций у числа n ? Обозначим количество композиций через a_n . Рассмотрим первое слагаемое k . Оно меняется в пределах $1 \leq k \leq n$. Оставшаяся часть $n - k$ может быть любой композицией. Отсюда рекуррента:

$$a_n = a_{n-1} + a_{n-2} + \dots + a_0, \quad (n \geq 1), \quad a_0 = 1.$$

Правая часть — сумма всех предыдущих членов: $a_n = \sum_{i=0}^{n-1} a_i$. Ряд частичных сумм имеет ОПФ $\frac{A(x)}{1-x}$. Его члены: $s_n = a_0 + \dots + a_n$. Ряд из a_n получается сдвигом ряда частичных сумм. Здесь $a_n = s_{n-1}$.

$$\begin{aligned}\sum_{n \geq 1} a_n x^n &= x \sum_{m \geq 0} s_m x^m \\ &= x \frac{A(x)}{1-x}.\end{aligned}$$

Левая часть — это $A(x) - a_0 = A(x) - 1$. Приравнивая, получаем:

$$\begin{aligned}A(x) - 1 &= x \frac{A(x)}{1-x}, \\ A(x) &= \frac{1-x}{1-2x} \\ &= \frac{1}{1-2x} - \frac{x}{1-2x}.\end{aligned}$$

Коэффициент: $[x^n]A(x) = 2^n - 2^{n-1} = 2^{n-1}$, $n \geq 1$. Начальное значение: $a_0 = 1$.

Ответ: $a_n = 2^{n-1}$, $n \geq 1$. Проверка: при $n = 3$ получается $2^2 = 4$ композиции. Интуитивно: у записи из одних единиц $(1 + 1 + \dots + 1)$ есть $n - 1$ промежутков между соседними. В каждом можно либо поставить плюс, либо объединить — 2^{n-1} вариантов.

Композиции — задача на точный счёт: мы искали количество вариантов. В анализе алгоритмов чаще спрашивают про среднее поведение, и случайность меняет характер ответа: следующий пример выводит ожидаемое число сравнений быстрой сортировки.

Пример Анализ быстрой сортировки

Ожидаемое число сравнений для случайного выбора опорного элемента (*pivot*) на входе размера n :

$$Q_n = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} Q_k, \quad Q_0 = 0.$$

Рекуррента содержит полную сумму всех предыдущих значений и деление на n . Умножение на n убирает дробь и готовит уравнение к суммированию:

$$nQ_n = n(n-1) + 2 \sum_{k=0}^{n-1} Q_k.$$

Умножаем обе части на x^n и суммируем по всем $n \geq 1$. Множитель n при коэффициенте ряда соответствует производной: $\sum_{n \geq 1} nQ_n x^n = xQ'(x)$. Первое слагаемое справа — вторая производная геометрической прогрессии: $\sum_{n \geq 1} n(n-1)x^n = 2\frac{x^2}{(1-x)^3}$. Второе слагаемое содержит частичную сумму, и правило частичных сумм даёт $\sum_{n \geq 1} x^n \sum_{k=0}^{n-1} Q_k = x \frac{Q(x)}{1-x}$. Уравнение:

$$xQ'(x) = 2\frac{x^2}{(1-x)^3} + 2x\frac{Q(x)}{1-x}.$$

После деления на x :

$$Q'(x) - \frac{2}{1-x}Q(x) = 2\frac{x}{(1-x)^3}.$$

Умножение на $(1-x)^2$ собирает слева производную произведения: $((1-x)^2Q(x))' = (1-x)^2Q'(x) - 2(1-x)Q(x)$. Уравнение принимает вид

$$((1-x)^2Q(x))' = 2\frac{x}{1-x}.$$

Первообразная правой части равна $-2\ln(1-x) - 2x$, и константа интегрирования определена условием $Q(0) = Q_0 = 0$. Отсюда

$$Q(x) = \frac{-2\ln(1-x) - 2x}{(1-x)^2}.$$

Для извлечения коэффициента нужен ряд логарифма. Производная $-\ln(1-x)$ равна геометрической прогрессии $\frac{1}{1-x}$, а значение в нуле равно нулю, поэтому $-\ln(1-x) = \sum_{k \geq 1} x^k/k$. Свёртка двух известных рядов даёт

$$\begin{aligned} [x^n] \frac{-\ln(1-x)}{(1-x)^2} &= \sum_{k=1}^n \frac{n-k+1}{k} \\ &= (n+1) \sum_{k=1}^n \frac{1}{k} - n. \end{aligned}$$

Обозначим через $H_n = \sum_{k=1}^n \frac{1}{k}$ гармонические числа. Второе слагаемое читается из таблицы базовых рядов: $[x^n] 2\frac{x}{(1-x)^2} = 2n$. Итого:

$$Q_n = 2(n+1)H_n - 4n, \quad (n \geq 1).$$

Проверка: $Q_1 = 0$, $Q_2 = 2 \cdot 3 \cdot \frac{3}{2} - 8 = 1$, $Q_3 = 2 \cdot 4 \cdot \frac{11}{6} - 12 = \frac{8}{3}$.

Используя $H_n \sim \ln n + \gamma$, где $\gamma \approx 0.577$ — постоянная Эйлера–Маскерони, получаем

$$Q_n \sim 2n \ln n,$$

классическое $O(n \log n)$.

Множитель n при неизвестном члене переводится в производную: $xQ'(x)$ собирает ряд с коэффициентами nQ_n . Так рекуррента с полиномиальным коэффициентом проходит через метод ОПФ: уравнение на $Q(x)$, решение, извлечение коэффициента.

Производящие функции применяются и за пределами комбинаторики — в теории вероятностей.

Для дискретной случайной величины X вероятностная производящая функция $G_X(s) = \sum p_k s^k$ кодирует распределение. Здесь $p_k = P(X = k)$ — вероятности значений. Матожидание и дисперсия выражаются через производные G_X в точке $s = 1$. Почленное дифференцирование ряда $\sum p_k s^k$ в единице даёт

$$E[X] = G_{X'}(1), \quad E[X(X-1)] = G_{X''}(1).$$

Отсюда

$$E[X^2] = G_{X''}(1) + G_{X'}(1), \quad \text{Var}[X] = G_{X''}(1) + G_{X'}(1) - (G_{X'}(1))^2.$$

Свёртке распределений соответствует произведение функций.¹⁶⁰

Пример Вероятностная производящая функция кубика

Для честного кубика $G(s) = \frac{s+s^2+s^3+s^4+s^5+s^6}{6}$.

Производные в единице:

$$G'(1) = \frac{1+2+3+4+5+6}{6} = 3.5$$

$$G''(1) = \frac{0+2+6+12+20+30}{6} = \frac{35}{3}.$$

Матожидание: $E[X] = G'(1) = 3.5$. Дисперсия: $\text{Var}[X] = G''(1) + G'(1) - (G'(1))^2 = \frac{35}{3} + 3.5 - 3.5^2 = \frac{35}{12}$. Те же числа, что и в главе о вероятности (20).

Произведение $G(s)^2$ кодирует распределение суммы двух кубиков. Коэффициент при s^7 равен $\frac{6}{36}$. Это те самые шесть исходов из открытия главы.

Проверка Применения производящих функций

1. Почему у числа n ровно 2^{n-1} композиций?
2. Где в рассуждении появляется умножение на $\frac{1}{1-x}$?
3. Как вероятностная производящая функция $G_X(s) = \sum p_k s^k$ кодирует распределение, и как извлечь из неё матожидание?

§ 21.4 * Комбинаторные виды

Производящие функции — часть более общей теории. За уравнениями $C(x) = 1 + xC(x)^2$, $T(x) = xe^{T(x)}$, $B(x) = e^{e^x-1}$ (Каталан, деревья, Белл) стоит общий принцип: комбинаторная конструкция переводится в операцию над производящей функцией. Теория комбинаторных видов Андре Жуайяля (combinatorial species, 1981) делает этот принцип точным.

¹⁶⁰W. Feller. «An Introduction to Probability Theory and Its Applications». Т. 1, 3-е изд., 1968, гл. XI.

Вид (*species*) — это правило, сопоставляющее каждому конечному множеству U (носителю) множество структур на U . Структура вида F на носителе U — это способ организовать элементы U в конструкцию типа F .

ОПРЕДЕЛЕНИЕ 21.4 Комбинаторный вид

Видом F называется правило. Каждому конечному множеству U оно сопоставляет конечное множество структур. Это множество обозначается $F[U]$.

Для каждой биекции $\sigma : U \rightarrow V$ задан перенос структур (*transport of structures*). Перенос переименовывает структуры по правилу σ . Перенос обязан уважать композицию биекций.

Число структур зависит только от $|U|$ и обозначается $|F[n]|$ — это количество F -структур на n помеченных элементах.

Пример Базовые виды

- Вид E — «множество»: на каждом носителе U есть ровно одна структура — само множество элементов.
- Вид X — «синглетон»: структура есть только на одноэлементном множестве, и она единственна.
- Вид S — «перестановка»: структуры суть все перестановки элементов носителя U . Их число — $n!$.
- Вид C — «цикл»: структуры суть циклические перестановки элементов носителя U . Их число — $(n - 1)!$.
- Вид T — «корневое дерево»: структуры суть корневые помеченные деревья с множеством вершин U .

Вид удобно сравнить с типом данных. Тип `List` описан для любых элементов. Конкретный список `[1, 2]` — значение этого типа на конкретных данных. Вид — это тип: правило сборки, заданное для любого носителя. Структура — значение: конкретная конструкция на конкретном множестве.

21.4.1 Операции над видами

Комбинаторные конструкции — объединение, произведение, подстановка — становятся операциями над видами. Каждая операция переводится в операцию над ЭПФ.

ОПРЕДЕЛЕНИЕ 21.5 Сумма

Суммой видов F и G называется вид $F + G$, структура которого на носителе U — это F -структура либо G -структура.

$$|(F + G)[n]| = |F[n]| + |G[n]|,$$

а ЭПФ суммы равна сумме ЭПФ.

Пример Сумма: множество или синглетон

Вид $E + X$ — «множество или синглетон». На одном элементе — две структуры (множество и синглетон), на $n \geq 2$ — одна (множество). ЭПФ: $e^x + x$ — сложение, как в определении.

ОПРЕДЕЛЕНИЕ 21.6 Произведение

Произведением видов F и G называется вид $F \cdot G$, структура которого получается разбиением носителя U на две упорядоченные части с F -структурой на первой и G -структурой на второй.

Число структур — биномиальная свёртка:

$$|(F \cdot G)[n]| = \sum_{k=0}^n \binom{n}{k} |F[k]| \cdot |G[n-k]|.$$

ЭПФ произведения равна произведению ЭПФ.

Пример Произведение: $E \cdot E = 2^n$

Пусть $F = G = E$ (вид множеств). Для каждого элемента $x \in U$ два варианта — в первую часть или во вторую — и выборы независимы:

$$|(E \cdot E)[n]| = 2^n.$$

ЭПФ произведения — $e^x \cdot e^x = e^{2x}$, а её разложение в ряд

$$e^{2x} = \sum_{n \geq 0} 2^n \frac{x^n}{n!}.$$

Коэффициент при $\frac{x^n}{n!}$ равен 2^n — комбинаторный подсчёт и алгебраический вывод совпали.

ОПРЕДЕЛЕНИЕ 21.7 Композиция

Композицией $F \circ G$ (подстановкой G в F) называется вид, в котором носитель U разбивается на блоки, на множестве блоков строится F -структура, а внутри каждого блока — G -структура. ЭПФ композиции равна композиции ЭПФ: $F(G(x))$.

Пример Композиция: корень и поддеревья

Уравнение корневых деревьев $T = X \cdot \text{Set}(T)$ — композиция. Корень — X . Неупорядоченный набор поддеревьев — $\text{Set}(T)$. Переходя к ЭПФ:

$$T(x) = x \cdot e^{T(x)}.$$

Экспонента появилась из набора, чья ЭПФ есть $e^{F(x)}$.

ОПРЕДЕЛЕНИЕ 21.8 Неупорядоченный набор

Неупорядоченным набором $\text{Set}(F)$ называется вид, в котором носитель U разбивается на блоки, а на каждом блоке строится F -структура. ЭПФ набора:

$$\text{Set}(F)(x) = e^{F(x)}.$$

Пример Набор: разбиения множества

Разбиение — это неупорядоченный набор непустых множеств $\text{Set}(E_+)$. Здесь $E_+(x) = e^x - 1$ — вид непустых множеств. ЭПФ разбиений:

$$B(x) = e^{e^x - 1},$$

и её коэффициенты — числа Белла. На трёх элементах разбиений пять: одно на один блок, три на два, одно на три.

Замечание Почему Set даёт экспоненту?

Набор из k компонент (порядок не важен) имеет ЭПФ $\frac{F(x)^k}{k!}$. Этот вывод мы уже проделали в заметке о числах Белла. Суммирование даёт $e^{F(x)}$. Экспонента возникает как прямое следствие комбинаторики: $\frac{1}{k!}$ в ЭПФ компенсирует перестановки блоков.

ОПРЕДЕЛЕНИЕ 21.9 Производная

Производной вида F называется вид F' с одной отмеченной точкой, обозначается $F'[U] = F[U \cup \{*\}]$. ЭПФ производной — производная ЭПФ.

Пример Производная: перестановки с отмеченным элементом

Производная помечает элемент: для вида перестановок

$$S'(x) = \frac{1}{(1-x)^2},$$

и $|S'[n]| = |S[n+1]| = (n+1)!$ — перестановки с отмеченным элементом.

Общий принцип: операция над видами переводится в операцию над ЭПФ, и одно подтверждает другое.

Замечание Почему на n метках ровно одна структура множества?

Вид E : на каждом носителе U ровно одна структура, обозначается $E[U] = \{U\}$. Путаница возникает, когда «структуры вида E » смешивают с «подмножествами

U ». Подмножество — это выбор: каждый элемент либо входит в него, либо нет. Выбор подмножества — это структура вида $E \cdot E$. Таких выборов ровно $2^{|U|}$.

Одно множество, два независимых выбора — два разных вида.

21.4.2 Откуда берутся формулы

Теперь знакомые формулы становятся следствием конструкций.

Доказательство

Докажем каждую формулу как следствие соответствующей конструкции.

Множества: $E = \text{Set}(X)$ — неупорядоченный набор синглетов, поэтому $E(x) = e^x$.

Перестановки: $S = \text{Set}(C)$ — неупорядоченный набор циклов.¹⁶¹

$$\begin{aligned} C(x) &= \sum_{n \geq 1} (n-1)! \frac{x^n}{n!} \\ &= -\ln(1-x), \end{aligned}$$

откуда $S(x) = e^{-\ln(1-x)} = \frac{1}{1-x}$.

Разбиения множества (числа Белла): разбиение — $\text{Set}(E_+)$, где $E_+(x) = e^x - 1$ — непустые множества, поэтому $B(x) = e^{e^x-1}$.

Корневые деревья: $T = X \cdot \text{Set}(T)$ — корень и набор поддеревьев, то есть $T(x) = x \cdot e^{T(x)}$.

Функции: $\text{Fun} = \text{Set}(\text{Cyc}(T))$ — множество циклов из деревьев, где $\text{Cyc}(x) = -\ln(1-x)$. Подстановка даёт $\text{Fun}(x) = \frac{1}{1-T(x)}$, а число функций из $[n]$ в $[n]$ равно n^n . Формула Лагранжа подтверждает: $[x^n] \frac{1}{1-T(x)} = \frac{n^n}{n!}$.¹⁶²

21.4.3 Применения видов

Теория видов служит инструментом для новых подсчётов, а не переформулировкой известных формул.

Помеченный граф на множестве вершин U — это выбор рёбер, где каждое ребро — неупорядоченная пара вершин. На языке видов граф — это неупорядоченный набор пар: $G = \text{Set}(P_2)$. Здесь P_2 — вид неупорядоченных пар. Число возможных рёбер — $C(n, 2)$. Каждое ребро либо есть, либо нет: $|G[n]| = 2^{C(n, 2)}$. ЭПФ графов:

$$G(x) = \sum_{n \geq 0} 2^{C(n, 2)} \frac{x^n}{n!}.$$

¹⁶¹На n элементах $(n-1)!$ различных циклов.

¹⁶² $T(x) = xe^{T(x)}$; формула Лагранжа: $[x^n] \frac{1}{1-T(x)} = \frac{n^n}{n!}$.

Замкнутой формулы вроде e^{2x} здесь нет: показатель $C(n, 2)$ растёт квадратично и не сворачивается. Это честное ограничение — не каждый вид сводится к композиции экспонент.

Деревья — это уже структуры данных. Бинарное дерево либо пусто, либо корень с двумя поддеревьями:

$$B = 1 + X \cdot B^2.$$

Уравнение кодирует рекурсивное определение типа — и сразу даёт производящую функцию

$$B(x) = 1 + xB(x)^2,$$

из которой извлекаются числа Каталана.

Сила видов — в разделении структуры и подсчёта. Описали конструкцию операциями — получили уравнение на производящую функцию. Симметрии учтены автоматически, делением на факториалы. Обращение Лагранжа, применённое к уравнению $T = X \cdot \text{Set}(T)$, даёт n^{n-1} корневых помеченных деревьев. Деление на n (выбор корня) оставляет n^{n-2} некорневых — формула Кэли.

Дальнейшее изучение: F. Bergeron, G. Labelle, P. Leroux. «Combinatorial Species and Tree-like Structures». 1998.

§ 21.5 * Формула Кэли

ТЕОРЕМА 21.5 Формула Кэли¹⁶³

Число помеченных деревьев на n вершинах равно

$$n^{n-2}.$$

Доказательство

Докажем в два этапа: сперва посчитаем корневые помеченные деревья, затем перейдём к некорневым.

Корневое помеченное дерево состоит из корня и неупорядоченного набора корневых помеченных деревьев-поддеревьев, присоединённых к корню. На языке видов: $T = X \cdot \text{Set}(T)$. Переходя к ЭПФ: $T(x) = x \cdot e^{T(x)}$.

Формула обращения Лагранжа для уравнения вида $T = x \cdot \varphi(T)$ даёт:

$$[x^n]T(x) = \frac{1}{n} [t^{n-1}] \varphi(t)^n.$$

Подставляя $\varphi(t) = e^t$:

¹⁶³ A. Cayley. «A Theorem on Trees». Quarterly Journal of Pure and Applied Mathematics, 1889.

$$\begin{aligned}
 [x^n]T(x) &= \frac{1}{n} [t^{n-1}]e^{nt} \\
 &= \frac{1}{n} \cdot \frac{n^{n-1}}{(n-1)!} \\
 &= \frac{n^{n-1}}{n!}.
 \end{aligned}$$

Число корневых помеченных деревьев на n вершинах:

$$n! \cdot [x^n]T(x) = n^{n-1}.$$

Некорневое дерево получается из корневого забыванием корня. Каждому некорневому дереву соответствует ровно n корневых: корнем можно выбрать любую из n вершин. Следовательно, число некорневых помеченных деревьев равно $\frac{n^{n-1}}{n} = n^{n-2}$.

Комбинаторное доказательство через код Прюфера приведено в главе 18.

§ 21.6 * Асимптотика коэффициентов

Метод производящих функций даёт точную формулу для a_n . Но точная формула не всегда удобна. Для средней глубины бинарного дерева поиска она занимает три строки. Для высоты случайного дерева замкнутой формулы (*closed form*) нет вовсе. Когда точное выражение громоздко или недоступно, достаточно асимптотики — ответа на вопрос, как быстро растёт a_n при больших n .

Производящая функция — одновременно формальный ряд и функция комплексного переменного. Ближайшая к нулю особенность $A(x)$ определяет асимптотику a_n . Почему именно особенность? Для рациональных функций это видно из простейших дробей. Коэффициент дроби $\frac{1}{(1-\rho x)^m}$ растёт как $n^{m-1}\rho^n$. Среди полюсов доминирует ближайший к нулю.

Аналитическая комбинаторика переносит ту же картину на точки ветвления и алгебраические функции. Грубое правило: полюс в точке $x = \rho$ даёт рост ρ^{-n} с полиномиальным множителем. Точка ветвления добавляет дробные степени n .

Пример Асимптотика чисел Каталана

ОПФ: $C(x) = \frac{1-\sqrt{1-4x}}{2x}$. Ближайшая особенность — точка ветвления при $x = \frac{1}{4}$. Вблизи этой точки:

$$C(x) \approx 2 - 2\sqrt{1-4x}.$$

¹⁶⁴P. Flajolet, R. Sedgewick. «Analytic Combinatorics». Гл. VI, 2009.

Слагаемое $\sqrt{1-4x}$ определяет асимптотику. По лемме о переносе¹⁶⁴ для α не равного целому неотрицательному:

$$[x^n] \left(1 - \frac{x}{\rho}\right)^\alpha \sim \rho^{-n} \frac{n^{-\alpha-1}}{\Gamma(-\alpha)}.$$

Подставляя $\alpha = \frac{1}{2}, \rho = \frac{1}{4}$:

$$[x^n] \sqrt{1-4x} \sim -\frac{4^n}{2\sqrt{\pi} n^{\frac{3}{2}}}.$$

Итого:

$$C_n \sim \frac{4^n}{\sqrt{\pi} n^{\frac{3}{2}}}.$$

Этот подход — аналитическая комбинаторика — систематически изложен в книге Флажоле и Седжвика «Analytic Combinatorics»¹⁶⁵. Метод работает для рациональных, алгебраических и дифференциально-конечных производящих функций. Ответ — асимптотика, не точная формула, и этого достаточно.

В анализе алгоритмов асимптотика часто важнее точного выражения. Средняя глубина бинарного дерева поиска растёт как $O(\log n)$. Среднее число сравнений в быстрой сортировке — $O(n \log n)$. Высота случайного бинарного дерева, равномерного среди каталановских, — $\Theta(\sqrt{n})$. Все эти результаты получены аналитической комбинаторикой из производящих функций соответствующих структур.

Доминирующий корень характеристического уравнения из главы 18 и ближайшая к нулю особенность рациональной ОПФ — один факт в двух языках. Полюс рациональной функции стоит в точке $1/\lambda$, поэтому больший корень даёт меньшую особенность. Алгебра корней отвечает за точные формулы, анализ особенностей — за асимптотики.

Одна запись несёт два прочтения, и прочтения делят между собой работу. Формальное прочтение отвечает за точность: равенство рядов означает равенство коэффициентов при каждом n . Аналитическое прочтение отвечает за масштаб: ближайшая особенность задаёт порядок роста коэффициентов. Тип конструкции определяет класс функции, класс функции определяет набор особенностей, а набор особенностей — формулы роста.

§ 21.7 Итоги главы

В основе главы лежит соответствие: последовательность a_n упаковывается в ряд $A(x) = \sum a_n x^n$, и один символ $A(x)$ несёт всю последовательность, от первого

¹⁶⁵P. Flajolet, R. Sedgewick. «Analytic Combinatorics». 2009.

члена до бесконечности. Каждая операция над последовательностями получает алгебраический двойник: сдвиг становится умножением на x , свёртка — произведением рядов, рекуррента — уравнением на $A(x)$, а извлечение коэффициента даёт замкнутую формулу. По этому конвейеру проходят и формула Бине для чисел Фибоначчи, и формула Каталана $C_n = \frac{1}{n+1} \binom{2n}{n}$, которую даёт уравнение $C(x) = 1 + xC(x)^2$.

Для помеченных объектов приём пополняется: экспоненциальная производящая функция с $\frac{x^n}{n!}$ восстанавливает в свёртке биномиальный коэффициент. Формализм держится на том, что операции над рядами определены через коэффициенты: сходимость для комбинаторных выводов не нужна — ряд работает как носитель чисел, а не как функция, в которую подставляют точку, — и анализ возвращается лишь на этапе асимптотики, где особенности задают рост коэффициентов.

Производящие функции связывают комбинаторные последовательности — числа Фибоначчи и Каталана — с алгебраическими уравнениями (глава 18), а формальные ряды ведут от конечных сумм к работе с бесконечностью (глава 22). Метод универсален: любая последовательность, заданная рекуррентой или конструкцией, получает производящую функцию, и величина её коэффициентов находится через алгебру рядов — вплоть до асимптотик анализа алгоритмов, вроде среднего числа сравнений быстрой сортировки.

Дальше вопрос сдвигается с последовательностей на сами числа: глава 22 построит их из множеств и покажет границы аксиоматического метода.

Проверка Проверьте себя

1. Почему производящая функция превращает рекурренту в уравнение?
2. Как извлечь коэффициент из рациональной ОПФ?
3. Чем ЭПФ отличается от ОПФ и когда использовать каждую?
4. Что означает запись $[x^n]A(x)$ и почему сходимость ряда не важна?
5. Какая операция над последовательностями соответствует умножению рядов?
6. Почему рекуррента чисел Каталана приводит к уравнению $C(x) = 1 + xC(x)^2$?

Что дальше?

Завершая инструментарий конечной математики, мы обращаемся к провокационному вопросу: насколько велика бесконечность? В главе 22 мы построим числа из множеств — от натуральных по фон Нейману до действительных сечениями Дедекинда. Там мы увидим границы аксиоматического метода: аксиому выбора и континуум-гипотезу.

§ 21.8 Упражнения

Обычные производящие функции.

1. Найдите обычную производящую функцию последовательности $a_n = n^2$. Подсказка: начните с $\sum nx^n = \frac{x}{(1-x)^2}$ и примените дифференцирование.
2. Найдите ОПФ последовательности $a_n = n \cdot 2^n$. Подсказка: начните с $\frac{1}{1-2x} = \sum 2^n x^n$ и продифференцируйте.
3. Найдите $[x^n] \frac{x}{(1-x)^3}$ двумя способами. Подсказка: используйте общую формулу $[x^n] \frac{1}{(1-x)^m}$ и дифференцирование геометрической прогрессии.
4. * Постройте ОПФ для количества способов набрать сумму n при бросании двух игральных костей (порядок важен — первый и второй бросок различимы). Сколько способов получить сумму 9?

Решение рекуррент через ОПФ.

5. Решите методом ОПФ рекурренту: $a_n = 3a_{n-1} - 2a_{n-2}, n \geq 2, a_0 = 0, a_1 = 1$.
6. Найдите $[x^n] \frac{1}{(1-2x)(1-3x)}$ с помощью разложения на простейшие дроби.
7. Решите методом ОПФ рекурренту с кратным корнем: $a_n = 4a_{n-1} - 4a_{n-2}, n \geq 2, a_0 = 1, a_1 = 6$.
8. * Пусть F_n — числа Фибоначчи ($F_0 = 0, F_1 = 1$). Найдите ОПФ для последовательности частичных сумм $s_n = \sum_{i=0}^n F_i$. Подсказка: частичные суммы соответствуют делению ОПФ на $1 - x$.
9. * Рекуррента Каталана: проверьте, что $C_n = \frac{1}{n+1} \binom{2n}{n}$. Подсказка: используйте уравнение $C(x) = 1 + xC(x)^2$ и обобщённый бином Ньютона: $\sqrt{1-4x} = \sum_{n=0}^{\infty} \binom{\frac{1}{2}}{n} (-4x)^n$.

Экспоненциальные производящие функции.

10. Найдите ЭПФ последовательности $a_n = c^n$. Проверьте: $e^{cx} \cdot e^{dx} = e^{(c+d)x}$.
11. * Выведите ЭПФ субфакториала ещё одним путём: беспорядок — это множество циклов, каждый длины не меньше 2. ЭПФ циклов длины $n \geq 2$ равна $\sum_{n \geq 2} \frac{x^n}{n} = -\ln(1-x) - x$, а множество структур имеет ЭПФ $e^{F(x)}$ (см. заметку о числах Белла). Сравните с выводом из рекурренты $D_n = nD_{n-1} + (-1)^n$ в примере выше.
12. * Докажите формулу Кэли: число помеченных деревьев на n вершинах равно

$$n^{n-2}.$$

Используйте связь между корневыми и некорневыми деревьями и результат из текста.

Применения производящих функций.

13. Сколько способов разменять n рублей монетами в 1, 2 и 5 рублей (порядок монет не важен)? Постройте ОПФ и найдите ответ для $n = 10$.
14. * Докажите тождество $\sum_{k=0}^n \binom{n}{k} = 2^n$ с помощью ОПФ. Подсказка: $(1+x)^n = \sum_{k=0}^n \binom{n}{k} x^k$.

ПРОЕКТ Счётчик разбиений по теореме Эйлера о пятиугольных числах

Эйлер использовал бесконечные суммы при подсчёте разбиений числа. Число разбиений $p(n)$ — количество представлений n в виде суммы положительных слагаемых, где порядок слагаемых не важен. У разбиений есть наивная рекуррента за $O(n^2)$. Теорема Эйлера утверждает: коэффициенты ряда $\prod_{k \geq 1} (1 - x^k)$ равны нулю везде, кроме обобщённых пятиугольных чисел. Обобщённые пятиугольные числа имеют вид $m(3m \pm 1)/2$. На них стоят знаки $(-1)^m$. Из этой теоремы вырастает рекуррента за $O(n^{3/2})$. Она даёт точные значения $p(n)$. Сверьте их с известными из литературы ориентирами. Соберите счётчик по шагам: наивный счёт, развёртывание производящей функции, самостоятельная проверка тождества Эйлера, быстрая рекуррента и проверка асимптотики $\log p(n)/\sqrt{n} \rightarrow \pi\sqrt{2/3}$.

① Наивный счёт

Соберите DP-треугольник $p(n, k) = p(n, k-1) + p(n-k, k)$ и квадратичную формулу перебора размеров частей. Длинной арифметикой сверьтесь с известными значениями $p(n)$: наивная и быстрая рекурренты должны совпадать.

② Производящая функция

Разверните произведение $\prod_{k \geq 1} \frac{1}{1-x^k}$ в усечённый многочлен усечённым умножением. Коэффициент при x^n — упаковка последовательности в ряд. Прочитайте его как $p(n)$.

③ Тождество Эйлера

Проверьте независимо, например по критерию точного квадрата, символьное разложение $\prod_{k \geq 1} (1 - x^k)$. Ненулевые коэффициенты должны стоять ровно на обобщённых пятиугольных числах $m(3m \pm 1)/2$. Знаки чередуются как $(-1)^m$.

④ Быстрая рекуррента

Выведите из тождества Эйлера рекурренту

$$p(n) = \sum_{m \geq 1} (-1)^{m+1} \left(p\left(n - \frac{m(3m-1)}{2}\right) + p\left(n - \frac{m(3m+1)}{2}\right) \right)$$

Ожидаемая сложность — $O(n^{3/2})$. Сверьте значения $p(n)$ с квадратичной формулой. Сравните время работы на замерах до $n = 10^4$.

⑤ Асимптотика

Проверьте сходимость на значениях до $n = 10^5$. Отношение $\log p(n)/\sqrt{n}$ должно монотонно сходиться. Предел — $\pi\sqrt{2/3}$ (Харди–Рамануджана).

Итог: быстрый счётчик разбиений, который заново открывает тождество Эйлера, измеряет ускорение, даваемое производящей функцией, и доводит подсчёт до асимптотики.

XXII

Конструкции чисел

В машине у числа всегда есть устройство: `int` — шестьдесят четыре бита, `float` — знак, мантисса и порядок, длинная арифметика — массив цифр. Сложение пускает устройство в дело: переносы, сдвиги, округление.

На бумаге устройство исчезает. Семёрка, минус три, три четверти, корень из двух — знаки, за которыми не видно ни битов, ни байтов. Ими пользуются каждый день, не спрашивая, из чего они сделаны.

Вопрос «что такое число» звучит праздно — ровно до библиотеки длинной арифметики или спецификации числового типа в языке. Там число обязано стать чем-то конкретным, и договорённость о том, чем именно, — инженерное решение со своей точностью, переполнением и скоростью.

Математика ответила на этот вопрос построением. Джон фон Нейман в 1923 году показал, как из пустого множества вырастает весь натуральный ряд: $0 = \emptyset$, $1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$ — и дальше одним и тем же правилом. Целые собираются из пар натуральных, рациональные — из пар целых, действительные — сечениями Дедекинда, разрезами множества рациональных на левую и правую части. На каждом шаге видно, из чего построено новое число и как в него вкладывается старое. А ординалы продолжают ту же конструкцию туда, где натуральные числа кончаются (глава 29).

Стройку сопровождает надзор. Аксиомы Пеано фиксируют требования к натуральному ряду, а алгебры Дедекинда добавляют: любая реализация с этими свойствами единственна с точностью до изоморфизма. Материал не влияет на арифметику — три яблока и три вложенных множества складываются одинаково.

Дальше начинается граница, где строить уже нечем. Аксиома выбора доказывает существование, ничего не строя, — и среди её следствий разрезание шара на пять частей, из которых собираются два шара, равных исходному. Сами аксиомы кончаются стеной: континуум-гипотеза не выводится из них — и её отрицание тоже.

Что такое число — и где проходит граница между построением и постулированием?

ИСТОРИЯ Дедекинд, Пеано и рождение аксиоматики

XIX век был временем пересмотра оснований математики. Открытие неевклидовых геометрий (Лобачевский, Бойяи, 1820–1830-е) показало, что геометрическая интуиция — не надёжный фундамент. Построение анализа на основе арифме-

тики стало ответом на этот кризис: если геометрия ненадёжна, то сведём всё к числам.

Дедекин (1872, 1888), Кантор (1872–1899), Фреге (1879–1903), Пеано (1889) — каждый внёс свой вклад в построение арифметического фундамента. Кульминацией программы стала аксиоматизация теории множеств Цермело и Френкелем (1908–1922) и доказательство Гёделем неполноты (1931). Арифметика не может быть формализована полностью: любая непротиворечивая теория, способная записать элементарную арифметику, неполна. То, что начиналось как попытка построить надёжный фундамент, привело к открытию принципиальных ограничений формальной математики.

ОБЗОР ГЛАВЫ

Мы ответим на вопрос «что такое число», построив его из множеств. Сначала аксиомы Пеано зафиксируют свойства натурального ряда, и мы увидим, почему логика первого порядка оставляет нестандартные модели, а второго — категорична, но неполна. Затем алгебры Дедекинда покажут ту же структуру с другой стороны: любые две просто бесконечные системы изоморфны, так что выбор материала не влияет на результат. Ординалы фон Неймана построят натуральные числа из пустого множества, из них соберутся целые и рациональные, а действительные выстроятся сечениями Дедекинда, с полнотой, которой не было у рациональных. В конце аксиома выбора с её эквивалентами — леммой Цорна и теоремой Цермело — откроет область неконструктивных существований, а независимость континуум-гипотезы от ZFC покажет границу самой аксиоматики.

После этой главы вы сможете:

1. формулировать аксиомы Пеано и объяснять, почему логика первого порядка оставляет нестандартные модели;
2. строить натуральные числа ординалами фон Неймана;
3. собирать целые и рациональные числа из классов эквивалентности пар;
4. строить действительные числа сечениями Дедекинда и объяснять полноту;
5. объяснять аксиому выбора, лемму Цорна и независимость континуум-гипотезы.

§ 22.1 Аксиомы Пеано

Джузеппе Пеано в 1889 году опубликовал работу «*Arithmetices principia, nova methodo exposita*». Она была написана на латыни и состояла из одних формул — ни одного слова на естественном языке. В ней девять аксиом: первые четыре задают свойства равенства, оставшиеся пять описывают натуральные числа. Свойства равенства в современной записи опускаются — равенство стало частью логики первого порядка, а не арифметики. Эти пять аксиом мы сегодня называем аксиомами Пеано.

В современной формулировке аксиомы Пеано описывают структуру $(\mathbb{N}, 0, S)$. Здесь $0 \in \mathbb{N}$ — выделенный элемент («нуль»). Функция $S : \mathbb{N} \rightarrow \mathbb{N}$ — «следование» (successor). Итак, аксиомы:

ОПРЕДЕЛЕНИЕ 22.1 Аксиомы Пеано

Структура $(\mathbb{N}, 0, S)$ состоит из множества $\mathbb{N}$ и выделенного элемента $0 \in \mathbb{N}$. Третья компонента — функция следования $S : \mathbb{N} \rightarrow \mathbb{N}$. Значение $S(n)$ — число, следующее за n .

1. $0 \in \mathbb{N}$.
2. $\forall n \in \mathbb{N}, S(n) \in \mathbb{N}$.
3. $\forall n \in \mathbb{N}, S(n) \neq 0$.
4. $\forall n, m \in \mathbb{N}, S(n) = S(m) \Rightarrow n = m$.
5. **(Аксиома индукции)** Для любого свойства P выполнено:

$$P(0) \wedge \forall n (P(n) \Rightarrow P(S(n))) \Rightarrow \forall n \in \mathbb{N}, P(n).$$

Первая аксиома фиксирует наличие нуля. Вторая — что за каждым числом идёт следующее. Третья — что нуль первый: ни одно число ему не предшествует. Четвёртая — что функция следования инъективна: разные числа имеют разные следующие, натуральный ряд не закичивается. Пятая — схема индукции: если свойство выполнено для нуля и передаётся от каждого числа к следующему, то оно выполнено для всех натуральных чисел.

Пример Натуральные числа как модель PA

Стандартная модель: $\mathbb{N} = \{0, 1, 2, \dots\}$, $0, S(n) = n + 1$.

Все пять аксиом выполняются. Проверка — это те же пять условий алгебры Дедекенда, взглянутые со стороны Пеано: после отождествления $0 = a$ и $S = f$ структура $(\mathbb{N}, 0, n \mapsto n + 1)$ удовлетворяет им обеими, и пошаговое рассмотрение в разделе 22.2 проверяет всё разом. Последняя аксиома — знакомый принцип математической индукции из главы 1.

Первые четыре аксиомы говорят об элементах и равенстве — это утверждения первого порядка. Пятая аксиома — схема индукции — устроена сложнее.

В формулировке «для любого свойства» она второго порядка. Мы квантифицируем по всем подмножествам $\mathbb{N}$. Дальше идёт вариант первого порядка. В логике первого порядка такая формулировка невозможна. Вместо этого мы выписываем бесконечное семейство аксиом — по одной на каждую формулу $\varphi(x)$ нашего языка:

$$[\varphi(0) \wedge \forall n (\varphi(n) \Rightarrow \varphi(S(n)))] \Rightarrow \forall n \varphi(n)$$

Это *схема аксиом*, а не одна аксиома, и разница принципиальна.

Аксиоматика второго порядка категорична: любые две её модели изоморфны. С точностью до изоморфизма существует ровно одна модель — стандартная: натуральный ряд $\{0, 1, 2, \dots\}$.

Аксиоматика первого порядка (РА) категоричностью не обладает. Она имеет нестандартные модели — структуры, удовлетворяющие всем аксиомам РА, но не изоморфные стандартной. В них есть «числа», которые больше любого натурального $0, 1, 2, \dots$ — так называемые *нестандартные элементы*. Развёрнутое обсуждение нестандартных моделей — в главе 29.

Разница объясняется выразительной силой логик:

1. Логика второго порядка «сильнее»: в ней можно выразить « P пробегает все подмножества универсума». Но такая логика неполна — для неё не существует полного и корректного исчисления (следствие теоремы Гёделя о неполноте).
2. Логика первого порядка «слабее»: в ней φ пробегает только *определимые* в языке подмножества. Но она полна (глава 3) — и именно полнота через теорему компактности порождает нестандартные модели (глава 29).

Это диалектическая ситуация. Чтобы зафиксировать натуральные числа однозначно, нужна логика второго порядка — но она не имеет полного исчисления. Логика первого порядка имеет полное исчисление — но не может зафиксировать натуральные числа однозначно. Мы не можем иметь и то и другое одновременно.

§ 22.2 Алгебры Дедекинда

За год до Пеано, в 1888 году, Рихард Дедекинд опубликовал работу «Was sind und was sollen die Zahlen?» — «Что такое числа и чем они должны быть?». В отличие от Пеано, писавшего на латыни и формулами, Дедекинд писал на немецком и прозой — и *конструировал* числа из более простых объектов.

Конструкция Дедекинда целиком опирается на «просто бесконечную систему» (einfach unendliches System). Дадим формальное определение.

ОПРЕДЕЛЕНИЕ 22.2 Алгебра Дедекинда

Тройка (A, a, f) называется **алгеброй Дедекинда** (или **просто бесконечной системой**), если выполнены пять условий:

1. $a \in A$ — выделенный элемент.
2. $f : A \rightarrow A$ — отображение множества в себя. Оно сопоставляет каждому элементу следующий.
3. $a \notin f(A)$. Выделенный элемент не принадлежит образу f .
4. f инъективно. Из равенства $f(x) = f(y)$ следует равенство аргументов.

5. Минимальность: если $B \subset A, a \in B, f(B) \subset B$, то $B = A$. Единственное подмножество A , содержащее a и замкнутое относительно f , — это всё A .

Разберём условия.

1. Первое условие означает, что в A есть выделенный элемент. Роль a — быть нулём, началом счёта.
2. Второе условие означает, что f сопоставляет каждому элементу следующий. Роль f — быть операцией перехода к следующему числу.
3. Третье условие означает, что a не следует ни за каким элементом. Нуль стоит первым — перед ним ничего нет.
4. Инъективность f означает, что разные элементы не могут иметь одно и то же следующее. Натуральный ряд не закичивается и не ветвится.
5. Условие минимальности означает, что A не содержит ничего лишнего. Каждый элемент A получается из выделенного элемента конечным числом применений f . Это принцип индукции: если свойство выполнено для a и передаётся от каждого элемента к следующему, то оно выполнено для всех элементов A .

Пример Натуральные числа как алгебра Дедекенда

Стандартная модель: $A = \mathbb{N}, a = 0, f(n) = n + 1$. Проверим выполнение каждого из пяти условий.

1. **Выделенный элемент.** $0 \in \mathbb{N}$.
2. **Отображение.** $n \mapsto n + 1$ переводит натуральное число в натуральное.
3. **Нуль не следует ни за кем.** Нуль не является значением $n + 1$ ни для какого натурального n .
4. **Инъективность.** Из равенства $n + 1 = m + 1$ следует $n = m$.
5. **Минимальность (индукция).** Если $B \subset \mathbb{N}$ содержит 0 и замкнуто относительно $n \mapsto n + 1$, то $B = \mathbb{N}$. Это обычный принцип математической индукции.

Итак, $(\mathbb{N}, 0, n \mapsto n + 1)$ — алгебра Дедекенда.

Эти пять условий — в точности пять аксиом Пеано из раздела 22.1: выделенный элемент играет роль нуля, а отображение f — роль следования S . Оба определения задают одну и ту же структуру, взглянутую с двух сторон.

Натуральные числа дали нам одну алгебру Дедекенда. Условие минимальности жёстко фиксирует структуру — может ли существовать алгебра Дедекенда, устроенная иначе? Ответ даёт следующая теорема.

ТЕОРЕМА 22.1 Изоморфизм просто бесконечных систем

Любые две алгебры Дедекинда изоморфны. Именно: если (A, a, f) и (B, b, g) — алгебры Дедекинда, то существует биекция $\varphi : A \rightarrow B$. Она переводит выделенный элемент первой алгебры в выделенный элемент второй. Она согласована со следованиями: $\varphi(f(x)) = g(\varphi(x))$ для всех элементов.

Доказательство

Докажем построением: определим φ рекурсивно и проверим корректность, инъективность и сюръективность.

Определим φ рекурсивно: положим $\varphi(a) = b$, а для произвольного элемента x зададим $\varphi(f(x)) = g(\varphi(x))$.

Сначала два факта об алгебре (B, b, g) .

Во-первых, $g(B) = B \setminus \{b\}$. Множество $g(B) \cup \{b\}$ содержит элемент b , замкнуто относительно g , и по минимальности это вся B . А $b \notin g(B)$ по определению алгебры Дедекинда.

Во-вторых, каждый элемент B получается из b конечным числом применений g . Множество таких элементов содержит b и замкнуто относительно g , поэтому по минимальности это вся B . Аналогично каждый элемент A получается из a конечным числом применений f .

Инъективность: пусть $\varphi(x) = \varphi(y)$. Элементы A имеют вид $x = f^{i(a)}$ и $y = f^{j(a)}$. Тогда $g^{i(b)} = g^{j(b)}$.

Рассмотрим два случая. Пусть $i > j$. При $j \geq 1$ обе части лежат в $g(B)$ — образе g . Поэтому можно применить $g^{-1}j$ раз и получить $g^{i-j}(b) = b$. При $j = 0$ равенство $g^{i(b)} = b$ уже имеет вид $g^{i-j}(b) = b$. Здесь $i - j = i \geq 1$. В обоих случаях получается $g^{k(b)} = b$ при $k \geq 1$.

Тогда $b = g(g^{k-1}(b)) \in g(B)$ противоречит $b \notin g(B)$. Значит, $i = j$ и $x = y$.

Сюръективность: образ $\varphi(A)$ содержит $b = \varphi(a)$. Он замкнут относительно g : если $y = \varphi(x)$, то $g(y) = \varphi(f(x))$. Значит, $g(y) \in \varphi(A)$. По условию минимальности для алгебры Дедекинда (B, b, g) множество $\varphi(A)$ должно совпадать со всем B .

Корректность определения: φ задано на всей A . A — минимальное множество, содержащее a и замкнутое относительно f . Рекурсивное правило согласовано: равенство $f(x_1) = f(x_2)$ не бывает. Это следует из инъективности f , поэтому и конфликтов нет.

Изоморфизм означает, что алгебры Дедекинда — это *единственная* (с точностью до изоморфизма) структура с данными свойствами. В этом смысле условия Дедекинда — как и аксиомы Пеано второго порядка — категоричны. Оговорка: у Дедекинда это

определяющие условия, а не аксиомы в формальном смысле, но по категоричности они работают так же. Они однозначно задают натуральный ряд.

Примечание Почему условия Дедекинда категоричны

Пятое условие — минимальность — говорит: если $B \subset A$ содержит a и замкнуто относительно f , то $B = A$. Это высказывание про *все* подмножества B сразу: квантор пробегает по подмножествам множества A . Такой квантор не выразим в логике первого порядка, и этим минимальность родственна второпорядковой аксиоме индукции из раздела 22.1.

Поэтому условия Дедекинда категоричны — как аксиоматика Пеано второго порядка. Аксиоматика первого порядка РА ослабляет пятое условие до схемы: вместо всех подмножеств она перечисляет только те, что задаются формулами языка, и именно в этом ослаблении прячется возможность нестандартных моделей. Минимальность в категоричном виде — причина, по которой у алгебр Дедекинда ровно один с точностью до изоморфизма представитель.

ЛОВУШКА Изоморфизм $\neq$ равенство

Из изоморфизма двух структур не следует их совпадение. Изоморфные структуры устроены одинаково, но это разные объекты — у них разные носители. Алгебры Дедекинда на разных множествах изоморфны, но не равны. Элементы $0 \in \mathbb{N}$, $a \in A$ устроены одинаково, но это разные элементы. Изоморфизм сохраняет свойства структуры, но не отождествляет элементы. Равенство — отношение между элементами одной структуры, изоморфизм — отношение между структурами.

Но где взять саму алгебру Дедекинда? Аксиомы описывают свойства, но не гарантируют существования объекта с этими свойствами.

Дедекинд рассуждал так: бесконечное множество существует, потому что существует «мир моих мыслей»¹⁶⁶.

Современная математика поступает иначе: она строит натуральные числа внутри теории множеств.

Проверка Алгебры Дедекинда

1. Какое из пяти условий алгебры Дедекинда отвечает за принцип индукции, и почему без него в A могли бы появиться «лишние» элементы?
2. Почему из изоморфизма двух алгебр Дедекинда не следует их равенство?

¹⁶⁶ «Рассмотрим множество всех возможных объектов моей мысли. Отображение s переводит мысль о предмете x в мысль «предмет x есть предмет моей мысли». Это отображение инъективно, а его образ не содержит мысль о моём «я», которая и служит начальным элементом.»

§ 22.3 Натуральные числа по фон Нейману

Джон фон Нейман предложил конструкцию, в которой каждое натуральное число — это множество всех меньших натуральных чисел:

$$\begin{aligned} 0 &= \emptyset, \\ 1 &= \{0\} = \{\emptyset\}, \\ 2 &= \{0, 1\} = \{\emptyset, \{\emptyset\}\}, \\ 3 &= \{0, 1, 2\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \\ &\dots \end{aligned}$$

Здесь $n + 1 = n \cup \{n\}$. Каждое натуральное число n содержит ровно n элементов. Оно является ординалом: транзитивным множеством, вполне упорядоченным (*well-ordered*) отношением принадлежности. В такой записи каждый следующий ординал — множество всех предыдущих: коробка числа n вмещает коробки чисел $0, \dots, n - 1$. Цепочка принадлежности $0 \in 1 \in 2 \in \dots$ превращается в картинку вложенных коробок, а внешний пунктирный контур — это ω , как на рис. 96.

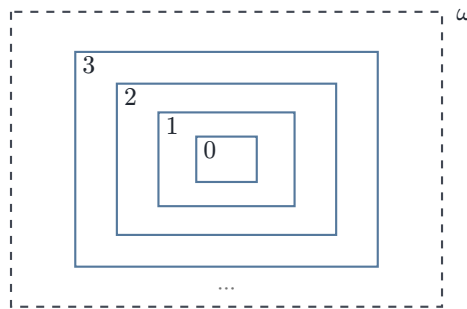


Рис. 96. Вложенность ординалов фон Неймана.

ОПРЕДЕЛЕНИЕ 22.3 Индуктивное множество

Множество I называется **индуктивным**, если выполнены два условия:

1. $\emptyset \in I$.
2. для любого $x \in I$ выполнено $x \cup \{x\} \in I$.

Индуктивное множество содержит пустое множество и замкнуто относительно операции взятия последователя.

Пример Индуктивные множества

Множество $I = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \dots\}$ индуктивно. Оно содержит $\emptyset$. Вместе с каждым элементом оно содержит и его последователь $x \cup \{x\}$. Для пустого множества последователь — это $\{\emptyset\}$, и так далее. Это множество в точности и есть натуральные числа фон Неймана, которые мы определим ниже.

Напротив, $\mathbb{R}$ не индуктивно. Причина: $\emptyset \notin \mathbb{R}$. Индуктивность — свойство именно порядковой структуры, а не любого множества.

Аксиома бесконечности в ZF утверждает: существует индуктивное множество. Пересечение всех индуктивных множеств и даёт натуральный ряд:

ОПРЕДЕЛЕНИЕ 22.4 Натуральные числа

Множество натуральных чисел определяется как пересечение всех возможных индуктивных множеств.

$$\omega = \bigcap_{I \text{ индуктивно}} I.$$

Для каждого $n \in \omega$ его «последователь»: $S(n) = n \cup \{n\}$. Так из пустого множества вырастает весь ряд.

Примечание Зачем пересечение всех индуктивных множеств

Произвольное индуктивное множество может содержать лишнее. Если I индуктивно, то индуктивно и $I \cup \{I\}$: пустое множество в нём есть, а последователь любого элемента остаётся внутри. В таком множестве лежит элемент I , натуральным числом не являющийся. Пересечение всех индуктивных множеств содержится в каждом из них, поэтому посторонний элемент отбрасывается: хотя бы одно индуктивное множество его не содержит. Остаётся ровно то, что обязано лежать в любом индуктивном множестве: ноль и вся цепочка его последовательных. Приём тот же, что у Дедекинда: минимальность выделяет структуру, порождённую начальным элементом и операцией следования.

Пример Первые натуральные числа фон Неймана

$0 = \emptyset$. $1 = \{\emptyset\} = \{0\}$. $2 = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$. $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0, 1, 2\}$.

Каждое число — множество всех предыдущих, поэтому порядок совпадает с принадлежностью: $n \in m \Leftrightarrow n < m$. Это удобная особенность кодирования: арифметика наследует отношение принадлежности из теории множеств.

Эта конструкция превращает $(\omega, \emptyset, S)$ в алгебру Дедекинда. Все аксиомы Пеано выполняются — и их доказательство становится теоремой, а не постулатом.

Фон Неймановская конструкция использует минимум средств:

- Каждое натуральное число — множество ровно из n элементов.
- Порядок задаётся принадлежностью: $n < m \Leftrightarrow n \in m$.
- Арифметические операции определяются рекурсивно. Корректность такой рекурсии гарантирует теорема о рекурсии. Для всякой функции f и элемента a существует единственная последовательность $u_0 = a, u_{n+1} = f(u_n)$. Так рекурсивные

определения получают строгий смысл. На ординалах аналогичная теорема позволяет определять функции на всех трансфинитных ступенях.

- Непривычно выглядит сама запись: число 2 — это $\{\emptyset, \{\emptyset\}\}$, а не цифра. Следующее число — тройка из трёх элементов, и так далее.

Пример Сложение по рекурсии

Сложение натуральных чисел задаётся двумя правилами:

$$\begin{aligned} m + 0 &= m, \\ m + S(n) &= S(m + n). \end{aligned}$$

Первое правило — база: прибавить ноль — значит ничего не менять. Второе — шаг: прибавить число, за которым идёт следующее — сначала прибавить само число, затем сделать шаг.

Вычислим $2 + 2$, помня, что $1 = S(0)$, $2 = S(1)$, $3 = S(2)$:

$$\begin{aligned} 2 + 2 &= 2 + S(1) \\ &= S(2 + 1) \\ &= S(2 + S(0)) \\ &= S(S(2 + 0)) \\ &= S(S(2)) \\ &= S(3) \\ &= 4. \end{aligned}$$

Каждое равенство — применение одного из двух правил, и цепочка обрывается, когда второе слагаемое доходит до нуля. Теорема о рекурсии гарантирует, что такая цепочка существует и единственна для любых m, n .

Проверка Натуральные числа по фон Нейману

1. Почему для определения ω берётся пересечение всех индуктивных множеств, а не произвольное индуктивное множество?
2. Чему в ординалах фон Неймана соответствует отношение порядка $n < m$?
3. Что именно гарантирует теорема о рекурсии при определении сложения правилами $m + 0 = m$ и $m + S(n) = S(m + n)$?

Замечание Число — это множество или нет?

Конструкция фон Неймана *реализует* натуральные числа как множества, но не утверждает, что число 2 — это $\{\emptyset, \{\emptyset\}\}$ по своей природе. Это *реализация*, а не определение. Так же как алгоритм можно реализовать на C, Python и Haskell, натуральные числа можно реализовать ординалами фон Неймана, алгебрами

Дедекинда или в любой другой категоричной аксиоматике. Важна структура, а не материал, из которого она сделана.

§ 22.4 Построение целых чисел

С натуральными числами есть неудобство: уравнение $x + 3 = 0$ не имеет решения в $\mathbb{N}$. Чтобы решать такие уравнения, нужны отрицательные числа. Но как их построить, не прибегая к формуле «добавим минус перед каждым натуральным»?

Целое число z можно представить как разность двух натуральных: $z = a - b$. Проблема в том, что разность $a - b$ определена не для всех пар. Нельзя вычесть большее из меньшего, оставаясь в $\mathbb{N}$. Решение — работать с самой парой (a, b) , а несколько пар считать представляющими одно и то же целое число.

ОПРЕДЕЛЕНИЕ 22.5 Целые числа

На множестве $\mathbb{N} \times \mathbb{N}$ введём отношение эквивалентности:

$$(a, b) \sim (c, d) \Leftrightarrow a + d = b + c$$

Целое число — это класс эквивалентности пары (a, b) .

Множество всех целых чисел обозначается $\mathbb{Z} = \frac{\mathbb{N} \times \mathbb{N}}{\sim}$.

Пара (a, b) представляет целое число $a - b$. Если $a \geq b$, то это обычная разность. Если $a < b$, то это отрицательное число. Например:

- $(3, 1)$ и $(5, 3)$ — одно и то же целое число $(+2)$. Проверка: $3 + 3 = 1 + 5$.
- $(1, 3)$ представляет $1 - 3 = -2$. Чтобы получить 3 из 1, не хватает 2 — мы «в долгу» на двойку.

Натуральное число n отождествляется с классом пары $(n, 0)$. Это задаёт инъекцию $\mathbb{N} \rightarrow \mathbb{Z}$, сохраняющую сложение.

Арифметические операции определяются через представителей:

$$\begin{aligned} [(a, b)] + [(c, d)] &= [(a + c, b + d)], \\ [(a, b)] \cdot [(c, d)] &= [(ac + bd, ad + bc)]. \end{aligned}$$

Формула умножения не взята с потолка: она повторяет раскрытие скобок для разностей. Класс $[(a, b)]$ представляет разность $a - b$, а класс $[(c, d)]$ — разность $c - d$. Тогда

$$\begin{aligned} (a - b)(c - d) &= ac - ad - bc + bd \\ &= (ac + bd) - (ad + bc), \end{aligned}$$

и пара $(ac + bd, ad + bc)$ — ровно та, что представляет произведение.

Проверим, что сумма и произведение не зависят от выбора представителей. Пусть $(a, b) \sim (a', b')$ и $(c, d) \sim (c', d')$. Тогда $(a + c, b + d) \sim (a' + c', b' + d')$. Это прямое следствие определения эквивалентности. Сложение и умножение корректно определены на классах. Полная проверка для обеих операций — хорошее упражнение на понимание конструкции.

Пример Сложение целых чисел через пары

Возьмём два целых числа и сложим их, пользуясь только парами натуральных.

1. **Представление чисел.** Число $+2$ представлено парой $(5, 3)$. Действительно, $5 - 3 = 2$. Число -3 — парой $(1, 4)$. Действительно, $1 - 4 = -3$.

2. **Сложение пар.** Складываем по определению:

$$(5, 3) + (1, 4) = (5 + 1, 3 + 4) = (6, 7).$$

3. **Интерпретация результата.** Пара $(6, 7)$ представляет разность $6 - 7 = -1$. Покажем это через эквивалентность: $(6, 7) \sim (0, 1)$. Действительно, $6 + 1 = 7 + 0$. А класс $(0, 1)$ соответствует числу $0 - 1 = -1$.

4. **Проверка.**

$$2 + (-3) = -1,$$

как и должно быть.

Мы не «добавляем отрицательные числа» — мы замечаем, что классы эквивалентности пар натуральных чисел уже содержат их.

Конструкция $\mathbb{Z}$ из $\mathbb{N}$ — это **групповое пополнение** моноида $(\mathbb{N}, +, 0)$. Смысл в том, что в целых числах вычитание становится всюду определённым:

1. Каждая пара (a, b) — формальная разность $a - b$.
2. Отношение $(a, b) \sim (c, d)$ означает равенство разностей: $a - b = c - d$. Оно переписывается без отрицательных чисел: $a + d = b + c$.
3. Отрицательные числа не постулируются — они *возникают* как классы пар, в которых первый компонент меньше второго.

§ 22.5 Построение рациональных чисел

Целых чисел достаточно для сложения, вычитания и умножения. Но деление в $\mathbb{Z}$ работает не всегда. Уравнение $2x = 3$ не имеет целого решения. Чтобы делить друг на друга любые числа (кроме нуля), переходят к рациональным.

Применим ту же стратегию, что и для целых. Рациональное число — это частное двух целых: $q = \frac{a}{b}$. Здесь $b \neq 0$. Вместо частного работаем с парой (a, b) . Эквивалентные дроби, вроде $\frac{1}{2}, \frac{2}{4}$, считаем одним и тем же рациональным числом. Запись дроби не важна, важен класс эквивалентности.

ОПРЕДЕЛЕНИЕ 22.6 Рациональные числа

На множестве $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$ введём отношение эквивалентности:

$$(a, b) \sim (c, d) \Leftrightarrow ad = bc$$

Рациональное число — это класс эквивалентности пары (a, b) .

Множество всех рациональных чисел обозначается $\mathbb{Q} = \frac{\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})}{\sim}$.

Целое число z отождествляется с классом пары $(z, 1)$. Это задаёт инъекцию $\mathbb{Z} \rightarrow \mathbb{Q}$, сохраняющую сложение и умножение.

Арифметические операции:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd},$$

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}.$$

С этими операциями $\mathbb{Q}$ образует поле. Каждый ненулевой элемент $\frac{a}{b}$ имеет обратный $\frac{b}{a}$.

ЛОВУШКА Почему деление на ноль запрещено

«Доказательство» равенства $1 = 2$ показывает, к чему приводит деление на ноль. Пусть $a = b$. Умножим на a , вычтем b^2 , разложим на множители:

$$\begin{aligned} a &= b \\ a^2 &= ab \\ a^2 - b^2 &= ab - b^2 \\ (a + b)(a - b) &= b(a - b) \\ a + b &= b \\ 2b &= b \\ 2 &= 1 \end{aligned}$$

Дыра на шаге сокращения на $a - b$. Из $a = b$ следует $a - b = 0$. Деление на ноль — незаконное преобразование. В поле обратный существует для каждого ненулевого элемента. У нуля обратного нет.

Пример Рациональные числа как классы пар

Дробь $\frac{1}{2}, \frac{2}{4}, -\frac{3}{-6}$ — один и тот же класс:

$$(1, 2) \sim (2, 4) \sim (-3, -6).$$

Проверим попарно:

1. $(1, 2) \sim (2, 4)$: произведения равны. А именно, $1 \cdot 4 = 2 \cdot 2 = 4$.

2. $(1, 2) \sim (-3, -6)$: произведения равны. А именно, $1 \cdot (-6) = 2 \cdot (-3) = -6$.

Все три пары задают одно и то же рациональное число: $\frac{1}{2}$.

$\mathbb{Q}$ плотно: между любыми двумя различными рациональными числами есть ещё одно. Для $p < q$ их среднее арифметическое $\frac{p+q}{2}$ лежит строго между ними. Оно рационально — сумма и деление рациональных чисел дают рациональное.

ЛОВУШКА Плотность $\mathbb{Q}$ не означает отсутствие дыр

Из того, что между любыми двумя рациональными числами есть третье, не следует, что рациональных чисел достаточно для непрерывности. Плотность — свойство порядка: между любыми двумя элементами найдётся третий. Полнота — свойство сходимости: каждая ограниченная монотонная последовательность имеет предел. $\mathbb{Q}$ плотен, но не полон. $\sqrt{2}$ — дыра, которую рациональными числами не заполнить.

Конкретный пример: функция $f(x) = x^2 - 2$ непрерывна на $\mathbb{Q}$. Она принимает отрицательное значение в 1 и положительное в 2. Но корня в $\mathbb{Q}$ у неё нет. $\sqrt{2}$ — не рациональное число. Чтобы заполнить такие «дыры», нужен следующий шаг.

Проверка Построение рациональных чисел

1. Чем плотность $\mathbb{Q}$ отличается от полноты, и почему наличие третьего числа между любыми двумя не заполняет дыру $\sqrt{2}$?
2. Почему сокращение на $a - b$ в «доказательстве» $1 = 2$ незаконно? Как это связано с отсутствием обратного у нуля?

§ 22.6 Сечения Дедекинда

— Так ты всё-таки добрался до финиша нашей дистанции? — спросила Черепаха. — Хотя она и состоит из бесконечного ряда расстояний? Я думала, какой-то всезнайка уже доказал, что это невозможно?

— Это возможно, — сказал Ахилл, — и уже сделано! *Solvitur ambulando*. Понимаешь, расстояния всё время уменьшались...

— Льюис Кэрролл

Мы установили, что в $\mathbb{Q}$ есть дыры: например, $\sqrt{2}$ — не рациональное число. Их заполнение и будет предметом этого раздела.

Рихард Дедекинд в 1872 году опубликовал работу «*Stetigkeit und irrationale Zahlen*» — «Непрерывность и иррациональные числа». Его идея: каждое действительное число r однозначно задаётся тем, *какие* рациональные числа меньше него. Множе-

ство $\{q \in \mathbb{Q} \mid q < r\}$ — это «сечение» рациональной прямой. Не мы определяем r через сечение. Мы *объявляем* само сечение числом r .

ОПРЕДЕЛЕНИЕ 22.7 Сечение Дедекинда

Подмножество $\alpha \subset \mathbb{Q}$ называется **сечением Дедекинда**, если выполнены три условия:

1. **Непустота и собственность.** $\alpha \neq \emptyset$ и $\alpha \neq \mathbb{Q}$.
2. **Замкнутость вниз.** Если $q \in \alpha$ и $p < q$, то $p \in \alpha$.
3. **Отсутствие наибольшего элемента.** Для любого $q \in \alpha$ существует $p \in \alpha$ с $p > q$.

Примечание Зачем условие отсутствия наибольшего элемента

Первые два условия уже дают непустые собственные подмножества, замкнутые вниз. Без третьего у каждого рационального числа было бы два сечения: $\{q \mid q < r\}$ и $\{q \mid q \leq r\}$. Оба множества отвечают одному и тому же числу r , но различны как множества, и в $\mathbb{R}$ появилось бы по два изображения каждого рационального числа. Запрет наибольшего элемента оставляет из этой пары только $\{q \mid q < r\}$, и каждое рациональное число получает единственное изображение. Для иррациональных сечений условие выполнено само собой: наибольшего рационального числа под $\sqrt{2}$ нет и без специального требования.

Пример Сечения Дедекинда

Сечение $\beta = \{q \in \mathbb{Q} \mid q < 1\}$ задаёт рациональное число 1. Оно непусто, замкнуто вниз, не совпадает со всем $\mathbb{Q}$ и не имеет наибольшего элемента. Каждое рациональное число r даёт сечение $\{q \in \mathbb{Q} \mid q < r\}$. Так $\mathbb{Q}$ вкладывается в действительные числа.

Но бывают сечения, которым рациональное число не соответствует: например, то, что отвечает $\sqrt{2}$ (пример ниже). Такие сечения — в точности иррациональные числа: рациональная арифметика описала объекты, которых в ней самой не было.

Рациональные сечения и иррациональные устроены одинаково — это подмножества $\mathbb{Q}$, замкнутые вниз и без наибольшего элемента. Соберём все такие подмножества в одно семейство — это и есть множество действительных чисел.

ОПРЕДЕЛЕНИЕ 22.8 Действительные числа

Действительное число — это сечение Дедекинда. Множество всех действительных чисел обозначается $\mathbb{R}$.

Рациональное число q отождествляется с сечением $\{p \in \mathbb{Q} \mid p < q\}$. Это задаёт инъекцию $\mathbb{Q} \rightarrow \mathbb{R}$, сохраняющую порядок и арифметические операции.

Порядок на $\mathbb{R}$ задаётся включением:

$$\alpha \leq \beta \Leftrightarrow \alpha \subseteq \beta.$$

Арифметические операции определяются поэлементно:

1. Сложение.

$$\alpha + \beta = \{p + q \mid p \in \alpha, q \in \beta\}.$$

2. Умножение. Для положительных сечений ($\alpha, \beta > 0$):

$$\alpha \cdot \beta = \{q \in \mathbb{Q} \mid q \leq 0\} \cup \{p \cdot q \mid p \in \alpha, q \in \beta, p > 0, q > 0\}.$$

Ограничение на положительные сомножители существенно. Иначе произведение сечения $\{q < 1\}$ само на себя содержало бы каждое рациональное число. Любое $r > 0$ записывается как $(-r)(-1)$, а оба сомножителя меньше единицы. Общий случай (с отрицательными) — разбором знаков, аналогично тому, как мы определяли умножение целых через пары. Выпишем его явно.

Если $\alpha, \beta < 0$, то $\alpha \cdot \beta = |\alpha| \cdot |\beta|$. Если ровно одно из сечений отрицательно, то $\alpha \cdot \beta = -(|\alpha| \cdot |\beta|)$. Для сечений с $\alpha = 0$ или $\beta = 0$ произведение равно нулю: $\alpha \cdot \beta = 0$.

Эти определения корректны: сумма и произведение сечений — снова сечения. Доказательство сводится к проверке трёх условий из определения сечения (непустота, замкнутость вниз, отсутствие наибольшего элемента).

Свойство $\mathbb{R}$, которого не было у $\mathbb{Q}$ — **полнота**. Каждое непустое ограниченное сверху подмножество $\mathbb{R}$ имеет точную верхнюю грань (*least upper bound*). Для сечений это следует прямо из определения.

Доказательство

Докажем, что объединение семейства сечений само является сечением и служит точной верхней гранью.

Супремум семейства сечений — их объединение. Объединение сечений, замкнутых вниз, само замкнуто вниз. Если ни одно из сечений семейства не имеет наибольшего элемента, то и объединение его не имеет. Поэтому объединение любого семейства сечений — снова сечение (при условии ограниченности сверху).

Осталось показать, что это объединение — точная верхняя грань. Оно содержит каждое сечение семейства и содержится в любой верхней грани семейства.

Пример $\sqrt{2}$ как сечение

Сечение, соответствующее $\sqrt{2}$:

$$\alpha = \{q \in \mathbb{Q} \mid q < 0 \text{ или } q^2 < 2\}$$

Это подмножество $\mathbb{Q}$ замкнуто вниз, не имеет наибольшего элемента и не совпадает со всем $\mathbb{Q}$. В $\mathbb{R}$ оно является квадратным корнем из 2. Можно проверить, что $\alpha \cdot \alpha = 2$ в смысле умножения сечений.

Арифметизация анализа — программа, начатая Дедекиндом и продолженная Вейерштрассом — показала, что весь аппарат математического анализа сводится к свойствам натуральных чисел. Цепочка $\mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q} \rightarrow \mathbb{R}$ — последовательное расширение: алгебраическое до $\mathbb{Q}$, затем пополнение до $\mathbb{R}$. Каждый шаг добавляет то, чего не хватало предыдущей структуре:

1. $\mathbb{Z}$ добавляет решения уравнений $x + n = 0$. Вычитание становится всюду определённым.
2. $\mathbb{Q}$ добавляет решения уравнений $ax = b$. Здесь $a \neq 0$. Деление становится всюду определённым, кроме деления на нуль.
3. $\mathbb{R}$ добавляет точные верхние грани для ограниченных множеств. Это эквивалентно существованию пределов и решений уравнений вроде $x^2 = 2$.

Следующий шаг — $\mathbb{C}$, алгебраическое замыкание $\mathbb{R}$. Оно добавляет решение уравнения $x^2 + 1 = 0$. Но $\mathbb{C}$ уже алгебраически замкнуто (основная теорема алгебры), так что дальнейшие алгебраические расширения новых корней не добавляют. Вдобавок $\mathbb{C}$ нельзя сделать упорядоченным полем. На этом цепочка обрывается.

Примечание Второе пополнение: последовательности Коши

Сечения — один из двух классических путей от $\mathbb{Q}$ к $\mathbb{R}$. Второй путь достраивает рациональные числа последовательностями Коши: последовательность $a_0, a_1, \dots$ называется последовательностью Коши, если для любого $\varepsilon > 0$ все её члены с достаточно большими номерами отличаются друг от друга меньше чем на ε . Две такие последовательности объявляются эквивалентными, если их разность стремится к нулю, а действительное число — это класс эквивалентности. Последовательности $1, 1.4, 1.41, 1.414, \dots$ и $2, 1.5, 1.42, 1.415, \dots$ расходятся в начале, но их разность стремится к нулю, поэтому класс у них один: оба способа выписывания задают одно и то же действительное число. Оба пополнения приводят к одному и тому же полю: упорядоченные поля сечений и классов последовательностей Коши изоморфны. Сечения опираются на порядок $\mathbb{Q}$, последовательности — на арифметику и оценку расстояний.

Проверка Сечения Дедекинда

1. Почему сложение сечений задаётся одной формулой, а умножение требует разбора знаков?
2. Почему объединение ограниченного сверху семейства сечений само является сечением и даёт точную верхнюю грань?
3. Какое из трёх условий определения нарушает множество $\{q \mid q \leq 1\}$, и что без этого условия случилось бы с изображением числа 1?

§ 22.7 Что утверждает аксиома выбора

Аксиома выбора утверждает: из любого семейства непустых множеств можно выбрать по элементу. На первый взгляд это прямое следствие здравого смысла. Если перед вами лежит набор непустых коробок, вы можете засунуть руку в каждую и что-то достать. Отрицание выглядит абсурдным: декартово произведение непустых множеств не может оказаться пустым.

Проблема в том, что «засунуть руку» — не математическая операция.

1. Если семейство конечно — никакой аксиомы не нужно: существование функции выбора доказывается индукцией.
2. Если множества имеют внутреннюю структуру — можно определить выбор явно. Например, из любого непустого подмножества $\mathbb{N}$ всегда можно выбрать наименьший элемент (принцип наименьшего натурального). Для действительных чисел это правило не работает. Наименьшего числа нет вовсе, а «наименьшее число после единицы» не существует тем более. За 1.01 следует 1.0001, за ним 1.00000001, и так без конца. Внутренняя структура, по которой можно выбрать элемент, — редкая удача, а не правило.
3. Но если семейство бесконечно, а множества не имеют выделенной структуры, то никакое конечное правило не укажет, какой элемент выбрать из каждого множества. Аксиома выбора утверждает существование функции выбора, но не предъявляет её. Это неконструктивное существование — именно против него возражали интуиционисты и конструктивисты в начале XX века.

ОПРЕДЕЛЕНИЕ 22.9 Аксиома выбора

Для любого семейства непустых множеств $\{A_i\}_{i \in I}$ существует **функция выбора** $f : I \rightarrow \bigcup A_i$, такая что $f(i) \in A_i$ для всех i .

Эквивалентная формулировка: декартово произведение непустого семейства непустых множеств непусто:

$$\prod_{i \in I} A_i \neq \emptyset.$$

Пример Функции выбора

Для семейства из двух множеств $A_0 = \{1, 2\}$ и $A_1 = \{3\}$ функция выбора задаётся явно. Например, $f(0) = 1$, $f(1) = 3$. АС нужна там, где в каждом множестве нет выделенного элемента и выбор нельзя описать правилом.

Классический пример — выбор представителя из каждого класса эквивалентности. Два вещественных числа эквивалентны, если их разность рациональна, — это отношение на $\mathbb{R}$ по модулю $\mathbb{Q}$. Каждый класс бесконечен и не имеет «самого маленького» представителя, поэтому без АС такой выбор не задать.

Несмотря на споры, подавляющее большинство математиков принимает АС. Почему — в ремарке ниже.

§ 22.8 Эквиваленты аксиомы выбора

В ZF (без аксиомы выбора) следующие утверждения эквивалентны АС. Каждое из них могло бы быть принято как аксиома вместо АС — и теория получилась бы той же самой. Доказательства эквивалентности — материал курса теории множеств. Здесь важна формулировка.

ТЕОРЕМА 22.2 Лемма Цорна

Если в частично упорядоченном множестве каждая цепь (линейно упорядоченное подмножество) имеет верхнюю грань, то множество содержит хотя бы один максимальный элемент.

Лемма Цорна — рабочий инструмент алгебраиста. Когда нужно доказать существование максимального идеала в кольце, максимального линейно независимого подмножества или алгебраического замыкания поля, строят частичный порядок, проверяют условие цепей и применяют лемму Цорна.

ТЕОРЕМА 22.3 Теорема о вполне упорядочении (*well-ordering theorem*)

Каждое множество может быть вполне упорядочено — то есть на любом множестве можно задать такой линейный порядок, в котором каждое непустое подмножество имеет наименьший элемент.

Эта теорема связывает АС с ординалами: если каждое множество можно вполне упорядочить, то каждое множество равномощно некоторому ординалу. Вполне порядок на $\mathbb{R}$ не похож на привычный. В нём 10^9 может стоять раньше 0.2. Наименьшего действительного числа не существует, но первый элемент во вполне порядке есть у любого непустого подмножества. Следовательно, любые два кардинала сравнимы.

Эрнст Цермело доказал теорему в 1904 году, впервые явно сформулировав аксиому выбора, — и вызвал бурные дебаты.¹⁶⁷

ТЕОРЕМА 22.4 Теорема Тихонова

Произведение любого семейства компактных топологических пространств компактно в топологии произведения.

Здесь встречаются понятия из топологии, которые мы не будем развивать. Этот принцип тоже эквивалентен аксиоме выбора.

ТЕОРЕМА 22.5 Существование базиса

Каждое векторное пространство имеет базис. Это верно и для бесконечномерных, например для $\mathbb{R}$ как векторного пространства над $\mathbb{Q}$.

Базис $\mathbb{R}$ над $\mathbb{Q}$ называется базисом Гамеля. Он несчётен, но его существование опирается на АС. Никто никогда не предъявил явного базиса Гамеля — и не сможет. В модели Соловея (без АС) $\mathbb{R}$ над $\mathbb{Q}$ не имеет базиса.

Эти четыре утверждения — разные грани одного принципа. Лемма Цорна удобна в алгебре, теорема Цермело — в теории множеств, теорема Тихонова — в топологии, существование базиса — в линейной алгебре. Выбор формулировки зависит от того, с какими объектами вы работаете, но логическая сила одна и та же.

§ 22.9 Парадокс Банаха–Тарского

Парадокс Банаха–Тарского (1924) — следствие аксиомы выбора, которое противоречит геометрической интуиции.

УТВЕРЖДЕНИЕ 22.6 Банах–Тарский

Трёхмерный шар единичного радиуса в $\mathbb{R}^3$ можно разбить на *пять* частей, переместить их с помощью вращений и переносов (без растяжений и деформаций) и собрать обратно в *два* шара. Каждый из двух шаров идентичен исходному — того же радиуса и объёма. Удвоение шара происходит исключительно за счёт перегруппировки частей.

Конструкция опирается на свободную группу F_2 . У неё две образующие: a и b . Свободную группу удобно представлять как слова из четырёх букв a, a^{-1}, b, b^{-1} . В словах запрещены соседние взаимно обратные буквы.

Сокращение — приём, на котором всё держится: повернуть на a , затем на a^{-1} — значит вернуться в исходную точку. Именно сокращение позволяет из части

¹⁶⁷E. Zermelo. «Beweis, daß jede Menge wohlgeordnet werden kann». Mathematische Annalen, 1904. Возражения опубликовали Борель, Лебег, Бэр и Адамар в том же журнале в 1905 году.

получить почти всё. Повернув часть $S(a)$ на a^{-1} , сотрём у каждого слова первую букву. Левое умножение на a^{-1} сокращает начальную a , и почти все слова шара останутся. Это как словарь, в котором из тома слов на букву «а» выбросили первую букву: получился снова весь словарь.

Посмотрим на конкретное слово, например $aba^{-1}b$. Умножение слева на a^{-1} даёт $a^{-1}aba^{-1}b = ba^{-1}b$: начальная буква a сократилась, слово укоротилось. Если слово начинается с a^{-1} , сокращать нечего: $a^{-1}a^{-1}ba$ после умножения на a^{-1} даёт $a^{-1}a^{-1}a^{-1}ba$, и первая буква остаётся на месте. Поворот части $S(a)$ на a^{-1} покрывает, стало быть, все слова, кроме начинающихся с a^{-1} . Одна из четырёх частей после одного поворота даёт почти всю сферу.

Эта группа действует вращениями на сфере — точнее, на всех точках сферы, кроме счётного множества неподвижных точек. Орбиты этого действия разбивают сферу на классы эквивалентности: две точки эквивалентны, если одну можно перевести в другую вращением из F_2 .

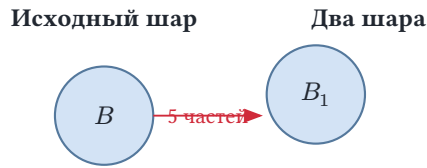
Построение идёт в несколько шагов:

1. **Выбор представителей.** Из каждой орбиты выберем по одному представителю. Здесь нужна аксиома выбора.
2. **Разбиение сферы на пять частей.** Четыре части соответствуют образующим группы и их обратным:
 - $S(a)$ — точки, достижимые из представителя применением a .
 - $S(a^{-1})$ — применением a^{-1} .
 - $S(b), S(b^{-1})$ — аналогично.

Пятая часть — множество неподвижных точек.

3. **Сборка двух сфер.** Повернём $S(a)$ с помощью a^{-1} . Получим почти всю сферу. Не хватает только точек, достижимых через a^{-1} , то есть множества $S(a^{-1})$. Аналогичные манёвры с остальными частями собирают две полные сферы.
4. **От сферы к шару.** Обобщение делается радиальной проекцией: каждая точка шара (кроме центра) однозначно проецируется на сферу. Разбиение сферы на пять частей продолжается внутрь шара вдоль радиусов. Сам центр шара неподвижен при всех вращениях, поэтому его относят к пятой части — вместе с неподвижными точками сферы. При сборке двух шаров центр едет вместе с этой частью.

Итоговое разбиение шара — на рис. 97: шар B распадается на пять частей, и движениями из этих частей собираются два шара B_1, B_2 того же радиуса, что и исходный.



Разбиение сферы на 5 частей (вращения + АС) $\rightarrow$ два шара того же радиуса.

Рис. 97. Из одного шара — два того же радиуса.

По крайней мере четыре из пяти частей, на которые разбивается шар, *неизмеримы по Лебегу* (пятая — счётное множество неподвижных точек — измерима) — для них нельзя корректно определить объём. Если бы объём был определён, то из сохранения объёма при движениях и аддитивности меры мы получили бы $1 = 2$, что действительно было бы противоречием. Но аксиома выбора позволяет строить множества, для которых понятие объёма не работает. Коротко итог парадокса записывают так — из одного шара получаются два, ничего не добавив, $1 = 2$.

Соловей (1970) построил модель теории множеств без АС, в которой все множества вещественных чисел измеримы по Лебегу.¹⁶⁸ Парадокс Банаха–Тарского невозможен без АС. Но отказ от АС делает невозможной и значительную часть функционального анализа (теорема Хана–Банаха, существование базисов Гамеля).



Аксиома выбора логически не обязательна — её принимают потому, что без неё рухнет слишком много полезных теорем. Это редкая ситуация для аксиомы: аксиому пустого множества или аксиому пары не обсуждают — их просто принимают. АС обсуждают до сих пор, и консенсус о её принятии — выбор в пользу богатства теории, а не логическая необходимость.

§ 22.10 Множество Витали

Парадокс Банаха–Тарского опирается на существование неизмеримых множеств — частей шара, которым нельзя приписать объём. Откуда вообще берутся такие множества? Джузеппе Витали в 1905 году построил первый явный пример: подмножество отрезка $[0, 1)$, которому нельзя приписать длину — *множество Витали*. Чтобы понять конструкцию, разберёмся сначала, что такое длина и почему она определена не для всякого множества.

Длина интервала — разница его концов: $[0, 1)$ имеет длину 1, $[2, 5)$ — длину 3. Длина определена и для множеств сложнее интервалов. Какова длина множества чисел из $[0, 1)$, у которых первая цифра дробной части равна 1? Это полуинтервал $[0.1, 0.2)$ длины 0.1, но вопрос сформулирован через свойство цифр, а не через геометрию.

¹⁶⁸R. Solovay. «A model of set-theory in which every set of reals is Lebesgue measurable». *Annals of Mathematics*, 1970. Модель требует аксиомы зависимого выбора (DC) и существования недостижимого кардинала.

Множество чисел, у которых вторая цифра дробной части равна 1, — уже не интервал: это десять полуинтервалов длины 0.01 каждый, и его длина 0.1 складывается из длин частей. Чтобы отвечать на такие вопросы, нужна длина как функция на множествах, а не формула для отрезков.

Длина — не мощность. У $[0, 1)$ и $[0, 2)$ мощность одна и та же (континуум), а длины разные: 1 и 2. У конечного множества точек длина 0. У счётного плотного множества (рациональные числа в $[0, 1)$) длина тоже 0, хотя точек бесконечно много. Мощность считает элементы, длина измеряет место множества на прямой.

Формализует длину *мера Лебега*: функция, сопоставляющая каждому измеримому множеству неотрицательное число и подчиняющаяся правилам:

1. неотрицательность: мера множества не меньше 0.
2. длина интервала $[a, b)$ равна $b - a$.
3. счётная аддитивность: мера объединения попарно непересекающихся измеримых множеств равна сумме их мер.
4. инвариантность к сдвигу: сдвиг множества не меняет его меры.

Правила выглядят безобидно, но выполнить их сразу для всех подмножеств $[0, 1)$ нельзя: некоторые множества не поддаются измерению. Идея конструкции проста.

Разобьём $[0, 1)$ на «рациональные слои»: две точки попадают в один слой, если их разность рациональна. В одном слое лежат точки, отличающиеся друг от друга на рациональное число. Из каждого слоя выберем по одной точке и соберём их в множество V . Правила выбора нет, поэтому нужна аксиома выбора.

Сдвинем V на каждое рациональное число из $[0, 1)$ «по модулю 1», заворачивая результат обратно в $[0, 1)$, как стрелка часов: 13:00 — это 1:00. Получится счётное семейство попарно непересекающихся копий V , покрывающее весь $[0, 1)$. Если бы у V была длина m , у каждой копии была бы та же длина m : сдвиг не меняет длину. Сумма счётного числа одинаковых слагаемых m равна 0 при $m = 0$ и бесконечности при $m > 0$. Но копии покрывают $[0, 1)$ длины 1, поэтому сумма их длин обязана быть единицей. Противоречие: длины у V нет.

Детали — в доказательстве теоремы.

ТЕОРЕМА 22.7 Множество Витали

Существует подмножество $V \subset [0, 1)$, которому нельзя приписать длину (меру Лебега) неппротиворечивым образом. Такое множество называется *неизмеримым*.

Доказательство

Докажем построением множества Витали. Построение разобьём на четыре шага.

Шаг 1: классы эквивалентности. На полуинтервале $[0, 1)$ введём отношение:

$$x \sim y \Leftrightarrow x - y \in \mathbb{Q}.$$

Два числа эквивалентны, если их разность рациональна. Это отношение рефлексивно, симметрично и транзитивно — отношение эквивалентности. Оно разбивает $[0, 1)$ на классы эквивалентности — «рациональные слои»: $x \sim y$ означает, что x и y лежат в одном слое.

Посмотрим на конкретный класс, например на класс точки 0.3. Он содержит 0.55 (разность 0.25 рациональна), 0.05 (разность -0.25 рациональна) и $0.3 + \frac{1}{3} = \frac{19}{30}$. Вообще, в класс попадают все точки $[0, 1)$, отличающиеся от 0.3 на рациональное число. Каждый класс счётен: он является подмножеством сдвига $x + \mathbb{Q}$, а сдвиг счётного множества счётен. Самых классов несчётно много: их объединение — весь $[0, 1)$, а объединение счётного числа счётных множеств счётно.

Шаг 2: выбор представителей. Представим классы как ящики, в каждом из которых лежит счётное плотное множество точек. По аксиоме выбора из каждого ящика возьмём ровно один предмет — представителя класса. Соберём выбранные точки в множество V — *множество Витали*.

Почему без аксиомы выбора не обойтись? Представителя нельзя указать правилом. Класс плотен: между любыми двумя его точками лежит ещё точка класса, поэтому «первого» элемента в классе нет и правило «возьми наименьший» не работает. Единственное исключение — класс рациональных чисел, который начинается с нуля, но это один класс из несчётного числа. Выбор из каждого класса — несчётное число одновременных произвольных решений, и именно такую возможность провозглашает аксиома выбора. Множество V существует, но его конкретный вид не предъявлен: он зависит от сделанного выбора.

Шаг 3: сдвиги по модулю 1. Для каждого рационального числа q из $[0, 1)$ определим сдвиг множества V на q по модулю 1:

$$V_q = \{v + q \bmod 1 \mid v \in V\}.$$

Запись $x \bmod 1$ означает дробную часть числа x : если $v + q \geq 1$, вычитаем 1, и результат снова лежит в $[0, 1)$. Это как стрелка часов: $0.9 + 0.3 = 1.2$, а после оборота стрелка показывает 0.2.

Семейство $\{V_q \mid q \in \mathbb{Q} \cap [0, 1)\}$ обладает двумя свойствами: дизъюнктностью и покрытием. Качественно: две разные копии не могут пересечься. Если бы точка x лежала в V_q и в $V_{q'}$, она получалась бы из двух представителей одного слоя, а в слое представитель один. Проверим формально.

Допустим, $x \in V_q \cap V_{q'}$. Тогда $x = v + q \bmod 1 = v' + q' \bmod 1$. Отсюда $v - v'$ рационально.

Но v и v' принадлежат V , а V содержит ровно по одному представителю из каждого класса. Если v и v' из разных классов, их разность не может быть рациональной. Значит, $v = v'$, и тогда $q = q'$.

Покрывтие тоже сначала опишем словами. Каждая точка $x \in [0, 1)$ лежит в своём слое. Представитель v этого слоя отличается от x на рациональное число r , и сдвиг V на r с заворачиванием доставляет копию, содержащую x .

Формально: класс эквивалентности x представлен в V некоторым элементом v . Разность $r = x - v$ рациональна. Приведём её в $[0, 1)$.

Если $r \geq 0$, положим $q = r$. Если $r < 0$, положим $q = r + 1$. Тогда $x \in V_q$.

Итак, счётное семейство попарно непересекающихся множеств $\{V_q\}$ покрывает $[0, 1)$.

Шаг 4: противоречие с мерой. Предположим, что у V есть длина m . Каждая копия V_q получается из V сдвигом, а сдвиг не меняет длину: $\mu(V_q) = m$ для каждого q . Мера счётно-аддитивна: мера объединения попарно непересекающихся множеств равна сумме их мер. Семейство копий попарно не пересекается и покрывает $[0, 1)$ длины 1, поэтому сумма их длин обязана быть равна 1. Следовательно,

$$1 = \mu([0, 1)) = \sum_{q \in \mathbb{Q} \cap [0, 1)} \mu(V_q) = \sum_{q \in \mathbb{Q} \cap [0, 1)} m.$$

Теперь наступает решающий момент. В сумме счётное число одинаковых слагаемых m . Если $m = 0$, сумма равна 0 — сумме счётного числа нулей. Если $m > 0$, сумма бесконечна: счётное число одинаковых положительных слагаемых. Ни 0, ни ∞ не равны 1. Противоречие.

Полученное противоречие означает, что V не может быть измеримым по Лебегу. Его длина не равна ни нулю, ни положительному числу — она *не определена*.

Множество Витали — прямой пример того, как аксиома выбора порождает объекты вне классической измеримости. АС используется ровно один раз — на шаге 2 — и этого достаточно. Сама конструкция не предъявляет явного вида V : она утверждает, что такое множество *существует*, но не даёт построения. Мы никогда не сможем «нарисовать» множество Витали или описать его формулой. В этом — неконструктивная природа аксиомы выбора: она гарантирует существование объектов, которые невозможно явно построить.

Что именно сломало меру? Противоречие собралось из трёх компонентов:

1. инвариантность длины к сдвигу: копии V_q имеют ту же длину, что и V .
2. счётная аддитивность: сумма длин копий обязана быть длиной их объединения.
3. выбор представителей без правила: само существование V .

Каждый компонент необходим: без сдвиговой инвариантности не было бы равенства $\mu(V_q) = m$, без счётной аддитивности сумма длин не была бы длиной объединения. Без аксиомы выбора множество V могло бы не существовать: в модели Соловея все множества измеримы.

Мера Лебега — обобщение длины, и измеримые множества устроены как «почти интервалы». Каждое из них либо собирается из интервалов счётными операциями объединения, пересечения и дополнения, либо отличается от такой постройки лишь на множество меры нуль. Множество Витали лежит вне этого класса: его нельзя собрать из интервалов даже с точностью до меры нуль, и именно поэтому длина на него не распространяется.

Неизмеримость — источник парадокса Банаха–Тарского. По крайней мере четыре из пяти частей, из которых перегруппировкой собираются два шара, неизмеримы, поэтому сохранению объёма не на что опереться: объём этих частей не определён. Два раздела — две стороны одного явления: аксиома выбора создаёт множества, для которых не работают ни длина, ни объём.

§ 22.11 Континуум-гипотеза

Мы знаем, что $|\mathbb{N}| < |\mathbb{R}|$. Естественный вопрос: а есть ли что-то между ними? Существует ли множество X с $|\mathbb{N}| < |X| < |\mathbb{R}|$? Кантор потратил годы, пытаясь доказать, что такого множества нет, но безуспешно. В 1900 году Гильберт поставил этот вопрос первым в своём списке 23 проблем.

УТВЕРЖДЕНИЕ 22.8 Континуум-гипотеза

Между $\aleph_0$ и $2^{\aleph_0}$ нет промежуточных кардиналов. Эквивалентно: любое несчётное подмножество $\mathbb{R}$ равномощно самому $\mathbb{R}$.

УТВЕРЖДЕНИЕ 22.9 Обобщённая континуум-гипотеза

$2^{\aleph_\alpha} = \aleph_{\alpha+1}$ для всех ординалов α .

Ответ пришёл с неожиданной стороны: СН нельзя ни доказать, ни опровергнуть в ZFC. Это не временная трудность — это свойство самой аксиоматики. История доказательства независимости СН — отдельная страница математики XX века.

22.11.1 Совместность континуум-гипотезы

В 1940 году Курт Гёдель доказал, что СН *совместима* с ZFC: добавив СН к аксиомам, нельзя получить противоречие (если ZFC вообще непротиворечива).

Для этого Гёдель построил *конструктивный универсум* (*constructible universe*) L — модель теории множеств, в которой все множества строятся явно, шаг за шагом, из более простых.

1. В L каждое множество появляется на некотором ординальном шаге через определённые операции над ранее построенными множествами.
2. Гёдель показал, что L удовлетворяет всем аксиомам ZFC.
3. Вдобавок в L истинна GCH (а значит, и СН).

Естественное возражение: «как можно *доказать*, что утверждение нельзя опровергнуть, и почему это не есть его доказательство?» Результат Гёделя означает: если бы CH *опровергалась* в ZFC, то ZFC была бы противоречива — из аксиом следовало бы одновременно CH и не- CH . Но из непротиворечивости ZFC не следует истинность CH . Следует лишь, что CH нельзя опровергнуть.

Оставался вопрос: а можно ли CH *доказать*?

22.11.2 Независимость континуум-гипотезы

В 1963 году Пол Коэн доказал, что отрицание CH также совместимо с ZFC. Вместе с результатом Гёделя это означает, что CH *независима* от стандартной аксиоматики: её нельзя ни доказать, ни опровергнуть.

Для доказательства Коэн изобрёл новую технику — *форсинг* (forcing). Идея в следующем.

1. **Исходная модель.** Возьмём счётную модель M теории ZFC. Мы хотим расширить M до модели $M[G]$, в которой CH ложна. То есть между $\aleph_0$ и $2^{\aleph_0}$ есть промежуточные кардиналы.
2. **Добавление новых множеств.** Для этого мы добавляем в M новые подмножества $\mathbb{N}$. Их нет в M . Добавлять их нужно аккуратно: каждое новое множество влияет на истинность утверждений в расширенной модели, и мы должны контролировать этот процесс.
3. **Множество условий.** Коэн строит частично упорядоченное *множество условий* P . Элементы P — конечные фрагменты информации о новых множествах. Эти фрагменты не противоречат друг другу и согласованы с тем, что мы уже знаем о множествах в M .
4. **Генерический фильтр.** Внешним диагональным аргументом строится подмножество $G \subset P$. Оно проходит через все плотные подмножества P и не содержит противоречивых условий. Модель M счётна. Поэтому плотные подмножества P , лежащие в M , перечислимы. По ним идут убывающей последовательностью условий. G не обязан принадлежать исходной модели M . Он приходит извне.
5. **Расширенная модель.** $M[G]$ — наименьшая модель ZFC, содержащая M и G . Все аксиомы ZFC в ней выполняются.

Выбирая подходящее P , Коэн построил модель, в которой $2^{\aleph_0} \geq \aleph_2$. То есть между $\aleph_0$ и континуумом есть как минимум один промежуточный кардинал ($\aleph_1$). В этой модели CH ложна.

ИСТОРИЯ Пол Коэн и самая неожиданная медаль

Пол Коэн (1934–2007) работал в Стэнфорде. До занятия теорией множеств он вёл семинар по математическому анализу и не интересовался логикой. Узнав о проблеме независимости CH , он заинтересовался и — по собственным словам —

выучил логику «с нуля» за несколько месяцев. Через год он изобрёл форсинг и доказал независимость СН.

За эту работу Коэн получил Филдсовскую медаль в 1966 году — единственный случай присуждения медали за результат в математической логике. Метод форсинга оказался универсальным: в последующие десятилетия с его помощью были доказаны десятки других результатов о независимости. Соломон Фейерман заметил: «Форсинг — это, возможно, самая важная техника в теории множеств после аксиом Цермело–Френкеля».

22.11.3 Что означает независимость?

Вообразим закон: все здания в городе — прямоугольные параллелепипеды. Из него следует, что число вершин минус число рёбер плюс число граней равно двум. Но не следует, что здания красные: бывает город из красных параллелепипедов и город из синих, и оба подчиняются закону. Цвет зданий — независимое свойство: закон ничего о нём не говорит. Так и ZFC ничего не говорит о континуум-гипотезе: есть вселенная, где она истинна, и вселенная, где ложна.

Независимость СН — факт о теории множеств. ZFC достаточно сильна, чтобы формализовать практически всю современную математику, но недостаточно сильна, чтобы ответить на вопрос «сколько точек на прямой». Это не значит, что вопрос лишён смысла. Это значит, что ZFC — не окончательная аксиоматика.

Платонистская позиция, которой придерживался Гёдель, исходит из того, что СН имеет определённое истинностное значение. L и модели форсинга — разные вселенные: в одной СН истинна, в другой ложна. Вопрос в том, какая из них соответствует математической реальности. ZFC просто недостаточно сильна, чтобы это установить. Гёдель надеялся, что будут найдены новые аксиомы — столь же естественные и убедительные, как аксиомы ZFC — которые позволят разрешить СН.

Формалистская позиция утверждает, что вопрос об истинности СН не имеет смысла вне конкретной модели. СН истинна в L и ложна в моделях форсинга — и обе одинаково легитимны как математические вселенные. Математик волен выбирать, в какой вселенной работать, в зависимости от того, какие теоремы ему нужны.

Программа поиска новых аксиом не привела к консенсусу. Среди кандидатов:

1. **Большие кардиналы** — утверждения о существовании «очень больших» множеств, далеко выходящих за пределы ZFC.
2. **Аксиома проективной детерминированности** — утверждение о разрешимости бесконечных игр с определённой структурой.
3. **Аксиома (*) Вудина** — максимальностный принцип о множествах наследственной мощности $< \aleph_2$. В присутствии больших кардиналов она влечёт $2^{\aleph_0} = \aleph_2$, то есть отрицание СН.

Ни один из кандидатов не является настолько же естественным и убедительным, как аксиомы ZFC.

СН остаётся открытым вопросом — в том смысле, что мы не уверены, правильно ли поставлен *сам вопрос*. У нас есть точная аксиоматика, формализующая почти всю математику, — и эта аксиоматика принципиально неполна в вопросе о мощности континуума.

Независимость СН — одновременно триумф и вызов. Триумф — потому что мы поняли границы ZFC. Вызов — потому что мы не знаем, как их преодолеть. Один из путей к преодолению — большие кардиналы.

§ 22.12 Недостижимые кардиналы и аксиомы больших кардиналов

В главе о мощности мы строили лестницу кардиналов: из любого кардинала κ булеан делает больший. А именно, $2^\kappa > \kappa$. Лестница не кончается, и возникает вопрос: существует ли кардинал, которого она не в состоянии достичь?

Его называют *недостижимым*. Он несчётен. Сильный предел и регулярность из главы 7 означают: булеан от любого $\lambda < \kappa$ снова даёт кардинал меньше κ . То же верно для объединения менее чем κ кардиналов, каждый из которых меньше κ . Счётная ω — сильный предел и регулярна, но не недостижима: несчётность — часть определения.

22.12.1 Ранняя вселенная V_κ

Понять, что это значит, помогает построение фон Неймана, с которого начиналась эта глава. Ординалы строятся по рецепту «множество всех меньших» и проходят через пределы (глава 29). По тому же рецепту устроены все множества: кумулятивная иерархия

$$V_0 \subseteq V_1 \subseteq V_2 \subseteq \dots$$

начинается с пустого множества, следующий уровень — булеан предыдущего, а на предельном ординале — объединение всех предыдущих уровней.

Пример Первые уровни

$V_0 = \emptyset$: пусто, из него ещё ничего не построено. $V_1 = \{\emptyset\}$ содержит единственное множество. V_2 — два, и так далее. V_ω — объединение всех конечных уровней. Это наследственно конечные множества — такие, чьи элементы и элементы элементов конечны — в том числе все натуральные числа фон Неймана. $V_{\omega+1}$ добавляет подмножества V_ω . Здесь уже континуум — число подмножеств счётного множества. Так строится вся иерархия.

Уровень V_κ — «ранняя вселенная». Это все множества, которые удаётся построить, не поднимаясь выше κ . Замкнутость недостижимого кардинала делает эту раннюю вселенную самодостаточной. Внутри V_κ снова выполняются все аксиомы ZFC.

Регулярность κ обеспечивает аксиому подстановки (*axiom of replacement*). Образ множества размера меньше κ снова имеет размер меньше κ . Сильный предел обеспечивает аксиому степени. Булеан любого множества из V_κ тоже лежит в V_κ . Мир, построенный до κ , сам оказывается моделью всей теории.

22.12.2 Почему существование недоказуемо

Из этой модели выводится недоказуемость существования κ . Если бы ZFC доказывала существование κ , она строила бы модель V_κ для себя самой.¹⁶⁹ Каждая формула, доказуемая в ZFC, истинна в любой её модели.

В том числе в V_κ . Противоречие не может быть истинным ни в какой модели. Значит, из существования V_κ следовала бы непротиворечивость ZFC. Тогда ZFC доказывала бы собственную непротиворечивость. Но по второй теореме Гёделя (глава 26) непротиворечивая теория, содержащая арифметику, не может доказать собственную непротиворечивость. Противоречие.

Существование недостижимого кардинала приходится принимать как аксиому. Это первый и простейший пример **аксиомы большого кардинала**: мы постулируем существование множества настолько огромного, что конструкция фон Неймана его не достигает — и получаем модель теории, которую сама теория построить не могла.

22.12.3 Иерархия больших кардиналов

Приняв одну такую аксиому, можно принять и следующую: постулировать кардинал, замкнутый ещё сильнее. Так складывается иерархия аксиом большого кардинала.

- **Кардиналы Мало:** ниже κ недостижимые кардиналы лежат так плотно, что любой класс, протянувшийся вплоть до κ без дыр вверху, задевает один из них. Плотность здесь означает, что таких кардиналов «много» в смысле стационарных множеств.
- **Неописуемые кардиналы:** никакая формула логики высших порядков не выделяет κ среди меньших кардиналов. Для любой такой формулы найдётся меньший кардинал, ей удовлетворяющий.
- **Измеримые кардиналы:** на них существует нетривиальная двузначная κ -аддитивная мера. Каждому подмножеству κ приписывается 0 или 1, одноэлементным — 0, а объединение менее чем κ множеств меры 0 снова имеет меру 0. Измеримый кардинал противоречит аксиоме конструкбельности $V = L$: мир с ним устроен иначе, чем конструктивный.

¹⁶⁹В отличие от универсума V , который является собственным классом, V_κ — множество. Его существование следует в ZFC из существования κ .

Каждая ступень иерархии — новая аксиома, из прежних не следующая. Очередной кардинал постулируется — решается, что он существует, — и иерархия показывает, как далеко можно уйти от ZFC осмысленными, но недоказуемыми постулатами.

22.12.4 Большие кардиналы и континуум

Некоторые из этих аксиом выбирают ответ на открытый вопрос о континууме. Программу новых аксиом мы обсуждали в разделе о континуум-гипотезе. Большие кардиналы — один из её кандидатов. Аксиома (*) Вудина даёт $2^{\aleph_0} = \aleph_2$. Это конкретное значение мощности континуума. Её непротиворечивость доказана при условии существования собственного класса кардиналов Вудина. Недостижимый кардинал — первая ступень этого пути. Но выбор аксиомы остаётся предметом спора, как и сама независимость континуум-гипотезы.

§ 22.13 Итоги главы

Вопрос о том, что такое число, получил два ответа: аксиоматический — свойства задаются аксиомами Пеано — и конструктивный — числа собираются из множеств. Натуральные числа выросли из пустого множества ординалами фон Неймана, целые и рациональные — из классов эквивалентности пар, действительные — из сечений Дедекинда. Конструкция показала, что материал на структуру не влияет: любые две алгебры Дедекинда изоморфны, поэтому выбор материала ничего не меняет.

Цепочка $\mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q} \rightarrow \mathbb{R}$ прослежена до конца: из простейшего понятия натурального числа вырастает весь аппарат анализа, с полнотой, которой не было у рациональных. Аксиоматика Пеано в логике первого порядка оставляет нестандартные модели — предмет главы 29 — тогда как аксиоматика второго порядка категорична.

Верхняя часть — границы метода: континуум-гипотеза независима от ZFC, а существование недостижимого кардинала лежит за пределами ZFC вовсе, и его приходится постулировать отдельной аксиомой. Аксиома выбора со своими эквивалентами — леммой Цорна и теоремой Цермело — открыла область неконструктивных существований, из которых вырастают парадокс Банаха–Тарского и множество Витали, а из постулатов о больших кардиналах складывается иерархия, уходящая вверх от ZFC.

Конструкция чисел дала ответ, из чего собраны объекты анализа, и показала, где аксиоматический метод достигает границы. Следующий предельный вопрос курса — что можно вычислить — и глава 23 начнёт ответ с простейшей модели: конечного автомата.

Проверка Проверьте себя

1. Почему аксиом первого порядка недостаточно, чтобы исключить нестандартные модели?

2. Что такое сечение Дедекинда и как оно задаёт действительное число?
3. Чем ординалы фон Неймана отличаются от просто чисел?
4. Как из пар натуральных чисел возникают целые, и почему $(a, b) \sim (c, d)$ при $a + d = b + c$?
5. Почему парадокс Банаха–Тарского невозможен без аксиомы выбора?
6. Что означает независимость континуум-гипотезы от ZFC?
7. Почему существование недостижимого кардинала нельзя доказать в ZFC?

Что дальше?

Мы построили числа от натуральных до действительных и увидели границы аксиоматического метода — континуум-гипотезу. Теперь мы переходим к другому предельному вопросу: что можно вычислить? В главе 23 мы начнём с простейшей модели — конечных автоматов.

§ 22.14 Упражнения

Аксиомы Пеано и алгебры Дедекинда.

1. Проверьте, что $(\mathbb{N}, 0, n \mapsto n + 1)$ является алгеброй Дедекинда: последовательно проверьте все пять условий определения.
2. Покажите, что любые две алгебры Дедекинда изоморфны, явно построив изоморфизм через рекурсию.
3. * Почему аксиоматика Пеано первого порядка допускает нестандартные модели, а второго порядка — нет? Какое свойство логики первого порядка за это отвечает?

Построение целых и рациональных чисел.

4. Проверьте, что отображение $n \mapsto [(n, 0)]$ является гомоморфизмом моноидов из $\mathbb{N}$ в $\mathbb{Z}$. Проверьте его инъективность. Оно сохраняет сложение и нуль.
5. Докажите, что $\mathbb{Z}$ как множество классов эквивалентности пар (a, b) является кольцом. Проверьте ассоциативность и коммутативность сложения, дистрибутивность.
6. Докажите, что $\mathbb{Q}$ является полем. Напомним: $\mathbb{Q}$ — это множество классов эквивалентности пар (a, b) с $b \neq 0$. Проверьте существование обратного элемента для умножения.
7. * Объясните, почему сокращение на $a - b$ в «доказательстве» равенства $1 = 2$ незаконно. Как это связано с отсутствием обратного у нуля в определении поля?

Действительные числа и сечения Дедекинда.

8. Докажите, что сечение Дедекинда, соответствующее $\sqrt{3}$, действительно является $\{q \in \mathbb{Q} \mid q < 0\}$ или сечением. А именно, $q^2 < 3\}$: проверьте три условия определения.

9. Докажите, что $\mathbb{R}$ как множество сечений Дедекинда является полным упорядоченным полем. Всякое непустое ограниченное сверху подмножество $\mathbb{R}$ имеет точную верхнюю грань. Подсказка: супремум семейства сечений — их объединение.

Аксиома выбора и её следствия.

10. Объясните, почему лемма Цорна эквивалентна аксиоме выбора (в ZF). Набросайте схему доказательства: как из AC вывести лемму Цорна, и наоборот.
11. * Почему парадокс Банаха–Тарского невозможен без аксиомы выбора? В каком именно месте доказательства используется AC?
12. * Докажите, что множество Витали неизмеримо по Лебегу: повторите четырёхшаговое доказательство из текста. Где именно в доказательстве используется аксиома выбора?

Континуум-гипотеза.

13. Что означает утверждение «континуум-гипотеза независима от ZFC»? Какие два результата (и чьи) вместе дают независимость?
14. * Почему модель L Гёделя доказывает совместимость CH с ZFC, а форсинг Коэна — совместимость отрицания CH? Опишите идею каждого доказательства (без технических деталей).

$$\alpha = \{q \in \mathbb{Q} \mid q < 0\}$$

или

15. * Рассмотрим сечение $q^2 < 2\}$. Докажите, что $\alpha \cdot \alpha = 2$ в смысле умножения сечений из текста. Подсказка: докажите оба включения, для обратного используйте плотность рациональных чисел и монотонность возведения в квадрат на положительных числах.

ПРОЕКТ Длинная арифметика

Целые типа фиксированной ширины заканчиваются на 2^{64} , а натуральный ряд продолжается. Соберите библиотеку длинной арифметики: целые числа произвольного размера со сложением, вычитанием, умножением и сравнением.

① Представление

Храните неотрицательное число массивом разрядов по основанию $B = 2^{32}$: младший разряд первым, ведущие нули запрещены. Знак держите отдельным признаком — та же идея, что в представлении целого разностью a —

b : знак фиксирует, какой компонент пары больше. Нуль — единственное число с двумя знаками, зафиксируйте его за плюсом.

② **Сложение и сравнение**

Реализуйте школьное сложение с переносом и поразрядное сравнение от старших разрядов. Для целых сложение сводится к натуральным по знакам: при одинаковых знаках складываются модули, при разных — из большего модуля вычитается меньший.

③ **Вычитание и умножение**

Вычитание модулей с заёмом определите только при $a \geq b$ и выразите общий случай через знак разности модулей. Умножение — столбиком за $O(n \cdot m)$ операций с разрядами: промежуточные суммы держите в удвоенной разрядности и потом нормуйте массив.

④ **Проверка свойств**

На случайных парах проверьте кольцевые свойства: коммутативность и ассоциативность сложения, дистрибутивность, согласованность порядка со сложением. В пределах ширины встроенных типов сверяйте результаты с ними — расхождение указывает на ошибку переноса или нормализации.

⑤ **Демонстрация**

Вычислите $100!$ и число Фибоначчи F_{1000} — значения, не помещающиеся в 64 бита, и выведите их десятичную запись. Проверьте известные факты: десятичная запись $100!$ заканчивается 24 нулями, F_{kn} делится на F_n .

Итог: библиотека длинной арифметики, которая хранит целые числа массивами разрядов, корректно выполняет сложение, вычитание, умножение и сравнение без ограничения ширины и проверяет кольцевые свойства на случайных данных.

XXIII

Конечные автоматы и регулярные языки

В строку поиска вводят шаблон — и редактор подсвечивает в длинном документе каждый номер телефона, каждую дату, каждый адрес. Шаблон занимает десяток символов, ответ приходит за время, сравнимое с размером текста, и машина не хранит ни одного найденного слова. Между запросом и ответом стоит вычисление: данные, правила, результат.

В прошлой главе 22 числа собирались из множеств: из пустого — натуральные, из них — целые, из целых — рациональные, из рациональных — действительные. Теперь строятся машины, и простейшая из них держит всю память в одном числе — номере текущего состояния.

Данные, с которыми работает машина, записываются строкой — конечной последовательностью символов. Число записывается цифрами, граф — перечнем рёбер, программа — текстом, и задача о любых данных сводится к задаче о строках: *всё, что машина читает, — строка*. Строки объединяются в языки. Язык — это множество слов, и про каждое слово можно спросить, принадлежит ли оно языку. Признак, по которому слово попадает в язык, бывает разным: первая буква, чётность числа единиц, последние два символа. Ответить на этот вопрос для каждого слова — значит распознать язык.

Распознавание — это вычисление: входом служит слово, ответом — «да» или «нет». Машина читает слово слева направо и в конце отвечает, принято оно или нет. Что машине нужно помнить по ходу чтения, зависит от языка. Простейшая машина помнит только, в каком состоянии она находится. Число состояний конечно и не растёт вместе со словом. Вся её память — номер состояния: ни стека, ни счётчика, ни истории. Это *конечный автомат*, а языки, которые он распознаёт, — *регулярные языки*.

Приложения этой машины повсюду. Лексер в начале каждого компилятора разбивает текст программы на токены — и токены описываются регулярными выражениями. Поиск по шаблону в тексте прогоняет документ через автомат, собранный из регулярного выражения. Протокол связи или микросхема с конечным числом состояний — тоже конечный автомат, и вопрос о достижимости аварийного состояния решается теми же средствами. Той же схемой описывают и поведение: охранник в стелс-игре — это конечный автомат, который патрулирует, подозревает и атакует. Между состояниями его переводят события — увидел игрока, потерял его из виду. Только роль символов здесь играют происшествия. Распознавание идёт за один

проход: каждый символ — один переход, без возвратов. Все эти задачи объединяет одно: конечного описания хватает, чтобы судить о бесконечном множестве слов.

Класс регулярных языков описывается с двух сторон: машинами — детерминированными и недетерминированными автоматами — и записями — регулярными выражениями. Теорема Клини связывает эти стороны в одно: выражения и автоматы распознают одно и то же.

Теория этих машин сложилась в 1950-е годы. Через два десятилетия инженерия подхватила теорию: Кен Томпсон скомпилировал регулярное выражение в автомат для редактора, и та же конструкция легла в основу `grep`. С тех пор регулярные выражения стали частью почти каждого языка программирования.

У конечной памяти есть граница: автомат не умеет считать. Чтобы распознать язык, в котором число нулей должно совпадать с числом единиц, нужно помнить, сколько нулей уже прочитано, — а число состояний фиксировано заранее. Стек даст памяти расти вместе со словом и откроет контекстно-свободные языки (глава 24), полное снятие ограничений выведет к машинам Тьюринга (глава 25). Что можно вычислить, если вся память машины — конечное число состояний?

ОБЗОР ГЛАВЫ

В этой главе мы уточним, что такое алфавит, слово, множество всех слов Σ^* и формальный язык. Затем построим машины — детерминированный и недетерминированный конечные автоматы — и убедимся, что недетерминизм не добавляет силы: конструкция подмножеств превращает НКА в ДКА. Запишем язык регулярным выражением, и теорема Клини покажет, что выражения и автоматы описывают один и тот же класс языков. Лемма о накачке и теорема Майхилла–Нерода дадут два инструмента: доказать, что язык не распознаётся конечной памятью, и найти для распознаваемого языка самый экономный автомат.

Конечность модели делает вопросы о ней разрешимыми — и это свойство не сохранится у машин с неограниченной памятью. А на практике та же теория работает в лексерах, в поиске по регулярным выражениям и в проверке свойств конечных систем.

После этой главы вы сможете:

1. определять алфавиты, слова и формальные языки;
2. строить детерминированные и недетерминированные конечные автоматы и переводить НКА в ДКА;
3. записывать регулярные языки выражениями и применять теорему Клини;
4. доказывать нерегулярность языков леммой о накачке;
5. минимизировать автоматы и пользоваться теоремой Майхилла–Нерода.

§ 23.1 Алфавит, слова и формальные языки

Прежде чем строить машину, нужно договориться, что именно она читает. Строка, алфавит и язык уже встречались нам. Теперь каждое слово получит точное определение. Начнём с задачи, к которой будем возвращаться ещё не раз.

Пример Язык слов, заканчивающихся на 01

Над двоичным алфавитом $\{0, 1\}$ рассмотрим множество слов, у которых последние два символа — это 0, затем 1. Слова 1001, 1000: первое этому множеству принадлежит, второе — нет. Мы хотим строить машины, которые читают слово слева направо и в конце отвечают, принадлежит ли оно заданному множеству.

Итак, зафиксируем точные определения.

ОПРЕДЕЛЕНИЕ 23.1 Алфавит

Алфавитом называется конечное множество Σ символов.

Символы — неделимые значки: буквы, цифры, служебные знаки, знаки препинания.

Два алфавита встречаются в этой главе особенно часто: двоичный и буквенный — $\{0, 1\}$ и $\{a, b\}$ соответственно.

ОПРЕДЕЛЕНИЕ 23.2 Слово

Словом над алфавитом Σ называется конечная последовательность символов из Σ .

Длина слова w — число его символов, обозначается $|w|$. **Пустое слово** ε — последовательность длины ноль.

ОПРЕДЕЛЕНИЕ 23.3 Множество всех слов

Через Σ^* обозначается множество всех слов над Σ , включая ε :

$$\Sigma^* = \{w \mid w \text{ — слово над } \Sigma\}.$$

ОПРЕДЕЛЕНИЕ 23.4 Конкатенация

Конкатенацией слов u, v называется слово, полученное приписыванием всех символов v к концу u . Обозначается uv .

Степень слова определяется индукцией по длине:

$$u^0 = \varepsilon, u^{n+1} = u^n u.$$

Конкатенация переносится и на языки:

$$L_1 L_2 = \{uv \mid u \in L_1, v \in L_2\}.$$

Например, $10 \cdot 01 = 1001$.

ОПРЕДЕЛЕНИЕ 23.5 Формальный язык

Формальным языком (или просто **языком**) над алфавитом Σ называется произвольное подмножество множества всех слов: $L \subseteq \Sigma^*$.

Вернёмся к примеру. Язык слов, заканчивающихся на 01, — подмножество $\{0, 1\}^*$: оно содержит 01, 101, 1001 и не содержит 10, 1000. Пустой язык $\emptyset$ (без единого слова) и полный язык Σ^* (со всеми словами) — тоже языки.

Распознать язык — значит для каждого слова w решить, принадлежит ли оно L . Языков над конечным алфавитом несчётно много, а конечных описаний — лишь счётное множество (глава 7). Значит, большинство языков конечной записью не описать. Мы будем работать с теми, которые можно.

Проверка Формальные языки

1. Язык слов над $\{0, 1\}$, заканчивающихся на 01: принадлежит ли слово 1001 языку? А слов 1000?
2. Чему равна длина слова 0110?
3. Принадлежит ли пустое слово ε языку из первого вопроса?

Осталась главная задача: по слову узнать, принадлежит ли оно языку. Делать это будет машина, и начинаем мы с простейшей из машин — с конечного автомата, память которого уместается в номер текущего состояния.

§ 23.2 Детерминированные конечные автоматы

Чтобы узнать, заканчивается ли слово на 01, достаточно трёх фактов.

- Прочитанный префикс не даёт информации о конце — состояние q_0 .
- Префикс заканчивается на 0 — возможно, это предпоследний символ финальной пары — состояние q_1 .
- Префикс заканчивается на 01 — цель достигнута — состояние q_2 .

Эти гипотезы — состояния автомата, а переходы между ними — то, как гипотеза обновляется при чтении очередного символа.

Детерминированный конечный автомат — математическая модель устройства, которое в каждый момент находится в одном из конечного числа состояний. Название происходит от латинского *determinare* — «определять»: поведение автомата предопределено. По текущему состоянию и прочитанному символу следующее

состояние известно заранее, без всякого выбора. Именно этой определённости недетерминизм позже лишит — зато приобретёт компактность описания. Автомат читает слово слева направо, переходя из состояния в состояние по фиксированной таблице переходов, и в конце либо принимает слово, либо отвергает. Вся память автомата — только номер текущего состояния. Но этого достаточно: чтобы узнать язык, не нужно помнить всё слово, нужно помнить только то, что имеет значение для будущего.

В главе 11 мы строили комбинационные схемы без памяти. Дополненные триггерами, они образуют последовательностные схемы, поведение которых определяется историей входов. Конечный автомат — математическая абстракция такого устройства.

ОПРЕДЕЛЕНИЕ 23.6 ДКА

Детерминированным конечным автоматом (ДКА) называется пятёрка

$$M = (Q, \Sigma, \delta, q_0, F),$$

где:

- Q — конечное множество состояний,
- Σ — конечный алфавит (множество допустимых символов),
- $\delta : Q \times \Sigma \rightarrow Q$ — всюду определённая функция переходов,
- $q_0 \in Q$ — начальное состояние,
- $F \subseteq Q$ — множество принимающих (допускающих) состояний.

Функция переходов δ задаёт правило: по текущему состоянию и прочитанному символу однозначно определяется следующее состояние. Таблица переходов — это и есть задание δ в конечном виде: для каждой пары (состояние, символ) — одна строка и одна клетка.

На практике таблицу переходов часто хранят *частичной*: пара (состояние, символ) без перехода означает неявный переход в *тупиковое состояние* — состояние, из которого нет пути к принимающему. Частичный ДКА эквивалентен полному: достаточно добавить тупиковое состояние с петлями по всем символам и перевести в него все отсутствующие переходы.

Пример ДКА для чётного числа единиц

Над алфавитом $\{0, 1\}$ рассмотрим язык слов с чётным числом единиц.

Два состояния:

- q_0 — прочитано чётное число единиц (начальное и принимающее).
- q_1 — нечётное.

Переходы простые: символ 0 состояние не меняет, символ 1 переключает его на противоположное. Начальное и единственное принимающее состояние — q_0 : в пустом слове ноль единиц, а ноль — чётное число.

Прогоны:

- Слово 011 принимается: $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_0$.
- Слово 01 отвергается: $q_0 \rightarrow q_0 \rightarrow q_1$.

Функция переходов знает, что делать с одним символом. Слово — это цепочка символов, и чтобы понять, как автомат ведёт себя на целом слове, функцию переходов нужно уметь протягивать вдоль всей цепочки.

ОПРЕДЕЛЕНИЕ 23.7 Расширенная функция переходов

Расширенная функция переходов $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ доопределяет δ на целые слова и задаётся рекурсивно:

- **Базис:** пустое слово не меняет состояния

$$\hat{\delta}(q, \varepsilon) = q.$$

- **Шаг:** слово, начинающееся с символа a , обрабатывается по δ , а остаток — рекурсивно

$$\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w) \quad \text{для } a \in \Sigma, w \in \Sigma^*.$$

Пример Расширенная функция переходов в действии

На автомате для чётного числа единиц вычислим $\hat{\delta}(q_0, 01)$. Раскладывая 01 на первый символ и остаток, получаем

$$\begin{aligned} \hat{\delta}(q_0, 01) &= \hat{\delta}(\delta(q_0, 0), 1) \\ &= \hat{\delta}(q_0, 1) \\ &= \delta(q_0, 1) \\ &= q_1. \end{aligned}$$

Автомат отвергает слово 01 — в нём одна единица.

С её помощью принятие слова превращается в простую проверку принадлежности.

ОПРЕДЕЛЕНИЕ 23.8 Язык, распознаваемый ДКА

Язык, распознаваемый автоматом M , — множество всех слов, которые переводят автомат из начального в принимающее состояние:

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}.$$

Автомат принимает слово w , если $w \in L(M)$. Иначе слово отвергается. Два автомата эквивалентны, если они распознают один и тот же язык.

Пример Язык автомата

Язык автомата для чётного числа единиц — все слова над $\{0, 1\}$ с чётным числом единиц. Язык автомата из начала раздела — все слова, заканчивающиеся на 01.

ОПРЕДЕЛЕНИЕ 23.9 Регулярный язык

Язык называется **регулярным**, если он распознаётся некоторым детерминированным конечным автоматом.

Класс всех регулярных языков — это то, что автоматы способны описать: класс языков, допускающих конечную память.

Примечание

Принятие слова — это цикл по символам слова: один шаг — один табличный переход, а ответ определяется состоянием в конце.

```
/// Принятие слова детерминированным конечным автоматом.
fn accepts(
    trans: &[Vec<usize>],
    start: usize,
    accept: &[bool],
    word: &[usize],
) -> bool {
    let mut q = start;
    for &sym in word { q = trans[q][sym]; }
    accept[q]
}
```

Массив `trans` — это и есть таблица переходов δ : для каждого состояния и символа — номер следующего состояния. Если переход не определён, в таблице стоит номер тупикового состояния: оно никогда не бывает принимающим.

Пример ДКА для слов, заканчивающихся на 01

Соберём автомат из гипотез, с которых начался раздел. Состояния:

- q_0 : последние символы не образуют префикса 01 (или слово ещё пусто).
- q_1 : слово заканчивается на 0 — потенциальное начало 01.
- q_2 : слово заканчивается на 01 — принимающее состояние.

Переходы:

- $\delta(q_0, 0) = q_1, \delta(q_0, 1) = q_0$.
- $\delta(q_1, 0) = q_1, \delta(q_1, 1) = q_2$ (по нулю гипотеза остаётся).
- $\delta(q_2, 0) = q_1, \delta(q_2, 1) = q_0$ (по нулю пара позади, но начался новый ноль).

Диаграмма переходов показана на рис. 98.

Прогон слова 1001: $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_2$ — слово принимается. Слово 1000: $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_1$ — отвергается.

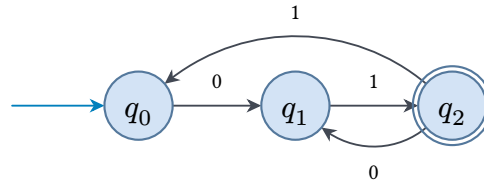


Рис. 98. ДКА для языка слов, заканчивающихся на 01.

Проверка Детерминированные конечные автоматы

1. Почему для распознавания слов, заканчивающихся на 01, автомату хватает трёх состояний, хотя сами слова бывают любой длины?
2. Почему пустое слово принимается ровно тогда, когда начальное состояние q_0 является принимающим?

§ 23.3 Недетерминированные конечные автоматы

У детерминизма есть цена: порой число состояний растёт экспоненциально. Язык, где третий с конца символ равен 1, — простейший пример такого роста (разбор ниже). *Недетерминизм* — способ описывать язык компактнее: автомат как бы «угадывает» верный путь. Как понимать это слово буквально, объясняют определение и замечание ниже.

ОПРЕДЕЛЕНИЕ 23.10 НКА

Недетерминированным конечным автоматом (НКА) называется пятёрка

$$M = (Q, \Sigma, \delta, q_0, F),$$

где:

- Q, Σ, q_0, F имеют тот же смысл, что и в ДКА,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ — функция переходов, возвращающая множество возможных следующих состояний.

Отличие от ДКА — в функции переходов: вместо одного следующего состояния она возвращает целое множество. Из состояния по символу автомат может перейти в любое из состояний этого множества, а по ε — сменить состояние спонтанно, не читая символ.

Примечание

НКА принимает слово, если существует *хотя бы один* путь вычисления из начального состояния в принимающее, следуя переходам. Пути, ведущие в

непринимаящие состояния или застревающие (нет перехода по очередному символу), игнорируются. Достаточно существования одного успешного пути.

Слово «угадывает», которым мы описали недетерминизм, легко понять превратно. На этом слове стоит остановиться прежде, чем переходить к примерам.

ЛОВУШКА НКА — не алгоритм с угадыванием

Считать, что недетерминированный автомат «угадывает» переходы или работает случайно, — ошибка. НКА — это декларативное описание: слово принимается, если существует путь до принимающего состояния. В определении языка действует лишь квантор существования, без всякой случайности или параллелизма. Конструкция подмножеств превращает НКА в ДКА без изменения языка, и этот факт показывает, что «угадывание» не добавляет вычислительной силы.

Пример НКА для слов, заканчивающихся на 01

Построим НКА для того же языка $L = \{w \in \{0, 1\}^* \mid w \text{ заканчивается на } 01\}$. Недетерминизм позволяет описать язык одной мыслью: на каждом символе 0 автомат «подозревает», что началась финальная пара 01. Он проверяет эту догадку следующим символом — тот должен быть 1, и на этом слово должно закончиться.

Состояния:

- q_0 — автомат читает слово и ждёт подходящего момента. На 0 недетерминированно выбирает: остаться в ожидании или перейти к проверке финала.
- q_1 — только что прочитан 0, возможно, предпоследний символ финальной пары.
- q_2 — слово заканчивается на 01 (принимающее).

Прогон слова 10101:

1. догадываемся, что финальная пара начинается на четвёртом символе.
2. прочитав 0, переходим в q_1 , а следующий символ 1 приводит в принимающее состояние q_2 .
3. слово заканчивается в принимающем состоянии — принято.

Догадка на втором символе привела бы в q_2 на два символа раньше конца: путь умирает, потому что из q_2 нет переходов. Принятие — существование хотя бы одного пути, поэтому слово принято. Слово 10110 отвергается: ни одна догадка не заканчивается в q_2 одновременно с концом слова.

Диаграмма этого НКА — на рис. 99. Из q_0 по символу 0 расходятся два перехода, воплощающие недетерминированный выбор.

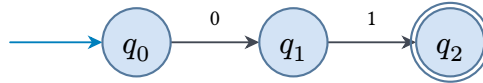


Рис. 99. НКА для языка слов, заканчивающихся на 01.

Пример НКА для слов, содержащих 00 или 11

Построим НКА для языка слов, содержащих подстроку 00 или 11.

Из начального состояния q_0 автомат «угадывает», какой из шаблонов встретится:

- По 0 переходит в q_1 (ожидание второй подряд 0).
- По 1 переходит в q_2 (ожидание второй подряд 1).
- Петли по 0 и по 1 в q_0 позволяют отложить догадку и продолжить поиск шаблона со следующей позиции.
- По второму 0 из q_1 — в принимающее состояние.
- По второй 1 из q_2 — в принимающее состояние.
- Петли по 0 и по 1 в принимающем состоянии удерживают слово до конца.

Обе ветви работают независимо.

Прогон слова 0110:

1. Первый символ 0 проходим по петле, оставаясь в q_0 .
2. На втором символе 1 угадываем шаблон 11 и переходим в q_2 .
3. Третий символ 1 приводит в принимающее состояние, которое удерживает слово до конца — слово принято.

Подстрока 11 на второй и третьей позициях — искомый шаблон.

Слово 010: ни 00, ни 11 не встречается, ни одна догадка не подтверждается — слово отвергнуто.

Диаграмма — на рис. 100. Две параллельные ветви, одна для 00, другая для 11, каждая со своим принимающим состоянием, и цикл в начальном состоянии, позволяющий искать шаблон с любой позиции.

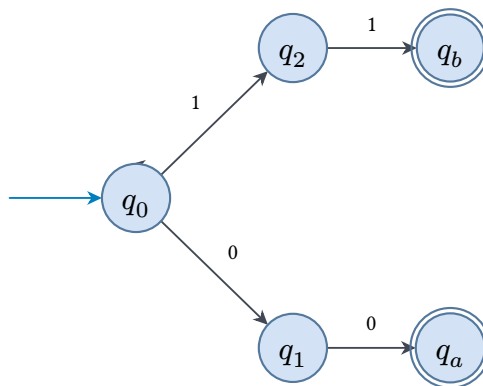


Рис. 100. НКА для языка слов, содержащих 00 или 11.

Во всех примерах этого раздела недетерминизм экономил только бумагу: те же языки распознаются и ДКА. Следующий язык показывает разницу посерьёзнее — она в самом числе состояний.

Пример НКА для «третий с конца символ равен 1»

Этот язык иллюстрирует экспоненциальный разрыв между НКА и ДКА.

Идея НКА проста: угадать момент, когда до конца слова осталось ровно три символа, проверить, что текущий символ равен 1, и убедиться, что после него идут ровно два любых символа. Требуется 4 состояния.

Минимальный ДКА для того же языка требует $2^3 = 8$ состояний: автомат должен различать все 8 возможных суффиксов длины 3. В общем случае язык « k -й с конца символ равен 1» требует 2^k состояний в ДКА. НКА для него хватает $k + 1$ состояний. Причина — в том, что ДКА должен различать все 2^k суффиксов длины k . Точную формулировку даст теорема Майхилла–Нерода из раздела ниже.

Прогон слова 1101: догадываемся на втором символе — он равен 1, и до конца осталось ровно два символа. Далее читаем два оставшихся символа и заканчиваем в принимающем состоянии. Слово 1001 отвергается: третий с конца символ равен 0, ни одна догадка не подтверждается.

ε -переходы из определения НКА — это спонтанные смены состояния: автомат меняет состояние, ничего при этом не читая. Разберём, как они работают, на простом примере.

Пример НКА с ε -переходами для a^*b^*

Рис. 101 показывает НКА с ε -переходами для языка a^*b^* : пунктирные переходы позволяют спонтанно переключиться от чтения a к чтению b .

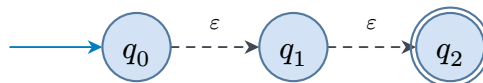


Рис. 101. НКА с ε -переходами для языка a^*b^* .

Прогон слова aab: прочитываем два символа a , по ε -переходу переключаемся на чтение b и прочитываем b — слово принято. Слово ba отвергается: после чтения b вернуться к чтению a нельзя — ε -переход работает только в одну сторону.

§ 23.4 Конструкция подмножеств

НКА выглядит мощнее ДКА, но распознают они в точности одни и те же языки. *Конструкция подмножеств* (Рабина–Скотта) превращает любой НКА в эквивалентный ДКА. Число состояний при этом может вырасти экспоненциально.

ТЕОРЕМА 23.1 Эквивалентность ДКА и НКА

Для любого НКА существует ДКА, распознающий тот же язык.

Доказательство

Докажем построением.

Пусть дан НКА $N = (Q, \Sigma, \delta, q_0, F)$. Построим ДКА, состояниями которого являются подмножества состояний НКА N :

$$D = (\mathcal{P}(Q), \Sigma, \delta', q_0', F').$$

Здесь $q_0' = \varepsilon\text{-closure}(\{q_0\})$ — ε -замыкание множества $\{q_0\}$: все состояния, достижимые из q_0 по ε -переходам.

Функция переходов ДКА: для множества $S \subseteq Q$ и символа a ,

$$\delta'(S, a) = \varepsilon\text{-closure}\left(\bigcup_{q \in S} \delta(q, a)\right).$$

То есть: из каждого состояния S берём все возможные переходы по a , собираем результаты и замыкаем по ε .

Принимающие состояния ДКА: $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$ — множества, содержащие хотя бы одно принимающее состояние НКА.

По построению, D симулирует все возможные пути N одновременно. Число состояний ДКА не превышает $2^{|Q|}$, и этот экспоненциальный рост неизбежен в худшем случае.

Конструкция подмножеств устраняет недетерминизм: любой недетерминированный конечный автомат моделируется детерминированным. Недетерминированный автомат выглядит свободнее: несколько переходов по одному символу, спонтанные ε -переходы, угадывание верного пути. От такой свободы естественно ждать большего — более широкого класса языков. Конструкция подмножеств показывает обратное.

Причина — в конечности пространства состояний. Булеан конечного множества из n элементов конечен. Он содержит ровно 2^n элементов. ДКА, построенный по конструкции подмножеств, имеет не более 2^n состояний — экспоненциально больше, чем исходный НКА, но всё ещё конечное число. Каждое состояние ДКА кодирует множество состояний, в которых НКА мог бы находиться после чтения очередного префикса слова. Детерминизм восстанавливается ценой экспоненциального роста — но не ценой потери конечности. На практике конструкция превращается в рецепт.

1. Начать с ε -замыкания начального множества $\{q_0\}$.
2. Для каждого уже созданного множества S и символа a собрать переходы по a из всех состояний S и замкнуть результат по ε — это и будет следующее состояние.

3. Новое состояние появляется только тогда, когда такого множества ещё не было.
4. Работа кончается, когда у каждого состояния заполнены все переходы.

Пример Конструкция подмножеств на примере языка 01

Применим конструкцию подмножеств к НКА из примера про 01. Начнём с множества $\{q_0\}$ и будем порождать переходы:

$$\begin{aligned}\delta'(\{q_0\}, 0) &= \{q_0, q_1\}, & \delta'(\{q_0\}, 1) &= \{q_0\}, \\ \delta'(\{q_0, q_1\}, 0) &= \{q_0, q_1\}, & \delta'(\{q_0, q_1\}, 1) &= \{q_0, q_2\}, \\ \delta'(\{q_0, q_2\}, 0) &= \{q_0, q_1\}, & \delta'(\{q_0, q_2\}, 1) &= \{q_0\}.\end{aligned}$$

Достижимы три множества, принимающее среди них одно — $\{q_0, q_2\}$, ведь оно содержит принимающее состояние q_2 НКА. Получившийся ДКА — ровно тот же автомат, что на рис. 98. Множества $\{q_0\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$ играют роли состояний q_0 , q_1 , q_2 .

Прогон слова 10110:

$$\{q_0\} \rightarrow \{q_0\} \rightarrow \{q_0, q_1\} \rightarrow \{q_0, q_2\} \rightarrow \{q_0\} \rightarrow \{q_0, q_1\}.$$

Конец — в непринимающем множестве, слово отвергается.

В примере выше у НКА не было ε -переходов, поэтому замыкание не меняло множества. На автомате для a^*b^* , показанном на рис. 101, оно работает в полную силу. ε -замыкание начального множества $\{q_0\}$ собирает все состояния, достижимые из него по ε -цепочкам: из q_0 — в q_1 , из q_1 — в q_2 . Начальное состояние ДКА — ε -closure($\{q_0\}$) = $\{q_0, q_1, q_2\}$. По символу a из этого множества достигим только q_1 (петля), и замыкание даёт $\{q_1, q_2\}$. По символу b — только q_2 (петля), и замыкание — $\{q_2\}$. Достижимы три состояния, и все они принимающие: каждое содержит состояние q_2 , а слова языка a^*b^* — ровно те, где ни одна b не стоит раньше a . В тупиковое состояние $\emptyset$ автомат попадает, прочитав a после блока b .

Примечание

Один шаг ДКА из конструкции подмножеств — это один параллельный шаг всех ветвей НКА: ДКА хранит множество всех состояний, в которых НКА мог бы находиться. Рост числа состояний экспоненциальный и неизбежен: существуют языки, где минимальный ДКА экспоненциально больше минимального НКА.

Замечание Два недетерминизма

В конечных автоматах недетерминизм устраним: любой НКА превращается в эквивалентный ДКА, и процедура всегда завершается, хотя число состояний при этом может вырасти экспоненциально. Все ветви вычисления помещаются в одно конечное множество, поэтому заикливание исключено. Для моделей с неограниченной памятью аналогичный вопрос, можно ли устранить недетер-

минимум без экспоненциального роста, остаётся открытым. К нему мы вернёмся в главах о вычислениях и сложности (глава 25, 30).

Разница между двумя моделями — в цене описания, а не в том, какие языки они распознают. Сведём эту разницу в таблицу.

Свойство	ДКА	НКА
Переход	$\delta(q, a) = \text{одно состояние}$	$\delta(q, a) = \text{множество состояний}$
ϵ -переходы	Запрещены	Разрешены (спонтанные)
Размер относительно НКА	Может быть экспоненциально больше	Обычно линейный или полиномиальный
Распознавание	Единственный путь вычисления	Хотя бы один путь ведёт к принятию
Реализация	Табличный, простой цикл	Возврат или симуляция подмножеств
Сложность построения	Сложнее (явно обрабатывать все случаи)	Проще (недетерминизм помогает)
Дополнение	Мгновенно (поменять F)	Требует предварительной детерминизации

Последняя строка таблицы — дополнение — требует отдельной оговорки. Здесь чаще всего подводит привычка, выработанная на ДКА.

ЛОВУШКА Дополнение НКА $\neq$ смена принимающих состояний

Дополнять язык НКА, поменяв принимающие и отвергающие состояния, как у ДКА, — ошибка. Для ДКА смена принимающих состояний дополняет язык: каждое слово либо принимается, либо отвергается. Для НКА тот же приём не работает: принятие — существование хотя бы одного принимающего пути, а дополнение требует отсутствия *всех* принимающих путей. Нужно сначала построить ДКА конструкцией подмножеств, затем менять принимающие состояния.

Примечание Зачем нужен НКА?

Вычислительной силы НКА не добавляют, но вводят их не за этим: выразительная сила — не единственная мера. НКА позволяет описывать язык *структурно*: объединение языков соответствует параллельному запуску двух автоматов, конкатенация — последовательному соединению, звезда Клини — закликиванию. В ДКА эти конструкции требовали бы явной детерминизации и угадывания точек склейки. Альтернативный путь — сразу работать с регулярными выражениями — теряет операциональную интуицию: автомат «читает и переходит», выраже-

ние — просто слово. НКА занимает промежуточное положение: нагляден как автомат и удобен как язык описания.

Проверка Недетерминизм и конструкция подмножеств

1. Почему язык «третий с конца символ равен 1» требует 2^3 состояний в ДКА, но всего 4 в НКА?
2. Зачем в конструкции подмножеств собирают переходы из всех состояний множества S и замыкают результат по ϵ ?

У этой теории короткая и бурная биография: три работы за четыре года.

ИСТОРИЯ Три работы за четыре года: рождение теории автоматов

Стивен Клини в 1951–1956 годах, развивая идеи нейронных сетей МакКаллока и Питтса (1943), формализовал понятие регулярного языка. Он доказал, что класс языков, описываемых регулярными выражениями, совпадает с классом языков, распознаваемых конечными автоматами. Ноам Хомский в 1956 году, работая над математической моделью естественного языка, построил иерархию грамматик — от регулярных до грамматик общего вида. Тем самым он создал структурную таксономию вычислимости.

Майкл Рабин и Дана Скотт в 1959 году опубликовали статью «Finite Automata and Their Decision Problems». В ней они ввели недетерминированные конечные автоматы (НКА) и доказали их эквивалентность детерминированным через конструкцию подмножеств. Эта работа также установила разрешимость проблем для автоматов: эквивалентность, пустота языка, минимизация. Это те вопросы, которые для более мощных моделей вычислений оказываются алгоритмически неразрешимыми.

Всего за четыре года — с 1956 по 1959 — три работы (Клини, Хомский, Рабин и Скотт) создали теорию автоматов и формальных языков как самостоятельную дисциплину. За эту работу Рабин и Скотт получили премию Тьюринга в 1976 году. Формулировка комитета особо отметила, что понятие недетерминизма стало «чрезвычайно ценным инструментом в теории сложности вычислений».

§ 23.5 Регулярные выражения

У языка есть два описания: как его *распознавать* и как его *записывать*. Автомат — распознаватель: читает слово и отвечает, входит ли оно в язык. Регулярное выражение — запись: короткая формула, из которой язык восстанавливается без машины. Такая запись нужна, даже когда автомат уже есть: она занимает десяток символов и восстанавливается в автомат без ручной работы. Стивен Клини ввёл регулярные выражения в 1951 году, чтобы не рисовать диаграммы: язык, заданный кружками и

стрелками, хотелось сжимать до строки, которую можно продиктовать по телефону. Знак повторения, звезда Клини — «ноль или более раз» — и есть сжатая запись цикла: сколько угодно раз пройти по одной и той же дуге. В формуле всего три операции: объединение (вертикальная черта), конкатенация (запись выражений подряд) и повторение (звезда).

Пример Базовые регулярные выражения

Над алфавитом $\{0, 1\}$:

- $0 \mid 1$ — язык из двух однобуквенных слов: 0 и 1.
- $(0 \mid 1)^*$ — все двоичные слова, включая пустое.
- 01^*0 — слова, начинающиеся и заканчивающиеся на 0, с любым количеством единиц между ними: 00, 010, 0110, 01110, ...
- $(0 \mid 1)^*01$ — слова, заканчивающиеся на 01 (язык, с которого началась глава).

Все записи из примера построены из одних и тех же операций — из тех, что войдут в определение.

ОПРЕДЕЛЕНИЕ 23.11 Регулярное выражение

Регулярным выражением над алфавитом Σ называется выражение, построенное по следующим правилам:

- $\emptyset$ обозначает пустой язык.
- ε обозначает язык, содержащий только пустое слово $\{\varepsilon\}$.
- Символ $a \in \Sigma$ обозначает язык $\{a\}$ (слово из одного символа).
- $R_1 \mid R_2$ обозначает объединение языков $L(R_1) \cup L(R_2)$.
- $R_1 R_2$ обозначает конкатенацию $\{uv \mid u \in L(R_1), v \in L(R_2)\}$.
- R^* обозначает итерацию (звезду Клини): ноль или более повторений R .

Приоритет операций (от высшего к низшему): *, конкатенация, $\mid$. Скобки расставляют приоритет явно: $(a \mid b)c \neq a \mid (bc)$.

Пример Код Грея как конкатенация

Рекурсивное построение кода Грея из главы 11 — уже знакомый пример конкатенации, просто мы не называли её этим словом. n -битный код Грея G_n задавался так:

$$G_{n+1} = [0G_n, 1G_n^R],$$

где запись $0G_n$ означала приписывание нуля слева к каждой строке списка G_n .

Теперь узнаём в этой записи конкатенацию. Слова, составляющие G_n , образуют конечный язык над алфавитом $\{0, 1\}$. Запись $0G_n$ — конкатенация однобуквенного языка $\{0\}$ и языка G_n :

$$\{0\}G_n = \{0w \mid w \in G_n\}.$$

Для $n = 2$ получаем: $G_2 = \{00, 01, 11, 10\}$, $\{0\}G_2 = \{000, 001, 011, 010\}$ — в точности слова трёхбитного кода, начинающиеся с 0.

Язык не различает порядок слов, поэтому обращение G_n^R — деталь *списка*, а не языка: языки $\{1\}G_n$ и $\{1\}G_n^R$ совпадают. Как множество G_n — это все двоичные слова длины n . Рекурсия Грея сводится к разбиению, верному для любого n :

$$\{0, 1\}^{n+1} = \{0\}\{0, 1\}^n \cup \{1\}\{0, 1\}^n :$$

слова длины $n + 1$ распадаются на начинающиеся с 0 и начинающиеся с 1. Конкатенация приписала слову символ слева и увеличила длину на единицу. Порядок же соседних слов — то, что делает код Грея кодом — она не несёт.

Регулярное выражение — запись для человека, но машина умеет компилировать её в конечный автомат и вести им поиск по тексту. Первым сделал это Кен Томпсон в редакторе QED (1968)¹⁷⁰. Тот же приём он перенёс в редактор ed (1971) и в утилиту грер (1973), а позже конструкция легла в основу генераторов лексеров (Lex, flex, ANTLR).

Инженерный успех Томпсона опирался на математический результат, полученный за два десятилетия до того.

ТЕОРЕМА 23.2 Теорема Клини¹⁷¹

Язык регулярен тогда и только тогда, когда он представим регулярным выражением: $AUT = REG$.

«Автоматный» и «регулярный» здесь — синонимы: по определению регулярный язык распознаётся ДКА, а теорема Клини утверждает, что в точности такие языки записываются регулярными выражениями. Распознавание и запись — две стороны одного класса.

Доказательство — два преобразования, по одному на каждое включение.

Регулярное выражение → автомат (включение $REG \subseteq AUT$). Строим по выражению ε -НКА конструкцией Томпсона: структурной индукцией по выражению (структурная индукция введена в главе 1). Для ε и отдельного символа a — автоматы из двух состояний и одного перехода, а для $\emptyset$ — из двух состояний и ни одного: из начального состояния нельзя попасть в принимающее. Для объединения, конкатенации и звезды — склейка автоматов подвыражений через ε -переходы. Та же картина, что в разделе про НКА: объединение — параллельный запуск, конкатенация — последовательное соединение, звезда — зацикливание.

¹⁷⁰K. Thompson. «Programming Techniques: Regular Expression Search Algorithm». Communications of the ACM, 1968.

¹⁷¹S. C. Kleene. «Representation of events in nerve nets and finite automata». RAND Corporation, 1951 (опубликована в сборнике «Automata Studies», 1956). Клини ввёл понятие регулярного выражения и доказал эквивалентность конечным автоматам.

Пример Конструкция Томпсона на $a \mid b$

Соберём автомат для простого объединения: $a \mid b$.

База — автоматы для отдельных символов: для a это два состояния и один переход $p_0 \rightarrow p_1$, для b — $r_0 \rightarrow r_1$.

Объединение $a \mid b$: новое начальное состояние s , из него ε -переходы в p_0 и в r_0 , принимающие состояния — p_1 и r_1 . Прогон начинается с ε -перехода, который выбирает ветвь: слово читается либо автоматом для a , либо автоматом для b . Выбор ветви недетерминирован, и это не случайность: объединение языков означает «одно из двух», а ε -переход — именно тот механизм, которым НКА записывает такой выбор.

Конкатенация и звезда склеиваются по той же схеме: последовательным соединением через ε -переход и замыканием в цикл.

Осталось превратить ε -НКА в ДКА: сначала ε -замыкание убирает ε -переходы, затем конструкция подмножеств Рабина–Скотта устраняет недетерминизм. Благодаря этому ДКА можно не строить сразу: достаточно НКА, а детерминизацию делает конструкция подмножеств.

Автомат $\rightarrow$ регулярное выражение (включение $\text{AUT} \subseteq \text{REG}$). Здесь работает алгоритм Клини, и мы разберём его по шагам.

Пронумеруем состояния автомата числами от 1 до n , начальному состоянию дадим номер 1. Обозначим через R_{ij}^k множество всех слов, которые переводят автомат из состояния q_i в состояние q_j . По пути (кроме начального и конечного состояний) разрешены только состояния с номерами не больше k . Когда $k = n$, ограничение ничем не мешает: тогда R_{ij}^n — это все слова, переводящие автомат из q_i в q_j . Язык автомата — объединение R_{1j}^n по всем принимающим состояниям q_j : путь начинается в начальном состоянии и заканчивается в принимающем.

Множества R_{ij}^k связаны рекуррентным соотношением:

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} (R_{kk}^{k-1})^* R_{kj}^{k-1}.$$

Слово, переводящее автомат из состояния q_i в состояние q_j через промежуточные состояния не выше k , либо вовсе не заходит в q_k (первое слагаемое). Либо оно заходит в q_k один или несколько раз: дойти до q_k , сколько угодно раз пройти цикл через q_k и выйти к q_j (второе слагаемое). Основание индукции:

$$R_{ij}^0 = \{a \in \Sigma \mid \delta(q_i, a) = q_j\},$$

а при $i = j$ к этому множеству добавляется пустое слово ε : из состояния в себя можно перейти, ничего не прочитав.

В записи R_{ij}^k участвуют только объединение, конкатенация и звезда — операции регулярных выражений. Индукцией по k превращаем каждое R_{ij}^k в регулярное выражение. На шаге n получаем выражение для языка автомата.

Пример Алгоритм Клини на автомате чётности

Разберём алгоритм на ДКА с двумя состояниями: q_1 — начальное и принимающее (чётное число единиц), а q_2 — нечётное. Основание индукции собирается из переходов:

$$\begin{aligned} R_{11}^0 &= \{\varepsilon, 0\}, R_{12}^0 = \{1\}, \\ R_{21}^0 &= \{1\}, R_{22}^0 = \{\varepsilon, 0\}. \end{aligned}$$

Первый шаг рекурсии ($k = 1$) разрешает промежуточное состояние q_1 :

$$\begin{aligned} R_{12}^1 &= R_{12}^0 \cup R_{11}^0 (R_{11}^0)^* R_{12}^0 \\ &= \{1\} \cup \{\varepsilon, 0\} \{\varepsilon, 0\}^* \{1\} \\ &= 0^*1. \end{aligned}$$

Упрощение последнего равенства — чистая алгебра множеств. По определению звезды $\{\varepsilon, 0\}^*$ — это все слова над алфавитом $\{0\}$, то есть 0^* . Тогда $\{\varepsilon, 0\} \{\varepsilon, 0\}^* = \{\varepsilon, 0\} 0^* = 0^*$: приписать к слову из нулей впереди пустое слово или ещё один ноль — значит снова получить слово из нулей. Каждое слово из нулей уже получается приписыванием пустого слова. Остаётся домножить на $\{1\}$ и получить 0^*1 . Множество 0^*1 — это все слова, переводящие автомат из чётного состояния в нечётное, если промежуточным может быть только состояние q_1 . Это сколько угодно нулей (оставаясь в q_1) и одна единица. На шаге $k = 2$ в игру вступает и q_2 , и рекурсия даёт полное выражение языка. Для языка автомата нужно R_{11}^2 : путь из начального состояния в принимающее, оба раза — состояние q_1 .

$$R_{11}^2 = R_{11}^1 \cup R_{12}^1 (R_{22}^1)^* R_{21}^1.$$

Вычисляем недостающие члены:

- $R_{11}^1 = \{\varepsilon, 0\}^* = 0^*$: нули, оставаясь в чётном состоянии.
- $R_{22}^1 = \{\varepsilon, 0\} \cup 10^*1$: вернуться в нечётное состояние, остаться на месте или пройти цикл с нечётным числом единиц.
- $R_{21}^1 = 10^*$: из нечётного в чётное, единица и сколько угодно нулей.

Внутри звезды далее важен лишь $0 \cup 10^*1$. Итог:

$$R_{11}^2 = 0^* \cup 0^*1(0 \cup 10^*1)^*10^*,$$

— все слова с чётным числом единиц. Для сравнения, эквивалентная запись через пары: $(0^*10^*1)^*0^*$.

Пример Самопроверка: алгоритм Клини на языке слов, заканчивающихся на 0

Прогоните алгоритм на ДКА для языка слов, заканчивающихся на 0. Состояния: q_1 — начальное (не принимающее), q_2 — принимающее, а переходы — $\delta(q_1, 0) = q_2, \delta(q_1, 1) = q_1, \delta(q_2, 0) = q_2, \delta(q_2, 1) = q_1$. Основание индукции: $R_{11}^0 = \{\varepsilon, 1\}, R_{12}^0 = \{0\}, R_{21}^0 = \{1\}, R_{22}^0 = \{\varepsilon, 0\}$. После двух шагов рекурсии должно получиться $R_{12}^2 = (0 \mid 1)^*0$ — все слова, заканчивающиеся на 0.

Выражение получается огромным, но оно существует для любого автомата.

Примечание

Алгоритм Клини — это ровно тот же приём, что и алгоритм Флойда–Уоршелла для кратчайших путей в графах. Оба — динамическое программирование, в котором разрешены промежуточные состояния с номерами не выше k . Клини описал его для автоматов в 1951 году. Флойд и Уоршелл переоткрыли его для графов в 1962-м.

Проверка Регулярные выражения

1. Чем различаются языки выражений $\emptyset$ и ε , и какие слова содержит язык $\emptyset^*$?
2. Как конструкция Томпсона склеивает автоматы для r и s в автомат для rs : что соединяет принимающее состояние первого автомата с начальным состоянием второго?
3. Что в алгоритме Клини играет ту же роль, что промежуточные вершины с номерами не выше k в алгоритме Флойда–Уоршелла, и из чего состоит основание R_{ij}^0 ?

§ 23.6 От выражения к автомату

Теорема Клини — одновременно математический факт и инженерный рецепт: язык задаётся регулярным выражением, а распознаётся конечным автоматом. Первый пример такой пары в действии — лексер, первая программа в каждом компиляторе.

Каждый компилятор или интерпретатор начинается с лексера — программы, разбивающей поток символов на токены: ключевые слова, идентификаторы, числа, операторы. Токены задаются регулярными выражениями (например, идентификатор — $[a-zA-Z_][a-zA-Z0-9_]*$). Каждое регулярное выражение компилируется в НКА, затем через конструкцию подмножеств — в ДКА, который и выполняет распознавание. ДКА обрабатывает каждый входной символ за $O(1)$ — один табличный переход по текущему состоянию и символу — поэтому лексеры оказываются одними из самых быстрых компонентов компилятора.

Лексер — первое звено конвейера компилятора: текст программы превращается в поток токенов, из токенов строится дерево разбора, дальше идут анализ и генерация кода. Регулярные языки заканчиваются на лексере: всё, что начинается после токенов — вложенные выражения, скобочные структуры, грамматика языка — требует уже контекстно-свободных грамматик. О них — глава 24.

Классическое изложение того, как регулярные выражения становятся рабочим инструментом компилятора, — в книге Альфреда Ахо, Рави Сети и Джеффри Ульмана. Книга называется «Компиляторы: принципы, технологии и инструменты»¹⁷². Тот же Ульман — соавтор учебника по теории автоматов, на который мы ссылались в доказательствах¹⁷³. Теория регулярных языков и инженерия компиляторов написаны одними людьми.

Утилита `grep` (Кен Томпсон, 1973) — пример автономного лексера: регулярное выражение компилируется в ДКА, фильтрующий строки текста. Тот же путь прошли генераторы лексеров: `Lex` (1975, Bell Labs), `flex` (Free Software Foundation, 1987), `ANTLR` (1989). За пределами компиляторов регулярные выражения — рабочий инструмент поиска и обработки текста: `sed` и `awk`, редакторы, стандартные библиотеки языков программирования.

§ 23.7 Свойства замкнутости

Имея автоматы для языков L_1, L_2 , можно собрать автоматы для их булевых комбинаций. Инструмент для этого — *прямое произведение*.

Идея простая: состояния нового автомата — это пары (q, r) , где q — состояние первого автомата, r — второго. Переходы определяются покомпонентно:

$$\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, a)).$$

Каждая компонента пары следит за своим автоматом, а вместе пара помнит состояние обеих машин сразу.

УТВЕРЖДЕНИЕ 23.3 Замкнутость относительно булевых операций

Имея ДКА для L_1, L_2 , можно построить ДКА для:

- **Объединения** $L_1 \cup L_2$: принимающие состояния — все пары, в которых хотя бы одна компонента принимающая: $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.
- **Пересечения** $L_1 \cap L_2$: принимающие состояния — пары, в которых обе компоненты принимающие: $F = F_1 \times F_2$.

¹⁷²A. V. Aho, R. Sethi, J. D. Ullman. «Compilers: Principles, Techniques, and Tools». 1986. Второе издание 2006 года с M. S. Lam. За обложку с драконом книгу называют Dragon Book. Лексеру и регулярным выражениям в ней посвящены главы 3–4.

¹⁷³J. E. Hopcroft, R. Motwani, J. D. Ullman. «Introduction to Automata Theory, Languages, and Computation». 3-е изд., 2006.

- **Дополнения $\overline{L_1}$:** меняем местами принимающие и непринимаящие состояния.

Набросок доказательства

Докажем построением прямого произведения.

Прямое произведение $M = M_1 \times M_2$ — две машины, работающие синхронно. После чтения слова w оно оказывается в паре $(\hat{\delta}_1(q_0, w), \hat{\delta}_2(r_0, w))$, и каждая компонента отслеживает свой автомат. Поэтому M принимает w ровно по тем правилам выбора принимающих состояний, которые заданы в формулировке. Например, для пересечения: $w \in L(M) \leftrightarrow \hat{\delta}_1(q_0, w) \in F_1 \wedge \hat{\delta}_2(r_0, w) \in F_2 \leftrightarrow w \in L_1 \cap L_2$.

Дополнение — смена принимающих состояний, и работает она только для ДКА. Почему для НКА тот же приём неверен, разобрано в разделе о недетерминированных автоматах.

Итог — класс регулярных языков образует *булеву алгебру*: он замкнут относительно объединения, пересечения и дополнения.

Сверх булевых операций регулярные языки замкнуты относительно конкатенации, звезды, обращения, гомоморфизмов и взятия префикса, суффикса и подстроки. Соберём все операции замкнутости в одну таблицу: для каждой — конструкция и короткое примечание.

Операция	Конструкция	Примечание
Объединение $L_1 \cup L_2$	Прямое произведение, $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$	Хотя бы одна компонента принимающая
Пересечение $L_1 \cap L_2$	Прямое произведение, $F = F_1 \times F_2$	Обе компоненты принимающие
Дополнение $\overline{L}$	Поменять принимающие и непринимаящие состояния	Только для ДКА, не для НКА
Конкатенация и звезда	По правилам регулярных выражений	Объединение — параллельный запуск, конкатенация — последовательный, звезда — цикл
Обращение L^R	Обратить переходы НКА и добавить новое начальное состояние с ε -переходами к прежним принимающим	Читает слово справа налево

Операция	Конструкция	Примечание
Гомоморфизм h	Заменить переход по символу a цепочкой переходов по слову $h(a)$	Символ превращается в слово
Обратный гомоморфизм h^{-1}	НКА раскладывает входной символ в подходящее слово образа	Слово превращается в символ
Префикс, суффикс, подстрока	Добавить ε -переходы, отрезающие начало или конец слова	Взять начало, конец или середину слова

Примечание

Приведём по одной конструкции на операцию. Гомоморфизм: при $h(0) = ab$ и $h(1) = a$ образ слова 010 — это слово $ab a ab$. Обратный гомоморфизм: при том же h слово 0 входит в $h^{-1}(\{ab\})$, потому что $h(0) = ab$. Префикс: для языка $L = \{abc\}$ язык всех начал его слов — это $\{\varepsilon, a, ab, abc\}$. Суффикс: для того же L язык всех концов его слов — это $\{\varepsilon, c, bc, abc\}$. Подстрока: для того же L это все слова из подряд идущих букв abc : $\{\varepsilon, a, b, c, ab, bc, abc\}$.

Пример Обращение языка

Язык $L = \{abc\}$ обращается в $L^R = \{cba\}$: слово читается справа налево.

Возьмём НКА для L : цепочка $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3$ по символам a, b, c , начальное состояние — q_0 , принимающее — q_3 . Обратим все переходы: цепочка превращается в $q_3 \rightarrow q_2 \rightarrow q_1 \rightarrow q_0$ по символам c, b, a . Начальное состояние q_0 становится принимающим: пустое слово при обращении остаётся собой. Новое начальное состояние s соединяется ε -переходами с q_3 — прежним принимающим.

Слово cba читается так: из s по ε в q_3 , затем по символам c, b, a вдоль обращённой цепочки до принимающего q_0 .

Свойства замкнутости позволяют строить новые регулярные языки из уже имеющих. Обратная задача труднее: доказать, что некоторый язык *не* регулярен. Нужно свойство, которым обладает каждый регулярный язык: если для языка оно не выполняется, то язык не регулярен.

Таким свойством обладает сама конечность памяти автомата. Если язык распознаётся автоматом с n состояниями, то любое слово длины не меньше n заставляет автомат пройти через $n + 1$ состояние. По принципу Дирихле хотя бы одно состояние повторяется. А если автомат прошёл через состояние дважды, то цикл между этими двумя вхождениями можно повторить сколько угодно раз — и автомат не заметит разницы.

Проверка Свойства замкнутости

1. Как прямое произведение автоматов даёт ДКА для пересечения $L_1 \cap L_2$, и что меняется в правиле выбора принимающих состояний при объединении?
2. Почему обращение языка L^R удобнее строить через НКА: что происходит с начальным и принимающим состояниями?

§ 23.8 Лемма о накачке**Пример Демонстрация идеи накачки**

Возьмём ДКА с тремя состояниями и любое слово длины 4, например 0001. Читая четыре символа, автомат проходит через $4 + 1 = 5$ состояний (считая начальное), а различных состояний всего три. По принципу Дирихле хотя бы одно состояние повторится.

Разбор:

1. Пусть состояние q встретилось на шагах i и j , где $i < j$.
2. Подслово между этими двумя вхождениями q — это цикл: пройдя его, автомат возвращается в q .
3. Автомат не различает, сколько раз он прошёл по этому циклу: один, два или десять — дальше он продолжает из того же состояния q и заканчивает в том же состоянии.

Именно это подслово и можно «накачать».

То, что мы увидели на одном слове и одном цикле, лемма о накачке превращает в утверждение о любом регулярном языке.

ТЕОРЕМА 23.4 Лемма о накачке для регулярных языков

Если язык L регулярен, то существует целое положительное число p (длина накачки) такое, что для любого слова $w \in L$ длины не меньше p существует разбиение $w = xyz$, удовлетворяющее трём условиям:

- $|y| > 0$ (накачиваемая часть непуста),
- $|xy| \leq p$ (накачиваемая часть находится в первых p символах),
- $xy^iz \in L$ для всех $i \geq 0$ (подслово y можно повторить любое число раз или удалить, и слово останется в языке).

Набросок доказательства

Докажем от принципа Дирихле: длинное слово заставляет автомат повторить состояние.

Пусть ДКА, распознающий L , имеет p состояний. Рассмотрим слово w длины не меньше p и первые p его символов. Читая эти p символов, автомат проходит через $p + 1$ состояние (считая начальное), а состояний всего p . По принципу Дирихле, среди этих $p + 1$ состояний есть повторяющееся — некоторое состояние s посещается дважды, и оба вхождения лежат среди первых $p + 1$ позиций.

Пусть x, y, z — префикс до первого вхождения состояния s , подслово между двумя вхождениями и оставшийся суффикс. Тогда $|y| > 0$, $|xy| \leq p$. Подслово y можно «накачать»: автомат будет ходить по циклу $s \rightarrow \dots \rightarrow s$, возвращаясь в то же состояние, или удалить — конечное состояние не изменится.

Геометрия доказательства — цикл в диаграмме переходов — показана на рис. 102. Участок y вместе с возвратной дугой к состоянию q_i можно пройти ноль раз, один раз или произвольное число k раз.

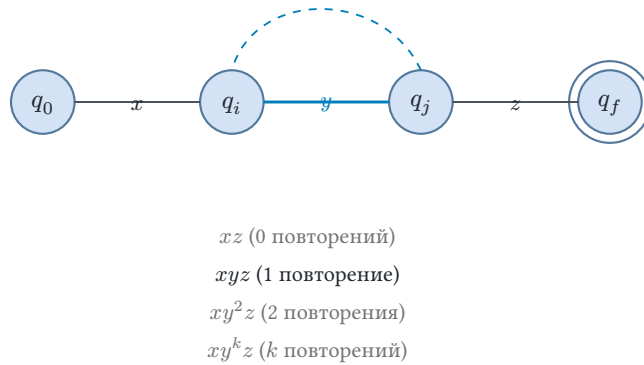


Рис. 102. Идея леммы о накачке: повторяющееся состояние образует цикл.

Пример Доказательство нерегулярности языка $\{0^n 1^n \mid n \geq 0\}$

Предположим, что $L = \{0^n 1^n \mid n \geq 0\}$ регулярен, и пусть p — его длина накачки.

Возьмём слово $w = 0^p 1^p \in L$. Его длина $2p \geq p$. По лемме о накачке $w = xyz$, $|xy| \leq p$, $|y| > 0$.

Шаги:

1. Так как $|xy| \leq p$, подстрока xy целиком лежит в первых p символах слова, то есть состоит только из нулей.
2. Значит, $y = 0^k$ для некоторого $k > 0$.
3. Тогда слово $xy^2z = 0^{p+k} 1^p$ должно принадлежать L .
4. Но в этом слове больше нулей, чем единиц — противоречие с определением L .

Следовательно, L не регулярен.

Примечание

Лемма о накачке даёт необходимое, но не достаточное условие регулярности. Существуют нерегулярные языки, которым она удовлетворяет, — пример ниже.

С её помощью регулярность нельзя доказать, только опровергнуть: для доказательства нужен более сильный инструмент — теорема Майхилла–Нерода.

Пример Лемма о накачке выполнена, но язык не регулярен

Рассмотрим язык

$$L = \{a^m b^n c^n \mid m \geq 1, n \geq 0\} \cup \{b^m c^n \mid m, n \geq 0\}.$$

Он нерегулярен: пересечение $L \cap ab^*c^* = \{ab^n c^n \mid n \geq 0\}$, а этот язык нерегулярен, как и $\{0^n 1^n\}$. Попробуем опровергнуть регулярность леммой о накачке и увидим, что не выйдет.

Пусть p — длина накачки.

Слово из первого слагаемого: $w = a^p b^p c^p$.

1. Так как $|xy| \leq p$, подслово xy состоит только из a .
2. Значит, $y = a^k$ для некоторого $k > 0$.
3. Накачанное слово $xy^2z = a^{p+k} b^p c^p$ по-прежнему принадлежит L : это слово первого вида с $m = p + k$.

Противоречия здесь нет: накачанное слово остаётся в языке.

Слово из второго слагаемого: $w = b^p c^p$.

1. Здесь $y = b^k$.
2. Тогда $xy^2z = b^{p+k} c^p$ принадлежит L (второй вид, $m = p + k$).

Опять противоречия нет.

Лемма «слепа» к структуре языка за пределами первых p символов: накачка всегда упирается в префикс из одинаковых символов. Значит, лемма о накачке не может ни доказать регулярность, ни опровергнуть её здесь — нужен критерий посильнее: теорема Майхилла–Нерода из следующего раздела.

Пример Другие нерегулярные языки

- Язык палиндромов над $\{a, b\}$ не регулярен (рассуждение аналогично $0^n 1^n$).
- Язык $\{a^{2^n} \mid n \geq 0\}$ не регулярен. Возьмём слово a^{2^p} при длине накачки p . По лемме $y = a^k$ для некоторого k , где $0 < k \leq p$. Тогда $xy^2z = a^{2^p+k}$ должна принадлежать языку. Но $2^p < 2^p + k \leq 2^p + p < 2^{p+1}$, то есть длина не степень двойки.
- Язык $\{a^n b^n c^n \mid n \geq 0\}$ не регулярен. Возьмём слово $a^p b^p c^p$ при длине накачки p . Так как $|xy| \leq p$, подслово y состоит только из a . Тогда $xy^2z = a^{p+|y|} b^p c^p$ содержит больше a , чем b и c , и не принадлежит языку.

Лемма о накачке доказывает только нерегулярность, и то не всегда. Теорема Майхилла–Нерода (1958) даёт одновременно необходимый и достаточный критерий. Она делает сразу два дела: классифицирует языки на регулярные и нерегулярные

и строит минимальный автомат напрямую по языку, через отношение неразличимости слов по продолжению.

§ 23.9 Теорема Майхилла–Нерода

Два слова могут вести себя одинаково относительно языка: какое продолжение ни припиши, результат одинаков. Отношение неразличимости по продолжению собирает такие слова в классы, а теорема Майхилла–Нерода утверждает, что язык регулярен тогда и только тогда, когда таких классов конечное число. Здесь мы определим отношение, применим критерий к примеру, построим через классы минимальный ДКА и увидим связь с теорией отношений.

23.9.1 Отношение неразличимости по продолжению

Возьмём конкретный язык: $L = \{w \in \{0, 1\}^* \mid w \text{ заканчивается на } 01\}$. Сравним три слова: ε (пустое), 0 и 00 . Продолжения покажут, какие из них «ведут себя одинаково» относительно L .

Сравним ε и 0 . Добавим к обоим продолжение 1 :

$$\varepsilon \cdot 1 = 1 \notin L, 0 \cdot 1 = 01 \in L.$$

Ответы разные — слова различимы: существует продолжение, на котором они расходятся.

Сравним 0 и 00 . Оба слова оканчиваются на 0 , а язык смотрит лишь на последние два символа. Продолжим оба, скажем, строкой 1 :

$$0 \cdot 1 = 01 \in L, 00 \cdot 1 = 001 \in L.$$

Попробуем другое продолжение, 01 :

$$0 \cdot 01 = 001 \in L, 00 \cdot 01 = 0001 \in L.$$

Ответы совпадают — и так будет на любом продолжении: $0z$ и $00z$ отличаются лишь лишним нулём в начале, а конец у них одинаков. Слова неразличимы: они лежат в одном классе.

Сравним ε и 1 . Язык L смотрит только на последние два символа. Для $|z| \geq 2$ хвосты продолжений z и $1z$ совпадают: лишний символ в начале ничего не меняет. Для коротких $z \in \{\varepsilon, 0, 1\}$ проверяется непосредственно, что ни z , ни $1z$ не заканчиваются на 01 — продолжения действуют одинаково. Слова неразличимы — класс ε не сводится к одиночному пустому слову.

Итак, слова ε и 0 , 00 дают лишь два класса: пустое неразлично с 1 , а 0 — с 00 . Третий класс составляют слова с хвостом 01 , например само это слово. Три класса — три состояния ДКА для L , как видно на рис. 98. Пустое слово и 1 ведут в начальное состояние q_0 . Слова 0 и 00 — в q_1 , а 01 — в принимающее состояние q_2 .

ОПРЕДЕЛЕНИЕ 23.12 Отношение неразличимости по продолжению

Для языка $L \subseteq \Sigma^*$ определим отношение $\simeq_L$ на Σ^* :

$$x \simeq_L y := \forall z \in \Sigma^* : (xz \in L \leftrightarrow yz \in L).$$

Слова эквивалентны, если добавление любого продолжения z справа даёт одинаковый ответ на вопрос «принадлежит ли результат языку L ?». Если хотя бы одно продолжение разводит ответы — слова различимы, и отношение $\simeq_L$ их не связывает.

$\simeq_L$ — отношение эквивалентности: рефлексивность, симметричность и транзитивность следуют непосредственно из определения. Равенство булевых значений для каждого z наследует эти свойства от самого равенства. Оно разбивает Σ^* на классы эквивалентности. Их число (конечное или бесконечное) называется *индексом* отношения.

В примере выше индекс $\simeq_L$ для языка слов, заканчивающихся на 01, равен трём — по числу состояний минимального ДКА. Это не совпадение: теорема Майхилла–Нерода утверждает, что индекс $\simeq_L$ всегда равен числу состояний минимального ДКА для L .

ТЕОРЕМА 23.5 Теорема Майхилла–Нерода¹⁷⁴

Для языка $L \subseteq \Sigma^*$ следующие утверждения эквивалентны:

1. L регулярен (распознаётся некоторым ДКА).
2. Отношение $\simeq_L$ имеет конечный индекс.
3. Язык L — объединение классов некоторой правой конгруэнции конечного индекса на Σ^* (отношения эквивалентности, согласованного с дописыванием символа справа).

Пункт 3 — это переформулировка пункта 2: само отношение неразличимости оказывается правой конгруэнцией. Требование правой конгруэнции — ровно то свойство, которое делает переход $[x] \rightarrow [xa]$ корректно определённым: переход не должен зависеть от того, каким представителем класса $[x]$ мы его задали. Если xRy , но xa и ya расходятся, один и тот же класс имел бы два разных перехода по a — такой автомат построить нельзя.

Примечание

Не всякое отношение эквивалентности на Σ^* — правая конгруэнция. Отношение «слова одновременно палиндромы или одновременно не палиндромы»

¹⁷⁴J. Myhill. «Finite automata and the representation of events». WADD TR-57-624, 1957. A. Nerode. «Linear automaton transformations». Proceedings of the AMS, 1958. Теорема даёт алгебраическую характеристику регулярных языков и конструкцию минимального ДКА.

связывает ab и ba : оба не палиндромы. Но после дописывания символа a слова расходятся — aba палиндром, а baa нет — и условие $xRy \Rightarrow xaRya$ нарушается.

Число состояний минимального ДКА для L равно индексу $\simeq_L$: по пункту (1) $\Rightarrow$ (2) классов не больше, чем состояний любого ДКА, а конструкция пункта (3) $\Rightarrow$ (1) строит автомат ровно с индексом состояний.

Набросок доказательства

Докажем эквивалентность трёх утверждений по циклу $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$.

(1) $\Rightarrow$ (2): Пусть L регулярен и распознаётся ДКА $M = (Q, \Sigma, \delta, q_0, F)$. Для каждого слова x определим состояние $\hat{\delta}(q_0, x)$, в которое M приходит после чтения x . Если $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$, то x и y неразличимы по продолжению: из одного состояния любой суффикс z ведёт к одному и тому же результату. Следовательно, слова, ведущие в одно состояние, попадают в один класс $\simeq_L$, и число классов не превышает $|Q|$ — конечно.

(2) $\Rightarrow$ (3): Само отношение неразличимости — правая конгруэнция конечного индекса: если $x \simeq_L y \Rightarrow xa \simeq_L ya$ (продолжение z для xa и ya — это одно и то же продолжение az для x и y). Язык L является объединением всех классов, содержащих слова из L .

(3) $\Rightarrow$ (1): Пусть дана правая конгруэнция R конечного индекса, и L — объединение некоторых её классов.

Построим ДКА: состояния — классы эквивалентности R . Начальное состояние — класс ε . Переход по a : $[x]_R \rightarrow [xa]_R$. Корректность: $xRy \Rightarrow xaRya$ по определению правой конгруэнции. Принимающие состояния — классы, составляющие L . Число состояний равно индексу R .

Теорема Майхилла–Нерода — структурная теорема: она утверждает, что классы неразличимости по продолжению *и есть* состояния минимального ДКА. Никакой другой автомат не может иметь меньше состояний.

23.9.2 Пример: доказательство нерегулярности через Майхилла–Нерода

Пример $L = \{a^n b^n \mid n \geq 0\}$ нерегулярен

Рассмотрим бесконечное множество слов $a, a^2, a^3, \dots$. Для любых $i < j$ слова a^i и a^j различимы продолжением:

$$z = b^i : a^i b^i \in L, a^j b^i \notin L.$$

Следовательно, a^i и a^j лежат в разных классах эквивалентности $\simeq_L$. Поскольку таких классов бесконечно много, индекс $\simeq_L$ бесконечен — по теореме Майхилла–Нерода язык L нерегулярен.

Сравним это доказательство с доказательством через лемму о накачке. Там мы предполагали регулярность, выбирали конкретное слово $a^p b^p$ и получали противоречие через накачку. Здесь мы не делаем предположения о регулярности — мы непосредственно доказываем, что само отношение $\simeq_L$ имеет бесконечный индекс. Оба доказательства корректны, но доказательство через Майхилла–Нерода конструктивно: оно явно предъявляет бесконечное множество попарно различных слов.

Теорема Майхилла–Нерода даёт *необходимый и достаточный* критерий регулярности. Для любого нерегулярного языка отношение $\simeq_L$ имеет бесконечный индекс. В отличие от леммы о накачке, которая позволяет доказывать только нерегулярность, теорема Майхилла–Нерода работает в обе стороны. Предъявление конечного индекса $\simeq_L$ доказывает регулярность языка — и заодно даёт минимальный ДКА.

23.9.3 Минимизация ДКА через классы эквивалентности

Доказательство достаточности в теореме Майхилла–Нерода конструктивно: оно строит ДКА непосредственно по классам $\simeq_L$. Этот ДКА и есть минимальный.

УТВЕРЖДЕНИЕ 23.6 Минимальный ДКА как фактор-автомат

Состояния минимального ДКА для языка L — это классы эквивалентности отношения $\simeq_L$. Переходы: из класса $[x]$ по символу a приходим в класс $[xa]$. Начальное состояние: $[\varepsilon]$. Принимающие состояния: все $[x]$, где $x \in L$.

Такой автомат корректен, поскольку:

- Переходы определены однозначно: $x \simeq_L y \Rightarrow xa \simeq_L ya$.
- Язык автомата есть L : слово w принимается тогда и только тогда, когда $[w]$ — принимающее состояние, то есть тогда и только тогда, когда $w \in L$.

Это построение — более концептуальный взгляд на минимизацию, чем алгоритм Мура из следующего раздела. Алгоритм Мура итеративно находит различные состояния. Теорема Майхилла–Нерода утверждает, что различимость состояний ДКА — это то же самое, что различимость слов продолжением. Два состояния эквивалентны тогда и только тогда, когда неразличимы слова, которые в них ведут: если x приводит в p , а y — в q , то $x \simeq_L y$. Эквивалентность состояний — это перенесённое на автомат отношение $\simeq_L$: состояния сливаются ровно тогда, когда склеиваются ведущие в них слова.

23.9.4 Связь с теорией отношений

Отношение $\simeq_L$ — это отношение эквивалентности на множестве слов Σ^* (глава 5). Минимальный ДКА для L есть *фактор-автомат*: его состояния — классы эквивалентности, а функция переходов индуцирована конкатенацией справа. Это та же конструкция, что и фактор-множество по отношению эквивалентности: элементы,

неразличимые относительно интересующего нас свойства (L), склеиваются в одно состояние.

Отношение $\simeq_L$ — конгруэнция относительно операции дописывания символа справа: если $x \simeq_L y$, то $xa \simeq_L ya$ для любого $a \in \Sigma$. Именно это свойство гарантирует корректность функции переходов в фактор-автомате. У отношения $\simeq_L$ в теории автоматов есть собственное имя: правая конгруэнция Майхилла–Нерода. Алгебраическое понятие конгруэнции оборачивается здесь вычислительной моделью — минимальным ДКА.

Разница между алгоритмом Мура и построением Майхилла–Нерода:

- Алгоритм Мура *операционный*: начинает с состояний конкретного ДКА и итеративно объединяет эквивалентные.
- Построение Майхилла–Нерода *структурное*: начинает с языка и строит минимальный ДКА напрямую из классов $\simeq_L$.

Алгоритм Мура требует уже иметь ДКА (возможно, избыточный). Построение Майхилла–Нерода получает автомат «с нуля» по языку. Плата — необходимость вычислить классы $\simeq_L$, что нетривиально для языка, заданного иным описанием.

§ 23.10 Минимизация ДКА

Среди всех ДКА, распознающих данный язык, существует единственный (с точностью до изоморфизма) минимальный. На практике минимизация окупается: меньше состояний — быстрее проверка, меньше памяти для хранения таблицы переходов.

Идея минимизации проста: объединить состояния, которые невозможно различить по дальнейшему поведению.

ОПРЕДЕЛЕНИЕ 23.13 Эквивалентность состояний

Состояния p, q называются **эквивалентными** (или **неразличимыми**), обозначается $p \sim q$, если для любого слова $w \in \Sigma^*$ выполняется:

$$\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F.$$

Оба состояния принимают в точности одни и те же слова.

Алгоритм минимизации систематически находит *различимые* пары состояний и объединяет неразличимые.

Метод таблицы различимостей.

1. Нарисовать таблицу всех неупорядоченных пар (p, q) с $p < q$.
2. Пометить пары, в которых ровно одно состояние принимающее (они различимы сразу).

3. Итеративно: для каждой непомеченной пары (p, q) и каждого символа a проверить пару $(\delta(p, a), \delta(q, a))$. Если эта пара помечена, пометить (p, q) (различимость распространяется назад).
4. Повторять, пока в таблице происходят изменения.
5. Непомеченные пары = эквивалентные состояния.
6. Склеить эквивалентные состояния в одно — получить минимальный ДКА.

Время работы: $O(|Q|^2 \cdot |\Sigma|)$.

Три варианта одной идеи — таблица различимостей, разбиение на блоки и ускоренная схема — удобно свести в одну таблицу.

Алгоритм	Идея	Сложность
Мур (1956), таблица различимостей	Помечаем различимые пары состояний и склеиваем неразличимые	$O(n^2 \cdot \Sigma)$
Мур (1956), уточнение разбиения	Та же идея блоками: блок расщепляется, если его состояния по некоторому символу расходятся по разным блокам	$O(n^2 \cdot \Sigma)$
Хопкрофт (1971)	Ускорение: при расщеплении перестраиваем переходы только меньшей половины блока	$O(n \cdot \Sigma \cdot \log n)$

Заполнение таблицы различимостей и есть табличная запись алгоритма Мура. Подробности блочной версии и ускорение Хопкрофта — в факультативном разделе ниже.

Алгоритм строит ДКА с минимальным числом состояний для данного языка. Следующая теорема утверждает, что этот результат каноничен.

ТЕОРЕМА 23.7 Единственность минимального ДКА

Каждый регулярный язык имеет единственный минимальный ДКА с точностью до изоморфизма (переименования состояний).

Два состояния неразличимы, если никакое слово не приводит из них к разным ответам. С точки зрения языка они — одно состояние. Алгоритм минимизации находит эту каноническую форму, склеивая неразличимые состояния: они избыточны, потому что по поведению это одно и то же состояние.

Набросок доказательства

Докажем построением изоморфизма между двумя минимальными ДКА.

Сначала докажем лемму: в минимальном ДКА любые два различных состояния различимы — есть слово, принимаемое из одного состояния и отвергаемое из другого. Иначе эти состояния можно было бы склеить и получить ДКА с меньшим числом состояний.

Пусть M_1, M_2 — два минимальных ДКА для языка L . По теореме Майхилла–Нерода число состояний каждого равно индексу $\simeq_L$: из пункта (1) $\Rightarrow$ (2) классов не больше, чем состояний любого ДКА. Конструкция пункта (3) $\Rightarrow$ (1) строит ДКА ровно с индексом состояний, так что меньше быть не может. Построим изоморфизм: состоянию M_1 , в которое ведёт слово x , сопоставим состояние M_2 , в которое ведёт то же слово x .

Это отображение корректно: если $x \simeq_L y$, то в каждом из автоматов M_1, M_2 оба слова ведут в одно состояние — иначе в автомате было бы два неразличимых состояния, что противоречит лемме. Сюръективность следует из минимальности: каждое состояние достижимо некоторым словом, иначе его можно удалить. Инъективность ссылается на контрапозицию леммы: если слова x и y ведут в разные состояния M_1 , они различимы, поэтому не могут вести в одно состояние M_2 . Различающее их продолжение z дало бы в M_2 один ответ, и M_2 не распознавал бы L .

Отображение сохраняет структуру: начальные состояния отвечают слову ε , принимающие — словам из L . Переход по a переводит состояние слова x в состояние слова xa в обоих автоматах. Значит, δ_1 и δ_2 , а также F_1 и F_2 согласованы, и построенное соответствие — изоморфизм.

Проверка эквивалентности двух регулярных языков сводится к сравнению их минимальных ДКА: достаточно построить обе канонические формы и убедиться, что они изоморфны.

Проверка Минимизация ДКА

1. Почему в алгоритме Мура пара состояний, из которых ровно одно принимающее, помечается как различимая сразу, ещё до первой итерации?
2. Почему минимальный ДКА единственен с точностью до изоморфизма, а не с точностью до числа состояний?

§ 23.11 * Минимизация ДКА (алгоритм Хопкрофта)

В разделе о минимизации мы рассмотрели алгоритм Мура — заполнение таблицы неразличимости за $O(n^2 \cdot |\Sigma|)$. Для учебного автомата с десятком состояний этого достаточно. Но лексер реального языка программирования может содержать сотни состояний, и квадратичный алгоритм становится узким местом.

Задача минимизации ДКА: по заданному автомату построить эквивалентный с минимальным числом состояний. Теорема Майхилла–Нерода даёт ответ: состояний ровно столько, сколько классов у отношения $\simeq_L$. Вопрос — как построить эффективно.

Алгоритм Мура (1956) — это уточнение разбиения (partition refinement). Начинаем с двух блоков: принимающие и непринимаящие. Для каждого блока и каждого символа проверяем, все ли состояния блока переходят в один и тот же блок. Если нет — блок расщепляется. Повторяем, пока разбиение не стабилизируется. Время работы — $O(n^2 \cdot |\Sigma|)$.

Хопкрофт в 1971 году заметил: при расщеплении блока на две части достаточно перестраивать переходы только для меньшей половины. Состояния, оставшиеся в большей части, сохраняют все переходы в другие блоки. Это даёт $O(n \cdot |\Sigma| \cdot \log n)$ — практически линейное время при малом алфавите.

Проиллюстрируем на примере. Возьмём ДКА $M = (\{0, 1, 2, 3, 4\}, \{a, b\}, 0, \{3, 4\})$ — состояния, алфавит, начальное и принимающие. Переходы по a : $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 3, 4 \rightarrow 4$. По b все состояния переходят в 4. Начальное разбиение — $\{0, 1, 2\}, \{3, 4\}$.

По символу a блок $\{0, 1, 2\}$ расщепляется: состояния 0 и 1 ведут внутрь блока, состояние 2 — в принимающий, и получается разбиение $\{0, 1\}, \{2\}, \{3, 4\}$. Проверяем блок $\{0, 1\}$ по символу a : 0 ведёт в 1 (тот же блок), 1 — в 2 (другой блок), и блок расщепляется на $\{0\}, \{1\}$.

По символу b все блоки устойчивы, ведь каждое состояние ведёт в 4. В итоге получается разбиение $\{0\}, \{1\}, \{2\}, \{3, 4\}$ — четыре состояния.

Минимальный ДКА каноничен. Два регулярных языка равны тогда и только тогда, когда их минимальные ДКА изоморфны. Эквивалентность языков, заданных ДКА, алгоритмически разрешима — в отличие от более мощных моделей вычислений (глава 26). Минимизация связывает математический факт (единственность канонической формы) с инженерной практикой (быстрая проверка эквивалентности).

§ 23.12 * Минимализация Бржозовски

Алгоритмы Мура и Хопкрофта начинают с готового ДКА и склеивают его состояния, пока склеиваются. Метод Бржозовски идёт с другого конца: минимальный автомат не ищется во входе, а собирается заново из двух операций, которые уже есть в этой главе. Обращение rev разворачивает рёбра и меняет ролями начальные и принимающие состояния, добавляя при нескольких принимающих новое начальное с ε -переходами — как при обращении языка из раздела о замкнутости. Детерминизация det — конструкция подмножеств с выбрасыванием недостижимых состояний, и на вход она принимает любой НКА. Рецепт — дважды применить пару из обращения и детерминизации к исходному автомату N :

$$\det(\text{rev}(\det(\text{rev}(N))))).$$

Подмножество, в которое слово w приводит детерминизацию обращённого автомата, — это множество состояний N , из которых w^R доводит до принятия. Первый проход сортирует по тому, чем пути могут быть завершены, — по классам обращённого отношения $\simeq_{\{L^R\}}$, а вторая пара замыкает сортировку до классов $\simeq_L$.

ТЕОРЕМА 23.8 Минимализация двойным обращением¹⁷⁵

Для любого НКА N автомат

$$\det(\text{rev}(\det(\text{rev}(N))))$$

— минимальный ДКА, распознающий язык N .

Набросок доказательства

Докажем в два шага: сначала лемма о двойном обращении для ДКА, затем её применение к выходу первой пары.

Лемма. Пусть D — ДКА для языка K , все состояния которого достижимы. Тогда

$$\det(\text{rev}(D))$$

— минимальный ДКА для K^R .

В детерминизации слово w приводит в подмножество $S_w = \{q \in Q \mid \hat{\delta}_D(q, w^R) \in F\}$ — всех состояний D , из которых w^R принимается. Пусть $S_w \neq S_v$. Выберем состояние q из их симметрической разности и слово z с $\hat{\delta}_D(q_0, z) = q$ — оно существует, потому что все состояния D достижимы. Чтение z из S_w ведёт детерминизацию в подмножество $\{p \mid \hat{\delta}_D(p, z) \in S_w\}$, и оно принимающее — содержит q_0 — тогда и только тогда, когда $q \in S_w$. Для S_v ответ обратный: $q \notin S_v$. Слово z различает S_w и S_v , значит, состояния

$$\det(\text{rev}(D))$$

попарно различимы, а достижимость у них есть по построению.

Разные состояния несут разные остаточные языки — множества слов, доводящих состояние до принятия. Различных остаточных языков у K^R ровно столько, сколько классов у $\simeq_{\{K^R\}}$ (теорема Майхилла–Нерода). Состояний у автомата не больше индекса — он и есть минимальный.

Сведение. Первая пара даёт $D_1 = \det(\text{rev}(N))$ — ДКА для L^R без недостижимых состояний. Одной пары недостаточно: лемма требует детерминированный вход,

¹⁷⁵J. A. Brzozowski. «Canonical regular expressions and minimal state graphs for definite events». Mathematical Theory of Automata, 1963. Метод известен как минимализация двойным обращением (double reversal).

поэтому первая пара лишь превращает НКА в ДКА, возможно, с избытком состояний, а гарантию минимальности даёт вторая.

Пример Двойное обращение на четырёх состояниях

Возьмём ДКА M для языка слов, заканчивающихся на ab , с одним лишним состоянием. Близнец q_3 повторяет строку таблицы начального q_0 , а приводит в него слово $abab$: переход из q_2 по b ведёт в близнеца.

Состояние	a	b
q_0 — старт	q_1	q_0
q_3 — близнец q_0	q_1	q_0
q_1 — хвост a	q_1	q_2
q_2 — принято	q_1	q_3

Минимальный ДКА для этого языка имеет три состояния.

Первый проход. Обращение: рёбра разворачиваются, начальным становится q_2 , принимающим — q_0 . Детерминизация из $\{q_2\}$ порождает четыре подмножества:

Подмножество	a	b
$Z_0 = \{q_2\}$ — старт	$\emptyset$	Z_1
$Z_1 = \{q_1\}$	Z_2	$\emptyset$
$Z_2 = \{q_0, q_1, q_2, q_3\}$ — принято	Z_2	Z_2
$\emptyset$ — тупиковое	$\emptyset$	$\emptyset$

Получился ДКА для L^R — слов, начинающихся с ba : по a из Z_0 детерминизация попадает в тупиковое подмножество, потому что в q_2 входного автомата ведёт только переход по b . Близнецы q_0 и q_3 везде встречаются только вместе: первая сортировка уже не различает их.

Второй проход. Снова обращение: начальным становится Z_2 , принимающим — Z_0 . Детерминизация из $\{Z_2\}$:

Подмножество	a	b
$W_0 = \{Z_2\}$ — старт	W_1	W_0
$W_1 = \{Z_1, Z_2\}$ — хвост a	W_1	W_2
$W_2 = \{Z_0, Z_1, Z_2\}$ — принято	W_1	W_0

Тупиковое подмножество не унаследовалось, а результат свёлся к трём состояниям — ровно тем классам $\simeq_L$, которые предписывает теорема Майхилла–Нерода: префикс без прогресса, префикс с хвостом a , префикс в L . Слово ab ведёт $W_0 \rightarrow W_1 \rightarrow W_2$ и принимается, а слово ba проходит путь $W_0 \rightarrow W_0 \rightarrow W_1$ и отвергается. Тот же автомат выдаёт на этом входе алгоритм Мура — теорема обещает ровно это на любом входе.

Плата за универсальность — экспонента. Конструкция подмножеств способна выдать 2^n состояний из n , и худшие входы реальны: НКА с $n + 1$ состоянием для языка слов, у которых n -й с конца символ равен 1, детерминизируется в 2^n состояний. Поэтому при детерминированном входе минимизируют Муром или Хопкрофтом — предсказуемо полиномиально. Метод Бржозовски ценен другим: он перерабатывает НКА целиком, без отдельной детерминизации, и выводит каноническую форму из двух простых операций — пригоден и как алгоритм, и как приём в доказательствах.

Замечание Автоматы с выходом

Автомат отвечает одним битом: принадлежит ли слово языку. Трансдюсер — конечный автомат с выходом: переход, кроме смены состояния, дописывает символ в выходное слово, и машина превращает входное слово в выходное. Так устроены лексические анализаторы с выдачей токенов, морфологические словари и нормализация текста. Обращение переносится на трансдюсеры дословно — метки переворачиваются вместе с рёбрами, — а двойное обращение уже не даёт канонической формы: у недетерминированного трансдюсера параллельные ветви выдают разные выходы, и конструкция подмножеств заменяется композицией отношений. Минимализация трансдюсеров — отдельная теория со своей канонической формой. Во взвешенном варианте, где переходы несут числа из полукольца, она применяется в распознавании речи и машинном переводе.

§ 23.13 * Автомат Кнута–Морриса–Пратта

Задача поиска подстроки звучит буднично: даны текст t длины n и слово-образец p длины m . Нужно найти все вхождения p в t . Наивный способ прикладывает образец к каждой позиции текста и сверяет символы — до $O(n \cdot m)$ сравнений. Худшие случаи не экзотика: на тексте из n букв a образец $a^{m-1}b$ отбрасывается с каждой позиции лишь на последнем символе, и работа выходит квадратичной. В 1977 году Дональд Кнут, Джеймс Моррис и Вон Пратт опубликовали алгоритм, решающий задачу за $O(n + m)$ сравнений¹⁷⁶. Моррис добрался до идеи ещё в 1970 году, отлаживая текстовый редактор и разбираясь, сколько сравнений съедает поиск подстроки. Кнут и Пратт построили алгоритм независимо, и совместная статья соединила три потока.

Прежде всякого алгоритма нужно решить, что вообще помнить о прочитанном куске текста. Теорема Майхилла–Нерода из раздела 23.9 отвечает: состояние распознающего автомата — это класс неразличимости прочитанного префикса. Пока вхождение не найдено, класс префикса текста определяется длиной наибольшего

¹⁷⁶D. E. Knuth, J. H. Morris, V. R. Pratt. «Fast pattern matching in strings». SIAM Journal on Computing, 1977.

префикса образца, являющегося суффиксом прочитанного. Совпадающий хвост означает: вхождение может достроиться впереди, и потому запоминать больше этого хвоста незачем. Различных состояний такой памяти $m + 1$: от нуля совпавших символов до целого образца.

ОПРЕДЕЛЕНИЕ 23.14 Префикс-функция

Префикс-функцией слова $p = p_1p_2\dots p_m$ называется функция π , обозначается $\pi(i)$, значением которой в точке i служит длина наибольшего собственного префикса отрезка $p_1\dots p_i$, являющегося его суффиксом. Если ни один собственный префикс не совпадает с суффиксом, $\pi(i) = 0$.

Собственный — значит не совпадающий со всем отрезком: у однобуквенного отрезка собственных префиксов нет вовсе.

Пример Префикс-функция слова abacaba

i	Отрезок	Общий префикс и суффикс	$\pi(i)$
1	a	—	0
2	ab	—	0
3	aba	a	1
4	abac	—	0
5	abaca	a	1
6	abacab	ab	2
7	abacaba	aba	3

Отрезок aba начинается и кончается на a, отрезок abacab — на ab. У отрезка abac общей части нет: он начинается на a, а кончается на c.

ОПРЕДЕЛЕНИЕ 23.15 Автомат Кнута–Морриса–Пратта

Автоматом Кнута–Морриса–Пратта для слова-образца $p = p_1p_2\dots p_m$ над алфавитом Σ называется ДКА с состояниями $0, 1, \dots, m$, начальным состоянием 0 и принимающим состоянием m . Переход из состояния $j < m$ по символу a ведёт в состояние, равное длине наибольшего префикса образца, являющегося суффиксом слова $p_1\dots p_ja$. Из принимающего состояния m все переходы ведут обратно в m .

Состояние j читается так: ровно j последних символов прочитанного совпадают с началом образца, и совпадение нельзя продлить. Слово принимается, когда оно содержит p как подслово: найдя вхождение, автомат уже его не забудет.

Префикс-функция даёт все переходы сразу. Если очередная буква a равна p_{j+1} , совпадение подрастает: $\delta(j, a) = j + 1$. Если нет — текущее совпадение обрывается до наибольшего префикса, который всё ещё годится, то есть до $\pi(j)$, и вопрос повторяется уже от него: $\delta(j, a) = \delta(\pi(j), a)$. Отсюда же способ выписать таблицу

переходов: каждая строка — это строка состояния $\pi(j)$, в которой заменена ровно одна ячейка, переход по букве p_{j+1} ведёт в $j + 1$. Префикс-функция вычисляется за $O(m)$, поэтому автомат строится за линейное от m время. Явная таблица $m \times |\Sigma|$ при желании разворачивается за $O(m \cdot |\Sigma|)$ — при фиксированном алфавите это та же линейность.

Пример Автомат для образца abab

Префикс-функция: $\pi(1) = 0, \pi(2) = 0, \pi(3) = 1, \pi(4) = 2$.

Состояние j	a	b
0	1	0
1	1	2
2	3	0
3	1	4
4 — принято	4	4

Строка состояния 3 — строка состояния $\pi(3) = 1$ с одной заменой: по букве b , четвёртой букве образца, переход ведёт в 4. Прогон текста aab: $0 \rightarrow 1 \rightarrow 1 \rightarrow 2$ — хвост ab совпал с началом образца, и до вхождения не хватает одной буквы.

Поиск — прогон этого автомата по тексту, только таблицу он не хранит. Достаточно одного числа j : если очередная буква текста равна p_{j+1} , оно растёт на единицу. Иначе j заменяется на $\pi(j)$, и проверка повторяется, пока совпадение не продлится или j не обнулится. При $j = m$ вхождение найдено. Для перекрывающихся вхождений поиск после рапорта сбрасывает j в $\pi(m)$ и продолжает чтение.

Вся арифметика окупается подсчётом сравнений: указатель по тексту движется только вперёд, а спуски по префикс-функции оплачиваются недавними подъёмами.

ТЕОРЕМА 23.9 Линейность поиска

Поиск всех вхождений образца p в текст t автоматом Кнута–Морриса–Пратта выполняется за $O(|t| + |p|)$: всего происходит не более $2|t| + 2|p|$ сравнений символов.

Доказательство

Докажем подсчётом сравнений: каждое сравнение символов заканчивается либо подъёмом, либо спуском.

Подъёмы. На каждом шаге поиска текущая длина совпадения j растёт не более чем на единицу — только при совпадении очередной буквы текста с p_{j+1} . За весь поиск подъёмов не более $|t|$.

Спуски. При несовпадении поиск спускается по префикс-функции: j заменяется на $\pi(j)$, что строго меньше j . Каждый спуск уменьшает j хотя бы на единицу,

а j не бывает меньше нуля, поэтому спусков не больше подъёмов — не более $|t|$. Итого на поиске не более $2|t|$ сравнений.

Построение префикс-функции. Тот же подсчёт работает и здесь: значение $\pi(i)$ ищется спуском по уже вычисленным значениям, подъёмов не больше $|p|$ — по одному на каждую букву образца, и спусков не больше, чем подъёмов. Итого не более $2|t|$ сравнений на поиске и $2|p|$ на подготовке.

Минимализм здесь не случайность: меньше состояний дать нельзя.

УТВЕРЖДЕНИЕ 23.10 Минимальность автомата поиска

Любой ДКА, распознающий язык слов, содержащих p как подслово, имеет не менее $|p| + 1$ состояний.

Доказательство

Докажем предъявлением $m + 1$ попарно различных слов, как в разделе 23.9. Префиксы $w_j = p_1 \dots p_j$, включая пустое $w_0 = \varepsilon$, различаются продолжением $z_j = p_{j+1} \dots p_m$, достраивающим w_j до целого образца.

Слово $w_j z_j = p$ содержит p и лежит в языке. При $i < j$ слово $w_i z_j$ имеет длину $i + (m - j) < m$: оно короче образца и потому не содержит его. Одно и то же продолжение z_j даёт для w_j и w_i разные ответы, значит, все $m + 1$ префиксов попарно различимы.

Индекс отношения $\simeq_L$ не меньше $m + 1$, и по теореме Майхилла–Нерода любой ДКА для L несёт не менее $m + 1$ состояний. Автомат Кнута–Морриса–Пратта ровно в эту оценку укладывается — он минимален.

Образец-слово — частный случай регулярного выражения, и его мог бы скомпилировать и Томпсон. НКА с петлёй в начальном состоянии из раздела о НКА ищет совпадения с любой позиции текста, и конструкция подмножеств выдала бы из него тот же автомат с $m + 1$ состояниями. Заслуга алгоритма 1977 года — префикс-функция: она выписывает этот автомат напрямую, за линейное время и без ε -замыканий, в то время как для произвольного выражения конструкция подмножеств способна раздуть автомат до 2^n состояний. Набор образцов обслуживает алгоритм Ахо–Корасика (1975): слова складываются в бор, а суффиксные связи между вершинами бора — та же префикс-функция, обобщённая на дерево¹⁷⁷. Что происходит, когда движок регулярных выражений идёт другим путём и перебирает пути НКА по одному — тема следующего раздела.

¹⁷⁷ A. V. Aho, M. J. Corasick. «Efficient string matching: an aid to bibliographic search». Communications of the ACM, 1975.

§ 23.14 * Регулярные выражения на практике: ДКА против backtracking

Теория этой главы обещает линейный поиск: выражение компилируется в НКА конструкцией Томпсона, НКА детерминизируется, и ДКА читает текст по переходу на символ. Большинство популярных библиотек — PCRE, Perl, Python, JavaScript — идут другим путём: их движки работают с возвратом (backtracking), а на неудачных парах выражение–текст время поиска взрывается экспоненциально. На типичных запросах возврат прост в реализации и быстр, поэтому он и стал нормой. Причина выбора прагматична: возврат легко надстраивается обратными ссылками, ленивыми квантификаторами и прочими расширениями, которых требует публика.

Модель возврата — обход дерева разбора в глубину. В выражении две точки выбора: ветвь $R_1 \mid R_2$ (какую альтернативу попробовать первой) и звезда R^* (сколько повторов взять). Порядок попыток задаёт жадность: квантификатор по умолчанию берёт максимум, ленивый — минимум. На класс языка это не влияет, на порядок нахождения совпадения и на время — влияет. Движок фиксирует выбор, идёт вперёд по тексту, а при неудаче откатывается к последней развилке и пробует следующий вариант. Жадный квантификатор сперва забирает всю цепочку, затем по букве отдаёт назад — так перебираются разбиения от длинного первого куска к короткому. Глубина стека при таком обходе растёт вместе с текстом: наивные реализации упрутся в предел рекурсии на длинных строках. По сути это перебор путей НКА по одному за раз — ровно то, от чего конструкция подмножеств избавляет детерминизацией.

Катастрофа ожидает даже простое выражение $(aa^*)^*b$. В синтаксисе библиотек его пишут $(a+)+b$, где $a+$ — «одна или больше». Язык простейший: цепочка из букв a , завершённая одной буквой b . Минимальный ДКА для него уместится в три состояния. На тексте из n букв a без завершающей b движок обязан перебрать все способы разбить цепочку на куски: внешняя звезда итерирует k раз, каждая итерация съедает непустой кусок, и для фиксированного k разбиений — $\binom{n-1}{k-1}$. Сумма по всем k даёт 2^{n-1} разборов одной и той же строки: тридцать букв a — это $2^{29} \approx 5.4 \times 10^8$ попыток, пятьдесят букв — $2^{49} \approx 5.6 \times 10^{14}$.

Экспонента вырастает из отсутствия запоминания. Возврат идёт по одному пути за раз и не замечает, что разные разбиения приводят в одно и то же положение: все 2^{n-1} путей НКА несут одну и ту же информацию — «прочитана цепочка из букв a , буквы b ждать». Конструкция подмножеств слила бы эти пути в одно состояние, и на каждый символ текста осталась бы константная работа.

Разобраться можно и без детерминизации — приёмом самого Томпсона: симулировать НКА множеством активных состояний. На каждом символе текста множество сдвигается по переходам и замыкается по ε . Работа на символ пропорциональна размеру НКА, то есть длине выражения, итого $O(n \cdot |R|)$ при любом разветвлении. Множество активных состояний — это все пути НКА разом: слово принимается, если среди активных есть принимающее, и квантор существования из раздела о НКА

обрабатывает без всякого перебора. Множества можно кэшировать: впервые встреченное обрабатывается построением, дальше — прямой переход по таблице. Такая ленивая детерминизация лежит в основе RE2 и крейта `gedex` языка Rust: линейная по длине текста гарантия без предварительного построения гигантского ДКА. Той же школой вырос `gedex` в Go.

Платой за гарантию служит отказ от обратных ссылок. Ссылка `\1` заставляет сравнить хвост текста со словом, захваченным скобками раньше. Для языка $L = \{ww \mid w \in \Sigma^*\}$ индекс $\simeq_L$ бесконечен: пример $a^n b^n$ из раздела 23.9 расширяется до него той же техникой. Конечный автомат такой язык не распознает. Задача совпадения для выражений с обратными ссылками NP-полна¹⁷⁸, поэтому библиотеки с возвратом не могут дать полиномиальной гарантии в принципе. RE2 и Rust-крейт отказываются от ссылок. PCRE и родственные движки сохраняют их — вместе с экспоненциальным худшим случаем.

Цена возврата измеряется не только синтетикой: в июле 2019 года одно выражение с катастрофическим возвратом в WAF-фильтре Cloudflare сделало недоступным заметный кусок интернета примерно на полчаса.

Примечание Скобки — это не ссылки

Захват подвыражений скобками сам по себе не требует возврата. Позиции пар скобок можно отслеживать в симуляции НКА, дописывая к каждому активному состоянию снимок открытых скобок: RE2 так и извлекает подвыражения, оставаясь линейным. За пределы регулярных языков выводят именно обратные ссылки: они сравнивают текст с захваченным, а не фиксируют, где произошёл захват.

Вопрос, детерминизировать заранее или разбираться на лету, обсуждали уже первые инженеры. Брайан Керниган для лексера языка PIS детерминизировал автомат заранее и мирился с тем, что построение занимало пятнадцать минут на машине VAX 750. Кен Томпсон в редакторе `ed` оставил недетерминизм и справлялся с ним по ходу чтения текста — той самой симуляцией множества состояний. Современные линейные движки соединили обе стратегии: НКА без заранее построенного ДКА даёт скорость без пятнадцатиминутных пауз, а кэш подмножеств возвращает табличную скорость на повторях.

Три стратегии расплачиваются за недетерминизм выражения по-разному: детерминизация — экспонентой памяти на построении, симуляция — размером НКА на каждом символе, возврат — экспонентой времени в худшем случае. Экспонента — цена способа записи, а не языка: автомат Кнута–Морриса–Пратта из предыдущего раздела выписывает минимальный автомат для поиска подстроки за линейное время.

¹⁷⁸ A. V. Aho. «Algorithms for finding patterns in strings». Handbook of Theoretical Computer Science, 1990.

§ 23.15 Об автомате можно узнать всё

У автомата есть свойство, которого нет у более мощных моделей: про него можно *всё выяснить*. Автомат — конечный объект: конечное число состояний, конечная таблица переходов. Его поведение целиком задано конечной диаграммой, а значит, вопросы о его поведении можно решать *алгоритмически*: написать процедуру, которая всегда завершается и даёт точный ответ.

Четыре вопроса об автомате сводятся к уже решённой задаче о пустоте языка и к поиску в конечном графе.

Вопрос	Сведение	Метод
Пуст ли язык $L(M)$?	Из q_0 не достижимо ни одно принимающее состояние	Обход графа состояний от q_0
Равен ли язык всем словам Σ^* ?	Пусто дополнение языка	Дополнение ДКА, затем проверка пустоты
Конечен ли язык?	Из q_0 достижимо состояние на цикле, из которого достижимо принимающее	Поиск циклов в графе состояний
Эквивалентны ли M_1 и M_2 ?	Пуст язык симметрической разности языков	Прямое произведение, затем проверка пустоты

Первый метод — обход от начального состояния — реализуется прямо по таблице переходов.

```

/// Пуст ли язык автомата? -- обход от начального состояния.
fn is_empty(trans: &Vec<usize>, start: usize, accept: &[bool]) -> bool {
    let mut seen = vec![false; trans.len()];
    let mut stack = vec![start];

    while let Some(q) = stack.pop() {
        if accept[q] { return false; } // нашли принимающее состояние
        for &r in &trans[q] {
            if !seen[r] { seen[r] = true; stack.push(r); }
        }
    }

    true
}

```

Массив `trans` — список переходов: для каждого состояния — состояния, в которые можно попасть по одному символу.

К таблице добавим три замечания. Принадлежит ли конкретное слово w языку, решается тем же проходом по символам w : достаточно посмотреть на итоговое состояние. Дополнение ДКА — смена принимающих и непринимавших состояний, но у НКА этот приём не работает (см. ловушку в разделе о конструкции подмно-

жеств). Принимающее состояние не обязано лежать на цикле: у языка a^*b на цикле лежит лишь начальное состояние, но из него достижимо принимающее. Если такого состояния нет, язык конечен: все его слова короче числа состояний.

Итог прост: четыре вопроса сводятся к поиску в конечном графе. Закономерность ясна: если модель конечна, любой вопрос о её поведении сводится к перебору конечного числа вариантов, а перебор всегда завершается. Вот что значит «вопрос разрешим»: существует алгоритм, который на любом автомате останавливается и выдаёт точный ответ.

Это первая встреча с большой темой — какие вопросы о вычислительной модели можно решить алгоритмически. Для конечных автоматов ответ почти всегда положителен. Для более мощных моделей с неограниченной памятью часть этих вопросов станет неразрешимой — неотъемлемое свойство модели, к которому мы вернёмся в главах 25 и 26.

§ 23.16 Верификация программ

Вопросы из прошлого раздела — не упражнение на аккуратность. На них держится целая инженерная область: проверка того, что программа ведёт себя правильно.

Программа, протокол или микросхема с конечным числом состояний — это конечный автомат. Состояние — память и регистры, переходы — шаги вычисления, а допустимые поведения — пути в графе состояний. Тогда вопрос «может ли программа попасть в аварийное состояние?» превращается в уже решённый вопрос о достижимости: достижимо ли из начального состояния некоторое плохое. Вопрос «завершится ли программа на любом входе?» — в вопрос о том, ведут ли все пути к принимающему состоянию.

Проверка свойств по такому описанию называется *верификацией моделей* (model checking). Программа или протокол задаются автоматом, свойство — набором плохих состояний или формулой, и машина отвечает, выполняется ли свойство, перебором конечного числа состояний. Именно конечность модели делает такую проверку возможной: перебор заведомо завершается.

Протоколы связи — классический пример. Состояние протокола — договорённости между отправителем и получателем: кто кого ждёт, что уже подтверждено. Вопрос, может ли протокол зайти в тупик или потерять сообщение безвозвратно, — вопрос о достижимости в конечном графе, и на него отвечает обход. Так проверяются ТСП, протоколы шины, контроллеры микросхем — везде, где число состояний конечно, а цена ошибки велика.

Отсюда же видно, где конечный автомат перестаёт хватать. Программа с рекурсией или динамической памятью может побывать в бесконечном числе состояний — их конечным числом не описать. Вопросы о такой программе в общем случае нераз-

решимы, и это уже тема машин Тьюринга и главы 26. Для конечных же моделей автомат даёт точный ответ — и только что мы разобрали, как именно.

§ 23.17 Итоги главы

Вся память автомата — номер состояния, и этого хватает, чтобы отвечать о бесконечном языке: всё, что нужно помнить о прочитанном слове, уместается в конечное множество состояний. Точные определения алфавита, слова и формального языка превратили этот факт в утверждения о классе регулярных языков, а на практике тот же аппарат работает в лексерах, поиске по регулярным выражениям и проверке свойств конечных систем.

ДКА, НКА и регулярные выражения описывают один и тот же класс языков — это теорема Клини, и она объясняет, почему недетерминизм оказывается удобством записи: его снимает конструкция подмножеств. Лемма о накачке и теорема Майхилла–Нерода дали две стороны одного вопроса: как отличить регулярный язык от нерегулярного и как найти каноническое описание языка.

Из конечности модели следует главная гарантия: любой вопрос об автомате — пустота, универсальность, эквивалентность — решается процедурой, которая всегда завершается. Гарантия исчезает, как только память перестаёт быть фиксированной: уже у контекстно-свободных языков (глава 24) часть таких вопросов становится неразрешимой. Минимизация связала математику с инженерией: единственный с точностью до изоморфизма минимальный ДКА служит канонической формой языка, и эквивалентность двух автоматов сводится к сравнению таких форм.

Модель исчерпана, и граница видна: язык $a^n b^n$ требует считать, а номер состояния этого не умеет. Следующая глава (24) даст памяти расти вместе со словом — стек добавит конечному автомату ровно ту возможность, которой ему не хватало.

Проверка Проверьте себя

1. Что такое формальный язык, и что значит «распознать язык»?
2. Почему НКА и ДКА распознают одни и те же языки?
3. Почему дополнение языка для ДКА — простая замена состояний, а для НКА тот же приём не работает?
4. Что утверждает теорема Клини и какие два формализма она связывает?
5. В чём суть леммы о накачке и почему она доказывает только нерегулярность, но не регулярность?
6. Что утверждает теорема Майхилла–Нерода и как из неё получается минимальный ДКА?
7. Почему вопрос «пуст ли язык автомата?» разрешим, хотя для более мощных моделей он неразрешим?
8. Чем конечный автомат отличается от комбинационной схемы?

Что дальше?

Автомат читает слово, хранит лишь номер состояния и отвечает в конце: конечная память, таблица переходов. Но граница уже видна — язык $a^n b^n$ регулярным не является: ему нужна память, способная считать. В следующей главе (24) мы перешагнём эту границу: контекстно-свободные грамматики и автоматы с магазинной памятью распознают $a^n b^n$ и языки со скобочной структурой. А затем, в главе 25, снимем ограничение памяти полностью — и увидим, как платой за универсальность становится неразрешимость.

§ 23.18 Упражнения

ДКА: построение и анализ.

1. Постройте ДКА, распознающий язык слов над $\{0, 1\}$, в которых число нулей делится на 3. Состояния должны соответствовать остатку от деления на 3.
2. Докажите, что любой конечный язык регулярен — построьте ДКА для произвольного конечного множества слов.
3. * Пусть $\Sigma = \{0, 1\}$. Постройте ДКА, распознающий язык слов, в которых пятый с конца символ равен 1. Сколько состояний потребовалось? Можно ли построить ДКА с меньшим числом состояний?

НКА и конструкция подмножеств.

4. Выполните конструкцию подмножеств для следующего НКА: состояния $\{q_0, q_1, q_2\}$, алфавит $\{0, 1\}$, переходы: $\delta(q_0, 0) = \{q_0, q_1\}$, $\delta(q_0, 1) = \{q_0\}$, $\delta(q_1, 1) = \{q_2\}$, принимающее состояние — q_2 . Нарисуйте диаграмму получившегося ДКА.

Регулярные выражения.

5. Запишите регулярное выражение для языка слов над $\{0, 1\}$, не содержащих подстроки 00 .
6. Запишите регулярное выражение для языка слов над $\{a, b\}$, содержащих не более двух букв b .
7. * Запишите регулярное выражение для языка слов над $\{0, 1\}$, в которых нет трёх идущих подряд одинаковых символов.

Свойства замкнутости.

8. Пусть L_1, L_2 — регулярные языки. Докажите, что $L_1 \setminus L_2$ (разность) также регулярен.
9. Докажите, что класс регулярных языков замкнут относительно обращения: если L регулярен, то язык $L^R = \{w^R \mid w \in L\}$ тоже регулярен. Подсказка: построьте НКА с ε -переходами, обратив переходы исходного автомата.

Лемма о накачке.

10. Примените лемму о накачке и докажите, что язык $\{a^n b^m \mid n > m\}$ нерегулярен.

Теорема Майхилла–Нерода и минимизация.

11. Используя теорему Майхилла–Нерода, докажите нерегулярность языка $\{a^n b^n \mid n \geq 0\}$. Предъявите бесконечное семейство попарно различных слов.
12. Постройте минимальный ДКА для языка слов над $\{0, 1\}$, заканчивающихся на 01 , напрямую через классы эквивалентности $\simeq_L$.
13. Минимизируйте следующий ДКА алгоритмом Мура: состояния q_0, q_1, q_2, q_3, q_4 , начальное — q_0 , принимающие — q_3 и q_4 . Переходы: $\delta(q_0, 0) = q_1, \delta(q_0, 1) = q_4, \delta(q_1, 0) = q_2, \delta(q_1, 1) = q_4, \delta(q_2, 0) = q_2, \delta(q_2, 1) = q_3, \delta(q_3, 0) = q_1, \delta(q_3, 1) = q_4, \delta(q_4, 0) = q_4, \delta(q_4, 1) = q_4$.
14. * Пусть язык L таков, что отношение $\simeq_L$ имеет ровно k классов эквивалентности. Докажите, что любой ДКА, распознающий L , имеет не менее k состояний, а ДКА, построенный по классам $\simeq_L$, минимален.

Разрешимость.

15. Опишите алгоритм, проверяющий, пуст ли язык данного ДКА. Оцените время его работы.
16. Опишите алгоритм, проверяющий эквивалентность двух ДКА. Какие свойства ДКА гарантируют, что алгоритм всегда завершается?
17. * Пусть L — регулярный язык. Определим $\text{HALF}(L) = \{x \mid \exists y : |y| = |x|, xy \in L\}$ — множество первых половин слов языка L . Докажите, что $\text{HALF}(L)$ регулярен. Подсказка: из ДКА для L постройте НКА, который «угадывает» середину слова.
18. * Рассмотрим язык $L = \{w \in \{0, 1\}^* \mid w \text{ в двоичной записи представляет число, делящееся на } 7\}$ (ведущие нули допустимы). Постройте ДКА, распознающий L , и объясните, почему он корректен. Указание: состояние — остаток от деления на 7. Чтение бита b удваивает значение и добавляет b .
19. * Найдите ошибку в следующем «доказательстве» регулярности языка $L = \{a^n b^n \mid n \geq 0\}$. Для длинного слова $a^k b^k$ пробуют найти разбиение $xyz = w$ с $|y| \geq 1$, при котором $xy^i z \in L$ для всех $i \geq 0$, и убеждаются, что никакое подходящее разбиение не существует. Отсюда заключают: лемма о накачке не опровергает регулярность L , следовательно, L регулярен. В чём логическая ошибка?
20. * Перемешивание строк u и v — множество $\text{SHUFFLE}(u, v)$ всех строк, получаемых слиянием символов u и v с сохранением порядка внутри каждой, например $\text{SHUFFLE}(ab, cd) = \{abcd, acbd, acdb, cabd, cadb, cdab\}$. Для языков положим $\text{SHUFFLE}(L_1, L_2) = \bigcup_{u \in L_1, v \in L_2} \text{SHUFFLE}(u, v)$. Докажите: если L_1

и L_2 регулярны, то $\text{SHUFFLE}(L_1, L_2)$ регулярен. Указание: постройте НКА с состояниями-парами.

ПРОЕКТ Регулярные выражения в исполняемый поиск по шаблону

Постройте работающий поиск по тексту, который компилирует регулярное выражение в автомат: парсер, автомат с ε -переходами, детерминизация и минимизация. Так устроена утилита `grep`: выражение компилируется в автомат, автомат прогоняется по тексту.

① *Парсер и автомат*

Соберите выражение в автомат: парсер ($|$, $*$, конкатенация, скобки, ε) и автомат с ε -переходами. Продумайте, как симулировать слово по автомату. Автомат должен принимать ровно слова языков «оканчивается на 01» и «чётное число единиц» — проверьте оба.

② *Детерминизация и операции*

Детерминизируйте автомат. Добавьте дополнение, объединение и пересечение языков. Продумайте, как проверить эквивалентность двух автоматов точно, а не на конечной выборке слов.

③ *Минимизация*

Приведите автомат к минимальному виду. Подумайте, как размер минимального автомата связан с числом классов отношения неразличимости, и проверьте эту связь на наборе языков.

④ *Раздувание*

Найдите семейство шаблонов, где выражение длины $O(n)$ порождает минимальный автомат с 2^n состояниями. Объясните, откуда берётся экспонента, и измерьте рост размеров по n .

⑤ *Ленивая детерминизация*

Детерминизация строит все подмножества сразу, но поиск по тексту обходит только достижимые. Реализуйте ленивую детерминизацию: состояния-подмножества порождаются на лету, когда слово действительно делает переход из них. Проверьте, что на коротком слове материализуется лишь малая часть полного ДКА. Сравните число материализованных состояний с полным размером детерминизации на выражении с экспоненциальным ДКА.

Итог: работающий поиск по шаблону от парсера до минимального автомата и измерение размеров на каждом звене конвейера.

XXIV

Контекстно-свободные языки

Конечный автомат из главы 23 не распознаёт язык $a^n b^n$ — слова, где сначала идут a , а затем столько же b . Причина — фиксированная память: у автомата с k состояниями префиксов $a^0, a^1, \dots, a^k$ на один больше, чем состояний, и по принципу Дирихле два из них — скажем, a^i и a^j при $i < j$ — приводят в одно состояние. Дописав к обоим префиксам b^i , автомат снова попадёт в одно состояние, хотя принимать обязан лишь $a^i b^i$, а не $a^j b^i$. Чтобы распознавать $a^n b^n$, нужна память, растущая вместе со словом.

Вложенность — вот что стоит за этой неудачей. Скобка, открытая раньше, закрывается позже: глубина вложенности растёт и падает, и этот след должен где-то храниться. Регулярные языки из прошлой главы плоские: слово — нитка из букв, без внутреннего устройства. Так устроены арифметические выражения: в $2 + (3 \cdot (4 + 1))$ первая открытая скобка замыкается последней. Тот же принцип в JSON: объект содержит массив, внутри которого — другие объекты. Так устроена и программа: вызов функции открывает блок, вложенные вызовы — блоки внутри блока. И разметка живёт по тому же правилу: тег открывается, внутри него открываются и закрываются вложенные теги, и только в конце — внешний.

Конечному автомату эта структура не по плечу: он видит символ и своё состояние, но не помнит, на какой глубине находится. Нужную память даёт *автомат с магазинной памятью (pushdown automaton)* — конечный автомат со стеком. Стек — это бесконечная память, но доступ к ней только сверху: символ, положенный последним, читается первым (принцип LIFO). Это память с одним концом: в середину не заглянуть — и в этом отличие от ленты, где доступ открыт в любом месте. Отсюда и название «магазинная память»: как в магазине винтовки, последним положенный патрон достаётся первым. Стек даёт автомату ровно одну новую способность — помнить глубину вложенности — и её хватает на всю скобочную структуру. По тому же правилу работают рекурсивные вызовы в программе: вызвал функцию — запомнил, куда вернуться, и последним достаётся самый свежий вызов.

До сих пор мы задавали язык распознаванием: автоматом или регулярным выражением. Но язык можно и порождать: задать правила, по которым из одного символа выводятся слова. Такое описание — *контекстно-свободная грамматика (context-free grammar)*: набор правил, в которых нетерминал заменяется строкой из терминалов и нетерминалов, и замена работает, где бы нетерминал ни стоял. Правило может звать само себя — рекурсия — и в этом вся сила: один шаг добавляет слой вложенности. Вывод слова — последовательность шагов, и её удобно читать как *дерево разбора*:

корень — стартовый символ, листья — буквы слова, ветви — применённые правила. Плоская строка прячет, какая скобка закрывает какую. Дерево разбора делает структуру видимой. Распознавание и порождение окажутся двумя взглядами на один и тот же класс языков, и на этом совпадении держится всё дальнейшее.

Нужда в таком описании пришла из языкознания и из проектирования компиляторов. Ноам Хомский в 1956 году опубликовал «Three Models for the Description of Language» — работу, в которой впервые выстроил грамматики в иерархию по их выразительности. Контекстно-свободные языки в этой иерархии — вторая ступень. Хомский строил модель естественного языка, но его иерархия описывает и языки программирования. Фрагмент такой модели выглядит знакомо: предложение — это подлежащее, сказуемое и дополнение, а подлежащее — это артикль с существительным. Несколько таких правил порождают множество осмысленных предложений — и это уже контекстно-свободная грамматика, только над словами.

ИСТОРИЯ Грамматики и языки программирования

Когда в конце 1950-х проектировали язык ALGOL 60, его синтаксис впервые описали формально. Джон Бэкус предложил способ записывать грамматику. Вскоре его уточнил Петер Наур, и получившаяся нотация получила имя Бэкуса–Наура (BNF). Синтаксис ALGOL 60 оказался контекстно-свободным, и с тех пор почти все языки программирования описываются именно так. Стек, добавленный к конечному автомату, — не абстракция. Алгоритм сортировочной станции, которым Дейкстра разбирал выражения по приоритету операторов, — это и есть работа со стеком.

За этим формализмом стоит работа компилятора. Спецификация синтаксиса языка — это контекстно-свободная грамматика, а парсер по тексту программы строит дерево разбора, по которому затем проводят проверки и генерируют код. Скобочная вложенность и вложенность конструкций языка — одна и та же структура. Разбор — механическая работа, и парсер по своему устройству — ближайший родственник автомата со стеком. Один и тот же аппарат описывает и абстрактный язык, и синтаксис языков программирования — в этом практическая отдача теории.

Но и у этой структуры есть предел: стек считает одну глубину, а два независимых счётчика уводят за пределы класса. Дальше по иерархии стек вырастет в ленту — бесконечную память с доступом в любом месте — и это будет машина Тьюринга (25). Наивная картина, в которой язык — просто список слов, здесь кончается: важна и сама строка, и то, как она собрана из частей. Как описать вложенность, если плоская строка прячет структуру, — и где кончается память со стеком?

ОБЗОР ГЛАВЫ

В этой главе мы зададим язык правилами вывода: контекстно-свободная грамматика заменяет нетерминалы на строки, и одного рекурсивного правила хватает,

чтобы породить вложенные скобки. Затем познакомимся со вторым описанием того же класса — автоматом с магазинной памятью — и увидим, что распознавание и порождение задают один и тот же класс языков. Деревья разбора покажут, что слово может иметь несколько структурных прочтений, и отсюда вырастет вопрос о неоднозначности грамматик.

Аналог леммы о накачке даст инструмент, которым доказывают, что язык не контекстно-свободен. Нормальная форма Хомского приведёт правила к единому виду и встанет за алгоритмом СЮК, а замкнутости покажут, какие операции класс выдерживает. В конце контекстно-свободные языки займут своё место в иерархии Хомского — между регулярными и контекстно-зависимыми.

После этой главы вы сможете:

1. задавать языки контекстно-свободными грамматиками и строить выводы слов;
2. строить автоматы с магазинной памятью и понимать их эквивалентность грамматикам;
3. читать деревья разбора и объяснять неоднозначность грамматик;
4. приводить грамматику к нормальной форме Хомского и применять лемму о накачке;
5. доказывать замкнутости КС-языков и проверять слово алгоритмом СЮК;
6. объяснять место контекстно-свободных языков в иерархии Хомского.

§ 24.1 Контекстно-свободные грамматики

Контекстно-свободная грамматика (КС-грамматика) — набор правил замены, по которым нетерминальный символ (имя синтаксической категории) заменяется на строку из терминалов (символов алфавита) и нетерминалов. Правила применяются независимо от контекста, в котором стоит нетерминал — отсюда название «контекстно-свободная». Процесс начинается со стартового нетерминала S . На каждом шаге любой нетерминал заменяется по одному из правил. Когда в строке не остаётся нетерминалов — получена терминальная строка, принадлежащая языку.

ОПРЕДЕЛЕНИЕ 24.1 Контекстно-свободная грамматика

Контекстно-свободной грамматикой называется четвёрка $G = (V, \Sigma, P, S)$, где:

- V — конечное множество нетерминалов (синтаксических категорий),
- Σ — конечный алфавит терминальных символов (входных символов языка),
- P — конечное множество продукций (правил вывода) вида $A \rightarrow \alpha$, где $A \in V$ — нетерминал, а $\alpha \in (V \cup \Sigma)^*$ — строка из терминалов и нетерминалов,
- $S \in V$ — стартовый нетерминал.

Язык, порождаемый грамматикой, обозначается $L(G)$ и состоит из всех терминальных строк, выводимых из S за конечное число шагов:

$$L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}.$$

Грамматика описывает язык порождением, а не перечислением: список слов бесконечен, а правила конечны. Несколько правил задают всё множество сразу — и это сжатие: грамматику можно записать, передать и разобрать, тогда как список — нельзя.

Регулярное выражение — это уже грамматика, только самая простая. Выражение $(a|b)^*a(a|b)$ задаёт язык строк с предпоследней буквой a . Тот же язык можно описать пятью правилами:

$$S \rightarrow aS, S \rightarrow bS, S \rightarrow aA, A \rightarrow a, A \rightarrow b$$

Грамматики, в которых разрешены лишь правила вида $A \rightarrow aB$, $A \rightarrow a$, называются *регулярными*. Для пустого слова допустимо ещё правило $S \rightarrow \varepsilon$. Они порождают в точности регулярные языки.

КС-грамматики снимают это ограничение: в правой части правила может стоять любая строка из терминалов и нетерминалов. Изменение, на вид незначительное, уже позволяет породить $a^n b^n$.

Пример КС-грамматика для $\{a^n b^n \mid n \geq 0\}$

$$S \rightarrow aSb \mid \varepsilon$$

Каждый вывод порождает одну a слева и одну b справа от S . Затем он либо снова раскрывает S , либо завершается. Завершение — это правило $S \rightarrow \varepsilon$. Пример вывода для $n = 3$:

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbbb = a^3 b^3.$$

Это минимальный пример языка, который не регулярен, но контекстно-свободен.

Следующий пример — чистый случай вложенности: кроме парных скобок в нём ничего нет. Рекурсия, которая в $a^n b^n$ порождает по одной паре на шаг, здесь строит всю структуру языка целиком.

Пример КС-грамматика для сбалансированных скобочных последовательностей

Язык правильных скобочных последовательностей (язык Дика) над алфавитом $\{ (,) \}$ задаётся грамматикой

$$S \rightarrow (S)S \mid \varepsilon.$$

Правило $S \rightarrow (S)S$ говорит: сбалансированные скобки — это либо пустая строка, либо пара скобок, внутри и после которой идут сбалансированные последовательности.

Вывод слова $((()))()$:

$$\begin{aligned}
 S &\rightarrow (S)S \\
 &\rightarrow ((S)S)S \\
 &\rightarrow ((\varepsilon)S)S \\
 &\rightarrow ((\varepsilon)\varepsilon)S \\
 &\rightarrow ((\varepsilon))S \\
 &\rightarrow ((\varepsilon))(\varepsilon)S \\
 &\rightarrow ((\varepsilon))(\varepsilon)S \\
 &\rightarrow ((\varepsilon))(\varepsilon)
 \end{aligned}$$

Распознать этот язык конечным автоматом нельзя: нужна память о глубине вложенности. Автомат с магазинной памятью кладёт $($ в стек при каждой открывающей скобке и снимает при каждой закрывающей.

Замечание

Рекурсивное правило $S \rightarrow (S)S$ порождает любую глубину вложенности, оставаясь одной строкой. Список конкретных конструкций — программ, выражений, разметки — конечен и потому не задаёт язык бесконечного множества строк. Грамматика сжимает это бесконечное множество в конечные правила. Левая часть правила описывает способ строить новую конструкцию, поэтому одно правило покрывает любую глубину.

Вложенность — не единственный источник правил. Грамматика может собирать слова из слов, и тогда язык оказывается множеством осмысленных фраз, а не всех строк над алфавитом. Два примера ниже показывают, как правила кодируют смысл.

Пример Грамматика предложений

Грамматика над словами, а не над буквами:

$$\begin{aligned}
 S &\rightarrow \text{NP VP}, \\
 \text{NP} &\rightarrow \text{Det } N \mid N, \\
 \text{VP} &\rightarrow V \text{ NP} \mid V, \\
 \text{Det} &\rightarrow \text{старый}, \\
 N &\rightarrow \text{кот} \mid \text{мышь}, \\
 V &\rightarrow \text{ловит} \mid \text{бежит}.
 \end{aligned}$$

Стартовый нетерминал S — предложение. Категория NP — именная группа, VP — глагольная группа, Det — определение, N — имя, V — глагол.

Левый вывод слова «старый кот ловит мышь»:

$$\begin{aligned}
S &\rightarrow NP \ VP \\
&\rightarrow \text{Det } N \ VP \\
&\rightarrow \text{старый } N \ VP \\
&\rightarrow \text{старый кот } VP \\
&\rightarrow \text{старый кот } V \ NP \\
&\rightarrow \text{старый кот ловит } NP \\
&\rightarrow \text{старый кот ловит } N \\
&\rightarrow \text{старый кот ловит мышь.}
\end{aligned}$$

На каждом шаге раскрывается самый *левый* нетерминал. Вывод слова «кот бежит» короче:

$$S \rightarrow NP \ VP \rightarrow N \ VP \rightarrow \text{кот } V \rightarrow \text{кот бежит.}$$

Не *всякая* последовательность слов выводится. Слово «кот старый» не принадлежит языку: в правиле $NP \rightarrow \text{Det } N$ определение стоит только перед именем, поэтому «старый» не может оказаться после «кот». Слово «ловит кот» тоже не выводится: предложение обязано начинаться с NP, а «ловит» — глагол.

Пример Палиндромы над словами

Скобки — не единственная вложенность: зеркально отражаться могут целые слова. Грамматика

$$S \rightarrow \text{кот } S \text{ кот} \mid \text{мышь } S \text{ мышь} \mid \text{кот} \mid \text{мышь} \mid \varepsilon$$

порождает *палиндромы*: слова, которые читаются одинаково слева направо и справа налево.

Вывод слова «кот мышь мышь кот»:

$$\begin{aligned}
S &\rightarrow \text{кот } S \text{ кот} \\
&\rightarrow \text{кот мышь } S \text{ мышь кот} \\
&\rightarrow \text{кот мышь мышь кот.}
\end{aligned}$$

Последний шаг применяет правило $S \rightarrow \varepsilon$: пустое слово не добавляет символов.

Слово «кот мышь» не выводится. Каждое применение правила $S \rightarrow wSw$ добавляет одно и то же слово слева и справа от S . Поэтому в любом непустом слове языка первое слово совпадает с последним. В слове «кот мышь» первое слово — «кот», последнее — «мышь», и они не совпадают.

§ 24.2 Автоматы с магазинной памятью

МП-автомат — конечный автомат со стеком, и вся новизна в этом «со стеком». Переход зависит от трёх вещей: от текущего состояния, от входного символа (или

от ε , если автомат действует спонтанно) и от символа на вершине стека. В результате перехода автомат заменяет вершину стека строкой символов: может положить несколько символов, один, либо снять верхушку — заменить её на ε .

ОПРЕДЕЛЕНИЕ 24.2 Автомат с магазинной памятью

Автоматом с магазинной памятью (МП-автоматом, pushdown automaton, PDA) называется семёрка

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

где:

- Q — конечное множество состояний,
- Σ — входной алфавит,
- Γ — алфавит стека: символы, которые можно класть в стек,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$ — функция переходов,
- $q_0 \in Q$ — начальное состояние,
- $Z_0 \in \Gamma$ — начальный символ стека,
- $F \subseteq Q$ — множество принимающих состояний.

Переход $(q', \gamma) \in \delta(q, a, A)$ читается так: находясь в состоянии q и прочитав символ a (либо ничего, если $a = \varepsilon$), увидев на вершине стека символ A , автомат переходит в состояние q' и заменяет символ A строкой γ . Замена на ε — это снятие вершины стека.

ОПРЕДЕЛЕНИЕ 24.3 Конфигурация МП-автомата

Конфигурацией МП-автомата называется тройка (q, w, γ) , где q — текущее состояние, w — неп прочитанный остаток входного слова, а γ — содержимое стека. Начальная конфигурация на слове w — тройка (q_0, w, Z_0) .

Слово принимается одним из двух равносильных способов. При *принятии по принимающему состоянию* автомат принимает слово w , если из начальной конфигурации существует последовательность переходов к конфигурации (q, ε, γ) с принимающим состоянием $q \in F$. При *принятии по пустому стеку* автомат принимает слово w , если существует последовательность переходов к конфигурации $(q, \varepsilon, \varepsilon)$.

Примечание Эквивалентность критериев принятия

Оба перехода можно построить явно. Чтобы принималось по пустому стеку, к автомату добавляют состояние-ловушку: из каждого принимающего состояния она опустошает стек за ε -переходы. Обратно, пустой стек имитируют донным символом, который никогда не снимают: автомат принимает, когда добирается до него. Детали конструкции — в стандартном учебнике по теории автоматов¹⁷⁹.

¹⁷⁹J. E. Hopcroft, R. Motwani, J. D. Ullman, глава 6.

Пример Распознавание $a^n b^n$ автоматом с магазинной памятью

Автомат работает в две фазы. В фазе q_0 , пока идут a , автомат кладёт в стек по символу A на каждую букву. Первая b переводит его в фазу q_1 , где на каждую b снимается по одному A .

Стек работает счётчиком: его глубина — число a , ещё не нашедших пару. Слово принимается по пустому стеку: к концу входа в стеке не должно остаться ни одного символа A . В фазе q_1 возврата к a уже нет. Пустое слово ε тоже принимается. Автомат сразу опустошает стек, снимая Z_0 .

Функция переходов:

$$\delta(q_0, a, Z_0) = \{(q_0, AZ_0)\}$$

$$\delta(q_0, a, A) = \{(q_0, AA)\}$$

$$\delta(q_0, b, A) = \{(q_1, \varepsilon)\}$$

$$\delta(q_1, b, A) = \{(q_1, \varepsilon)\}$$

$$\delta(q_1, \varepsilon, Z_0) = \{(q_1, \varepsilon)\}$$

$$\delta(q_0, \varepsilon, Z_0) = \{(q_1, \varepsilon)\}.$$

Переходы $\delta(q_1, a, \cdot)$, $\delta(q_0, b, Z_0)$ не определены. Это чтение a в фазе q_1 и чтение b на пустом счётчике. Попад в такую ситуацию, автомат застревает, и слово отвергается.

Прогон слова $aabb$ (стартовый символ стека — Z_0):

$$(q_0, aabb, Z_0) \rightarrow (q_0, abb, AZ_0) \rightarrow (q_0, bb, AAZ_0) \rightarrow (q_1, b, AZ_0) \rightarrow (q_1, \varepsilon, Z_0) \rightarrow (q_1, \varepsilon, \varepsilon).$$

Последний шаг снимает и Z_0 . Стек пуст — слово принято. Слово aba отвергается. В фазе q_1 символ a прочитать нельзя: перехода нет.

Пример Распознавание палиндромов

Палиндромы над $\{a, b\}$ требуют недетерминизма: автомат должен *угадать середину* слова. До середины он кладёт символы в стек, после середины — снимает и сравнивает.

Состояния: q_0 — фаза записи, q_1 — фаза сравнения. Переходы (для любого верхнего символа стека $Z \in \{a, b, Z_0\}$):

$$\begin{aligned}
\delta(q_0, a, Z) &= \{(q_0, aZ), (q_1, Z)\} \\
\delta(q_0, b, Z) &= \{(q_0, bZ), (q_1, Z)\} \\
\delta(q_0, \varepsilon, Z) &= \{(q_1, Z)\} \\
\delta(q_1, a, a) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, b, b) &= \{(q_1, \varepsilon)\} \\
\delta(q_1, \varepsilon, Z_0) &= \{(q_1, \varepsilon)\}.
\end{aligned}$$

Первая пара переходов нетривиальна: по каждому символу у автомата два выбора — продолжить запись или перейти к сравнению, не положив символ в стек. Второй выбор пропускает средний символ нечётного палиндрома, ничего не кладя в стек. Переход по ε — то же решение, принятое до чтения символа: так устроена середина чётного палиндрома. В фазе сравнения вершина стека обязана совпасть со входным символом.

Прогон слова aba :

$$(q_0, aba, Z_0) \rightarrow (q_0, ba, aZ_0) \rightarrow (q_1, a, aZ_0) \rightarrow (q_1, \varepsilon, Z_0) \rightarrow (q_1, \varepsilon, \varepsilon).$$

На втором шаге автомат угадал середину на символе b : с этого момента стек сравнивается со входом, и слово принимается по пустому стеку. Для не-палиндрома ab ни один выбор середины не приводит к пустому стеку: автомат застревает, не дочитав или не опустошив стек.

В регулярном случае недетерминизм был лишь удобством записи: конструкция подмножеств снимала его без потери языка. Для палиндромов это не так: детерминированный автомат не знает, где середина, угадать её может только недетерминированный. Язык палиндромов над алфавитом из двух букв — стандартный пример языка, который не распознаётся ни одним детерминированным МП-автоматом.

ТЕОРЕМА 24.1 Эквивалентность КСГ и PDA

Язык порождается контекстно-свободной грамматикой тогда и только тогда, когда он распознаётся автоматом с магазинной памятью.

Набросок доказательства

Докажем эквивалентность в обе стороны построениями.

Из грамматики строится недетерминированный МП-автомат с одним состоянием, который имитирует левый вывод: нетерминалы на вершине стека заменяются по правилам грамматики и сравниваются с терминалами входной строки. Обратно, по автомату строится грамматика, нетерминалы которой кодируют переходы между состояниями с учётом содержимого стека. Эта конструкция аналогична доказательству эквивалентности НКА и регулярных выражений

через исключение состояний, но сложнее из-за стека. Детали — в стандартном учебнике по теории автоматов и формальных языков¹⁸⁰.

Пример Имитация левого вывода на слове $aabb$

Автомат, построенный в наброске доказательства, работает по правилам грамматики $S \rightarrow aSb \mid \varepsilon$. В стеке живут нетерминалы и терминалы, ожидающие совпадения с входом. Прогон слова $aabb$ (конфигурации — тройки (q, w, γ)):

$$(q, aabb, S) \rightarrow (q, aabb, aSb) \rightarrow (q, abb, Sb) \rightarrow (q, abb, aSbb) \rightarrow (q, bb, Sbb) \rightarrow (q, bb, bb) \rightarrow (q, b, b) \rightarrow (q, \varepsilon, \varepsilon).$$

Шаги с ε -переходом разворачивают верхний нетерминал по одному из правил. Правила — $S \rightarrow aSb$ или $S \rightarrow \varepsilon$. Шаги по терминалу сверяют вершину стека со входным символом и снимают её. Последовательность разворотов повторяет левый вывод $S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$. Автомат строит вывод по ходу работы. В конце стек пуст, и слово принято по пустому стеку.

Пример Автомат для сбалансированных скобок

Язык Дика из первого раздела получает особенно простой автомат: один символ в стек кладётся на открывающую скобку и снимается на закрывающую. Устройство — одно состояние q и стек с донным символом Z_0 , принимает по пустому стеку. Переходы:

$$\begin{aligned}\delta(q, (, Z) &= \{(q, (Z)\} \text{ для } Z \in \{ (, Z_0 \} \\ \delta(q,), () &= \{(q, \varepsilon)\} \\ \delta(q, \varepsilon, Z_0) &= \{(q, \varepsilon)\}.\end{aligned}$$

Первая группа кладёт скобку на любую вершину. Вторая снимает пару. Третий переход убирает дно, когда вход кончился и лишних скобок нет: так принимается и пустое слово.

Прогон слова $((()))()$:

$$(q, ((()))(), Z_0) \rightarrow (q, (())(), (Z_0) \rightarrow (q, (())(), ((Z_0) \rightarrow (q, (())(), (Z_0) \rightarrow (q, (), Z_0) \rightarrow (q,), (Z_0) \rightarrow (q, \varepsilon, Z_0) \rightarrow (q, \varepsilon, \varepsilon)$$

Каждая закрывающая скобка снимает ровно ту открывающую, что лежит сверху, поэтому скобки разных пар не могут перемешаться.

В отличие от палиндромов, автомат здесь детерминирован: открывающая скобка всегда кладётся, закрывающая всегда снимает, выбора нет. Язык Дика — стандартный пример контекстно-свободного языка, который распознаётся детерминированным автоматом с магазинной памятью.

¹⁸⁰J. E. Hopcroft, R. Motwani, J. D. Ullman. «Introduction to Automata Theory, Languages, and Computation». 3-е изд., 2006, глава 6.

§ 24.3 Структура контекстно-свободных языков

Вывод в КС-грамматике удобно представлять *деревом разбора*: корень — S , внутренние вершины — нетерминалы, листья — терминалы. Линейная запись строки прячет структуру: на одной строке не видно, какая скобка закрывает какую. Дерево разбора возвращает этой структуре объём — сразу видно, что внутри чего, и компилятору, и человеку.

ОПРЕДЕЛЕНИЕ 24.4 Дерево разбора

Деревом разбора слова w по грамматике G называется дерево, в котором:

- Корень помечен стартовым нетерминалом S .
- Каждая внутренняя вершина помечена нетерминалом A .
- Её дети слева направо помечены символами правой части правила $A \rightarrow \alpha \in P$.
- Конкатенация меток листьев слева направо равна w .

Пример Дерево разбора слова a^3b^3

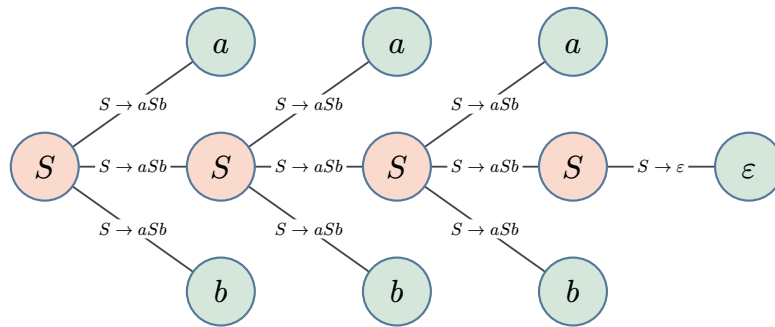
Грамматика $S \rightarrow aSb \mid \varepsilon$ порождает слово a^3b^3 тремя вложениями. Дерево разбора повторяет эту вложенность:

1. Корень помечен стартовым символом S .
2. Корень раскрыт по правилу $S \rightarrow aSb$: дети — a, S, b .
3. Внутренний S раскрыт по тому же правилу: снова дети a, S, b .
4. Следующий S — ещё раз: дети a, S, b .
5. Самый нижний S раскрыт по правилу $S \rightarrow \varepsilon$: единственный ребёнок — пустая строка.

Листья слева направо: $a, a, a, \varepsilon, b, b, b$. Пустой лист не даёт символов, поэтому конкатенация листьев — слово a^3b^3 .

Дерево содержит четыре вхождения S : корень и три вложенных нетерминала. Три вложения соответствуют трём парам (a, b) . Глубина стека при разборе совпадает с числом вложений.

То же дерево изображено на рис. 103: корень S стоит слева, листья — справа, каждое раскрытие S помечено применённым правилом.

Рис. 103. Дерево разбора слова a^3b^3 .

Строка может иметь несколько деревьев разбора — см. определение ниже.

ОПРЕДЕЛЕНИЕ 24.5 Неоднозначная грамматика

Грамматика называется **неоднозначной**, если некоторое слово языка $L(G)$ имеет более одного дерева разбора.

Неоднозначность — указание на множественность структурных интерпретаций: при разборе программы разные деревья дают разный смысл. Классический пример — приоритет операторов.

Пример Неоднозначность: приоритет операторов

Грамматика арифметических выражений

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

порождает слово $\text{id} + \text{id} * \text{id}$ двумя способами.

Два дерева разбора:

1. Одно склеивает сначала $\text{id} + \text{id}$.

Это соответствует чтению $(\text{id} + \text{id}) * \text{id}$.

1. Другое — сначала $\text{id} * \text{id}$.

Это чтение $\text{id} + (\text{id} * \text{id})$.

При конкретных числах значения расходятся: $(2 + 3) \cdot 4 = 20 \neq 14 = 2 + (3 \cdot 4)$.

Чтобы однозначно описать приоритет, правила разделяют на уровни:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}.$$

Слагаемые строятся из множителей, умножение связывается раньше сложения. Слово $\text{id} + \text{id} * \text{id}$ получает единственное дерево разбора. Это чтение $\text{id} + (\text{id} * \text{id})$.

Замечание

Неоднозначность не ждёт разных приоритетов: один оператор на одном уровне тоже допускает две группировки. Строку $8 / 4 / 2$ можно прочитать по-разному. Левая группировка даёт $\frac{8}{2} = 4$. Правая — $\frac{8}{4} = 2$. Умножение такую разницу прячет — при любой группировке $8 \cdot 4 \cdot 2 = 64$ — а деление немедленно выдаёт. Дерево разбора отвечает и за то, что с чем соединяется, и за то, какая пара соединяется первой.

Пример Пять деревьев для одного слова

Грамматика $E \rightarrow E + E \mid \text{id}$ порождает слово $\text{id} + \text{id} + \text{id} + \text{id}$ пятью разными деревьями. Пять деревьев соответствуют пяти расстановкам скобок:

1. $((\text{id} + \text{id}) + \text{id}) + \text{id}$
2. $(\text{id} + (\text{id} + \text{id})) + \text{id}$
3. $\text{id} + ((\text{id} + \text{id}) + \text{id})$
4. $\text{id} + (\text{id} + (\text{id} + \text{id}))$
5. $(\text{id} + \text{id}) + (\text{id} + \text{id})$

Четыре операнда дают пять деревьев — число Каталана $C_3 = 5$. Для трёх операндов деревьев два, для пяти — четырнадцать. Все пять деревьев вычисляют одно и то же значение: сложение ассоциативно. Неоднозначность — свойство синтаксиса: структура у слова есть, но не одна.

Для программиста это — рабочий инструмент. Нотация Бэкуса–Наура (BNF), в которой записывают спецификации языков, — это синтаксический вариант КС-грамматик, а парсеры — алгоритмы построения дерева разбора: рекурсивный спуск (recursive descent), Эрли (Earley), Кок–Янггер–Касами (СҮК). Каждый налагает свои условия: рекурсивный спуск не терпит левой рекурсии (правил вида $A \rightarrow A\alpha$, когда нетерминал порождает сам себя слева). СҮК требует нормальной формы Хомского, алгоритм Эрли работает для произвольных КС-грамматик за $O(n^3)$.

Парсеры отвечают на вопрос о конкретном слове: есть ли у него дерево разбора. Обратный вопрос — какие языки вообще под силу контекстно-свободным грамматикам — требует инструмента, который работает с языком целиком, а не с одним словом.

Для контекстно-свободных языков существует аналог леммы о накачке, и отличие от регулярного случая видно уже в формулировке.

ТЕОРЕМА 24.2 Лемма о накачке для контекстно-свободных языков

Пусть язык L контекстно-свободен. Тогда существует целое число $p \geq 1$ такое, что любое слово $w \in L$, $|w| \geq p$ разбивается на пять частей. Запишем разбиение как $w = uvxyz$, где:

- $|vy| > 0$ (накачиваемая пара непуста),

- $|vxy| \leq p$ (подстрока vxy ограничена по длине),
- $\forall i \geq 0 : uv^i xy^i z \in L$ (обе подстроки повторяются одновременно).

Накачиваются *две* подстроки одновременно и симметрично — структурное следствие того, что стек растёт и уменьшается согласованно. В регулярном случае подстрока одна — цикл в конечном графе переходов. Здесь их две.

Набросок доказательства

Докажем построением по дереву разбора длинного слова.

Сначала приведём грамматику к нормальной форме Хомского (раздел ниже): каждое правило в ней имеет вид $A \rightarrow BC$ или $A \rightarrow a$, дерево разбора становится бинарным, и его проще мерить по высоте. Пусть в грамматике m нетерминалов, положим $p = 2^m$ и возьмём слово $w \in L$ с $|w| \geq p$.

В бинарном дереве с более чем 2^m листьями есть путь от корня к листу, на котором лежит более m нетерминалов. На этом пути больше вершин, чем разных нетерминалов, поэтому по принципу Дирихле некоторый нетерминал A встретится дважды. Выберем два вхождения A , расположенные ближе всего к листу: на участке от верхнего из них до листа все нетерминалы попарно различны.

Разрежем слово по этим двум вершинам. Нижнее вхождение A порождает подстроку x , верхнее порождает vxy : слева от x стоит v , справа — y . Всё, что осталось вокруг поддерева верхнего A , образует префикс u и суффикс z , итого $w = uvxyz$.

Теперь накачаем поддерево. Заменив поддерево верхнего A на поддерево нижнего, получим $uxz = uv^0 xy^0 z$, а заменив наоборот — $uv^2 xy^2 z$. Повторяя замену, получаем $\forall i \geq 0 : uv^i xy^i z \in L$. Обе вершины помечены одним нетерминалом A , поэтому каждая подстановка снова даёт корректное дерево разбора.

Осталось объяснить два ограничения. Поддерево верхнего вхождения A невысокое: ниже него повторов нет, поэтому его высота не превосходит m , и в бинарном дереве оно порождает не более $2^m = p$ терминалов. Все они лежат внутри vxy , откуда $|vxy| \leq p$.

Наконец, $|vy| > 0$. У нетерминалов, отличных от стартового, ε -правил в нормальной форме нет, поэтому каждый нетерминал на пути между двумя вхождениями A имеет соседнее поддерево с хотя бы одним терминалом. Все такие терминалы попадают в v или в y .

Пример $\{a^n b^n c^n \mid n \geq 0\}$ не контекстно-свободен

Предположим противное: пусть язык $L = \{a^n b^n c^n\}$ контекстно-свободен, и пусть p — его длина накачки. Возьмём слово $w = a^p b^p c^p$, его длина $3p \geq p$, поэтому по лемме w разбивается как $uvxyz$ с $|vxy| \leq p$ и $|vy| \geq 1$.

Подстрока vxy не может захватить и буквы a , и буквы c : между ними стоит блок из p букв b , и тогда все три блока вместе заняли бы больше p символов. Значит, vxy не содержит либо c , либо a . Пусть в vxy нет буквы c , тогда меняется количество только букв a и b , а количество c остаётся равным p . Так как $|vy| \geq 1$, меняется хотя бы одно из количеств a или b , и слово перестаёт иметь поровну всех трёх букв. Случай, когда в vxy нет буквы a , разбирается симметрично.

В каждом случае накачанное слово $uv^2xy^2z \notin L$, а это противоречие, поэтому язык $a^n b^n c^n$ не контекстно-свободен.

Пример Две пары счётчиков: $\{a^i b^j c^i d^j\}$

Язык $L = \{a^i b^j c^i d^j \mid i, j \geq 0\}$ требует два независимых равенства: количеств a и c , а также b и d . Применим лемму о накачке к слову $w = a^p b^p c^p d^p$, подстрока vxy задевает не более двух соседних блоков.

Случай 1. Блок d не задет. Количество d остаётся p , а хотя бы одно из количеств a, b, c меняется: если изменилось количество a или c , нарушено равенство количеств a и c , а если изменилось количество b , нарушено равенство количеств b и d .

Случай 2. Блок d задет. Тогда блок b не задет, количество b остаётся p , а количество d меняется, и равенство количеств b и d нарушено.

В каждом случае накачанное слово uv^2xy^2z не лежит в L , поэтому язык L не контекстно-свободен.

§ 24.4 Нормальная форма Хомского

Лемма о накачке опиралась на грамматику с однообразными правилами, и требование однообразия оформляется в отдельную нормальную форму. Нормальная форма Хомского — приведённая КС-грамматика, в которой каждое правило либо порождает один терминал, либо склеивает строку из двух нетерминалов.

ОПРЕДЕЛЕНИЕ 24.6 Нормальная форма Хомского

Грамматика называется **приведённой к нормальной форме Хомского**, если каждое правило имеет вид $A \rightarrow BC$ или $A \rightarrow a$, где A, B, C — нетерминалы, а a — терминал. Для языков, содержащих пустое слово, дополнительно разрешается одно правило $S_0 \rightarrow \varepsilon$, где S_0 — стартовый нетерминал, который не встречается в правых частях правил.

Произвольная грамматика так не выглядит: правые части бывают длинными, содержат терминалы среди нетерминалов, а иные правила вовсе ничего не порождают. Три препятствия убираются механически: ε -правила вида $A \rightarrow \varepsilon$, цепные правила

вида $A \rightarrow B$ и правые части длины три и больше. Каждый шаг сохраняет язык, и весь процесс завершается за полиномиальное от размера грамматики время.

Примечание Пустое слово при приведении

Полный запрет ε -правил сдвигает язык: грамматика без них не порождает ε , а он может принадлежать $L(G)$. Переживает приведение единственное ε -правило — правило нового стартового символа $S_0 \rightarrow \varepsilon$, и потому S_0 не пускают в правые части. Остальные ε -правила убирают так: сначала находят все нетерминалы, из которых выводится ε (обнуляемые), затем в каждом правиле, содержащем обнуляемый символ, добавляют все варианты с вычеркнутыми вхождениями. Из правила $S \rightarrow aSb$ с обнуляемым S вырастает правило $S \rightarrow ab$: без него приведённая грамматика потеряла бы слово ab .

АЛГОРИТМ Приведение КС-грамматики к нормальной форме Хомского

Вход: КС-грамматика G .

1. Ввести новый стартовый нетерминал S_0 с правилом $S_0 \rightarrow S$, а если $\varepsilon \in L(G)$, то и с правилом $S_0 \rightarrow \varepsilon$.
2. Найти все обнуляемые нетерминалы, удалить ε -правила, кроме $S_0 \rightarrow \varepsilon$, и в каждое правило с обнуляемыми символами добавить все варианты с вычеркнутыми вхождениями, отбросив дубликаты и бесполезные правила вида $A \rightarrow A$.
3. Устранить цепные правила: для каждой пары A, B , где $A \xrightarrow{*} B$ по цепным правилам, добавить все нецепные правила $B \rightarrow \gamma$, а цепные правила удалить.
4. В правых частях длины два и больше заменить каждый терминал a новым нетерминалом T_a с единственным правилом $T_a \rightarrow a$.
5. Расщепить правые части длины три и больше: правило $A \rightarrow X_1 X_2 \dots X_k$ заменить цепочкой правил $A \rightarrow X_1 N_2, N_2 \rightarrow X_2 N_3, \dots, N_{k-1} \rightarrow X_{k-1} X_k$ с новыми нетерминалами N_i .

Пример Полное приведение грамматики $S \rightarrow aSb \mid \varepsilon$

Новый стартовый символ: $S_0 \rightarrow S$ и $S_0 \rightarrow \varepsilon$, потому что ε принадлежит языку. Правило $S \rightarrow \varepsilon$ удаляется, и из $S \rightarrow aSb$ с обнуляемым S появляется вариант $S \rightarrow ab$. Цепное правило $S_0 \rightarrow S$ заменяется нецепными правилами S : вместо него появляются $S_0 \rightarrow aSb$ и $S_0 \rightarrow ab$. Терминалы в длинных правых частях заменяются: $A \rightarrow a, B \rightarrow b$. Расщепление тройки ASB даёт $S \rightarrow AX$ и $X \rightarrow SB$, а правила $S_0 \rightarrow ASB$ расщепляются так же: $S_0 \rightarrow AX$. Итог:

$$S_0 \rightarrow AX \mid AB \mid \varepsilon, \quad S \rightarrow AX \mid AB, \quad X \rightarrow SB, \quad A \rightarrow a, \quad B \rightarrow b.$$

Вывод слова a^2b^2 :

$$S_0 \rightarrow AX \rightarrow aX \rightarrow aSB \rightarrow aABB \rightarrow aaBB \rightarrow aabB \rightarrow aabb.$$

После замены терминалов $S \rightarrow ab$ превращается в $S \rightarrow AB$, и дерево разбора в итоговой грамматике бинарно.

Проверка Нормальная форма Хомского

1. Почему нельзя просто запретить все ε -правила и что делают вместо этого?
2. Правило $A \rightarrow BCD$ запрещено в нормальной форме. Какие правила появятся после его расщепления?
3. Правило $A \rightarrow aB$ не имеет допустимого вида. Как его привести к нормальной форме?

§ 24.5 Замкнутость КС-языков

Замкнутость говорит, какие операции над языками сохраняют класс, а какие выводят из него. У регулярных языков ответ сплошь положительный: класс замкнут относительно булевых операций, конкатенации и звезды Клини (23). У контекстно-свободных спектр уже: одни операции класс сохраняет, другие — нет.

УТВЕРЖДЕНИЕ 24.3 Замкнутость относительно объединения, конкатенации и звезды

Если языки L_1 и L_2 контекстно-свободны, то контекстно-свободны также $L_1 \cup L_2$, конкатенация $L_1 L_2$ и звезда Клини L_1^* .

Доказательство

Докажем построением грамматик.

Пусть G_1 и G_2 порождают L_1 и L_2 со стартовыми символами S_1 и S_2 . Без ограничения общности нетерминалы грамматик не пересекаются — при необходимости их переименовывают. Новый стартовый символ S добавляет по одному правилу в каждом из трёх случаев.

Объединение. Правило $S \rightarrow S_1 \text{ mid } S_2$ — первый шаг вывода выбирает грамматику, дальше она работает без помех, потому что замена нетерминала происходит в любом контексте.

Конкатенация. Правило $S \rightarrow S_1 S_2$: сначала порождается слово из L_1 , затем нетерминал S_2 раскрывается в слово из L_2 .

Звезда. Правила $S \rightarrow S_1 S \text{ mid } S \rightarrow \varepsilon$: каждый виток добавляет очередное слово из L_1 , а переход по ε заканчивает вывод.

УТВЕРЖДЕНИЕ 24.4 Замкнутость относительно гомоморфизмов

Пусть $L \subseteq \Sigma^*$ контекстно-свободен, а $h : \Sigma^* \rightarrow \Delta^*$ — гомоморфизм, то есть отображение с $h(uv) = h(u)h(v)$ и $h(\varepsilon) = \varepsilon$. Тогда образ $h(L)$ контекстно-свободен.

Набросок доказательства

Докажем заменой терминалов в правилах. В каждом правиле грамматики терминал a заменяется строкой $h(a)$, и вывод слова w превращается в вывод $h(w)$. Обратно, всякое слово из $h(L)$ получается из вывода какого-то прообраза, поэтому язык совпадает точно. Замкнутость относительно обратного образа $h^{-1}(L)$ тоже верна: МП-автомат, прочитав символ a , держит в стеке строку $h(a)$ и скормливает её моделируемому автомату по одному символу.

УТВЕРЖДЕНИЕ 24.5 Замкнутость относительно пересечения с регулярным языком

Если L контекстно-свободен, а R регулярен, то пересечение $L \cap R$ контекстно-свободно.

Набросок доказательства

Докажем произведением автоматов. Пусть МП-автомат распознаёт L , а регулярный язык R распознаёт детерминированный конечный автомат. Строится произведение: состояние — пара состояний двух автоматов, стек один, входной символ синхронно сдвигает обе модели, и слово принимается, когда обе модели принимают. Недетерминизм МП-автомата ничему не мешает: пары переходов перебираются так же, как раньше.

Пересечение с регулярным языком — рабочий инструмент доказательств: чтобы показать, что язык не контекстно-свободен, его часто пересекают с регулярным и применяют лемму о накачке к результату. За пределами этого списка класс уже не держится: пересечение двух контекстно-свободных языков может не быть контекстно-свободным. Контрпример собирается из языка $a^n b^n c^n$, который не контекстно-свободен (пример выше):

$$\{a^i b^i c^j \mid i, j \geq 0\} \cap \{a^i b^j c^j \mid i, j \geq 0\} = \{a^n b^n c^n \mid n \geq 0\}.$$

Первый язык требует равенства количеств a и b и безразличен к c : его порождает грамматика $S_1 \rightarrow AC$ с правилами $A \rightarrow aAb \mid \varepsilon$ и $C \rightarrow cC \mid \varepsilon$. Второй строится зеркально: $S_2 \rightarrow AB$, $A \rightarrow aA \mid \varepsilon$, $B \rightarrow bBc \mid \varepsilon$. Если бы пересечение двух КС-языков всегда было КС-языком, язык $a^n b^n c^n$ оказался бы контекстно-свободным — противоречие.

Не замкнут класс и относительно дополнения. Будь дополнение каждого КС-языка КС-языком, пересечение выражалось бы через объединение по законам де Моргана:

$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, а объединение замкнуто, как показано выше. Противоречие показывает: дополнение к контекстно-свободному языку может не быть контекстно-свободным.

Проверка Замкнутость КС-языков

1. Зачем при построении грамматики для $L_1 \cup L_2$ переименовывать нетерминалы?
2. Языки $\{a^i b^i c^j\}$ и $\{a^i b^j c^j\}$ контекстно-свободны. Что известно об их пересечении?
3. Каким построением доказывается замкнутость относительно пересечения с регулярным языком и почему достаточно одного стека?

§ 24.6 От грамматики к парсеру

До сих пор КС-грамматика задавала язык: множество слов, которые можно вывести. Программист сталкивается с обратной задачей — *разбором* (parsing): дана строка, нужно найти её вывод в грамматике, то есть построить дерево разбора. Именно это делает компилятор, когда превращает текст программы в синтаксическое дерево.

Нисходящий разбор (top-down) начинает со стартового символа и разворачивает продукции, стремясь получить строку. Восходящий разбор (bottom-up) начинает со строки и сворачивает правые части продукций к стартовому символу. Вся трудность — в выборе: в нисходящем разборе — какую продукцию применить к нетерминалу, в восходящем — где и что свернуть. Нисходящий разбор удобно читать как работу со стеком: нетерминалы, которые ещё нужно раскрыть, ведут себя как цели на стеке. Новая продукция кладёт цели сверху, совпадение с входной строкой — снимает верхнюю — и парсер по устройству снова оказывается автоматом со стеком.

24.6.1 Алгоритм Кока–Янгера–Касами

Алгоритм СУК проверяет принадлежность слова языку и попутно даёт материал для дерева разбора. Работает он с грамматикой в нормальной форме Хомского, и именно единообразие правил делает алгоритм простым: каждое правило либо порождает один терминал, либо склеивает строку из двух частей. Идея — динамическое программирование по подстрокам. Для слова $w = w_1 w_2 \dots w_n$ заполняется таблица $X_{i,j}$, где клетка хранит множество нетерминалов, выводящих подстроку $w_i \dots w_j$. База — подстроки длины один: $A \in X_{i,i}$, если в грамматике есть правило $A \rightarrow w_i$. Шаг — подстроки длиннее: подстрока $w_i \dots w_j$ разрезается на префикс и суффикс в каждой точке k , и если B выводит префикс, C — суффикс, а правило $A \rightarrow BC$ есть в грамматике, то A выводит всю подстроку:

$$X_{i,j} = \bigcup_{k=i}^{j-1} \{A \mid A \rightarrow BC, B \in X_{i,k}, C \in X_{k+1,j}\}.$$

Слово принадлежит языку, когда S попадает в клетку $X_{1,n}$.

АЛГОРИТМ Распознавание алгоритмом СΥΚ

Вход: грамматика в нормальной форме Хомского и слово $w_1 \dots w_n$.

1. Для каждого i положить $X_{i,i} = \{A \mid A \rightarrow w_i \in P\}$.
2. Для длин $l = 2, \dots, n$ и начал i вычислить $X_{i,i+l-1}$ по формуле выше, перебирая все точки разреза k .
3. Ответ: слово принадлежит языку, если $S \in X_{1,n}$.

Пример Работа СΥΚ на слове $aabb$

Возьмём приведённую выше грамматику для $a^n b^n$ без стартового S_0 и правила $S_0 \rightarrow \varepsilon$: $S \rightarrow AX \mid AB, X \rightarrow SB, A \rightarrow a, B \rightarrow b$.

1. Длина один: в клетках для букв a стоит A , для букв b — B .
2. Длина два: подстроки ab дают S по правилу $S \rightarrow AB$, а клетки подстрок aa и bb пусты.
3. Длина три: подстрока abb склеивается из S в клетке для ab и B в клетке для последней b по правилу $X \rightarrow SB$, и в её клетку входит X .
4. Длина четыре: всё слово склеивается из A в клетке для первой a и X в клетке для abb по правилу $S \rightarrow AX$, и в клетку $X_{1,4}$ входит S — слово принадлежит языку.

Дерево разбора восстанавливается обратным ходом: каждое попадание нетерминала в клетку запоминает правило и точку разреза, и от клетки $X_{1,4}$ дерево раскрывается до листьев.

Примечание СΥΚ как динамическое программирование

СΥΚ — динамическое программирование по подстрокам: ответ для подстроки собирается из ответов для её частей, каждая клетка вычисляется один раз и работает на многие клетки сверху. Клеток $O(n^2)$, и на каждую перебирается $O(n)$ точек разреза, поэтому время работы $O(n^3 \cdot |G|)$, где $|G|$ — размер грамматики. Тот же приём промежуточной точки k работает в алгоритме Флойда–Уоршелла (9): там из коротких путей собираются длинные, здесь — из коротких выводов длинные.

24.6.2 FIRST, FOLLOW и LL(1)-разбор

Нисходящему разбору хочется выбрать продукцию сразу, без перебора и возвратов. Выбор подсказывают два множества. $\text{FIRST}(A)$ — терминалы, с которых может начинаться строка, выводимая из A . Если $A \xrightarrow{*} \varepsilon$, в множество входит и ε . $\text{FOLLOW}(A)$ — терминалы, которые могут стоять в выводе непосредственно после A , а за стартовым символом ставят служебный маркер конца слова. Оба множества вычисляются обходом правил до неподвижной точки. Для правила $A \rightarrow \alpha B \beta$ терминалы из $\text{FIRST}(\beta)$ попадают в $\text{FOLLOW}(B)$, а если β выводит ε , то добавляется и весь $\text{FOLLOW}(A)$. По этим множествам строится предсказывающая

таблица: строки — нетерминалы, столбцы — терминалы подглядывания, в клетку (A, a) попадают продукции $A \rightarrow \alpha$, которые могут начать вывод с символа a . Грамматика называется LL(1), если ни одна клетка не содержит двух продукций: тогда один символ подглядывания решает выбор полностью, и разбор идёт без возвратов. Левая рекурсия гарантирует конфликт: продукции $A \rightarrow A\alpha$ и $A \rightarrow \beta$ претендуют на пересекающиеся множества начальных терминалов. Поэтому перед LL(1)-разбором левую рекурсию устраняют.

Практические классы детерминированы. Грамматики LL(k) разбираются нисходяще с подглядыванием на k символов вперёд, а грамматики LR(k) — восходяще. Большинство грамматик языков программирования — LR (или их упрощение LALR), поэтому для них существуют генераторы парсеров: yacc, bison, ANTLR. Рекурсивный спуск — простейший нисходящий парсер: по одной функции на нетерминал, его пишут вручную.

LL- и LR-языки — собственные подклассы КС-языков, и не всякая КС-грамматика разбирается детерминированно: недетерминизм — цена выразительности. Это тот же компромисс, что с автоматами: недетерминированная модель описывает больше, а детерминированная разбирается эффективно.

Примечание Неоднозначность алгоритмически неразрешима

СЮК и Эрли отвечают на вопрос, есть ли у слова хотя бы одно дерево разбора, и деревья одного слова пересчитать несложно. Вопрос о грамматике целиком — существует ли слово с двумя разными деревьями — алгоритмически неразрешим¹⁸¹. Доказательство переносит неразрешимость проблемы соответствий Поста — вопроса о том, существует ли непустая последовательность индексов, по которой два списка слов дают одинаковые конкатенации. Поэтому генераторы парсеров проверяют не неоднозначность, а достаточные условия вроде LL(1) и LR(1): конфликт в таблице — повод пересмотреть грамматику, а не приговор языку.

Проверка Разбор: СЮК и LL(1)

1. Почему СЮК работает только с нормальной формой Хомского и что именно лежит в клетке таблицы?
2. Слово принято таблицей СЮК. Как по ней восстановить дерево разбора?
3. Продукции $A \rightarrow \alpha$ и $A \rightarrow \beta$ имеют общие терминалы в FIRST. Что это значит для LL(1)-разбора?

¹⁸¹D. G. Cantor. «On the Ambiguity Problem of Backus Systems». Journal of the ACM, 1962; R. W. Floyd. «On Ambiguity in Phrase Structure Languages». Communications of the ACM, 1962.

§ 24.7 Контекстно-зависимые языки

Язык $a^n b^n c^n$ мы только что исключили из контекстно-свободных. Сам язык прост по описанию: три буквы, и каждой поровну. Значит, у него есть место в иерархии, и это место — следующая ступень.

КС-грамматика не смогла сосчитать два независимых счётчика. Правило в ней заменяет один нетерминал, и безразлично, что вокруг него. Следующий шаг — разрешить правилу оглядываться на окружение: заменять нетерминал A только тогда, когда по бокам от него стоят нужные строки.

ОПРЕДЕЛЕНИЕ 24.7 Контекстно-зависимая грамматика

Контекстно-зависимой грамматикой (КЗ-грамматикой, от англ. context-sensitive grammar) называется четвёрка $G = (V, \Sigma, P, S)$, в которой каждое правило имеет вид

$$\alpha A \beta \rightarrow \alpha \gamma \beta,$$

где $A \in V$ — нетерминал, строки $\alpha, \beta \in (V \cup \Sigma)^*$ образуют контекст, а $\gamma \in (V \cup \Sigma)^+$ непуста.

КЗ-правило $\alpha A \beta \rightarrow \alpha \gamma \beta$ не укорачивает строку. Правая часть длиннее левой на $|\gamma| - 1 \geq 0$ символов. Этим свойством, *монотонностью*, мы воспользуемся ниже, когда будем распознавать язык алгоритмически.

Пример КЗ-грамматика для $\{a^n b^n c^n \mid n \geq 1\}$

Язык, с которым не справилась КС-грамматика, теперь поддаётся:

$$\begin{aligned} S &\rightarrow aSBC \mid aBC, \quad CB \rightarrow BC, \\ aB &\rightarrow ab, \quad bB \rightarrow bb, \quad bC \rightarrow bc, \quad cC \rightarrow cc. \end{aligned}$$

Правила $S \rightarrow aSBC$ и $S \rightarrow aBC$ добавляют по одной тройке aBC , а правило $CB \rightarrow BC$ переставляет соседние B и C , и повторное применение собирает все B перед всеми C . Остальные правила по очереди выполняют замены $B \rightarrow b$ и $C \rightarrow c$.

Вывод слова $a^2 b^2 c^2$:

$$\begin{aligned} S &\rightarrow aSBC \\ &\rightarrow aaBCBC \\ &\rightarrow aaBBCC \\ &\rightarrow aabBCC \\ &\rightarrow aabbCC \\ &\rightarrow aabb cC \\ &\rightarrow aabbcc. \end{aligned}$$

Правила вида $CB \rightarrow BC$ записаны в монотонной форме. Монотонные грамматики (все правила $\alpha \rightarrow \beta$, $|\beta| \geq |\alpha|$) порождают в точности контекстно-зависимые языки. Так что монотонная запись ничего не меняет.

Чем КЗ-грамматика отличается от КС, видно на этом примере. КС-правило $A \rightarrow \gamma$ заменяет нетерминал где угодно. КЗ-правило $\alpha A \beta \rightarrow \alpha \gamma \beta$ — только в подходящем контексте. Всякое КС-правило с непустой правой частью — частный случай КЗ-правила с пустым контекстом, поэтому контекстно-свободные языки — подмножество контекстно-зависимых. Вложение строгое: язык $a^n b^n c^n$ контекстно-зависим, но не контекстно-свободен.

На каком устройстве распознавать контекстно-зависимые языки? Для регулярных языков устройство — конечный автомат, для контекстно-свободных — автомат со стеком. Напрашивается продолжение: взять автомат со стеком и заменить стек на ленту — бесконечную память, где можно читать и писать в любом месте. Вместе с небольшими изменениями конечного управления это и есть машина Тьюринга, которой посвящена следующая глава (25). А если ограничить длину ленты линейной функцией от длины входа, получится *линейно-ограниченный автомат* — модель чуть слабее полной машины Тьюринга, и распознаёт она в точности контекстно-зависимые языки.

Машина Тьюринга ещё впереди, поэтому здесь мы не строим распознаватель для КЗ-языков. Но распознавать их можно и без абстрактной машины — алгоритмически. Свойство, на котором всё держится, — монотонность: КЗ-правила не укорачивают строку. Значит, в выводе слова w ни одна промежуточная строка не длиннее $|w|$. Строк над конечным алфавитом $V \cup \Sigma$ длины не больше $|w|$ — конечное число. Остаётся перебрать все достижимые строки подходящей длины и проверить, найдётся ли среди них w .

АЛГОРИТМ Распознавание слова контекстно-зависимой грамматикой

Вход: КЗ-грамматика G и слово w .

1. Положить $R = \{S\}$.
2. Пока появляются новые строки, применить каждое правило грамматики к каждой строке из R .
3. Добавить результаты длиной не больше $|w|$.
4. Ответ: $w \in R$.

Процедура заведомо завершается: кандидатов конечное число, и каждый проход либо добавляет новую строку, либо останавливает процесс. Так распознавание КЗ-языков оказывается разрешимым, хотя и заметно дороже, чем у контекстно-свободных: перебор растёт экспоненциально с длиной слова.

§ 24.8 Иерархия Хомского

Четыре ступени иерархии Хомского вложены друг в друга: каждый класс — подмножество следующего. Регулярные языки — внутри контекстно-свободных, те — внутри контекстно-зависимых, а вся цепочка — внутри рекурсивно перечислимых. Рис. 104 показывает эту вложенность.

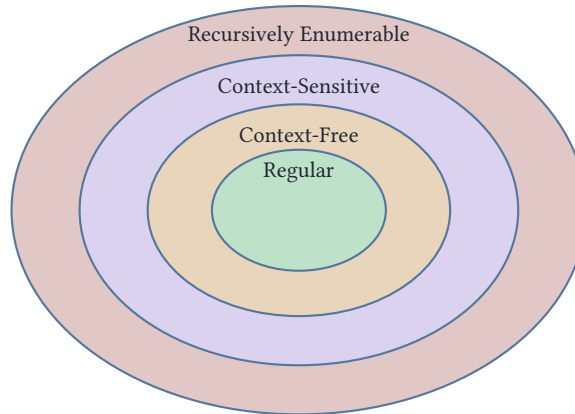


Рис. 104. Иерархия Хомского.

КС-языки — вторая ступень иерархии Хомского:

- **Тип 3 (регулярные):** конечные автоматы. Вести неограниченный счёт они не умеют: память фиксирована заранее, счётчика в ней нет (по модулю фиксированного числа считать можно).
- **Тип 2 (контекстно-свободные):** МП-автоматы. Считать умеют, но одну величину — глубину стека — и только сверху. Язык $a^n b^n c^n$ им уже не по силам: нужны два независимых счётчика.
- **Тип 1 (контекстно-зависимые):** линейно-ограниченные автоматы (раздел выше) — модель с памятью, ограниченной линейной функцией от длины входа.
- **Тип 0 (рекурсивно перечислимые):** машины Тьюринга без ограничений (глава 25).

Подъём по иерархии расширяет класс языков — но каждая ступень платит за это алгоритмическими гарантиями:

- Регулярные языки: разрешимы все стандартные вопросы — эквивалентность, пустота, включение.
- КС-языки: разрешима пустота, но эквивалентность — уже нет.
- Рекурсивно перечислимые языки: даже пустота неразрешима (глава 26).

Граница проходит там, где память перестаёт быть ограниченной заранее: уже для КС-языков эквивалентность неразрешима, хотя пустота ещё разрешима.

§ 24.9 Итоги главы

Контекстно-свободные языки — это языки вложенности: $a^n b^n$ и скобочные структуры становятся распознаваемыми, как только к конечному автомату добавляется

стек. Два описания класса — порождающее (контекстно-свободные грамматики) и распознающее (автоматы с магазинной памятью) — задают один и тот же класс языков, как два описания регулярных языков в главе 23. Правило вроде $S \rightarrow aSb \mid \varepsilon$ заменяет нетерминал на строку без оглядки на контекст, и одного рекурсивного правила хватает, чтобы породить вложенные скобки.

Деревья разбора добавили к описанию языка структуру: одна строка может иметь несколько прочтений, и неоднозначность грамматики становится вопросом о том, какие интерпретации допускает язык. Нормальная форма Хомского привела правила к единому виду, и на этой форме работает алгоритм СҮК, проверяющий принадлежность слова за кубическое время. Класс выдерживает объединение, конкатенацию, звезду, гомоморфизмы и пересечение с регулярным языком, но пересечение двух контекстно-свободных языков может не быть контекстно-свободным.

Лемма о накачке для КС-языков показала предел стека: накачиваются две подстроки одновременно и симметрично, потому что стек растёт и уменьшается согласованно, и этого недостаточно для двух независимых счётчиков вроде $a^n b^n c^n$. Иерархия Хомского расставила классы по памяти: каждая ступень добавляет памяти и теряет алгоритмические гарантии, и разрешимость эквивалентности кончается уже на контекстно-свободной ступени, хотя пустота там ещё разрешима.

Дальнейшие шаги по иерархии меняют саму модель памяти: замена стека лентой с произвольным доступом даёт машину Тьюринга (глава 25), и там алгоритмические гарантии почти исчезают.

Проверка Проверьте себя

1. Что такое контекстно-свободная грамматика и чем её правила отличаются от регулярных?
2. Почему автомат с магазинной памятью эквивалентен КС-грамматике?
3. Что такое дерево разбора и когда грамматика называется неоднозначной?
4. Как лемма о накачке для КС-языков отличается от регулярной версии?
5. Где в иерархии Хомского лежат контекстно-свободные языки?

Что дальше?

Контекстно-свободные языки — вторая ступень иерархии, но и стек не даёт универсальности. В следующей главе (25) мы снимем ограничение памяти полностью: машина Тьюринга с бесконечной лентой — универсальная модель вычислений. Её универсальность — и одновременно предел: в главе 26 мы увидим, что для таких моделей почти все вопросы о языке неразрешимы.

§ 24.10 Упражнения

КС-грамматики: построение.

1. Постройте КС-грамматику для языка строк над $\{a, b\}$. В каждом слове букв a, b поровну.
2. Постройте КС-грамматику для языка палиндромов над алфавитом $\{a, b\}$.
3. Постройте КС-грамматику для языка $\{a^n b^m \mid n \geq m \geq 0\}$. Объясните, почему достаточно одного нетерминала.
4. * Постройте КС-грамматику для языка $\{w c w^R \mid w \in \{a, b\}^*\}$ — слов вида «строка, затем разделитель c , затем та же строка в обратном порядке».

Деревья разбора и неоднозначность.

5. Для грамматики $S \rightarrow aSb \mid \varepsilon$ нарисуйте дерево разбора строки. Строка: $a \ a \ b \ b$.
6. Грамматика $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$ неоднозначна. Приведите слово, имеющее два разных дерева разбора, и нарисуйте оба дерева. Объясните, какое из них соответствует приоритету умножения над сложением.
7. * Постройте КС-грамматику для языка из двух типов скобок — круглых и квадратных — в котором скобки разных типов не пересекаются. То есть $([])$ запрещено. А $(())$ разрешено. Докажите, что ваша грамматика однозначна (или объясните, почему она неоднозначна).

МП-автоматы.

8. Постройте МП-автомат, распознающий язык $\{a^n b^n \mid n \geq 0\}$ по пустому стеку. Выпишите функцию переходов и покажите последовательность конфигураций для слова $a \ a \ b \ b$.
9. Постройте МП-автомат, распознающий язык палиндромов над $\{a, b\}$. Опишите стратегию: как стек помогает сравнить первую половину слова со второй?
10. * Постройте МП-автомат для языка $\{a^i b^j c^k \mid i = j \text{ или } j = k\}$. Объясните, почему автомат обязан быть недетерминированным.

Лемма о накачке для КС-языков.

11. Докажите с помощью леммы о накачке, что язык $\{a^n b^n c^n \mid n \geq 0\}$ не контекстно-свободен.
12. Докажите, что язык $\{a^n b^n a^n b^n \mid n \geq 0\}$ не контекстно-свободен.
13. * Приведите пример языка, который не контекстно-свободен, но удовлетворяет лемме о накачке для КС-языков. Подсказка: по аналогии с главой 23, где был приведён язык, удовлетворяющий регулярной лемме о накачке, но не регулярный.

Нормальная форма Хомского и СЮК.

14. Приведите к нормальной форме Хомского грамматику $S \rightarrow aSb \mid SS \mid \varepsilon$. Укажите, какие правила добавляются при устранении ε -правил, и назовите добавленное правило, которое сразу бесполезно.

15. Заполните таблицу СΥΚ для слова $a a b b b$ по грамматике из примера о приведении к нормальной форме (без стартового S_0). Определите, в каких клетках стоит S , и восстановите по таблице дерево разбора.

Замкнутость.

16. Пусть L — язык всех слов над алфавитом $\{a, b, c\}$, в которых количества букв a , b и c совпадают. Используя замкнутость относительно пересечения с регулярными языками и лемму о накачке, докажите, что L не контекстно-свободен.

Иерархия Хомского.

17. Расположите следующие языки по ступеням иерархии Хомского (регулярный / КС / КЗ / рекурсивно перечислимый): a^*b^* , $a^n b^n$, $a^n b^n c^n$, проблема останова.
18. * Постройте КЗ-грамматику для языка $\{a^n b^n c^n \mid n \geq 1\}$. Либо, пользуясь монотонной грамматикой из главы, выведите слово $a a b b c c$ с указанием каждого шага.
19. * Рассмотрите грамматику с единственным нетерминалом S и правилами $S \rightarrow SS \mid a$. Покажите, что она неоднозначна: найдите слово с двумя разными деревьями разбора. Является ли сам язык, порождаемый этой грамматикой, неоднозначным, то есть не допускающим никакой однозначной грамматики?
20. * Пусть G_1 и G_2 — контекстно-свободные грамматики, порождающие языки L_1 и L_2 . Покажите, что $L_1 \cup L_2$, конкатенация $L_1 L_2$ и звезда Клини L_1^* контекстно-свободны, явно построив для каждого языка грамматику.

ПРОЕКТ Проверка однозначности контекстно-свободных грамматик

Одна строка может иметь несколько деревьев разбора: грамматика арифметических выражений $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$ даёт слову $\text{id}+\text{id}*\text{id}$ два прочтения. Строка $8/4/2$ — две группировки с разными значениями. Компилятор, разобравший программу не тем деревом, молча меняет её смысл. Постройте инструмент, который по любой грамматике либо выдаёт кратчайшее слово с двумя деревьями разбора, либо выписывает сертификат однозначности: ни одно слово длины до N не имеет двух деревьев.

① Разбор всех деревьев

Соберите представление КС-грамматики (нетерминалы, терминалы, правила, стартовый символ) и перебор с мемоизацией, возвращающий все деревья разбора слова, а не первое попавшееся.

② Неоднозначность и приоритет

Перебирайте слова по возрастанию длины и подсчитывайте число деревьев. Первое слово с двумя деревьями — доказательство неоднозначности. Проверьте на грамматике $S \rightarrow SS \mid a$: слово aaa должно получить два дерева — левую и правую группировки. Та же проверка на грамматике арифметических выражений $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$ даёт слову $\text{id}+\text{id}*\text{id}$ два дерева. Грамматика с делением $E \rightarrow E / E \mid 8 \mid 4 \mid 2$ — пример неоднознач-

ности. Убедитесь, что у строки $8/4/2$ находятся две группировки, дающие значения $\frac{8}{\frac{4}{2}} = 1$, $\frac{8}{\frac{4}{2}} = 4$. Объясните, какая группировка соответствует соглашению о левой ассоциативности операторов одного приоритета.

③ **Устранение неоднозначности и язык Дика**

Разделите правила на уровни приоритета ($E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow (E) \mid \text{id}$) и проверьте, что слову $\text{id}+\text{id}*\text{id}$ соответствует единственное дерево. Исчерпывающий поиск до длины N должен выдать сертификат однозначности. Получите такой сертификат для грамматики языка Дика $S \rightarrow (S)S \mid \varepsilon$. Для наивной грамматики $S \rightarrow SS \mid (S) \mid \varepsilon$ найдите минимальное ломающее слово: ожидается $()()()$ с двумя разбиениями $() \mid ()()$, $()() \mid ()$.

④ **Парсер Эрли**

Алгоритм Эрли разбирает произвольную КС-грамматику за $O(n^3)$ без предварительных преобразований. Постройте таблицу множеств $S_0, \dots, S_n$ из пунктов $[A \rightarrow \alpha \cdot \beta, i]$: правило с точкой между α и β плюс начальная позиция i . Применяйте три операции до насыщения: предсказание (за точкой нетерминал), сканирование (за точкой терминал, совпавший с символом слова), завершение (правило закончено — продвиньте точку в пунктах, ожидавших этот нетерминал). Примите слово, если в S_n лежит пункт $[S \rightarrow \alpha \cdot, 0]$ с законченным стартовым правилом. Проверьте приём на однозначной грамматике языка Дика и отказ на слове вне языка, сверив результат с перебором всех деревьев.

⑤ **Устранение левой рекурсии**

Рекурсивный спуск и LL-разбор застревают на правилах вида $A \rightarrow A\alpha$: парсер, не прочитав ни одного символа, снова вызывает себя. Реализуйте проверку левой рекурсии: прямая — это правило $A \rightarrow A\alpha$, косвенная — когда A выводит строку, начинающуюся с самого A , через цепочку нетерминалов. Устраните прямую левую рекурсию: правила $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_k \mid \beta_1 \mid \dots \mid \beta_m$ замените на $A \rightarrow \beta_1 A' \mid \dots \mid \beta_m A'$ и $A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_k A' \mid \varepsilon$. Проверьте, что новая грамматика порождает тот же язык: слова принимаются и отвергаются одинаково до и после устранения.

⑥ **LL(1)-разбор**

LL(1)-парсер выбирает продукцию по одному символу взгляда вперёд, без возврата. Реализуйте FIRST (символы, с которых может начинаться выводимая строка) и FOLLOW (символы, допустимые сразу после нетерминала). Если $A \rightarrow \beta$ и β выводит ε , то $\varepsilon \in \text{FIRST}(A)$. При продукции $A \rightarrow \alpha B \beta$ символы из $\text{FIRST}(\beta)$ без ε входят в $\text{FOLLOW}(B)$, а если β выводит ε — ещё и всё $\text{FOLLOW}(A)$. Маркер конца строки (его обычно записывают как $\$$) входит в FOLLOW стартового символа. Для цепочки $X_1 \dots X_n$ символы берутся из $\text{FIRST}(X_1)$, а если X_1 выводит ε — ещё и из $\text{FIRST}(X_2)$, и так до первого элемента без ε . Постройте предсказывающую таблицу. В клетку (A, a) попадают продукции $A \rightarrow \alpha$, у которых $a \in \text{FIRST}(\alpha)$. Если

$a \in \text{FOLLOW}(A)$ — ещё и все ε -продукции. Грамматика является LL(1), если ни в одной клетке нет двух продукций. Проверьте: грамматика приоритетов с левой рекурсией — не LL(1), после устранения левой рекурсии — LL(1), и слово `id+id*id` разбирается по таблице без возврата. Язык Дика $S \rightarrow (S)S \mid \varepsilon$ окажется LL(1), а грамматика `dangling-else` — нет: на взгляде i конфликтуют две продукции. Выведите последовательность применённых продукций и сверьте её с деревом разбора из переборного этапа.

Итог: инструмент «грамматика на вход — сертификат однозначности до N либо конкретное слово с двумя деревьями», проверенный на приоритете операторов и языке Дика, плюс LL(1)-парсер с FIRST/FOLLOW-множествами и предсказывающей таблицей, который разбирает слово без возврата или предъявляет конфликт продукций.

XXV

Машины Тьюринга

Человек вычисляет карандашом на бумаге. Он смотрит на символ, применяет одно правило из конечного списка, пишет новый символ и переходит к следующему месту. Вся школьная арифметика — умножение столбиком, деление углом — это последовательность таких шагов. Правил немного, бумага дешёвая, и от шага к шагу работающий почти не думает. Если из этой картинки убрать всё случайное — манеру письма, почерк, усталость — останется скелет вычисления.

В предыдущей главе (24) конечный автомат получил стек. Память выросла с номера состояния до глубины вложенности, и это позволило распознать язык $a^n b^n$. Но $a^n b^n c^n$ — два независимых счётчика — стеку уже не по силам. У стека достаточно памяти, однако доступ есть лишь к одной точке. Следующий шаг — снять и это ограничение: лента, на которой можно читать и писать в любом месте.

Машина Тьюринга — это бумага и карандаш, доведённые до предела: бесконечная лента, головка, конечный список правил. На такой машине можно прибавить единицу к двоичному числу, проверить палиндром, сравнить два независимых счётчика. Каждая программа на любом языке сводится к таким же шагам: прочитать из памяти, решить по правилу, записать результат. Машина Тьюринга симулирует любую программу, если программу записать на ленту как данные. Что вычислимо на машине Тьюринга — то вычислимо вообще.

До 1936 года у понятия алгоритма не было строгого определения — только интуиция и надежда. Машина Тьюринга — чистый математический объект, который фиксирует, что значит *вычислять*. Универсальная машина покажет, как одна фиксированная машина выполняет любую другую, — прямой предок *хранимой программы*, на которой стоят современные компьютеры. Тезис Чёрча–Тьюринга делает машину мерой для остальных моделей: лямбда-исчисление (глава 27) докажет свою эквивалентность ей, а теория сложности (глава 30) возьмёт машину как меру времени и памяти.

Математическая машина сильнее любого реального компьютера: лента бесконечна, время никто не ограничивает. Но та же сила подводит к границе: универсальная машина выполняет любую программу, и в следующей главе 26 появятся задачи, которые не решит ни одна программа.

Какие задачи не решит ни одна машина?

ИСТОРИЯ Машина, которой не нужен механизм: Тьюринг и Entscheidungsproblem

История машины начинается с вопроса, который старше неё на восемь лет. Давид Гильберт в 1928 году сформулировал Entscheidungsproblem — проблему разрешения: существует ли алгоритм, определяющий истинность любого математического утверждения, записанного в логике первого порядка? Корни вопроса глубже: Лейбниц мечтал о «calculus ratiocinator», универсальном методе разрешать споры вычислением, — а Гильберт встроил эту мечту в программу полной формализации математики. Сам Гильберт не сомневался в ответе: «В математике нет ignorabimus».

Алан Тьюринг в 1936 году, двадцатичетырёхлетним аспирантом, опубликовал статью «On Computable Numbers, with an Application to the Entscheidungsproblem». Чтобы сформулировать, что такое алгоритм, он описал гипотетическое устройство с бесконечной лентой и конечным набором правил — машину Тьюринга — и показал, что у точного понятия вычисления есть границы. Раз некоторые проблемы не решаются никаким алгоритмом, отрицательное решение Entscheidungsproblem следовало автоматически: нет и алгоритма, определяющего истинность любого математического утверждения, записанного в логике первого порядка.

Алонзо Чёрч пришёл к тому же выводу на месяц с небольшим раньше, в апреле 1936 года, — через λ -исчисление, формальную систему, где вычисление есть последовательная замена символов по правилам редукции. Тьюринг о работе Чёрча не знал — узнав, он добавил в свою статью доказательство эквивалентности λ -исчисления и машин и уехал в Принстон писать диссертацию под руководством Чёрча. Клини дополнил картину: λ -исчисление оказалось эквивалентно и общерекурсивным функциям. Три формализма, построенные независимо, описали один класс функций.

Курт Гёдель встретил формализмы по-разному. К λ -исчислению он отнёсся скептически: оснований считать, что определимость в нём исчерпывает вычислимость, Гёдель не видел. Машину Тьюринга с её лентой и механическими шагами он принял сразу. «Только Тьюрингова работа, — писал Гёдель позднее, — дала точную и несомненно адекватную дефиницию формальной системы».

Так машина, которой не нужен механизм, — чистый математический объект — определила границы вычислимости.

ОБЗОР ГЛАВЫ

В этой главе мы опишем машину: бесконечная лента, головка, конечное управление — и научимся записывать вычисления конфигурациями, мгновенными снимками работы машины. Программирование покажет, как из простых правил складывают-

ся нетривиальные алгоритмы: сравнение двух счётчиков, проверка палиндрома. Важным поворотом станет устойчивость модели: многоленточные и недетерминированные варианты эквивалентны по силе, отличаясь лишь временем работы. Тезис Чёрча–Тьюринга закрепит машину как эталон вычислимости, а универсальная машина покажет, что программа — это данные, которые читает другая программа. В конце останется одна модель, к которой сводится любое вычисление, и именно её мы будем иметь в виду, когда говорим о вычислимом.

После этой главы вы сможете:

1. определять машину Тьюринга и записывать её вычисления конфигурациями;
2. программировать машины для простых задач: сдвиги, сравнение счётчиков, палиндром;
3. объяснять устойчивость модели: многоленточные и недетерминированные варианты сохраняют силу;
4. формулировать тезис Чёрча–Тьюринга и объяснять роль универсальной машины;
5. объяснять, как машина Тьюринга предвосхищает хранимую программу.

§ 25.1 Определение машины Тьюринга

В той статье Тьюринг описал вычислительное устройство, которое мы теперь называем *машиной Тьюринга* (МТ). Устройство предельно просто: бесконечная лента, головка чтения-записи, конечное управление. И этой простоты хватает, чтобы вычислить на МТ всё, что вычислит любой современный компьютер.

ОПРЕДЕЛЕНИЕ 25.1 Машина Тьюринга

Машиной Тьюринга называется семёрка

$$M = (Q, \Gamma, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

с компонентами:

- Q — конечное множество состояний,
- Γ — алфавит ленты, включающий пустой символ $\square$,
- $\Sigma \subseteq \Gamma \setminus \{\square\}$ — входной алфавит,
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ — частичная функция переходов,
- $q_0 \in Q$ — начальное состояние,
- $q_{\text{accept}} \in Q$ — принимающее состояние (остановка с ответом «да»),
- $q_{\text{reject}} \in Q$ — отвергающее состояние (остановка с ответом «нет»), причём $q_{\text{reject}} \neq q_{\text{accept}}$.

Пример МТ в виде семёрки

Простейшая МТ, принимающая все строки, заканчивающиеся на 0:

- $Q = \{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\}$

- $\Gamma = \{0, 1, \square\}$
- $\Sigma = \{0, 1\}$
- δ :
 - $(q_0, 0) \rightarrow (q_1, 0, R)$
 - $(q_0, 1) \rightarrow (q_0, 1, R)$
 - $(q_0, \square) \rightarrow (q_{\text{reject}}, \square, R)$
 - $(q_1, 0) \rightarrow (q_1, 0, R)$
 - $(q_1, 1) \rightarrow (q_0, 1, R)$
 - $(q_1, \square) \rightarrow (q_{\text{accept}}, \square, R)$
- q_0 — начальное состояние.
- $q_{\text{accept}}, q_{\text{reject}}$ — стандартные.

Состояние q_0 означает «последний увиденный символ — 1», а q_1 — «последний увиденный символ — 0». Если лента закончилась в состоянии q_1 , строка заканчивается на 0 — принимаем, а если в состоянии q_0 — отвергаем.

Примечание Почему именно такое определение?

Бесконечная лента — минимальное расширение конечного автомата, дающее универсальность, а не просто новые языки: стек из главы о контекстно-свободных языках уже вывел нас за пределы регулярных. Доступ в любую точку ленты позволяет одной машине симулировать любую другую. Конечное управление сохраняет финитность описания: программа машины записывается конечной строкой. Одна головка достаточна для универсальности, несколько не добавляют выразительности. Альтернативные формализмы эквивалентны МТ:

1. лямбда-исчисление — подстановка вместо ленты.
2. рекурсивные функции — композиция вместо состояний.
3. клеточные автоматы — параллелизм вместо последовательного шага.

Но МТ остаётся самой механистически наглядной: лента, символы, сдвиг — это буквально модель работы человека-вычислителя с карандашом и бумагой.

У машины три части, и каждая отвечает за своё:

- *Лента*: бесконечная в обе стороны, разбита на ячейки. Изначально содержит входную строку w , окружённую пустыми символами.¹⁸²
- *Головка*: в каждый момент указывает на одну ячейку ленты. Читает её символ, записывает новый и сдвигается на одну позицию влево или вправо (L или R). Представьте её как коробочку на колёсиках, бегущую над лентой и заглядывающую в одну ячейку за раз.
- *Управление*: конечный автомат, задающий логику машины через функцию переходов δ . Мысленно держите под рукой книгу инструкций. Если машина в

¹⁸²Бесконечная лента программно моделируется двумя стеками — по одному с каждой стороны головки, пустые ячейки соответствуют отсутствию элемента в стеке.

состоянии q_{23} видит под головкой 0, книга предписывает записать 1, сдвинуться вправо и перейти в состояние q_{359} . δ и есть такая книга.

Все три части собраны на рис. 105.

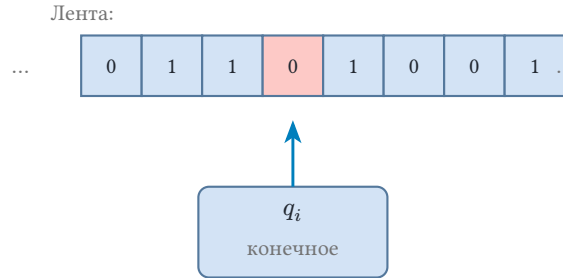


Рис. 105. Схема машины Тьюринга.

Один шаг машины — прочитать символ под головкой, найти по δ подходящий переход, записать новый символ, сдвинуть головку и перейти в новое состояние.

ОПРЕДЕЛЕНИЕ 25.2 Конфигурация

Конфигурацией МТ называется запись:

$$uqv,$$

где:

- uv — содержимое ленты, без начальных и конечных пустых символов; если головка стоит на пустой ячейке, v начинается с $\square$,
- q — текущее состояние,
- головка указывает на первый символ строки v .

Начальная конфигурация на входе w — это запись q_0w , в которой головка стоит на первом символе входа. Конфигурация, содержащая состояние q_{accept} , называется *принимавшей*, а содержащая состояние q_{reject} — *отвергающей*.

Пример Запись конфигураций

Для МТ, принимающей строки, заканчивающиеся на 0, рассмотрим вход 10:

- Начальная конфигурация: q_010 .
- После первого шага: МТ в состоянии q_0 читает 1, пишет 1, сдвигается вправо. Конфигурация: $1q_00$.
- После второго шага: МТ в состоянии q_0 читает 0, переходит в q_1 и сдвигается вправо. Конфигурация: $10q_1\square$.
- МТ в состоянии q_1 видит $\square$ и переходит в q_{accept} — принимающая конфигурация.

Нотация uqv компактна:

- u — часть ленты слева от головки,
- q — текущее состояние,
- v — часть ленты от головки до последнего непустого символа.

У работы машины на данном входе три исхода, и полезно держать их в голове порознь:

- МТ *принимает* вход w , если последовательность конфигураций, начинающаяся с q_0w , достигает принимающей конфигурации за конечное число шагов.
- МТ *отвергает*, если достигает отвергающей конфигурации.
- Третий исход — машина может *зациклиться*: никогда не достигнуть ни принимающего, ни отвергающего состояния.

Примечание Остановка вне финальных состояний

Функция переходов δ частична, поэтому машина может остановиться в состоянии, которое не является ни принимающим, ни отвергающим: для текущей пары состояния и символа правила в таблице нет. Такая остановка по соглашению считается отверганием. Принять машина может только явным попаданием в q_{accept} , поэтому определение распознаваемости достаточно проверять лишь по приёму: всё остальное — уже ответ «нет».

Пример трассы — на рис. 106: машина считает единицы по модулю 2 и принимает слово из двух единиц. В состоянии q_0 она помнит чётное число единиц, в q_1 — нечётное, а головка на рисунке отмечена стрелкой вниз.

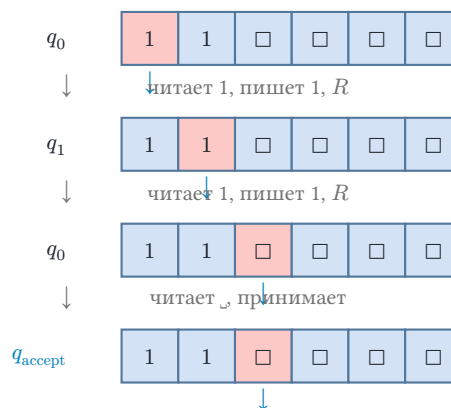


Рис. 106. Трасса вычисления МТ.

ОПРЕДЕЛЕНИЕ 25.3 Распознаваемость

Язык L называется **распознаваемым**, если существует МТ M , которая принимает слово w в точности тогда, когда $w \in L$.

На входах $w \notin L$ машина может отвергнуть или зациклиться. Если машина при этом всегда останавливается (принимая или отвергая), язык называется *разрешимым* (*decidable*). Систематическое изучение разрешимости — предмет главы 26: там появится язык, для которого всегда останавливающуюся машину построить нельзя.

Проверка Конфигурации и распознаваемость

1. Почему в определении распознаваемости машина обязана принять все строки языка, но на строках вне языка ей позволено заикнуться?
2. Что означает запись конфигурации uv ? Как выглядит начальная конфигурация для входа 10 из примера?

§ 25.2 Примеры машин Тьюринга

Пример Первый алгоритм: инверсия битов

Начнём с задачи проще некуда: заменить в двоичной строке все 0 на 1, а все 1 на 0. Головка стартует на первом символе строки.

1. Прочитать символ под головкой.
2. Если это 0 — записать 1. Если 1 — записать 0.
3. Сдвинуться вправо и повторить.
4. На пробеле после строки — принять.

Функция переходов:

- $(q_0, 0) \rightarrow (q_0, 1, R)$,
- $(q_0, 1) \rightarrow (q_0, 0, R)$,
- $(q_0, \square) \rightarrow (q_{\text{accept}}, \square, R)$.

Машина имеет одно состояние, а весь алгоритм — три правила. Это первая программа, которая *вычисляет* функцию — превращает вход в выход.

Пример Удвоение унарного числа: $1^n \rightarrow 1^{2n}$

Построим машину, которая по унарному входу 1^n оставляет на ленте 1^{2n} . Обработываем вход справа налево, каждую единицу помечаем нулём и дописываем в конец две новые единицы. В конце метки стираются, и на ленте остаётся только дописанный блок.

Состояние	Символ	Переход
q_{start}	1	$(q_{\text{start}}, 1, R)$
q_{start}	$\square$	$(q_0, \square, L)$
q_0	1	$(q_1, 0, R)$
q_0	0	$(q_0, 0, L)$
q_0	$\square$	$(q_{\text{clean}}, \square, R)$
q_1	0	$(q_1, 0, R)$
q_1	1	$(q_1, 1, R)$
q_1	$\square$	$(q_2, 1, R)$
q_2	$\square$	$(q_3, 1, L)$
q_3	0	$(q_3, 0, L)$

Состояние	Символ	Переход
q_3	1	$(q_3, 1, L)$
q_3	$\square$	$(q_0, \square, R)$
q_{clean}	0	$(q_{\text{clean}}, \square, R)$
q_{clean}	1	$(q_{\text{clean}}, 1, R)$
q_{clean}	$\square$	$(q_{\text{accept}}, \square, R)$

Тройка в столбце перехода — (q', b, d) : новое состояние, записываемый символ, направление сдвига. Состояние q_{start} один раз доходит до конца входа и переводит головку на последнюю единицу. Состояние q_0 ищет справа налево непомеченную единицу. Состояния q_1, q_2 дописывают две единицы в конец. Состояние q_3 возвращает головку к началу. Состояние q_{clean} стирает метки.

Трасса на входе 11:

Шаг	Конфигурация	Комментарий
0	$q_{\text{start}}11$	Проход вправо до пробела.
1	$1q_01$	Шаг влево: головка на последней единице.
2	$10q_1\square$	Единица помечена, поход к концу.
3	$101q_2\square$	Дописана первая единица.
4	$10q_311$	Дописана вторая, возврат.
5	q_01011	Головка у начала.
6	$0q_1011$	Вторая единица помечена, снова к концу.
7	$00111q_2\square$	Дописана первая из новой пары.
8	$q_0001111$	Возврат к началу.
9	$q_0\square001111$	q_0 ушёл влево от меток.
10	$q_{\text{clean}}001111$	Вход обработан: стирание меток.
11	$q_{\text{accept}}1111$	На ленте 1^4 : результат $2 \cdot 2$.

Пример МТ для языка $\{0^n 1^n \mid n \geq 0\}$

Покажем, что язык $\{0^n 1^n\}$ разрешим — построим МТ, всегда останавливающуюся.

Стратегия: многократно вычёркиваем один 0 слева и одну 1 справа. Если в итоге все символы вычеркнуты — принимаем. Если обнаружено нарушение формата (1 раньше 0, несовпадение количества) — отвергаем.

Состояния:

- q_0 : поиск первого невычеркнутого 0.
- q_1 : движение вправо к концу строки.
- q_2 : поиск первой невычеркнутой 1 справа.
- q_3 : движение влево до границы вычеркнутой области, отмеченной X .

Инвариант: q_0 всегда входит в первый невычеркнутый символ. Если это X — все символы справа уже вычеркнуты, принимаем. Если невычеркнутый 0 — вычёркиваем. Если невычеркнутая 1 — отвергаем.

Переходы на вычеркнутые символы: в следующих проходах машина натывается на X :

- q_1 — пропускает их вправо,
- q_2 — влево,
- q_3 — влево до границы вычеркнутой области, затем шаг вправо в q_0 .

Пример выполнения на строке 0011:

1. q_0 : читает 0, вычёркивает, пишет X , сдвиг вправо, переход в q_1 .
2. q_1 : пропускает оставшийся 0 и две 1, доходит до пробела, сдвиг влево, переход в q_2 .
3. q_2 : читает 1, вычёркивает, пишет X , сдвиг влево, переход в q_3 .
4. q_3 : движется влево мимо 1 и 0, доходит до X , сдвиг вправо, переход в q_0 .
5. Повтор:

$$X011 \rightarrow X01X \rightarrow XX1X \rightarrow XXXX$$

6. q_0 на X : все символы вычеркнуты — принимаем.
7. q_0 на пробеле с самого начала: вход пуст, $n = 0$ — принимаем.

Отклонение устроено так. Если 0 закончились, а 1 остались (строка 0111), q_0 при поиске невычеркнутого 0 натывается на невычеркнутую 1 — отвергаем. Если 1 закончились, а 0 остались (строка 001), q_2 при поиске 1 справа доходит до пробела — отвергаем. Если формат нарушен (1 раньше 0), q_0 увидит 1 в самом начале — отвергаем.

Предыдущий пример был про счёт: машина вычёркивала пары и так сравнивала количества. Палиндром требует другого приёма — сопоставить символы на концах строки. Для этого нужно запомнить увиденное и дойти до другого конца ленты.

Пример МТ для палиндромов

Язык палиндромов над $\{a, b\}$ состоит из строк, читающихся одинаково слева направо и справа налево. Например, слова $abba$, $ba\bar{b}$ и ε .

Стратегия: многократно сравниваем первый и последний символ строки, стираем их и повторяем. Если в итоге все символы стёрты — палиндром. Если на каком-то шаге первый и последний не совпали — не палиндром.

Алгоритм (неформально).

1. Запомнить первый символ (через переход в состояние «видел a » или «видел b »), стереть его.
2. Двигать головку вправо до последнего нестёртого символа.

3. Если последний символ совпадает с запомненным — стереть его, вернуться в начало ленты.
4. Если не совпадает — отвергнуть.
5. Когда все символы стёрты — принять.

Состояния:

- q_0 : чтение первого нестёртого символа строки. Стёртые X пропускаются сдвигом вправо. Если пробел — все символы стёрты, палиндром, принять. Если a — запомнить, стереть, перейти в q_a . Если b — аналогично, перейти в q_b .
- q_a : движение вправо до конца строки (до пробела). Затем шаг влево и пропуск стёртых X до первого нестёртого символа. Если этот символ — a , стереть, перейти в q_{back} . Если b — отвергнуть (несовпадение). Если нестёртых символов не осталось (слева пробел) — средний символ нечётного палиндрома только что стёрт, принять.
- q_b : симметрично, для b .
- q_{back} : движение влево до начала строки (до пробела), пропуская стёртые X , затем шаг вправо и переход в q_0 .

Этот пример показывает стандартный приём программирования на МТ: запоминание символа через состояние, движение к краю строки, сравнение, возврат. В отличие от ДКА, МТ может «видеть» оба конца строки — но за это приходится платить многократными проходами по ленте.

Пример Трасса палиндрома $abba$

Прогоним машину для палиндромов на слове $abba$ и запишем каждый шаг конфигурацией uqv .

Шаг	Конфигурация	Комментарий
0	$q_0 abba$	Головка на первом символе.
1	$Xq_a bba$	a запомнен и стёрт.
2	$Xbq_{\text{back}}bX$	q_a сравнил конец с a , стёр его.
3	$q_0 XbbX$	q_{back} вернул головку к началу.
4	$XXq_b bX$	b запомнен и стёрт.
5	$XXq_{\text{back}}XX$	q_b сравнил конец с b , стёр его.
6	$q_0 XXXX$	Возврат к началу.
7	$XXXXq_0 \square$	Пробел: все символы стёрты.
8	q_{accept}	Слово « $abba$ » — палиндром.

Шаги 1–2 сравнивают первую и последнюю буквы a , шаги 4–5 — буквы b . Каждая пара стирается целиком, и головка после сравнения возвращается к началу ленты.

Пример Отвергание не-палиндрома

Посмотрим, как машина отвергает слово, не являющееся палиндромом, на коротком примере ab :

$$q_0ab \rightarrow Xq_ab \rightarrow Xbq_a\Box \rightarrow Xq_ab.$$

Первый символ a запомнен и стёрт, головка дошла до конца слова и вернулась на последний символ — читает b , не совпадающий с запомненным a . Отвергание происходит при первом же несовпадении: остаток слова машина уже не читает.

§ 25.3 Стандартные приёмы программирования

Примеры выше — ручная сборка из состояний, переходов и движения головки. Как и в обычном программировании, здесь есть повторяющиеся приёмы, из которых собираются машины.

Запоминание через состояние. ДКА помнит ровно то, что закодировано в состоянии, и МТ — то же. Чтобы запомнить текущий символ a , машина переходит в состояние «видел a » и несёт эту информацию, пока она не понадобится. Так состояние работает как переменная, принимающая конечное число значений.

Проход к краю и возврат. Многие алгоритмы требуют дойти до конца строки и вернуться к началу. Для этого служат состояния-двигатели: «идти вправо, пока не пробел», затем шаг влево и переключение. Проход по ленте — это цикл, а условие его завершения — край строки.

Вычёркивание и маркеры. Чтобы не учитывать обработанные символы повторно, их заменяют маркером. В примере с 0^n1^n маркером служит X . Маркер отделяет «уже учтено» от «ещё нет» без дополнительной памяти. Счёт на МТ — это всегда расстановка и пересчёт маркеров.

Композиция машин. Если $M_1 : u \rightarrow v$, $M_2 : v \rightarrow w$, то из них собирается одна машина: состояние останова M_1 отождествляется с начальным состоянием M_2 , остальные состояния перенумеровываются. Так из простых блоков — «найти край», «скопировать», «стереть» — строятся большие программы.

Пример Композиция: удвоение, затем инверсия

Соберём из двух машин третью. Машина M_1 — удвоение унарного числа из примера выше: вход 1^n , выход 1^{2n} . Машина M_2 — инверсия битов: $0 \rightarrow 1$, $1 \rightarrow 0$. Конструкция отождествляет принимающее состояние M_1 с начальным состоянием M_2 . На входе 11 машина M_1 оставляет на ленте 1111 . Машина M_2 превращает его в 0000 . Составная машина вычисляет функцию $1^n \rightarrow 0^{2n}$. Её таблица переходов — объединение таблиц: состояния M_2 добавляются к состояниям M_1 , а переходы, ведущие в q_{accept} , заменяются первыми шагами M_2 .

Эти приёмы — прообраз настоящего процессора: состояние играет роль счётчика команд, маркер — флага, проход по ленте — цикла. Понимая, как машина собирается из примитивов, легче поверить в универсальную машину: программа на ленте — это те же символы, которые конечное управление интерпретирует как инструкции.

§ 25.4 Устойчивость и варианты

Определение машины содержит решения, которые легко принять за произвол. Лента бесконечна в обе стороны, головка сдвигается ровно на одну ячейку и только влево или вправо, за шаг исполняется ровно одно правило. С тем же основанием можно было бы разрешить головке стоять на месте, обрезать ленту с одной стороны или выдать машине несколько лент. Если бы класс вычислимых функций зависел от этих решений, слово «вычислимо» описывало бы одну конкретную конструкцию, и каждую оценку границ вычислимости пришлось бы доказывать заново для каждой версии модели. Эквивалентность вариантов закрывает вопрос: все перечисленные ниже изменения оставляют класс распознаваемых языков прежним. Сила модели определяется задачами, а не деталями конструкции, поэтому о вычислимости можно говорить без указания на версию машины.

УТВЕРЖДЕНИЕ 25.1 Эквивалентность вариантов

Каждая из следующих модификаций сохраняет класс языков, распознаваемых машинами Тьюринга:

1. односторонняя лента, бесконечная только вправо.
2. дополнительный сдвиг S (остаться на месте) наряду с L и R .
3. k независимых лент, каждая со своей головкой.
4. недетерминированный выбор: функция переходов возвращает множество вариантов, и машина принимает, если хотя бы одна ветвь вычисления достигает $q_{\text{акцепт}}$.

Набросок доказательства

Докажем симуляцией: для каждой модификации опишем стандартную одноленточную детерминированную машину, воспроизводящую её работу шаг за шагом.

Односторонняя лента. Правая половина обычной ленты укладывается на верхнюю дорожку, левая — на нижнюю отражённо, так что соседние ячейки исходной ленты остаются соседними, а у левого края стоит граничный маркер. Сдвиг влево за маркер переводит головку на другую дорожку, и вычисление продолжается по ней. Шаг исходной машины стоит постоянного числа шагов симулирующей.

Переход на месте. Каждый переход $(q, a) \rightarrow (q', b, S)$ заменяется парой через свежее состояние r : сначала $(q, a) \rightarrow (r, b, L)$, затем $(r, x) \rightarrow (q', x, R)$ для всех

$x \in \Gamma$. Головка возвращается на прежнюю ячейку, и два шага симулирующей машины дают один шаг исходной.

Несколько лент. Содержимое всех лент укладывается на одну: сначала первые ячейки всех лент, затем вторые, и так далее, а позиции головок отмечены маркерными символами. Один шаг k -ленточной машины — несколько проходов по занятой части общей ленты: прочитать символы под маркерами, найти переходы, записать новые символы, сдвинуть маркеры. Каждый шаг исходной машины удлиняет занятую часть на постоянное число ячеек, поэтому после t шагов она занимает $O(t)$ ячеек, а вся симуляция стоит $O(t^2)$ шагов.

Недетерминизм. Вычисление недетерминированной машины — дерево конфигураций, из каждой вершины которого исходит не более b переходов. Детерминированная машина обходит дерево в ширину, продвигая все ветви поочерёдно на один шаг, поэтому бесконечная ветвь не блокирует проверку остальных. Если принимающая ветвь существует, обход её найдёт, и языки совпадут, хотя число шагов доходит до b^t .

Примечание Дорожки

Дорожка — часть символа ячейки: ячейка хранит кортеж символов, по одному на дорожку. Лента с дорожками — та же машина Тьюринга с алфавитом-произведением $\Gamma_1 \times \dots \times \Gamma_k$, а произведение конечных алфавитов конечно. Выигрыш состоит в проходах: то, что разнесено по соседним ячейкам, одна головка читает за один шаг.

Модификации меняют удобство программирования и время работы, но не силу. Многоленточной машиной удобнее писать алгоритмы с несколькими массивами данных, недетерминированной — описывать задачи, где решение достаточно угадать и проверить.

Лента имитирует и привычные формы памяти, поэтому двухстековые и счётчиковые машины Минского и регистровые машины RAM распознают те же языки. Стек — участок ленты, у которого вершина находится под головкой: положить символ значит записать его и сдвинуться, снять — стереть и вернуться. Два стека укладываются по разные стороны от головки, а счётчик — натуральное число в унарной записи, для которого увеличение означает дописать единицу.

Отдельный случай — альтернативные формализмы: λ -исчисление Чёрча (глава 27) и общерекурсивные функции Гёделя. Это независимые определения понятия «вычисление», а не модификации одной машины: они сходятся к тому же классу функций, что и МТ. То, что ни одна разумная попытка формализовать алгоритм не выходит за границы класса вычислимых функций, — сильное эмпирическое свидетельство в пользу тезиса Чёрча–Тьюринга.

Проверка Устойчивость и варианты

1. Почему детерминированная МТ, симулирующая недетерминированную, обходит дерево конфигураций в ширину, а не в глубину?
2. Во сколько раз одноленточная МТ замедляет симуляцию k -ленточной, и откуда берётся этот множитель?

§ 25.5 Тезис Чёрча–Тьюринга

Машина построена, её варианты сведены друг к другу, и можно спросить, что именно измеряет эта модель. Тезис Чёрча–Тьюринга утверждает, что класс функций, вычислимых на машине Тьюринга, совпадает с классом интуитивно вычислимых функций. Из тезиса следует практическое правило: если задача неразрешима на машине Тьюринга, она неразрешима и в любом другом формализме вычислений.

ОПРЕДЕЛЕНИЕ 25.4 Вычисление функции

Машина Тьюринга **вычисляет** функцию

$$f : \Sigma^* \mapsto \Sigma^*,$$

если, начав работу на слове w , она останавливается с $f(w)$ на ленте в точности тогда, когда $f(w)$ определено.

Вычисляемая функция может быть *частичной*: на входах, где она не определена, машина не останавливается вовсе. Функция, определённая на всех словах, называется *всюду определённой* (*total*).

Замечание Почему тезис — не теорема?

Теорема связывает два формальных объекта, а у тезиса Чёрча–Тьюринга одна из сторон неформальна. «Интуитивно вычислимая функция» — представление о любом конечном процессе, исполнимом по точным правилам, существующее до всякой формальной модели, и класса таких функций в математике не существует. Совпадение формального класса с неформальным понятием нельзя доказать: обе стороны доказательства должны быть утверждениями теории. Статус у тезиса другой: это определение, принятое и проверяемое опытом. Проверка идёт с двух сторон. Независимые формализмы вычисления сходятся к одному классу функций, а процедура, которая вычисляет что-то вне машинного класса, за девяносто лет не предъявлена.

§ 25.6 Универсальная машина Тьюринга

До сих пор каждая машина решала свою задачу: свой язык, свои переходы. Универсальная машина снимает это ограничение — она выполняет любую программу, потому что программа лежит на ленте как данные. Программа и данные впервые становятся неразличимы — идея, которую десятилетие спустя заимствуют первые компьютеры. Сегодня в телевизоре и в стиральной машине стоит одна и та же электронная схема — различает их только программа. Одна машина, много программ — это и есть замысел Тьюринга.

ТЕОРЕМА 25.2 Универсальная МТ

Существует машина Тьюринга U , которая по данной на входе кодировке произвольной МТ, обозначаемой $\langle M \rangle$, и строке w симулирует работу этой машины на w . Если M останавливается на этом входе, то U останавливается с тем же результатом.

Универсальная машина ведёт себя в точности как M :

- принимает, когда принимает M .
- отвергает, когда отвергает M .
- заикливаясь, когда M заикливается.

Набросок доказательства

Докажем построением универсальной машины U , работающей по программе, записанной на ленте.

Кодируем МТ $M = (Q, \Gamma, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ строкой символов фиксированного алфавита. Каждое состояние, символ алфавита и направление получают уникальный код. Переходы кодируются как пятёрки (q, a, q', b, d) , где d — направление.

Универсальная МТ U использует три дорожки на ленте. Первая дорожка хранит закодированную таблицу переходов M — программу, вторая — текущее содержимое ленты симулируемой M , а третья — текущее состояние M и позицию её головки, отмеченную специальным маркером.

Каждый шаг U состоит из трёх действий: прочесть символ под маркером на дорожке 2 и текущее состояние на дорожке 3, найти в таблице переходов (дорожка 1) подходящую пятёрку и обновить дорожки 2 и 3 согласно переходу. Построение U с фиксированным числом состояний — технически громоздкая, но принципиально несложная задача, подобная написанию эмулятора на ассемблере.

Пример Универсальная машина на конкретной паре

Возьмём в качестве M машину инверсии битов. Её таблица кодируется числами и символами:

- состояния получают номера $q_0 = 1, q_{\text{accept}} = 2$;
- символы: $0 = a, 1 = b, \square = c$;
- направления: $L = l, R = r$.

Каждый переход $\delta(q, s) = (q', s', D)$ кодируется пятёркой. Например, переход $\delta(q_0, 0) = (q_0, 1, R)$ даёт пятёрку $(1, a, 1, b, r)$. Вся таблица — строка $\langle M \rangle = (1, a, 1, b, r)(1, b, 1, a, r)(1, c, 2, c, r)$.

На ленте универсальной машины лежит $\langle M \rangle \ 001$: кодировка, разделитель, входное слово. Машина U читает текущее состояние и символ под маркером, находит в таблице подходящую пятёрку и обновляет ленту. Для состояния 1 и символа a подходит первая пятёрка: символ меняется на b , состояние остаётся 1, головка сдвигается вправо. Прогон повторяет шаги самой M .

После трёх инверсий головка читает разделительный пробел, и переход $(1, c, 2, c, r)$ переводит U в состояние 2: машина останавливается. На ленте остаётся 110.

Описание $\langle M \rangle$ в примере выше лежит на ленте как данные, и универсальная машина читает его, чтобы исполнить. Печатасть строки умеет любая машина — а её описание и есть строка, поэтому найдётся машина, печатающая собственное описание. Программа, печатающая собственный текст, называется *квайном* (*quine*) — термин ввёл Дуглас Хофштадтер в честь логика Уилларда ван Ормана Куайна¹⁸³. Лобовое решение — вписать в программу её собственный текст — буксует: вписанная копия короче целого, ведь сама она обёрнута в команду печати, и точного совпадения не получается. Квайн выходит из круга разделением ролей: в данных лежит шаблон — текст программы с одной дырой, — а код печатает шаблон, вклеивая в дыру цитату данных.

Пример Квайн: шаблон и подстановка

Договоримся о двух операциях над строками. `quote(x)` возвращает цитату строки x — ту же строку в кавычках, где перевод строки записан как `\n`. `подставить(s, y)` заменяет в s единственный маркер `<>` на строку y . Вот программа из двух строк:

```
s = "s = <>\nпечать(подставить(s, quote(s)))"
печать(подставить(s, quote(s)))
```

Проверим, что вывод совпадает с текстом. Значение s — шаблон: строка $s = \langle \rangle$, за ней перевод строки и текст `печать(подставить(s, quote(s)))`. Цитата `quote(s)`

¹⁸³D. Hofstadter. «Gödel, Escher, Bach: an Eternal Golden Braid». 1979.

— тот же шаблон в кавычках: `"s = <>\nпечать(подставить(s, quote(s)))"`. Подстановка вклеивает цитату вместо маркера, и на печать выходит следующее:

```
s = "s = <>\nпечать(подставить(s, quote(s)))"
печать(подставить(s, quote(s)))
```

Получились в точности две строки программы. Дырка в шаблоне одна, подстановка однократная, поэтому программа собирает собственный текст в момент печати, а не хранит его.

Замечание Почему квайн — не трюк?

Конструкция с шаблоном и подстановкой — проявление общего факта, известного как теорема Клина о рекурсии. Для всякой вычислимой трансформации t , которая по описанию машины $\langle M \rangle$ выдаёт описание машины $t(\langle M \rangle)$, существует машина P , вычисляющая ту же функцию, что и машина $t(\langle P \rangle)$. Программа эквивалентна собственному образу под трансформацией.

Квайн получается, когда t по $\langle M \rangle$ строит машину, печатающую $\langle M \rangle$: неподвижная точка такой трансформации печатает собственное описание. Тем же механизмом самоссылки держится доказательство неразрешимости проблемы остановки в главе 26 — машина там спрашивает решатель о собственном описании.

За конструкцией стоит идея хранимой программы (stored-program concept): программа — данные, которые другая программа читает и исполняет. Тьюринг описал универсальную машину в 1936 году. Архитектура фон Неймана, где программа и данные размещаются в общей памяти, была сформулирована в 1945 году в «First Draft of a Report on the EDVAC»¹⁸⁴. Точная связь между работами Тьюринга и фон Неймана остаётся предметом дискуссий историков науки¹⁸⁵.



Универсальная машина — это прообраз интерпретатора: программы, которая выполняет другие программы. Компилятор переводит текст в код, интерпретатор читает код и исполняет его на лету, а виртуальная машина — Java, .NET, WebAssembly — симулирует машину внутри машины. Вся эта инженерия повторяет конструкцию Тьюринга: программа — данные, данные читает другая программа, и никакого различия между ними в природе вычисления нет. Различие — инженерное соглашение о том, что сегодня считается программой, а что — данными.

¹⁸⁴M. Davis. «Engines of Logic: Mathematicians and the Origin of the Computer». 2000.

¹⁸⁵B. J. Copeland. «The Essential Turing». 2004.

§ 25.7 Итоги главы

Машина Тьюринга — предел пути, начатого конечным автоматом: память, которая росла от номера состояния до стека, стала лентой с произвольным доступом, и на ней любой алгоритм разворачивается в простые шаги — прочитать, решить по правилу, записать результат. Стек видел лишь одну точку, лента открыла любую ячейку, и этого хватило, чтобы сравнить два счётчика и проверить палиндром. Конечное управление сохраняет финитность описания: программа машины записывается конечной строкой, а вычисление — последовательностью конфигураций.

Модель устойчива: многоленточные и недетерминированные варианты сохраняют вычислительную силу и меняют лишь время работы, так что выбор версии машины — дело удобства, а не выразительности.

Тезис Чёрча–Тьюринга закрепил за машиной роль меры: лямбда-исчисление (глава 27) и рекурсивные функции сходятся к тому же классу, а универсальная машина воплотила идею хранимой программы — программа стала данными, которые другая программа читает и исполняет. С этого момента курс получает единый язык для вопроса о том, что можно вычислить, и теория сложности (глава 30) возьмёт машину как меру времени и памяти.

Универсальность машины ставит вопрос о границе вычислимого: если одна машина выполняет любую программу, то все ли задачи ей по силам. Глава 26 отвечает — есть задачи, которые не решит ни одна машина, и проблема остановки первая среди них.

Проверка Проверьте себя

1. Что утверждает тезис Чёрча–Тьюринга и почему это не теорема?
2. Как устроена машина Тьюринга: лента, головка, конечное управление?
3. Чем конфигурация отличается от состояния и зачем нужна такая запись?
4. Почему многоленточные и недетерминированные МТ не сильнее одноленточной?
5. Что такое универсальная машина и почему программа — это данные?

Что дальше?

Машина Тьюринга — одна модель вычислений, но универсальная: тезис Чёрча–Тьюринга связывает её со всеми разумными формализмами. Но и она решает не всё. В следующей главе (26) мы докажем, что существуют задачи, которые не решит ни одна машина: проблема остановки и другие неразрешимые проблемы. Затем мы увидим альтернативную модель — лямбда-исчисление (глава 27) — а в главе 30 классифицируем разрешимые задачи по требуемым ресурсам.

§ 25.8 Упражнения

Определение МТ и конфигурации.

1. Запишите машину Тьюринга, принимающую язык $\{0^n 1^n \mid n \geq 0\}$, в виде семёрки $(Q, \Gamma, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$. Опишите стратегию и выпишите функцию переходов.
2. Запишите последовательность конфигураций МТ из предыдущего упражнения на входе 0011. Укажите, на каком шаге головка меняет направление.
3. МТ, принимающая строки, заканчивающиеся на 0, была определена в примере главы. Запишите последовательность конфигураций для входов 10, 11. В каком состоянии и на каком символе происходит остановка в каждом случае?
4. * Докажите, что множество всех машин Тьюринга счётно. Какое следствие это имеет для существования неразрешимых языков?

Построение машин Тьюринга.

5. Постройте машину Тьюринга, которая прибавляет 1 к двоичному числу на ленте (младший бит слева). Опишите состояния и переходы.
6. Постройте машину Тьюринга, распознающую язык чётных палиндромов $\{ww^R \mid w \in \{a, b\}^*\}$ — слов, в которых вторая половина равна обращению первой. Опишите стратегию.
7. * Постройте машину Тьюринга, распознающую язык $\{a^n b^n c^n \mid n \geq 0\}$. Опишите стратегию: как проверить равенство трёх счётчиков одной головкой?
8. * Постройте машину Тьюринга, которая копирует входную строку: если в начале на ленте w , то к моменту остановки на ленте w , разделитель (например, #) и снова w .

Варианты, устойчивость и универсальность.

9. Объясните, как одноленточная детерминированная МТ симулирует k -ленточную. Во сколько раз замедляется вычисление и почему это замедление не отменяет эквивалентности моделей?
10. Почему детерминированная МТ, симулирующая недетерминированную, обходит дерево конфигураций в ширину (BFS), а не в глубину (DFS)?
11. Что такое универсальная машина Тьюринга и как она связана с идеей хранимой программы (stored-program concept) из архитектуры фон Неймана?
12. * Тезис Чёрча–Тьюринга утверждает, что класс интуитивно вычислимых функций совпадает с классом функций, вычислимых на МТ. Почему это утверждение не может быть теоремой в обычном смысле? Какие аргументы приводят в его пользу?

ПРОЕКТ Симулятор машин Тьюринга и универсальная машина

Состояние МТ — это счётчик команд, маркер — флаг, проход по ленте — цикл: машина Тьюринга — процессор без железа. Универсальную машину можно собрать из считанных состояний, и задача о минимальном размере машины под конкретную функцию — способ проверить понимание модели. Постройте симулятор и для каждой задачи ниже соберите машину с наименьшим числом состояний.

① Этап 1. Симулятор

Напишите симулятор, читающий семёрку $(Q, \Gamma, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$. Запускайте машины с пределом шагов и печатайте конфигурации uqv .

② Этап 2. Машины для задач

Соберите машины для инкремента двоичного числа, палиндрома и языка $0^n 1^n$ с вычёркиванием маркером. Удвоение унарного числа: $1^n \rightarrow 1^{2n}$.

③ Этап 3. Композиция

Реализуйте композицию машин (отождествление принимающего состояния с начальным следующей) и соберите $w\#w$ из примитивов «найти край», «стереть», «скопировать».

④ Этап 4. Минимизация состояний

Сокращайте число состояний вручную и систематическим поиском. Оценивайте размер машины суммой числа состояний и числа символов ленты. Составьте таблицу результатов.

⑤ Этап 5. Универсальность

Закодируйте свои машины строкой и постройте машину U , которая по кодировке и слову симулирует любую из них — программа как данные.

Итог: симулятор с таблицей минимальных машин, составная машина для $w\#w$ с доказательством корректности и работающая универсальная машина U .

XXVI

Разрешимость и неразрешимость

Программа, которая не останавливается, — знакомая беда. Она крутится и крутится, а когда кончится — неизвестно. Хочется инструмент, который посмотрит на текст программы и заранее скажет: завершится она или заикнется. Такой инструмент стоил бы дорого: он ловил бы ошибки до запуска, и в компиляторе, и в тестовой системе, и во всяком анализаторе чужого кода. Доказано, что его не существует. Инструмент противоречив сам по себе: никакой поиск не помог бы.

В предыдущей главе (25) построена машина Тьюринга — точное определение того, что вообще можно вычислить. Машина Тьюринга умеет всё: симулирует любую программу, проводит любой перебор, дожидается любого результата, если он рано или поздно придёт. Но у этого могущества есть граница, и границу можно доказать. Универсальность не означает всемогущества: есть задачи, которые не решит ни одна машина, какую бы память она ни имела и сколько бы ни работала. Это теорема, и у неё есть доказательство.

Первая такая задача — *проблема остановки*. Даны описание машины и входные данные, и спрашивается, остановится ли машина. Кажется, ответ где-то рядом: можно симулировать машину и смотреть. Но симуляция не отвечает, когда машина заикнулась: ждать можно вечно, а решение нужно принять заранее. Доказательство неразрешимости проблемы остановки — *диагональный аргумент*, тот же приём, которым Кантор доказал несчётность действительных чисел.

Диагонализация — закон выразительности. Как только формальная система достаточно богата, чтобы говорить о себе, в ней неизбежно рождается задача, которую система решить не может. Этот закон демонстрируют и машины Тьюринга, и языки программирования, и формальные системы. Неразрешимость — структурное свойство выразительности.

Курт Гёдель в 1931 году доказал, что любая достаточно выразительная непротиворечивая формальная система неполна. В ней есть утверждения, которые нельзя ни доказать, ни опровергнуть. Алонзо Чёрч в 1936 году показал, что в лямбда-исчислении проблема разрешимости неразрешима. Алан Тьюринг в том же 1936-м, независимо от Чёрча, доказал неразрешимость проблемы остановки. Разные системы и формулировки, но одна диагональная схема. Тьюринг и Чёрч дали понятию вычислимого точное определение, а Гёдель показал, что граница проходит и вокруг доказательств.

Граница вычислимого не умозрительна: она проходит через повседневную инженерию:

- Компилятор не может гарантировать завершение произвольной программы.
- Верификатор не может решить, эквивалентны ли две произвольные программы.
- Статический анализатор обязан выбирать: пропустить ошибку или дать ложное срабатывание.

Система проверки типов, оптимизатор, планировщик — все упираются в ту же границу, когда рассуждают о произвольной программе. Теорема Райса объяснит, почему этот выбор неизбежен. А вопрос о том, что задача разрешима «в принципе», но недостижима «на практике», подведёт к теории сложности (глава 30).

Масштаб границы виден из простого подсчёта: машин Тьюринга счётное число, а языков над любым непустым алфавитом — континуум. Разрешимое и даже распознаваемое — крошечный остров в несчётном море языков. Но остров не пуст: задачи, которые решают реальные программы, живут на нём. Неразрешимость не парализует вычисления — она очерчивает их пределы. Где проходит граница вычислимого и почему её не обойти никакой программой?

ИСТОРИЯ Эмиль Пост и степени неразрешимости

Гильберт поставил вопрос о механическом тесте на истинность, Чёрч и Тьюринг в 1936 году дали отрицательный ответ, Гёдель перенёс границу на доказательства. В этой истории была четвёртая фигура, чья роль стала видна позже: Эмиль Пост (1897–1954). Ещё в начале 1920-х годов, за десятилетие до Гёделя, он пришёл к мысли, что механический перебор формул не обязан давать ответ на каждый вопрос, и к идее формальных систем, порождающих строки по правилам. Довести замысел до теорем Пост не смог: с детства он болел диабетом, взрослую жизнь отравляло тяжёлое психическое расстройство, а записи 1920-х годов остались в архиве. Логика прочли их лишь в 1965 году, когда Мартин Дэвис подготовил посмертную публикацию¹⁸⁶. Историки науки считают эти записи самым ранним предвосхищением неполноты и тезиса Чёрча—Тьюринга.

В 1936 году Пост независимо от Тьюринга опубликовал определение вычислителя: лента на клетки, головка, конечная программа из инструкций чтения, записи и сдвига¹⁸⁷. Его статья вышла на несколько месяцев позже тьюринговской, и совпадение двух независимых формулировок стало весомым аргументом в пользу того, что понятие вычислимости определено верно.

Главный вклад Поста в предмет главы — программа сравнения неразрешимостей. В статье 1944 года он ввёл сведения задач друг к другу и *степени неразрешимости* и спросил, существует ли перечислимое множество, строго промежуточное по степени между разрешимыми множествами и проблемой

¹⁸⁶E. L. Post. «Absolutely Unsolvability Problems and Relatively Undecidable Propositions». В сб. «The Undecidable» (ред. M. Davis), 1965.

¹⁸⁷E. L. Post. «Finite Combinatory Processes, Formulation 1». Journal of Symbolic Logic, 1936.

остановки¹⁸⁸. Задачу Поста решили в 1956–1957 годах Альберт Мучник и Ричард Фридберг независимо друг от друга: промежуточные степени существуют, а для их построения понадобился метод приоритетов — новый стандартный инструмент теории вычислимости. Лестницу кванторных классов, известную как арифметическая иерархия, построили Клини и Мостовский в 1940-е годы, а связывающая её ступени с оракулами теорема Поста превращает лестницу в измерительный инструмент: класс Δ_{n+1} — это в точности задачи, разрешимые с оракулом уровня Σ_n .

Посту принадлежит и классическая неразрешимая задача: проблема соответствий 1946 года — по конечному набору пар строк спросить, существует ли непустая последовательность пар, у которой верхние и нижние строки совпадают. Его порождающие системы, переписывающие строки по правилам, стали прообразом формальных грамматик, когда в 1950-е годы Хомский строил иерархию языков.

ОБЗОР ГЛАВЫ

Точное определение разрешимости отделит её от распознаваемости. Проблема остановки станет первой доказанной неразрешимостью, а диагональный аргумент покажет, почему выразительная система неизбежно упирается в свой предел. Сведения позволяют переносить неразрешимость с задачи на задачу, а теорема Райса одним утверждением закроет все нетривиальные свойства распознаваемых языков. Оракулы и арифметическая иерархия расставят неразрешимые задачи по ступеням сложности, и окажется, что одни неразрешимы строже других. В конце функция занятого бобра покажет, как максимум по конечному множеству машин ускользает от любого алгоритма.

После этой главы вы сможете:

1. различать разрешимые и распознаваемые языки;
2. воспроизводить диагональное доказательство неразрешимости проблемы остановки;
3. переносить неразрешимость сведениями между задачами;
4. применять теорему Райса к свойствам программ;
5. объяснять арифметическую иерархию и функцию занятого бобра.

¹⁸⁸E. L. Post. «Recursively Enumerable Sets of Positive Integers and Their Decision Problems». Bulletin of the American Mathematical Society, 1944.

§ 26.1 Разрешимость

ОПРЕДЕЛЕНИЕ 26.1 Разрешимость (*decidability*)

Язык L называется **разрешимым**, если существует машина Тьюринга M , которая всегда останавливается и принимает вход w тогда и только тогда, когда $w \in L$.

Разрешимость влечёт распознаваемость (см. главу 25). Обратное неверно: примером служит проблема остановки, до которой мы дойдём в следующем разделе.

ЛОВУШКА Вычислимо — не значит практически решаемо

Путать разрешимость с практичностью — ошибка. Разрешимость означает, что алгоритм существует. Практичность — что он успевает за разумное время. Разрешимая задача может иметь лишь астрономически медленный алгоритм. Вычислимость — граница «в принципе», а сложность — граница «на практике». Об этом глава 30.

Пример Распознаваемость vs разрешимость

Язык $\{0^n 1^n \mid n \geq 0\}$ разрешим: мы построили в главе 25 МТ, которая всегда останавливается и принимает в точности строки этого языка.

Проблема остановки HALT распознаваема: универсальная МТ может симулировать M на входе w и принять, если симуляция остановится. Но HALT не разрешима — мы докажем это в следующем разделе. Это пример языка, который распознаваем, но не разрешим.

Язык L разрешим тогда и только тогда, когда распознаваемы и он сам, и его дополнение $\bar{L}$. Действительно, если обе машины принимают свои языки, можно запускать их параллельно, чередуя шаги. Каждый вход принадлежит либо L , либо $\bar{L}$, поэтому одна из машин обязательно остановится.



Заметим, что разрешимость = вычислимость «с двух сторон» — положительной и отрицательной. Та же форма возникает в топологии (clopen = closed + open) и в арифметической иерархии, к которой вернёмся ниже.

Проверка Разрешимость

1. Почему из распознаваемости языка L и его дополнения $\bar{L}$ следует разрешимость L , хотя ни одна из машин не обязана останавливаться?
2. Чем разрешимость отличается от практичности: почему разрешимая задача может не иметь быстрого алгоритма?

§ 26.2 Проблема останковки

Сейчас мы докажем, что нельзя написать программу, которая по тексту другой программы и её входным данным определяет, завершится ли эта программа или заикнется. В любом языке программирования, на любой архитектуре, в любом Turing-полном формализме — такой программы не существует. Сам объект «универсальный анализатор останковки» логически *противоречив*. Доказательство использует диагональный аргумент — тот же метод, которым Кантор доказал несчётность действительных чисел. Вопрос пришёл из логики — Гильберт спросил, есть ли механический тест, который по посылкам и заключению решает, следует ли одно из другого. Тьюринг показал, что такой тест решал бы и проблему останковки, а её не существует. Так неразрешимость логики и неразрешимость останковки оказались одной границей.

Существуют ли задачи, которые ни одна машина Тьюринга не может решить? Ответ — да, и их неизмеримо больше, чем разрешимых. Существует лишь счётное число машин Тьюринга — каждую можно закодировать конечной строкой — тогда как языков над любым непустым алфавитом континуум. Проблема останковки — первая доказанная неразрешимая задача, к ней сводят почти все остальные.

ОПРЕДЕЛЕНИЕ 26.2 Проблема останковки

Проблемой останковки (*halting problem*) называется задача — по паре $(\langle M \rangle, w)$, состоящей из кодировки машины и строки, определить, останавливается ли M на входе w .

Формально она описывается языком

$$\text{HALT} = \{ \langle M \rangle w \mid M \text{ останавливается на входе } w \}.$$

Пример Конкретный запрос к проблеме останковки

Пусть M_{loop} — машина, которая на любом входе немедленно заикливается. В состоянии q_0 она при любом символе переходит в q_0 , не меняя ленту и не сдвигаясь. Тогда $\langle M_{\text{loop}} \rangle 0 \notin \text{HALT}$ (машина заикливается).

Пусть M_{acc} — машина, которая на любом входе немедленно принимает. В состоянии q_0 она при любом символе переходит в q_{accept} . Тогда $\langle M_{\text{acc}} \rangle \varepsilon \in \text{HALT}$ (машина останавливается).

Эти тривиальные случаи разрешимы локально, но универсального алгоритма, работающего для всех пар $\langle M \rangle w$, не существует.

На первый взгляд кажется, что проблему останковки можно решить: симулируем машину M на входе w , и если M остановится — мы это увидим. Но что если M заиклилась? Можно ли определить это за конечное время?

Можно попробовать подождать: секунду, десять секунд, час. Любой конечный предел произволен: заиклившаяся машина может остановиться на секунду позже. Анализатор, который просто ждёт, ведёт себя как симулируемая машина: заиклилась та, заиклился и он. Он лишь воспроизводит проблему. Тьюринг доказал, что нельзя.

ТЕОРЕМА 26.1 Неразрешимость проблемы остановки

Язык HALT неразрешим.

Доказательство

Докажем от противного диагонализацией.

Предположим противное: существует машина Тьюринга H , разрешающая язык «HALT». То есть $H(\langle M \rangle w)$ принимает, если M останавливается на w , и отвергает, если M заикливается на w . При этом H всегда завершает работу. Самоприменение не требует магии: описание $\langle M \rangle$ — такая же строка, как любая другая, а машина умеет читать строки. Для компьютера программа — это текст, и подать машине её собственное описание не страннее, чем передать функции число.

Построим машину D , которая спрашивает машину о ней самой. На входе $\langle M \rangle$ она запускает $H(\langle M \rangle \langle M \rangle)$ — спрашивает, остановится ли M , если ей дать её собственное описание. Если H принимает, то есть M останавливается на $\langle M \rangle$, то D заикливается. Если H отвергает, то есть M заикливается на $\langle M \rangle$, то D останавливается.

Теперь подставим D в саму себя и посмотрим, что делает $D(\langle D \rangle)$. Если D останавливается на $\langle D \rangle$, то H на $(\langle D \rangle, \langle D \rangle)$ должна была отвергнуть. По её ответу D обязана была заиклиться, но она остановилась — противоречие. Если же D заикливается на $\langle D \rangle$, то H на $(\langle D \rangle, \langle D \rangle)$ должна была принять. По её ответу D обязана была остановиться, но она заиклилась — противоречие.

В обоих случаях H ошибается. Следовательно, никакая МТ не может разрешать HALT.

Примечание

- HALT распознаема: универсальная МТ симулирует M на входе w и принимает, если M останавливается. Если M заикливается, симуляция тоже заиклится — поэтому HALT не разрешима.
- $\overline{\text{HALT}}$ — множество пар $(\langle M \rangle, w)$, на которых M заикливается — даже не распознаема. Если бы она была распознаема, «HALT» была бы разрешима: разрешимость = распознаваемость и ко-распознаваемость.

Примечание

Диагональный аргумент Кантора, доказательство Тьюринга, первая теорема Гёделя о неполноте, теорема Тарского о невыразимости истины и парадокс Рассела — все они используют одну и ту же логическую схему. Построить объект (число, машину, формулу, множество), который отличается от каждого элемента предполагаемого перечисления в специально подобранной позиции. Отличие гарантирует, что объект отсутствует в перечислении. Способ построения гарантирует, что он должен был бы принадлежать ему, если бы перечисление было полным. Как только система становится достаточно выразительной, чтобы говорить о себе, диагонализация порождает неразрешимость. Проблема остановки неразрешима потому, что выразительность машин Тьюринга достаточно высока для самоприменимости — а самоприменимость и диагонализация неизбежно порождают парадокс.

Замечание Граница между постулируемым и доказуемым

Тезис Чёрча—Тьюринга утверждает: класс вычислимого совпадает с классом тьюринг-вычислимого. Это определение, фиксирующее границы понятия «алгоритм»: совпадение интуитивной и тьюринг-вычислимости мы принимаем как рабочее, и за 90 лет оно не дало сбой.

Но как только определение принято, внутри очерченной границы всё становится строгим. Проблема остановки неразрешима — и это теорема, доказанная диагональным аргументом. Различим два типа утверждений. «Мы не знаем, разрешима ли эта проблема» — открытый вопрос, пробел в знании. «Мы доказали, что эта проблема неразрешима» — закрытый вопрос, и его результат: доказана невозможность узнать.

Тезис Чёрча—Тьюринга проводит границу между постулируемым и доказуемым в теории вычислимости. Определения выбираются. Теоремы следуют из определений с необходимостью. Проблема остановки принадлежит второй категории.

26.2.1 Неразрешимость логики первого порядка

Вопрос Гильберта — существует ли механический тест, который по формуле логики первого порядка решает, общезначима ли она — называется *проблемой разрешения* (*Entscheidungsproblem*). Ответ отрицательный.

ТЕОРЕМА 26.2 Теорема Чёрча (1936)

Множество общезначимых формул логики первого порядка неразрешимо: нет алгоритма, который по формуле отвечает, общезначима ли она.

Идея доказательства — сведение проблемы остановки. По паре (M, x) — машина Тьюринга и её вход — строится формула логики первого порядка: константы для состояний и клеток, отношения для переходов, аксиомы для шагов.

Например, переход $(q, a) \rightarrow (q', b, R)$ превращается в аксиому: если в момент t машина находится в состоянии q и читает символ a , то в момент $t + 1$ она находится в состоянии q' . Символ под головкой заменён на b , головка сдвинута вправо. Каждый переход даёт одну такую аксиому, и вся таблица переходов превращается в конечный список формул.

Формула утверждает существование конечного прогона: последовательность конфигураций начинается с начальной конфигурации машины M на входе x , согласована с таблицей переходов и заканчивается в останавливающем состоянии. Она выполнима тогда и только тогда, когда существует останавливающийся прогон — поэтому алгоритм проверки выполнимости решал бы проблему остановки. Неразрешимость выполнимости влечёт неразрешимость общезначимости. Формула A общезначима тогда и только тогда, когда невыполнима $\neg A$.

Из полноты (глава 3) следует, что общезначимость *перечислима*: она совпадает с выводимостью, а выводы можно перечислять механически. Итак, логика первого порядка полуразрешима: для общезначимой формулы алгоритм рано или поздно подтвердит это, а для необщезначимой может искать вечно. Именно поэтому практические инструменты — SMT-солверы (глава 15) — работают с разрешимыми фрагментами логики.

§ 26.3 Сведения

Доказав неразрешимость одной задачи, мы можем доказать неразрешимость других задач с помощью *сведений*.

Сведение $A \leq_m B$ означает, что умение решать задачу B автоматически даёт умение решать задачу A .

Пусть у нас есть «чёрный ящик», решающий задачу B , и мы хотим решить задачу A .

1. Преобразуем вход задачи A во вход задачи B : «да» для A переходит в «да» для B , «нет» — в «нет».
2. Подаём преобразованный вход в чёрный ящик и получаем ответ — это и есть ответ для задачи A .

Если такое преобразование возможно, то задача A не труднее задачи B : решив B , мы автоматически решили A .

Теперь контрапозиция — если задача A неразрешима и сводится к B , то B тоже неразрешима. Иначе, имея разрешитель для B , мы построили бы разрешитель для A — противоречие.

Эта логическая цепочка — основной инструмент доказательства неразрешимости.

ОПРЕДЕЛЕНИЕ 26.3 Сведение (*reduction*)

Язык A называется **сводимым** (*many-one reducible*) к языку B , обозначается $A \leq_m B$, если существует вычислимая тотальная функция f , для которой

$$w \in A \Leftrightarrow f(w) \in B.$$

Функция f переносит принадлежность: вопрос « $w \in A$?» превращается в вопрос « $f(w) \in B$?», как на рис. 107. Точка в A переходит в точку в B , а точка вне A — в точку вне B . Поэтому разрешитель для B отвечает сразу на оба вопроса.

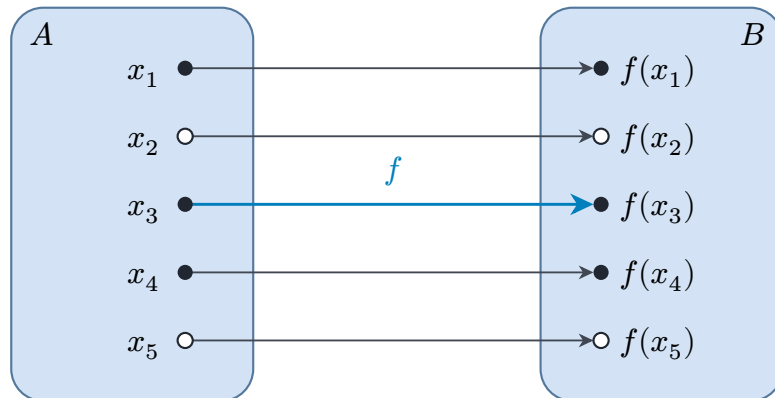


Рис. 107. Сведение $A \leq_m B$ через функцию f .

УТВЕРЖДЕНИЕ 26.3 Использование сведений

- Если $A \leq_m B$ и задача B разрешима, то разрешима и задача A . Вычисляем образ $f(w)$ и подаём его на вход разрешителю B .
- Контрапозитивно: если $A \leq_m B$ и задача A неразрешима, то неразрешима и задача B .

Стандартная стратегия — сводим HALT (известную неразрешимой) к интересующей нас задаче, тем самым доказывая её неразрешимость.

ЛОВУШКА Сведение в неправильную сторону

Доказывать неразрешимость задачи A , сводя её к разрешимой задаче B , — ошибка. Из $A \leq_m B$ при разрешимой B следует разрешимость A , а не неразрешимость. Чтобы показать неразрешимость задачи A , сводят к ней уже *неразрешимую*: например, $\text{HALT} \leq_m A$. Если бы задача A была разрешима, разрешима была бы и «HALT» — противоречие.

Пример Сведение: HALT к проблеме пустоты языка

Проблема пустоты: $E_{\text{TM}} = \{\langle M \rangle \mid L(M) = \emptyset\}$ — множество описаний МТ, язык которых пуст. Покажем, что E_{TM} неразрешима. На рис. 108 показано само све-

дение: по паре $(\langle M \rangle, w)$ строится вспомогательная машина M' , чей язык пуст тогда и только тогда, когда M закичивается на w .

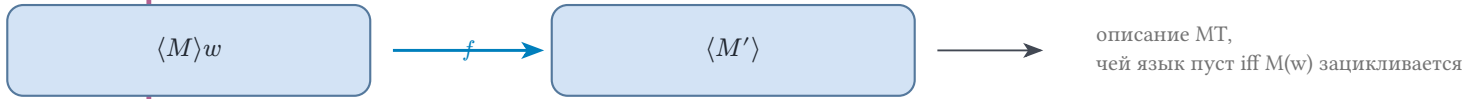


Рис. 108. Сведение HALT к проблеме пустоты.

Сведение: предположим, существует разрешитель E для проблемы пустоты E_{TM} . Построим разрешитель H для языка «HALT»:

На входе $\langle M \rangle w$:

1. Построим вспомогательную машину M' , которая на любом входе x делает следующее:
 - Игнорирует x .
 - Симулирует машину M на входе w .
 - Если симуляция останавливается, M' принимает x .

Таким образом, язык новой машины — либо множество всех строк, либо пустой язык, в зависимости от поведения исходной на входе w :

$$L(M') = \Sigma^* \text{ если } M \text{ останавливается на } w$$

$$L(M') = \emptyset \text{ если } M \text{ закичивается на } w$$

2. Подадим описание $\langle M' \rangle$ на вход проверяющей машине E .
3. Если E принимает, язык новой машины пуст: $L(M') = \emptyset \Rightarrow M$ закичивается на $w \Rightarrow$ отвергаем. Если E отвергает, язык новой машины непуст: $L(M') \neq \emptyset \Rightarrow M$ останавливается на $w \Rightarrow$ принимаем.

Тогда HALT разрешима — противоречие. Следовательно, E_{TM} неразрешима.

Пример Другие неразрешимые задачи

Все следующие задачи неразрешимы (сведения от HALT):

- **Тотальность:** останавливается ли M на каждом входе? Сведение: новая машина M' игнорирует свой вход и симулирует машину M на входе w . Она тотальна тогда и только тогда, когда M останавливается на w .
- **Эквивалентность:** $L(M_1) = L(M_2)$? Даже проверка $L(M) = \emptyset$ (частный случай) неразрешима.
- **Регулярность:** является ли $L(M)$ регулярным языком? Неразрешима.
- **Контекстная свобода:** является ли $L(M)$ контекстно-свободным языком? Неразрешима.

Пример Сведение: HALT к остановке на пустом входе

Язык $\text{HALT}_\varepsilon = \{\langle M \rangle \mid M \text{ останавливается на пустом входе}\}$ состоит из описаний машин, которые останавливаются, получив на ленту пустую строку. Покажем $\text{HALT} \leq_m \text{HALT}_\varepsilon$ и разберём сведение на конкретной паре.

Возьмём машину M_1 , которая читает первый символ входа. Если это 1, машина сразу принимает. Если это 0, машина переходит в состояние, из которого нет выхода, и зацикливается. Поэтому $\langle M_1 \rangle 1 \in \text{HALT}$. А $\langle M_1 \rangle 0 \notin \text{HALT}$.

По паре $(\langle M_1 \rangle, 1)$ построим вспомогательную машину. Назовём её M' . Сначала M' записывает на ленту строку 1, игнорируя собственный вход, и возвращает головку в начало. Затем M' ведёт себя в точности как исходная машина. На пустом входе машина M' сама готовит себе вход 1. Дальше она работает как исходная машина, поэтому останавливается на пустом входе тогда и только тогда, когда исходная машина останавливается на 1. Здесь она останавливается: $\langle M' \rangle \in \text{HALT}_\varepsilon$.

Отрицательный случай устроен так же. По паре $(\langle M_1 \rangle, 0)$ построим машину, которая записывает 0 и симулирует исходную. Она зацикливается на пустом входе и в HALT_ε не входит.

Общая схема: $f(\langle M \rangle, w) = \langle M' \rangle$, где новая машина записывает w и симулирует исходную. Тогда $\langle M \rangle w \in \text{HALT} \Leftrightarrow f(\langle M \rangle, w) \in \text{HALT}_\varepsilon$. Разрешитель для HALT_ε дал бы разрешитель для HALT . Значит, HALT_ε неразрешима.

Мы доказали неразрешимость нескольких задач: проблема остановки, проблема пустоты, проблема тотальности. Каждое доказательство требует своего сведения, своей конструкции вспомогательной машины, своего трюка. Кажется, что для каждой новой задачи придётся изобретать новое сведение — и доказывать неразрешимость будет бесконечной работой.

Но присмотримся к этим задачам: пустота языка, тотальность, регулярность, эквивалентность. Все они спрашивают о свойствах *языка*, который распознаёт машина Тьюринга. Внутреннее устройство машины здесь ни при чём. И все они обладают одним и тем же структурным свойством: ответ зависит только от $L(M)$, и каждое из них нетривиально — не выполнено ни для всех машин, ни для одной. Этих двух условий достаточно для неразрешимости.

Теорема Райса (1953) утверждает, что *любое* нетривиальное свойство языка, распознаваемого машиной Тьюринга, алгоритмически неразрешимо. Теорема Райса *снимает* сам вопрос: отдельные доказательства неразрешимости становятся излишними. После теоремы Райса вопрос сводится к тривиальности свойства P . Если нет — оно неразрешимо автоматически. Весь класс экстенциональных свойств программ закрывается одним утверждением.

§ 26.4 Теорема Райса

ТЕОРЕМА 26.4 Теорема Райса¹⁸⁹

Пусть P — свойство распознаваемых языков (формально: множество описаний МТ), такое что:

- Свойство зависит только от языка: $L(M_1) = L(M_2) \Rightarrow (\langle M_1 \rangle \in P \Leftrightarrow \langle M_2 \rangle \in P)$.
- Свойство нетривиально: найдутся машины M_1, M_2 такие, что $L(M_1) \in P$ и $L(M_2) \notin P$.

Тогда язык $\{\langle M \rangle \mid L(M) \text{ удовлетворяет } P\}$ неразрешим.

Набросок доказательства

Докажем сведением проблемы останова к P .

Сводим «HALT» к P . Без ограничения общности считаем, что пустой язык $\emptyset$ не обладает свойством P . Иначе рассмотрим дополнение $\bar{P}$ и будем рассуждать про него. Так как свойство нетривиально, существует машина A , чей язык $L(A)$ обладает этим свойством.

По данному описанию машины и её входу построим вспомогательную машину M' , которая на любом входе x симулирует машину M на входе w . Если M останавливается на w , машина M' ведёт себя как A на входе x и возвращает результат. Тогда $L(M') = L(A)$, если M останавливается на w , и $L(M') = \emptyset$, если M закикливается на w . Первое множество обладает свойством P , а второе им не обладает.

Таким образом, $\langle M' \rangle \in P$ тогда и только тогда, когда M останавливается на w . Разрешитель для P дал бы разрешитель для «HALT» — противоречие.

Рассмотрение дополнения $\bar{P}$ вместо P законно: разрешимость свойства P равносильна разрешимости дополнения, ведь ответ на один вопрос — отрицание ответа на другой. Поэтому, доказав неразрешимость дополнения, мы доказали её и для исходного свойства.

Пример Неразрешимые свойства по теореме Райса

Все следующие свойства неразрешимы:

- Является ли $L(M)$ пустым, конечным или регулярным?
- Является ли $L(M)$ контекстно-свободным? Разрешимым?
- Содержит ли $L(M)$ конкретную строку? Бесконечен ли $L(M)$?
- Верно ли, что $L(M) = \Sigma^*$?

¹⁸⁹H. G. Rice. «Classes of Recursively Enumerable Sets and Their Decision Problems». Transactions of the AMS, 1953.

Единственные разрешимые свойства — тривиальные: те, которые выполнены для всех распознаваемых языков или ни для одного.

Пример Теорема Райса на конкретном свойстве

Возьмём свойство P : «язык машины содержит пустую строку», то есть машина *принимает* пустой вход.

Свойство экстенционально: $L(M_1) = L(M_2) \Rightarrow (\varepsilon \in L(M_1) \Leftrightarrow \varepsilon \in L(M_2))$, ведь язык — это в точности множество принимаемых входов. Оно нетривиально: машина, принимающая всё, обладает им, а машина, не принимающая ничего, — нет. По теореме Райса язык P неразрешим.

Разберём сведение от HALT явно, по схеме из доказательства. Пустой язык $\emptyset$ свойством P не обладает, тогда как машина A , принимающая любой вход, — им обладает. Построим вспомогательную машину M' . На входе x она симулирует M на входе w , а если симуляция остановилась, принимает x . Тогда:

- Если M останавливается на w , машина M' принимает любой вход, в том числе ε . В этом случае $\langle M' \rangle \in P$.
- Если M закикливается на w , машина M' не принимает ничего. В этом случае $\langle M' \rangle \notin P$.

Итак, $\langle M' \rangle \in P$ тогда и только тогда, когда M останавливается на w . Разрешитель для P решал бы проблему остановки — противоречие.

Здесь нужна точная формулировка: свойство «машина *останавливается* на пустом входе» экстенциональным не является. Две машины с одинаковым языком могут вести себя на отвергаемом входе по-разному: одна останавливается, другая закикливается. К нему теорема Райса напрямую не применима: неразрешимость доказывается отдельным сведением, и такое сведение мы построили выше, в разделе «Сведения».

§ 26.5 За пределами машин Тьюринга

Проблема остановки — лишь начало бесконечной иерархии неразрешимости.

ОПРЕДЕЛЕНИЕ 26.4 МТ с оракулом

Машина Тьюринга с оракулом для языка A — это МТ, имеющая дополнительную ленту-оракул и три специальных состояния: запрос, ответ-да, ответ-нет. Машина может записать строку на ленту-оракул и перейти в состояние запроса. На следующем шаге она переходит в ответ-да или ответ-нет в зависимости от того, принадлежит ли записанная строка языку A . Оракул отвечает мгновенно — это «чёрный ящик», решающий A .

МТ с оракулом формализует относительную вычислимость — вопрос о том, что можно вычислить, имея гипотетическую возможность решать A .

Пример Оракул для проблемы остановки

Имея оракул для «HALT», можно было бы попытаться разрешить проблему тотальности: для данной машины M спросить, останавливается ли M на каждом конкретном входе? Но вопрос «для всех ли x машина M останавливается на x ?» требует бесконечного числа запросов к оракулу: по одному на каждый возможный вход. Оракул для «HALT» даёт возможность решать Σ_1 -вопросы. Задача тотальности находится на уровне Π_2 и остаётся неразрешимой даже с этим оракулом:

$$\forall x. \exists t. M \text{ останавливается на } x \text{ за } t \text{ шагов}$$

Так возникает иерархия: каждый новый оракул открывает одни задачи и оставляет неразрешимыми другие.

Замечание Оракул как внешняя подпрограмма

Оракульная машина Тьюринга — вычислитель, часть работы которого выполняет внешний источник, за один шаг отвечающий на вопрос о принадлежности строки языку A . Это как вызов подпрограммы, которая отвечает мгновенно: остальная часть машины работает обычными переходами. Тезис Чёрча–Тьюринга описывает вычисление *без* такого внешнего источника — что вычислитель делает в одиночку. А что вычислитель может, обращаясь к среде, — отдельный вопрос. На нём и построена иерархия ниже.

Наличие оракула порождает иерархию. Имея оракул для HALT, можно решить некоторые задачи, неразрешимые без оракула, — но появляется новая проблема остановки *для машин с оракулом*, которая неразрешима с этим оракулом. То же повторится с оракулом для этой новой проблемы, и так далее.

Эта иерархия формализуется в виде *арифметической иерархии*.

§ 26.6 Арифметическая иерархия

Теорема Райса утверждает: любое нетривиальное свойство распознаваемых языков неразрешимо. Но значит ли это, что все неразрешимые задачи «одинаково неразрешимы»? Нет: между разрешимостью (Δ_1) и полной неразрешимостью лежит бесконечная лестница — арифметическая иерархия, классифицирующая неразрешимые задачи по логической сложности их определения. Связка лестницы с оракулами и сама идея сравнивать неразрешимости по трудности — программа Поста, разобранный в исторической справке в начале главы.

Идея. Неразрешимость возникает из-за неограниченных кванторов. Проблема остановки: $\exists t$ («существует момент времени, когда МТ остановится»). Её дополнение: $\forall t$ («для любого момента времени МТ не остановится»). Добавление каждого нового квантора поднимает задачу на следующий уровень иерархии.

ОПРЕДЕЛЕНИЕ 26.5 Уровни арифметической иерархии

Пусть $P(x, n_1, \dots, n_k)$ — разрешимый предикат: существует машина, всегда останавливающаяся и правильно его вычисляющая. Определим классы:

- Σ_1 : один квантор существования. Языки этого класса определимы формулой $\exists n.P(x, n)$.

Это в точности класс распознаваемых языков: $w \in L$ означает, что существует конечный принимающий прогон M на w . Существование такого прогона проверяется разрешимым предикатом — то есть задаётся формулой с одним квантором существования. Проблема остановки «HALT» Σ_1 -полна.

- Π_1 : один квантор всеобщности. Языки этого класса определимы формулой $\forall n.P(x, n)$.

Это класс ко-распознаваемых языков (дополнения Σ_1). Дополнение проблемы остановки Π_1 -полно.

- $\Delta_1 = \Sigma_1 \cap \Pi_1$: языки, определимые обеими формулами. Класс разрешимых языков: для задачи из Δ_1 существует машина, всегда останавливающаяся.
- Σ_2 : два квантора, начиная с $\exists$. Языки этого класса определимы формулой $\exists m.\forall n.P(x, m, n)$.

Содержит задачи вроде «МТ останавливается лишь на конечном числе входов» (Σ_2 -полная задача).

- Π_2 : два квантора, начиная с $\forall$. Языки этого класса определимы формулой $\forall m.\exists n.P(x, m, n)$.

Задача тотальности МТ («останавливается ли МТ на каждом входе?») Π_2 -полна.

- Классы $\Sigma_n, \Pi_n, \Delta_n = \Sigma_n \cap \Pi_n, n \geq 1$ определяются аналогично, чередованием кванторов.

Включения строгие при стандартных теоретико-множественных предположениях:

$$\Delta_1 \subset \Sigma_1 \subset \Delta_2 \subset \Sigma_2 \subset \dots$$

(и симметрично для Π_n).

Слово *полна* здесь — в смысле сведений из раздела выше — означает, что полная задача сама принадлежит своему классу, а любая задача этого класса сводится к ней. Разрешив полную задачу, мы решили бы сразу весь класс. К Σ_1 -полной задаче сводится и проблема остановки, поэтому всякая такая задача неразрешима: её разрешитель решал бы и HALT.

УТВЕРЖДЕНИЕ 26.5 Примеры задач на разных уровнях

Уровень	Задача	Формулировка
Δ_1	Разрешимые задачи	Существует МТ, всегда дающая ответ
Σ_1	Проблема остановки	$\exists t$: МТ останавливается за t шагов
Π_1	Дополнение остановки	$\forall t$: МТ не остановилась за t шагов
Σ_2	Конечность языка	МТ принимает лишь конечное число строк
Π_2	Тотальность	$\forall x. \exists t$: МТ останавливается на x за t шагов
Σ_3	Коконечность языка	МТ не принимает лишь конечное число строк

Иерархия расставляет по уровням лишь языки, определимые арифметической формулой, а таких среди всех языков меньшинство.

26.6.1 Почему иерархия?

Каждый уровень добавляет ровно один неограниченный квантор.

- Разрешимая задача (Δ_1) не требует кванторов: ответ вычисляется алгоритмом.
- Σ_1 добавляет квантор существования $\exists$: нужно угадать сертификат, машина угадывает момент остановки.
- Σ_2 добавляет два квантора $\exists\forall$: нужно угадать объект, который работает для всех возможных продолжений.
- Σ_3 добавляет три квантора $\exists\forall\exists$, и так далее.

Этот шаблон — чередование кванторов как мера сложности — универсален. Он проявляется в теории сложности (полиномиальная иерархия), в дескриптивной теории множеств (борелевская иерархия) и в теории вычислимости (степени неразрешимости). Везде один и тот же принцип: каждый дополнительный квантор — это ещё один уровень принципиальной невычислимости, которую нельзя устранить никаким алгоритмическим трюком.

Практическое следствие — если задача Π_2 -полна, то она *строго* труднее проблемы остановки. Оракул для проблемы остановки — это оракул уровня Σ_1 . Π_2 -полную задачу он не решает: для неё нужен оракул на уровень выше.

Проверка Оракулы и арифметическая иерархия

1. Почему добавление оракула для HALT не ставит точку в иерархии: какая проблема остаётся неразрешимой даже для машин с этим оракулом?
2. Почему задача «МТ останавливается на бесконечно многих входах» — уровня Π_2 ? Какой квантор добавляется по сравнению с Σ_1 ?

§ 26.7 Теоремы Гёделя о неполноте

Гипотеза Гольдбаха: каждое чётное число, большее двух, — сумма двух простых. Её проверили до невообразимых границ, и она выглядит истинной, но доказательства нет. Гёдель показал, что в любой достаточно сильной непротиворечивой системе есть истинные утверждения без доказательства, и наши усилия тут ни при чём. Быть может, гипотеза Гольдбаха — одно из них.

Арифметическая иерархия упорядочивает задачи по логической сложности определений. Теоремы Гёделя о неполноте — тот же диагональный аргумент, применённый к формальным системам, а не к алгоритмам. Формальная система фиксирует язык, аксиомы и правила вывода. *Гёделева нумерация* сопоставляет каждой формуле её номер — код из натуральных чисел. Раз коды формул и выводов — числа, теория говорит о себе: о своих формулах, выводах и доказуемости. Требование, чтобы множество аксиом было *рекурсивно аксиоматизируемым*, означает, что алгоритм отличает аксиому от не-аксиомы. Достаточно даже перечислимости аксиом, лишь бы множество теорем было перечислимым (*recursively enumerable*). Без него теорема теряет силу: если взять в аксиомы все истинные предложения арифметики, всякое истинное станет доказуемым, и неполнота исчезнет.

ТЕОРЕМА 26.6 Первая теорема Гёделя о неполноте

Пусть T — непротиворечивая рекурсивно аксиоматизируемая теория, содержащая арифметику. Тогда существует предложение φ , которое T не может ни доказать, ни опровергнуть. В арифметике Пеано такое φ можно выбрать истинным в стандартной модели — в обычном ряду натуральных чисел.

Доказательство идёт той же диагональной схемой, что и неразрешимость проблемы остановки, — только вместо машин кодируются формулы. Как строится предложение, говорящее о себе? Самоссылка — старая знакомая: на одной стороне карточки написано «утверждение на обороте ложно», на обороте — «утверждение на лицевой стороне истинно». Карточка закликивается и не даёт ответа. Гёделево предложение отвечает иначе: истинно, но недоказуемо. Разница в том, что у Гёделя самоотрицание обернуто в числа — и из парадокса становится теоремой.

ЛЕММА 26.7 Диагональная лемма (лемма о неподвижной точке)

Пусть T — теория, в языке которой выразима гёделева нумерация формул. Для любой формулы $\varphi(x)$ с одной свободной переменной найдётся предложение, для которого теория доказывает эквивалентность

$$T \vdash \psi \Leftrightarrow \varphi(\langle \psi \rangle).$$

Набросок доказательства

Докажем построением неподвижной точки через подстановочную функцию.

Код формулы — число, поэтому механическая обработка формул превращается в вычисление над числами. Определим *подстановочную функцию* d : по коду x формулы $\theta(y)$ с одной свободной переменной она вычисляет код формулы $\theta(\langle x \rangle)$, полученной подстановкой числового термина для x вместо y . Подстановка — механическая перестановка символов по синтаксическим правилам, поэтому d вычислима, и в языке T есть формула $D(x, z)$, которая на каждой конкретной паре доказуемо истинна ровно тогда, когда z равен $d(x)$.

Теперь самоприменение. Возьмём $\theta(y) := \exists z. D(y, z) \wedge \varphi(z)$: формула говорит, что выполняется φ на коде результата самоподстановки её аргумента. Пусть ψ — предложение $\theta(\langle \theta \rangle)$. По определению d код ψ равен $d(\langle \theta \rangle)$, поэтому T доказывает $D(\langle \theta \rangle, \langle \psi \rangle)$. Из ψ , то есть из $\exists z. D(\langle \theta \rangle, z) \wedge \varphi(z)$, теория выводит $\varphi(\langle \psi \rangle)$, поскольку доказуемое $D(\langle \theta \rangle, \langle \psi \rangle)$ указывает единственное значение z .

Верно и обратное. Из $\varphi(\langle \psi \rangle)$ и того же $D(\langle \theta \rangle, \langle \psi \rangle)$ выводится $\exists z. D(\langle \theta \rangle, z) \wedge \varphi(z)$, то есть ψ . Значит, T доказывает эквивалентность $\psi \Leftrightarrow \varphi(\langle \psi \rangle)$, и ψ — искомая неподвижная точка.

Лемма — та же диагональ, что и в проблеме остановки: там машина спрашивала решатель о собственном коде, здесь формула говорит о собственном номере.

Проверка конечного вывода — разрешимая процедура, поэтому существование вывода записывается арифметической формулой уровня Σ_1 : «найден код вывода». Её отрицание — формула $P(x)$: она утверждает, что формула с номером x недоказуема в T . Применим диагональную лемму к $P(x)$. Она даёт предложение G , для которого доказуема эквивалентность

$$G \Leftrightarrow P(\langle G \rangle).$$

Но $P(\langle G \rangle)$ — это и есть утверждение « G недоказуемо», так что предложение говорит о себе: « G недоказуемо в T ». Если бы теория доказывала G , она доказывала бы и утверждение « G доказуемо»: когда вывод есть, теория видит его код. Но G утверждает противоположное, и это противоположное утверждение тоже было бы доказано — противоречие. Если же теория не доказывает G , предложение истинно, но недоказуемо. Аргумент показал недоказуемость G . Неопровержимость (недоказуемость $\neg G$) — отдельный шаг, и для произвольной теории его обеспечивает форма Россера, о которой ниже.

Из аргумента следует больше, чем независимость: G истинно — так в арифметике появляются истинные, но недоказуемые утверждения. Доказуемость — свойство уровня Σ_1 : существует код вывода. Поэтому утверждение « G недоказуемо» имеет уровень Π_1 : неполнота даёт истинное недоказуемое утверждение этого уровня.

Напрашивается возражение: добавим G в аксиомы, и оно станет доказуемым. Гёдель отвечает: не поможет — в расширенной системе он построит новое недоказуемое предложение. Приём повторяется бесконечно, как с простыми числами:

добавь недостающее простое в список, и Евклид отыщет следующее пропущенное. Сколько ни расширяй аксиомы, пропуск останется.

Формулировка «ни доказать, ни опровергнуть» в общей теореме — это форма Россера. Гёдель в оригинале гарантировал неопровержимость при более сильном условии ω -непротиворечивости. Россер показал, что достаточно одной непротиворечивости, ценой более хитрого предложения. Для арифметики Пеано шаг закрывается звуковостью: стандартная модель — модель PA , так что ложное в ней из « PA » не выводится.

ТЕОРЕМА 26.8 Вторая теорема Гёделя о неполноте

Пусть T — непротиворечивая рекурсивно аксиоматизируемая теория, содержащая арифметику. Тогда T не доказывает предложение $\text{Con}(T)$, выражающее её непротиворечивость.

Раз доказуемость формализуется внутри теории, формализуется и утверждение «в T нет противоречия»: нет вывода $0 = 1$. Предложение $\text{Con}(T)$ записывается в языке самой T . Рассуждение первой теоремы тоже формализуется в T : для разговора о собственной доказуемости хватает гёделева нумерации. Так первая теорема внутри T принимает вид $\text{Con}(T) \rightarrow G$: из непротиворечивости следует недоказуемость гёделева предложения. Если бы T доказывала $\text{Con}(T)$, она доказывала бы и G , что невозможно. Значит, непротиворечивая теория не может доказать собственную непротиворечивость.

Следствие для арифметики: непротиворечивость PA недоказуема в самой « PA ». Её доказали внешними средствами: Генцен использовал трансфинитную индукцию до ε_0 — то есть шаг за пределы того, что « PA » способна доказать (глава 29).

Граница доказуемого и граница вычислимого — две стороны одной диагонали. Функция занятого бобра покажет третью: максимум по конечному множеству машин оказывается невычислимым.

§ 26.8 Функция занятого бобра

Машина Тьюринга с n состояниями и двухсимвольным алфавитом $\{0, 1\}$ запускается на пустой ленте. Если она останавливается, она записывает на ленте некоторое количество единиц. Сколько единиц можно записать максимально, прежде чем остановиться?

ОПРЕДЕЛЕНИЕ 26.6 Busy Beaver

Функция занятого бобра $\Sigma(n)$ — максимальное число единиц, которое может записать на ленте останавливающаяся машина Тьюринга с n состояниями, бу-

дучи запущенной на пустой ленте. Машины имеют n состояний плюс состояние останова и двухсимвольный алфавит $\{0, 1\}$: символы 0 и 1.

Функция $S(n)$ — максимум числа шагов, которое может совершить останавливающаяся машина с n состояниями на пустой ленте.

Известные значения:

- $\Sigma(1) = 1$
- $\Sigma(2) = 4$
- $\Sigma(3) = 6$
- $\Sigma(4) = 13$
- $\Sigma(5) = 4098$: нижнюю оценку нашли Buntrock и Marxen в 1990, равенство доказано в 2020-е годы.
- $\Sigma(6)$: нижняя оценка превышает башню из 15 экспонент. Это число невозможно записать в обычной нотации — оно больше числа атомов в наблюдаемой Вселенной.

Пример Занятые бобры для одного и двух состояний

При $n = 1$ максимум достигается машиной

$$A : 0 \rightarrow 1, R, \text{halt}$$

В состоянии A на пустой клетке машина пишет единицу, сдвигается вправо и останавливается. На ленте одна единица: $\Sigma(1) = 1$. Больше одной единицы записать нечем: у машины одно состояние, и первый же переход либо ведёт в останов, либо закидывает машину навсегда.

При $n = 2$ рекорд держит машина Радо:

$$A : 0 \rightarrow 1, R, B$$

$$A : 1 \rightarrow 1, L, B$$

$$B : 0 \rightarrow 1, L, A$$

$$B : 1 \rightarrow 1, R, \text{halt}$$

Проследим её работу на пустой ленте.

1. $(A, 0) \rightarrow (1, R, B)$: первая единица на ленте.
2. $(B, 0) \rightarrow (1, L, A)$: вторая единица.
3. $(A, 1) \rightarrow (1, L, B)$: головка уходит влево от записанного блока.
4. $(B, 0) \rightarrow (1, L, A)$: третья единица.
5. $(A, 0) \rightarrow (1, R, B)$: четвёртая единица.
6. $(B, 1) \rightarrow (1, R, \text{halt})$: головка вернулась на блок, машина останавливается.

Итог — четыре единицы за шесть шагов. Перебор всех $12^4 = 20736$ таблиц переходов двухсостояниеовых машин показывает: ни одна останавливающаяся

машина с двумя состояниями не записывает больше четырёх единиц. Поэтому $\Sigma(2) = 4, S(2) = 6$.

Функция $\Sigma(n)$ растёт быстрее любой вычислимой функции. Причина в том, что она кодирует проблему остановки. Если бы мы умели вычислять $S(n)$ — максимум числа шагов среди останавливающихся машин с n состояниями — мы бы решили проблему остановки для таких машин. Машина, не остановившаяся за это число шагов, не остановится никогда. Знание $S(n)$ дало бы и $\Sigma(n)$ — прогнав каждую n -состояниеую машину на $S(n)$ шагов, возьмём максимум записанных единиц. Обе функции невычислимы — это доказывает теорема ниже.

ТЕОРЕМА 26.9 Невычислимость занятого бобра

Функции $\Sigma(n), S(n)$ невычислимы. Они асимптотически растут быстрее любой вычислимой функции: для любой вычислимой функции f найдётся порог n_0 , после которого $\Sigma(n) > f(n)$ для всех $n \geq n_0$.

Набросок доказательства

Докажем построением, что Σ растёт быстрее любой вычислимой функции, а затем от противного, что Σ невычислима. Для S рассуждение то же.

Сначала докажем рост. Пусть $f : \mathbb{N} \rightarrow \mathbb{N}$ — произвольная вычислимая функция. Машина F , которая её вычисляет, имеет c состояний. По унарному входу 1^n она записывает $f(n)$ единиц и дописывает ещё одну. Для каждого n соберём машину, работающую на пустой ленте: n начальных состояний записывают n единиц — унарный код числа n — после чего машина F превращает их в $f(n) + 1$ единиц и останавливается. Итого, $n + c$ состояний, $f(n) + 1$ единиц, значит

$$\Sigma(n + c) \geq f(n) + 1.$$

Сдвиг c устраним: записать n единиц можно за $O(\log n)$ состояний — пишем число n в двоичном виде и переводим его в унарный код постоянным числом состояний. Поэтому, $O(\log n) + c$ состояний, $f(n) + 1$ единиц, то есть

$$\Sigma(O(\log n) + c) \geq f(n) + 1.$$

При достаточно больших n выполнено $O(\log n) + c < n$. Функция Σ монотонна, поэтому $\Sigma(n) \geq f(n) + 1 > f(n)$: рост быстрее любой вычислимой функции.

Теперь от противного. Если бы Σ была вычислимой, вычислима была бы и функция $g(n) = \Sigma(2n)$. Эту функцию вычисляет машина с c_g состояниями. По неравенству $\Sigma(n + c_g) \geq g(n) + 1 = \Sigma(2n) + 1$. Выполнено $n \geq c_g \Rightarrow n + c_g \leq 2n$, а функция монотонна. Поэтому $\Sigma(n + c_g) \leq \Sigma(2n)$. Получаем $\Sigma(2n) \geq \Sigma(2n) + 1$ — противоречие.

Для S рассуждение то же: машина, которая после вычисления $f(n)$ делает ещё $f(n)$ шагов и останавливается, имеет $O(\log n) + c$ состояний и совершает больше $f(n)$ шагов. По монотонности $S(n) > f(n)$ для всех достаточно больших n .

Функция занятого бобра¹⁹⁰ — точная граница вычислимости: она мажорирует любую вычислимую функцию.



$\Sigma(n)$ — максимум по конечному множеству. Останавливающихся машин с n состояниями конечное число. Для любого фиксированного n это конкретное целое число. Но чтобы найти его перебором, нужно для каждой n -состояниеовой машины ответить на вопрос «останавливается ли она» — частный случай проблемы остановки. Так максимум конечного набора целых чисел остаётся за пределами вычислимого.

За ней — область, куда не может зайти ни один алгоритм. Конкретные значения для малых n известны. Поведение для $n \geq 6$ — открытое поле, где теория вычислимости встречается с комбинаторной оптимизацией¹⁹¹.

§ 26.9 * Теорема Райса и статический анализ программ

Для программиста теорема Райса формулирует дилемму статического анализа. Любой инструмент, проверяющий нетривиальное поведенческое свойство программ, вынужден выбирать:

- *Корректность (soundness)*: находить все потенциальные ошибки, допуская ложные срабатывания.
- *Полнота (completeness)*: сообщать только о реальных ошибках, рискуя пропустить некоторые.

Одновременно корректный и полный анализатор для нетривиального семантического свойства эквивалентен алгоритму, решающему неразрешимую задачу. Такой анализатор невозможен.

Инструменты спасает оговорка: теорема говорит о свойствах языка, распознаваемого машиной, а не о свойствах её текста. «Программа содержит разыменованное нулевого указателя» — свойство *интенциональное*, и теорема Райса к нему напрямую не применяется. Тем не менее анализатор аппроксимирует поведение программы, и компромисс между корректностью и полнотой остаётся.

Практические инструменты выбирают сторону осознанно. Rust borrow checker корректен: если он принимает программу, ошибок работы с памятью нет. Но он кон-

¹⁹⁰T. Radó. «On Non-Computable Functions». Bell System Technical Journal, 1962.

¹⁹¹A. H. Brady. «The Busy Beaver Game and the Meaning of Life». В сб. «The Universal Turing Machine: A Half-Century Survey» (ред. R. Herken), 1988.

сервативен: существуют безопасные программы, которые borrow checker отвергает. TypeScript некорректен (unsound): типизированный код может упасть в рантайме. Ложных срабатываний на корректном JavaScript он при этом не даёт (полнота сохранена). Coverity и Infer находят реальные баги, не гарантируя нахождение всех.

Это инженерные компромиссы, неизбежность которых доказана математически.

Проблема остановки — частный случай общего принципа. Теорема Райса обобщает её на любое нетривиальное поведенческое свойство программ.

§ 26.10 * Десятая проблема Гильберта

Уравнения в целых числах ведут себя разнородно: $x^2 + y^2 = z^2$ решается бесконечным числом способов — это пифагоровы тройки, а решения уравнения Пелля $x^2 - 2y^2 = 1$ выписываются по одной формуле, через степени числа $3 + 2\sqrt{2}$. У уравнения $x^n + y^n = z^n$ при $n \geq 3$ решений в натуральных числах нет вовсе — на доказательство этого факта, великой теоремы Ферма, ушли три с половиной столетия. Напрашивается общий вопрос: можно ли по виду уравнения заранее определить, есть ли у него решение?

ОПРЕДЕЛЕНИЕ 26.7 Диофантово уравнение

Уравнение $p(x_1, \dots, x_n) = 0$, где p — многочлен с целыми коэффициентами, называется **диофантовым**.

Диофант Александрийский решал такие уравнения ещё в III веке, а в повестку математики их поставил Гильберт в списке из двадцати трёх проблем 1900 года. Десятая проблема просила указать способ, определяющий за конечное число операций, имеет ли уравнение решение в целых числах. Точного понятия «способ» тогда не было: машина Тьюринга появилась тридцать шесть лет спустя и превратила проблему в корректный вопрос о вычислителе.

ОПРЕДЕЛЕНИЕ 26.8 Диофантово множество

Множество $A \subseteq \mathbb{N}^m$ называется **диофантовым**, если существует многочлен p с целыми коэффициентами, для которого $a \in A \Leftrightarrow \exists y_1, \dots, y_k \in \mathbb{N}. p(a_1, \dots, a_m, y_1, \dots, y_k) = 0$.

Всякое диофантово множество перечислимо: алгоритм перебирает кортежи значений переменных и печатает те $a_1, \dots, a_m$, при которых уравнение обратилось в ноль. Перечислимо и множество пар (t, w) , на которых машина t останавливается на входе w (раздел 26.2). Десятая проблема превратилась в вопрос, всякое ли перечислимое множество чисел задаётся полиномиальным условием.

Мартин Дэвис, Хилари Патнэм и Джулия Робинсон свели конструкцию к одному

кирпичу¹⁹²: если полиномиально выражается возведение в степень $b = a^n$, то диофантово всякое перечислимое множество. Кирпич нашёл в 1970 году двадцатидвухлетний Юрий Матиясевич, и итог носит имя всех четверых — теорема MRDP.

ТЕОРЕМА 26.10 Теорема MRDP¹⁹³

Множество $A \subseteq \mathbb{N}^k$ диофантово тогда и только тогда, когда оно перечислимо.

Доказательство теоремы — за пределами курса: полное изложение тянет на отдельную книгу, но его ход стоит показать в одном абзаце. Программа машины — рекуррентные правила: следующая конфигурация получается из предыдущей локальной перестановкой символов. Остановка означает существование конечной последовательности конфигураций, соседи которой связаны простыми условиями, — перебор, оформленный кванторами существования. Препятствие в масштабе: за t шагов машина успевает записать числа порядка 2^t , и закодировать такой прогон многочленом в лоб не выходит. Возведение в степень — ровно тот инструмент, которого алгебре не хватает. Недостающее отношение Матиясевич извлёк из чисел Фибоначчи: они задаются рекуррентностью $F_{n+2} = F_{n+1} + F_n$ и растут экспоненциально, как φ^n при $\varphi = \frac{1+\sqrt{5}}{2}$. Критерий же принадлежности полиномиален: b — число Фибоначчи тогда и только тогда, когда $5b^2 + 4$ или $5b^2 - 4$ — точный квадрат. Экспоненциальный рост спрятан в решения уравнения Пелля: перебор, который машина ведёт шагами, многочлен ведёт лишними переменными. Заодно снимается вопрос о форме решений: замена каждой переменной x_i разностью $u_i - v_i$ сводит целочисленные решения уравнения $p = 0$ к натуральным решениям уравнения $p(u_1 - v_1, \dots, u_n - v_n) = 0$. Натуральные решения целочисленны и так, поэтому обе формулировки десятой проблемы равносильны.

СЛЕДСТВИЕ 26.11 Неразрешимость десятой проблемы

Не существует алгоритма, который по любому диофантову уравнению определяет, имеет ли оно решение в целых числах.

Доказательство

Докажем сведением проблемы остановки к десятой проблеме.

Множество пар (m, w) , на которых машина с номером m останавливается, перечислимо (см. выше). По теореме MRDP оно диофантово: существует многочлен p с целыми коэффициентами, для которого разрешимость уравнения $p(m, w, y_1, \dots, y_k) = 0$ равносильна остановке машины m на входе w .

Пусть существовал бы разрешитель D , отвечающий на вопрос о разрешимости любого диофантова уравнения. На входе (m, w) подставим числа m и w в p и

¹⁹²M. Davis, H. Putnam, J. Robinson. «The Decision Problem for Exponential Diophantine Equations». Annals of Mathematics, 1961.

¹⁹³Ю. В. Матиясевич. «Диофантовость перечислимых множеств». Доклады АН СССР, 1970.

спросим D об уравнении $p(m, w, y_1, \dots, y_k) = 0$. Его ответ совпадает с ответом на вопрос, останавливается ли машина m на w . Разрешитель десятой проблемы оказался бы разрешителем проблемы остановки — противоречие.

Контраст с арифметикой Пресбургера обнажает роль умножения: там его нет, и теория разрешима, хотя SMT-солверы расплачиваются двойной экспонентой от длины формулы (глава 15). Без умножения нечего кодировать — ни последовательностей, ни машин, ни следов вычисления. С умножением разрешимость рушится на самой нижней ступени: десятая проблема — экзистенциальная теория арифметики, один квантор существования поверх разрешимой проверки.

В терминах арифметической иерархии теорема MRDP означает: диофантовы множества — это в точности класс Σ_1 , а полиномиальное уравнение — форма одного квантора существования. Гильберт спрашивал о способе, Тьюринг дал этому слову точный смысл, а диагональный аргумент и сведение показали, что способа не существует. Вопрос, заданный за тридцать шесть лет до понятия алгоритма, получил ответ — отрицательный и окончательный.

§ 26.11 * Арифметика Пресбургера

Выведем из арифметики умножение и посмотрим, кто останется в строю. Язык сохраняет натуральные числа, ноль, сложение и порядок. Терм в нём — линейная сумма вида $2x + 3$, перемножать переменные нельзя. Порядок сводится к сложению: $x \leq y$ равносильно $\exists z. y = x + z$, так что единственный арифметический герой этой сцены — сумма. Теорию этого языка называют арифметикой Пресбургера, и в главе 15 она появилась как разрешимый, но дорогой случай. Здесь мы разберём, откуда берутся оба свойства.

Скупость словаря обманчива: начальный запас теории чисел помещается в этот язык. Атом $2 \mid x$ — сокращение для $\exists y. y + y = x$, и точно так же $d \mid x$ раскладывается в $\exists y. x = y + \dots + y$, где слагаемых y ровно d . Рядом встают сравнения по модулю константы: $x \equiv r \pmod{d}$ означает $d \mid (x - r)$. Сокращения законны: каждое переводится в чистую формулу языка, поэтому разрешимость от них не зависит. А вот о квадратах, простых числах и степенях язык сказать ничего не может — скоро станет ясно, почему.

Ответ даёт элиминация кванторов: разрешимость арифметики сложения Пресбургера доказал ещё в 1929 году, но общепринятой стала процедура Купера, выписывающая кванторы из формулы по одному за виток¹⁹⁴. Формулу приводят к дизъюнктивной нормальной форме, и $\exists$ разносится по дизъюнктам — достаточно научиться выкидывать один $\exists y$ из конъюнкции литералов. Литералы о y бывают четырёх сортов: нижние границы $t \leq y$, верхние границы $y \leq u$, равенства $y = t$ и делимости $d \mid$

¹⁹⁴D. C. Cooper. «Theorem Proving in Arithmetic without Multiplication». Machine Intelligence, 1972.

y , где t и u не содержат y . Строгие сравнения $t < y$ предварительно превращаются в $t + 1 \leq y$: между соседними целыми числами никого нет. Равенство исчезает первым: подставив t вместо y во все остальные литералы, мы выбрасываем переменную вместе с уравнением. Если при y стоит коэффициент, как в $2y = x$, подстановка $y = \frac{x}{2}$ может увести из целых чисел. Тогда равенство распадается на пару: условие делимости $2 \mid x$ и подстановка $y = \frac{x}{2}$, а все остальные литералы умножаются на 2, чтобы дробей не осталось. Из $\exists y. 2y = x \wedge y < 3$ так получается $2 \mid x \wedge x < 6$: неравенство умножилось на положительную двойку без искажений.

Если равенств нет, вступает канонический свидетель. Пусть наибольшая нижняя граница равна L , а шаг делимости равен d . Без делимостей берём $d = 1$, а без нижних границ — $L = 0$. Наименьшее значение y , проходящее все нижние границы и делимость, отстоит от L меньше чем на d : иначе $y - d$ тоже проходит все проверки, будучи меньше. Кандидатов, значит, ровно d штук: $L, L + 1, \dots, L + d - 1$, и квантор разворачивается в конечную дизъюнкцию подстановок. Верхние границы и свободная от y часть формулы переходят в проверку каждого кандидата.

Пройдём один цикл целиком и проверим его вручную. Спросим, для каких x среди чисел от x до $x + 3$ есть чётное:

$$\exists y. x \leq y \wedge y \leq x + 3 \wedge 2 \mid y$$

Нижняя граница одна — $x \leq y$, верхняя — $y \leq x + 3$, делимость задаёт шаг $d = 2$. Равенств нет, и канонический свидетель — наименьшее чётное число, не меньше x . Он отстоит от x меньше чем на 2: иначе $y - 2$ тоже чётно и тоже не меньше x . Кандидатов двое: $y = x$ и $y = x + 1$. Подстановка первого даёт тривиальную границу $x \leq x + 3$ и остаток $2 \mid x$. Подстановка второго даёт $x + 1 \leq x + 3$ и остаток $2 \mid x + 1$. Итог элиминации: исходная формула равносильна $2 \mid x \vee 2 \mid x + 1$. Эта формула истинна при каждом x : чётность достаётся либо самому x , либо его соседу. Граница $x + 3$ ни на что не повлияла — оба кандидата проходят её автоматически, и ручная проверка согласна: среди четырёх последовательных чисел всегда есть чётное.

Метод не привязан к примеру: цикл применим к любому конъюнкту, и на каждом витке число кванторов строго убывает. Отсюда и разрешимость.

ТЕОРЕМА 26.12 Разрешимость арифметики Пресбургера

Существует алгоритм, который по любой замкнутой формуле арифметики Пресбургера определяет, истинна ли она на натуральных числах.

Набросок доказательства

Докажем элиминацией кванторов.

Каждый цикл Купера сохраняет равносильность: равенства подставляются, а оставшиеся свидетели перебираются конечной дизъюнкцией, как в примере с чётным числом в отрезке. Если ни один кандидат не проходит проверку,

дизъюнкция вырождается в ложь — отрицательный ответ получается тем же механизмом, что и положительный. Подстановка линейного терма не выводит из фрагмента: термы остаются линейными суммами, а константы в делимостях перемножаются, оставаясь константами. Универсальный квантор сводится к существованию двойным отрицанием: $\forall y. \varphi$ равносильно $\neg \exists y. \neg \varphi$.

Кванторов после каждого витка на один меньше, поэтому индукция заканчивает работу на бескванторной формуле. Для замкнутой формулы все термы — конкретные числа: каждый атом вычисляется, и ответ даёт таблица истинности.

Цена разрешимости видна в самой процедуре: виток с d кандидатами разворачивает квантор в d -членную дизъюнкцию, следующий виток разветвляет каждый её член, и после n кванторов размер промежуточных формул растёт степенной башней. Фишер и Рабин превратили наблюдение в точную оценку: всякий решающий алгоритм на некоторых формулах длины n делает не меньше $2^{2^{cn}}$ шагов для некоторой константы $c > 0$ ¹⁹⁵. Стена общая для всех процедур — двойная экспонента принадлежит самой теории, поэтому полной элиминацией кванторов промышленные инструменты не занимаются.

Контраст с предыдущим разделом ложится на ту же страницу и расставляет всё по местам. Кванторы ни при чём: арифметика Пресбургера допускает любые чередования $\forall$ и $\exists$ и остаётся разрешимой. Всю работу делает умножение: с ним язык кодирует пары и последовательности, и теорема MRDP показывает, что одного экзистенциального квантора над диофантовым уравнением хватает, чтобы выразить машину Тьюринга. Без умножения выразимый мир периодичен: всякое определяемое множество натуральных чисел за некоторым порогом склеивается из арифметических прогрессий. Квадраты, простые числа, степени двойки не периодичны даже в этом слабом смысле, и язык без умножения их не выговаривает. На языке арифметической иерархии контраст предельный: без умножения разрешима вся башня чередований, с умножением неразрешим уже первый этаж — экзистенциальная теория.

Наследие Купера при этом живо: SMT-солверы (глава 15) решают линейную арифметику как повседневную рутину, аккуратно обходя полную элиминацию и её двойную экспоненту.

§ 26.12 Итоги главы

Ответ на вопрос, можно ли по тексту программы заранее узнать, завершится ли она, отрицателен — и это доказано. Проблема остановки — первая неразрешимость: диагональный аргумент показывает, что никакая машина Тьюринга не может по описанию другой машины и её входу решить, остановится ли та. Разрешимость мы

¹⁹⁵M. J. Fischer, M. O. Rabin. «Super-Exponential Complexity of Presburger Arithmetic». 1974.

отделили от распознаваемости: HALT распознаваема, но не разрешима, и разрешимость оказалась вычислимостью с двух сторон — положительной и отрицательной.

Техника сведений переносит этот результат с задачи на задачу, а теорема Райса закрывает весь класс одним утверждением: любое нетривиальное свойство распознаваемого языка алгоритмически неразрешимо, хотя синтаксические свойства вроде числа состояний она не покрывает.

За первым рубежом открылась структура неразрешимости. Арифметическая иерархия расставляет задачи по числу чередующихся кванторов в их определениях: тотальность строго труднее проблемы остановки, и лестница продолжается вверх, а оракулы порождают бесконечную иерархию. Та же диагональная схема, перенесённая с машин на формальные системы, даёт теоремы Гёделя: в достаточно сильной непротиворечивой теории есть истинные, но недоказуемые утверждения, а собственную непротиворечивость она не доказывает — для PA это показал Генцен, выйдя к трансфинитной индукции до ε_0 (глава 29). Функция занятого бобра $\Sigma(n)$ подводит черту: максимум по конечному множеству останавливающихся машин — величина с простым комбинаторным определением — растёт быстрее любой вычислимой функции.

К концу главы граница вычислимого превращается из запрета в инструмент: неразрешимость можно доказывать и переносить между задачами, а иерархия позволяет сравнивать разные неразрешимости по трудности. Дальше вычисление предстанет с другой стороны — как подстановка символов по правилам — и эта смена ракурса приведёт нас к лямбда-исчислению и теории типов.

Проверка Проверьте себя

1. Почему проблема остановки неразрешима и где диагональный аргумент опирается на существование решателя?
2. Что даёт техника сведений и в каком направлении её применяют?
3. Что утверждает теорема Райса и почему она не покрывает свойство «машина имеет не более 100 состояний»?
4. Чем распознаваемость отличается от разрешимости?
5. Что измеряет уровень арифметической иерархии и почему тотальность труднее проблемы остановки?
6. Почему из диагонального аргумента следует неполнота формальных систем, а не только неразрешимость проблем?

Что дальше?

Мы увидели, где проходит граница вычислимого: за ней — неразрешимые задачи, и их больше, чем разрешимых. В следующей главе мы взглянем на вычисление с другой стороны — как на подстановку символов по правилам: лямбда-исчисление (глава 27). Оно не добавляет вычислительной силы, но даёт новый взгляд на природу вычисления. Этот взгляд приводит к теории типов, в которой программы

и доказательства оказываются одним и тем же (глава 28). Затем, в главе 30, мы перейдём от вопроса «что можно вычислить» к вопросу «как быстро» — и подойдём к главной открытой проблеме computer science: P vs NP.

§ 26.13 Упражнения

Разрешимость и проблема остановки.

1. Объясните различие между распознаваемостью и разрешимостью. Приведите пример языка, который распознаваем, но не разрешим.
2. Докажите, что язык L разрешим тогда и только тогда, когда распознаваемы и он сам, и его дополнение $\bar{L}$.
3. Воспроизведите диагональный аргумент неразрешимости проблемы остановки. Укажите, в какой момент доказательства используется предположение о существовании универсального решателя H , и какая именно конфигурация приводит к противоречию.

Сведения.

4. Докажите, что язык $L_1 = \{\langle M \rangle \mid M \text{ останавливается на пустом входе}\}$ неразрешим. Сведите к нему проблему остановки.
5. Докажите, что язык $L_{\text{empty}} = \{\langle M \rangle \mid L(M) = \emptyset\}$ неразрешим. Используйте сведение от проблемы остановки.
6. * Докажите неразрешимость языка $L_{\text{univ}} = \{\langle M \rangle \mid L(M) = \Sigma^*\}$. Указание: сведите к нему L_{empty} или проблему остановки.
7. * Докажите, что задача проверки эквивалентности двух МТ ($L(M_1) = L(M_2)$) неразрешима — сведением от задачи пустоты языка.

Теорема Райса.

8. Сформулируйте теорему Райса. Почему она утверждает неразрешимость «одним ударом» для целого класса задач?
9. Объясните, почему теорема Райса не применима к свойству «машина Тьюринга имеет не более 100 состояний». Какова принципиальная разница между этим свойством и свойствами из теоремы Райса?
10. * Воспроизведите доказательство теоремы Райса: сведите проблему остановки к произвольному нетривиальному экстенциональному свойству P . Разберите случай, когда пустой язык не обладает свойством P , и объясните, почему противоположный случай сводится к нему через дополнение.

Арифметическая иерархия.

11. К какому уровню арифметической иерархии принадлежит проблема остановки? А задача тотальности («останавливается ли МТ на каждом входе»)? Объясните разницу в кванторной структуре.
12. Какие задачи остаются неразрешимыми для МТ с оракулом для HALT? Как этот факт порождает бесконечную иерархию оракулов?

Функция занятого бобра.

13. Что такое функция занятого бобра $\Sigma(n)$? Почему она определена как максимум по конечному множеству, но невычислима?
14. * Докажите, что $\Sigma(n)$ растёт быстрее любой вычислимой функции: для любой вычислимой функции $f : \mathbb{N} \rightarrow \mathbb{N}$ существует порог n_0 , после которого $\Sigma(n) > f(n)$ для всех $n \geq n_0$. Почему из этого сразу следует невычислимость Σ ?
15. ** Докажите, что не существует машины Тьюринга, которая по числу n вычисляет $\Sigma(n)$. Подсказка: предположите существование такой машины и постройте на её основе разрешитель проблемы остановки.

ПРОЕКТ Охота на занятого бобра

Занятой бобёр определён как максимум по конечному множеству машин, и для малых n это множество можно честно перебрать. Для двух состояний пример выше насчитал $12^4 = 20736$ машин и рекорд в четыре единицы. Соберите переборщик для трёх состояний и воспроизведите значения, которые в 1960-е годы потребовали отдельного компьютерного исследования.

① Кодировка и перечисление

Таблица переходов трёхсостояниеовой машины состоит из шести клеток — по одной на комбинацию рабочего состояния и читаемого символа. Каждая клетка выбирает из $2 \times 2 \times 4 = 16$ вариантов: какой символ записать, куда сдвинуть головку и каким будет следующее состояние — одно из трёх рабочих или останов. Закодируйте таблицу одним числом от 0 до $16^6 - 1 = 16777215$ и научитесь декодировать его обратно. Проверка: при двух состояниях тот же код перечисляет $12^4 = 20736$ машин и находит машину Радо из примера выше.

② Симулятор с потолком

Реализуйте пошаговую симуляцию на пустой ленте: лента как словарь позиций, текущее состояние, позиция головки. Остановившуюся машину характеризуйте парой чисел: шагов до останова и единиц на ленте. Машины, не остановившиеся за потолок шагов, помечайте отдельно: для поиска кандидатов потолка в тысячу шагов достаточно с запасом, а доказательство отсутствия останова — отдельная работа, ей посвящён этап ниже.

③ Рекорд

Просмотрите все 16^6 машин и найдите максимум единиц и максимум шагов среди остановившихся. Значения должны совпасть с известными:

$\Sigma(3) = 6$ и $S(3) = 21$ ¹⁹⁶. Выведите таблицу машины-рекордсмена в нотации примера выше и сравните с двумя состояниями: шесть единиц против четырёх.

④ *Перебор против доказательства*

Неостановившиеся машины разберите по классам: останов недостижим, машина возвращается в уже виденную конфигурацию, лента растёт без остановочным сдвигом в одну сторону. Подсчитайте, сколько машин попало в каждый класс и сколько осталось необъяснёнными. Ровно здесь проходит граница из теоремы выше: перебор находит кандидатов, а утверждение «не остановится никогда» требует отдельного доказательства для каждой машины.

⑤ *Масштабирование*

Прикиньте, что даёт наивный перебор при четырёх состояниях: $20^8 \approx 2.6 \times 10^{10}$ машин, и полный просмотр уже тяжёл. Опишите отсев очевидно закливающих таблиц до симуляции: недостижимый останов, немедленное повторение клетки. Значение $\Sigma(4) = 13$ из списка выше доказал Аллен Брэди в 1983 году, комбинируя перебор с доказательствами закливания отдельных машин.

Итог: переборщик, который воспроизводит $\Sigma(3) = 6$ и $S(3) = 21$, печатает таблицу рекордсмена в нотации главы и показывает, где кончается сила перебора и начинается доказательство отсутствия останова.

¹⁹⁶S. Lin, T. Radó. «Computer Studies of Turing Machine Problems». Journal of the ACM, 1965.

XXVII

Бестиповое λ -исчисление

Школьное упрощение $2 \cdot 3 + 4 = 10$ — тоже вычисление, и для него не нужны ни машина, ни программист. Есть выражение, и есть правила переписывания: умножить, сложить, зачеркнуть выполненное. Вычисление как переписывание — второй путь к алгоритму, столь же строгий, как машина с лентой из главы 25.

Там каждый шаг задавал механизм: состояние, символ под головкой, сдвиг, перезапись. Здесь механизма нет вовсе. Остаются функции и единственное действие — *подстановка* аргумента в тело функции. Ни памяти, ни счётчика команд, ни присваивания: шаг вычисления — всегда замена одного подвыражения другим.

Квадрат — функция $x \mapsto x \cdot x$. Применённая к пяти, она даёт двадцать пять. Из записи выкидывается всё лишнее: имя функции, знак умножения, сами числа. Остаётся скелет: $\lambda x.xx$ — функция, применяющая свой аргумент к самому себе. Один шаг подстановки в применении $\lambda x.xx$ к аргументу 5 даёт 55 — число, применённое к числу. В школьной записи это бессмыслица, но язык ещё не достроен: числа в нём сами станут функциями, и применение обретёт смысл. Весь язык уместается в три правила построения выражений и одно правило вычисления.

Подстановка — не кабинетная экзотика: ровно её выполняет интерпретатор при вызове любой функции в Python, JavaScript или Haskell. Анонимные функции, замыкания, `map` и `filter` — прямые потомки этого исчисления. Императивные языки унаследовали машину Тьюринга, функциональные — алгебру подстановок, а тезис Чёрча–Тьюринга утверждает, что класс вычислимого у них один. Теорема Чёрча–Россера обещает: если у выражения есть нормальная форма, она единственна, и порядок шагов на итог не влияет.

Что такое вычисление, если в нём нет ни памяти, ни состояний — а только замена выражений?

ИСТОРИЯ Чёрч и функциональный взгляд на вычисление

Алонзо Чёрч (1903–1995) разработал λ -исчисление в начале 1930-х годов. Первая публикация — 1932 год. Систематическое изложение появилось в 1941-м в монографии «The Calculi of Lambda-Conversion».

Изначальная цель была амбициозной: построить логическую систему, в которой вся математика выражается через функции. Система оказалась противоречивой (Клини и Россер в 1935-м нашли парадокс), но вычислительное ядро — чистое λ -исчисление — выжило.

В 1936-м Чёрч использовал λ -исчисление, чтобы доказать неразрешимость Entscheidungsproblem Гильберта — на месяц раньше Тьюринга и независимо от него. Он определил понятие лямбда-определимой функции и показал, что проблема распознавания равенства двух лямбда-термов алгоритмически неразрешима.

Тьюринг в 1937 году доказал эквивалентность λ -определимости и вычислимости на МТ. Две модели связаны и математически, и биографически: в 1936–1938 годах Тьюринг работал в Принстоне, и его научным руководителем был Чёрч. Эквивалентность, доказанная Тьюрингом в 1937-м, — результат работы ученика и учителя. С тех пор λ -исчисление и машины Тьюринга — два канонических формализма, определяющих границу вычислимого.

В 1958 году Джон Маккарти положил λ -нотацию в основу языка Lisp, а впоследствии идея функций как значений обрела вторую жизнь в ML, Scheme и Clojure. Бестиповое ядро осталось тем же самым, которое описал Чёрч в 1932-м.

ОБЗОР ГЛАВЫ

В этой главе мы построим лямбда-термы из переменных, абстракций и аппликаций и научимся различать свободные и связанные переменные, выполнять альфа-конверсию и подстановку без захвата. β -редукция — правило вычисления — доведёт термы до нормальных форм, а теорема Чёрча–Россера покажет, что порядок редукции на результат не влияет. На одних функциях мы закодируем булевы значения и числа Чёрча, научимся складывать, умножать и строить условные выражения — без встроенных примитивов. Рекурсия выразится через неподвижные точки и Y -комбинатор: функция сможет вызывать саму себя в мире без имён. В конце главы мы увидим, что этот мир функций очерчивает ту же границу вычислимости, что и машина Тьюринга: класс λ -определимых функций совпадает с классом вычислимых.

После этой главы вы сможете:

1. строить термы из переменных, абстракций и аппликаций и различать свободные и связанные переменные;
2. выполнять подстановку без захвата и бета-редукцию до нормальной формы;
3. применять теорему Чёрча–Россера;
4. кодировать булевы значения, числа Чёрча и арифметику;
5. объяснять Y -комбинатор и эквивалентность λ -исчисления машине Тьюринга.

§ 27.1 Три способа строить термы

Начнём с конкретного терма: $\lambda x.x$ — тождественная функция: «функция, которая принимает x и возвращает x ». Применим её к аргументу y : $(\lambda x.x)y$. Подставляем y вместо x в тело x и получаем y , то есть $(\lambda x.x)y \xrightarrow{\beta} y$.

Возьмём терм посложнее: $\lambda x.\lambda y.x$ — функция, которая принимает x и возвращает функцию, принимающую y и игнорирующую его (возвращающую x). Применим к a и b :

$$(\lambda x.\lambda y.x)ab \xrightarrow{\beta} (\lambda y.a)b \xrightarrow{\beta} a$$

Два шага — первый аргумент получен, второй проигнорирован. Терм $\lambda x.\lambda y.x$ — *комбинатор*: терм без свободных переменных, ни от чего извне он не зависит. Он возвращает первый аргумент и игнорирует второй.

Три действия — и больше ничего для построения термов не требуется:

- *Переменная*: x обозначает некоторое значение — неизвестное, но имя у него есть.
- *Абстракция*: $\lambda x.M$ означает «функция от x с телом M ». Точка разделяет параметр и тело — как в математике пишут $x \mapsto f(x)$.
- *Аппликация*: MN означает «применить функцию M к аргументу N ».

Никаких чисел, строк или циклов — только переменные, построение функций и применение функций. Этого достаточно, чтобы выразить любое вычисление.

ОПРЕДЕЛЕНИЕ 27.1 λ -термы

Множество Λ всех лямбда-термов строится индуктивно:

1. **Переменная**: всякая переменная — терм: $x \in \Lambda$.
2. **Аппликация**: применение терма к терму — терм: $(MN) \in \Lambda$.
3. **Абстракция**: абстракция по переменной из терма — терм: $\lambda x.M \in \Lambda$.

Никаких других λ -термов не существует.

Примечание

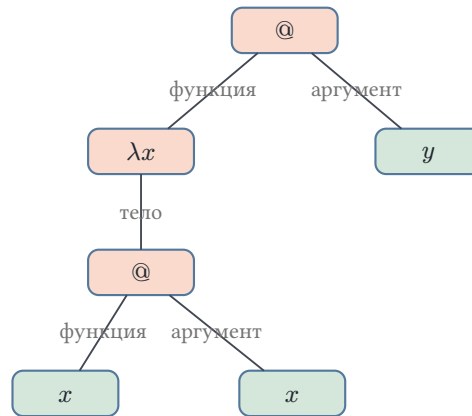
Три правила индуктивного определения — это три конструктора типа. На Rust:

```
/// Термы  $\lambda$ -исчисления: переменная, абстракция, аппликация.
enum Term {
    Var(String),
    Abs(String, Box<Term>),
    App(Box<Term>, Box<Term>),
}
```

Бета-редекс $(\lambda x.M)N$ — это аппликация, где левый терм — абстракция. Подстановка $M[x := N]$ требует избежания захвата переменных — к ней мы вернёмся дальше в главе.

Три правила, три строки — и мы имеем универсальный язык, который выглядит почти обманчиво простым.

Синтаксис терма удобно читать как дерево: у аппликации два поддерева, у абстракции — одно. Дерево терма $(\lambda x.xx)y$ показано на рис. 109: корень — аппликация, левый сын — абстракция λx с телом xx , правый сын — переменная y .



Синтаксическое дерево терма $(\lambda x.xx)y$

Рис. 109. Синтаксическое дерево терма $(\lambda x.xx)y$.

Дерево понадобится нам для работы с переменными: позже мы будем определять, какие переменные свободны, а какие связаны, отслеживая путь от вхождения переменной к корню. Каждый λ -терм — либо лист-переменная, либо узел-аппликация с двумя поддеревьями, либо узел-абстракция с параметром и одним поддеревом.

Пример Простейшие λ -термы

1. x, y, z — переменные, а значит, лямбда-термы.
2. (xx) — аппликация переменной к самой себе. Синтаксически корректен, хотя смысл пока неясен.
3. $\lambda x.x$ — тождественная функция, комбинатор I .
4. $\lambda x.\lambda y.x$ — комбинатор K : принимает x и возвращает функцию, которая принимает y и игнорирует его.
5. $(\lambda x.x)y$ — аппликация тождественной функции к y .

Чтобы не утонуть в скобках, принимаем стандартные соглашения:

- Внешние скобки опускаем: пишем $\lambda x.M$ вместо $(\lambda x.M)$.
- Аппликация лево-ассоциативна: $MNP = ((MN)P)$.
- Абстракция право-ассоциативна: $\lambda xy.M = \lambda x.(\lambda y.M)$.

- Аппликация связывает сильнее абстракции: $\lambda x.MN$ означает $\lambda x.(MN)$, а не $(\lambda x.M)N$.

Соглашения экономят чернила, но требуют привычки. Особенно коварна левая ассоциативность аппликации: $MNP = (MN)P \neq M(NP)$. Инженерная аналогия: скобки в λ -исчислении — как знаки приоритета операций в арифметике. Опускаешь — полагаешься на конвенцию.

ЛОВУШКА Аппликация левоассоциативна

Считать xuz аппликацией x к (yz) — ошибка. По соглашению $xuz = (xu)z$: аппликация левоассоциативна, скобки группируются слева. Терм $x(yz)$ — другая аппликация: функция x применяется к результату yz . Ошибка в расстановке скобок меняет смысл терма.

27.1.1 Свободные и связанные переменные

Вернёмся к дереву терма и рассмотрим $\lambda x.xu$. В терме три вхождения переменных: связывающая x (после абстракции), x в теле и u .

Геометрический критерий: идём от вхождения переменной вверх к корню дерева. Если на пути встречается узел-абстракция с той же переменной — вхождение *связано*. Не встречается — *свободно*.

В $\lambda x.xu$ путь от x в теле идёт вверх и сразу упирается в λx — x связана. Путь от u идёт вверх, проходит через аппликацию, через λx (которая связывает x , а не u) и доходит до корня — ни одна λu не встретила, u свободна. Простой геометрический критерий, работающий безошибочно.

Ещё пример: $x(\lambda x.x)$. Первое вхождение x (левое) свободно — на пути к корню нет λx . Второе и третье вхождения x (внутри абстракции) связаны — на их пути стоит λx .

И ещё: $(\lambda a.(\lambda b.ab))a$. Скобки вокруг абстракции обязательны: без них её тело продолжилось бы вправо, и последнее a попало бы в область действия λa . a после λ (первая) — связывающая. a в теле внутренней абстракции связана внешней λa (путь проходит через λb , которая не связывает a , и упирается в λa). b после λ — связывающая. b в теле связана λb . Последнее a — аргумент, к которому применяется абстракция — свободно: оно вне области действия λa .

ОПРЕДЕЛЕНИЕ 27.2 Свободные переменные

Множество свободных переменных $FV(M)$ определяется рекурсивно:

1. $FV(x) = \{x\}$,
2. $FV(MN) = FV(M) \cup FV(N)$,
3. $FV(\lambda x.M) = FV(M) \setminus \{x\}$.

ОПРЕДЕЛЕНИЕ 27.3 Комбинатор

Терм называется комбинатором, если он не содержит свободных переменных.

Пример Комбинаторы и не-комбинаторы

1. $\lambda x.x$ — комбинатор: свободных переменных нет.
2. $\lambda x.xy$ — не комбинатор: y свободна.

$I = \lambda x.x$, $K = \lambda xy.x$ — комбинаторы. Комбинаторы — программы, которым не нужен внешний контекст для вычисления.

Примечание Свободные и связанные: аналогия с программированием

Свободная переменная — как параметр функции в Python, значение которого будет передано извне при вызове. Связанная переменная — как локальная переменная функции, объявленная в её заголовке. Различие существенно для подстановки: свободные переменные мы заменяем, связанные — нет. Если заменить связанную переменную, программа сломается — как если бы вы переименовали параметр функции в середине её тела, но не в заголовке.

27.1.2 α -конверсия

Функция $\lambda x.x$ — тождественная. Функция $\lambda y.y$ — тоже тождественная. Это одна и та же функция, записанная с разными именами параметра.

Альфа-конверсия, или *альфа-эквивалентность*, — отождествление термов, отличающихся только именами связанных переменных.

ОПРЕДЕЛЕНИЕ 27.4 α -конверсия

Терм $\lambda x.M \stackrel{\alpha}{=} \lambda y.M'$, где M' получен из M заменой вхождений x , связанных этой абстракцией, на y (переименование связанной переменной). Условие: y не входит свободно в M и y не является связывающей переменной в M .

ОПРЕДЕЛЕНИЕ 27.5 α -эквивалентность

Два терма называются α -эквивалентными, если один может быть получен из другого последовательностью α -конверсий.

ЛОВУШКА α -эквивалентность — не равенство термов

Считать, что $\lambda x.x \neq \lambda y.y$, — ошибка. Это одна и та же функция: α -эквивалентные термы отождествляются. Переименование связанной переменной не меняет смысла, но при подстановке нужно избегать захвата свободных переменных. Иначе $(\lambda x.y)[y := x] = \lambda x.x$ — неверный результат.

Пример Корректное и некорректное переименование

- $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$: переименование корректно, y не входит свободно в тело x .
- $\lambda x.y \neq \lambda y.y$: y свободна в исходном терме, переименование захватывает её.
- $\lambda x.(\lambda y.x) \neq \lambda y.(\lambda y.y)$: внутренний x попадает в область действия λy .

Условия — не бюрократия. Без них переименование меняет смысл.

Рассмотрим $\lambda x.y$. Переименуем x в y : получим $\lambda y.y$. Первый терм — константная функция, всегда возвращающая свободную y . Второй — тождественная функция. Переименование испортило программу: свободная y из тела стала связанной.

Другой пример: $\lambda x.(\lambda y.x)$. Переименуем x в y : получим $\lambda y.(\lambda y.y)$. В исходном терме внутренний x связан внешней λx . После переименования он оказался связан внутренней λy — и функция теперь возвращает второй аргумент, а не первый.

С этого момента мы считаем α -эквивалентные термы одним и тем же термом. $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$ — один и тот же терм. Пишем «с точностью до α -конверсии».

27.1.3 Подстановка

Подстановка — операция, которая заменяет выполнение программы. Мы берём терм M и заменяем в нём все свободные вхождения переменной x на терм N , обозначается $M[x := N]$.

Разберём четыре примера — от простого к тонкому.

Пример 1. $M = x$, заменяем x на 5. $x[x := 5] = 5$. Переменная совпала — заменили.

Пример 2. $M = y$, заменяем x на 5. $y[x := 5] = y$. Переменная не совпала — оставили как есть.

Пример 3. $M = (xz)$, заменяем x на $\lambda w.w$. $(xz)[x := \lambda w.w] = (\lambda w.w)z$. Рекурсивно заменили в обеих частях аппликации.

Пример 4. $M = \lambda y.xy$, заменяем x на y .

Наивный подход: идём по терму, видим свободное вхождение x , заменяем на y — получаем $\lambda y.yu$. Но исходный терм $\lambda y.xy$ — это функция, которая принимает аргумент и применяет к нему свободную переменную x . После наивной замены получился $\lambda y.yu$ — функция, применяющая свой аргумент к самому себе. Смысл термина изменился.

Проблема в том, что свободная переменная y , которую мы подставили вместо x , попала в область действия λy и стала связанной. Это называется *захватом переменной*: подставляемая переменная y оказалась захвачена связыванием λy .

Правильное решение: перед подстановкой переименовать связанную переменную в свежее имя. Переименуем y в z (альфа-конверсия): $(\lambda y.xy) \stackrel{\alpha}{=} (\lambda z.xz)$. Теперь подставляем y вместо x : $(\lambda z.xz)[x := y] = \lambda z.yz$.

Свободная переменная y осталась свободной. Связанная переменная z осталась связанной. Смысл термина не изменился.

ОПРЕДЕЛЕНИЕ 27.6 Подстановка

Подстановка термина N вместо свободной переменной x в терме M определяется рекурсивно:

1. $x[x := N] = N$,
2. $y[x := N] = y$ (y и x различны),
3. $(M_1 M_2)[x := N] = (M_1[x := N])(M_2[x := N])$,
4. $(\lambda x.M)[x := N] = \lambda x.M$ (связанная переменная не заменяется),
5. $(\lambda y.M)[x := N] = \lambda y.(M[x := N])$ (x и y различны, y не входит свободно в N),
6. $(\lambda y.M)[x := N] = \lambda z.(M[y := z][x := N])$ (z — свежая переменная, x и y различны, y входит свободно в N).

Последний пункт — тот самый случай захвата. Везде далее под $M[x := N]$ мы понимаем подстановку с избеганием захвата.

Замечание Почему подстановка определена так сложно?

Может показаться, что переименование при подстановке — искусственное усложнение. Почему бы просто не запретить термы, в которых возможно перекрытие имён?

Потому что тогда мы лишимся возможности рассуждать о термах как о чисто структурных объектах. Комбинатор $K = \lambda xy.x$ должен работать одинаково, подадим ли мы ему аргумент y или z . Захват переменной — следствие того, что имена переменных локальны, а термы строятся композиционно. Решение с переименованием сохраняет оба свойства: локальность имён и композиционность построения.

Проверка Синтаксис: скобки и переименование

1. Почему $xyz = (xy)z \neq x(yz)$? Что меняется при перестановке скобок?
2. Почему $\lambda x.x \stackrel{\alpha}{=} \lambda y.y$, а $\lambda x.y$ и $\lambda y.y$ — нет? Какое условие нарушено во второй паре?

§ 27.2 β -редукция

Синтаксис даёт нам термы — выражения, построенные по правилам. Семантика даёт правило их упрощения.

Правило упрощения — β -редукция. Оно говорит: если у вас есть функция $(\lambda x.M)$ и аргумент N , подставьте аргумент в тело функции.

ОПРЕДЕЛЕНИЕ 27.7 β -редукция

Бета-редексом называется терм вида $(\lambda x.M)N$. Одношаговая β -редукция, обозначаемая $\rightarrow_\beta$, определяется правилом:

$$(\lambda x.M)N \xrightarrow{\beta} M[x := N]$$

и расширяется на подтермы:

1. $M \xrightarrow{\beta} M' \Rightarrow MN \xrightarrow{\beta} M'N$,
2. $N \xrightarrow{\beta} N' \Rightarrow MN \xrightarrow{\beta} MN'$,
3. $M \xrightarrow{\beta} M' \Rightarrow \lambda x.M \xrightarrow{\beta} \lambda x.M'$.

Многошаговая β -редукция $\rightarrow_\beta^*$ — рефлексивно-транзитивное замыкание $\rightarrow_\beta$.

Говоря проще: находим подвыражение вида $(\lambda x.M)N \xrightarrow{\beta} M[x := N]$, заменяем, повторяем — пока есть что заменять. Обычно количество редексов при этом убывает, но шаг может и породить новые.

Например, терм $(\lambda f.f(fx))(\lambda y.y)$ содержит один редекс — всю аппликацию. Шаг подстановки *удваивает* вхождение аргумента:

$$(\lambda f.f(fx))(\lambda y.y) \xrightarrow{\beta} (\lambda y.y)((\lambda y.y)x).$$

В результате редексов уже два: внешняя аппликация и внутренняя $(\lambda y.y)x$. Затем оба схлопываются, и терм сводится к x .

Пример Два аргумента (комбинатор K)

$$(\lambda x.\lambda y.x)ab \xrightarrow{\beta} (\lambda y.a)b \xrightarrow{\beta} a$$

Первый шаг: подставили a вместо x . Второй шаг: подставили b вместо y — но y не входит в тело a , поэтому результат просто a . Комбинатор K возвращает первый аргумент и игнорирует второй.

До сих пор редукция занимала один-два шага. Дальше понадобится целая цепочка подстановок.

Пример Композиция функций

$$(\lambda f.\lambda x.fx)(\lambda y.y)z \xrightarrow{\beta} (\lambda x.(\lambda y.y)x)z \xrightarrow{\beta} (\lambda y.y)z \xrightarrow{\beta} z$$

Три шага: подставили функцию $(\lambda y.y)$ вместо f , подставили z вместо x , применили тождественную функцию к z .

Все предыдущие примеры завершались. Покажем, что так бывает не всегда.

Пример Бесконечная редукция (Ω)

$$\begin{aligned}
 (\lambda x.xx)(\lambda x.xx) &\xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx) \\
 &\xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx) \\
 &\xrightarrow{\beta} \dots
 \end{aligned}$$

Терм $\Omega = (\lambda x.xx)(\lambda x.xx)$ — «вечный двигатель» лямбда-исчисления. Единственный возможный шаг редукции воспроизводит исходный терм. В языке с именами простейшая рекурсивная программа тривиальна: `loop = loop` — определение, которое, развернувшись, возвращается к самому себе. Ω — тот же «loop = loop», только без имён: один терм, который разворачивается в самого себя. Редукция никогда не завершается.

Ω не завершается — просто «крутится вечно», лямбда-аналог бесконечного цикла. Проблема определения, остановится ли произвольный λ -терм, алгоритмически неразрешима — тот же факт, что и для машин Тьюринга.

Примечание β -редукция и вызов функции

β -редукция — это в точности то, что происходит при вызове функции в любом языке программирования. Вы объявляете функцию `def f(x): return x + 1`. Вызываете `f(5)`. Интерпретатор подставляет 5 вместо `x` в тело `x + 1` и вычисляет `5 + 1 = 6`.

В λ -исчислении нет встроенного сложения — только подстановка. Но принцип тот же: аргумент подставляется вместо параметра, и выражение упрощается дальше.

Терм, в котором больше нет β -редексов, находится в *нормальной форме*: редуцировать больше нечего, программа завершилась.

ОПРЕДЕЛЕНИЕ 27.8 Нормальная форма

Терм M находится в нормальной форме, если он не содержит бета-редексов — подтермов вида $(\lambda x.P)Q$.

Говорят, что M имеет нормальную форму N , если $M \xrightarrow{\beta} N$ и N — в нормальной форме.

Пример Нормальная форма и её отсутствие

Терм $(\lambda x.x)(\lambda y.y)$ имеет нормальную форму: $(\lambda x.x)(\lambda y.y) \xrightarrow{\beta} \lambda y.y$, и $\lambda y.y$ — в нормальной форме (нет редексов).

Терм $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ не имеет нормальной формы: единственный шаг даёт $\Omega \xrightarrow{\beta} \Omega$, и процесс никогда не завершается.

Не каждый терм имеет нормальную форму. Ω не имеет. Задача определения, имеет ли произвольный λ -терм нормальную форму, алгоритмически неразрешима.

Однако если нормальная форма существует, она единственна. И путь к ней не имеет значения.

ТЕОРЕМА 27.1 Чёрча–Россера

Если $M \xrightarrow{\beta} N_1 \wedge M \xrightarrow{\beta} N_2$, то существует терм L такой, что

$$N_1 \xrightarrow{\beta} L$$

$$N_2 \xrightarrow{\beta} L.$$

Набросок доказательства

Докажем конфлюэнтность через параллельную редукцию: усилим одношаговую редукцию до отношения, обладающего свойством ромба.

Один шаг $\xrightarrow{\beta}$ не обладает свойством ромба (*diamond property*): если из M можно сделать один шаг в N_1 и один шаг в N_2 , эти два терма не обязаны сходиться за один шаг. Но свойством ромба обладает *параллельная редукция* $\succeq$ — отношение, позволяющее редуцировать несколько непересекающихся редексов одновременно.

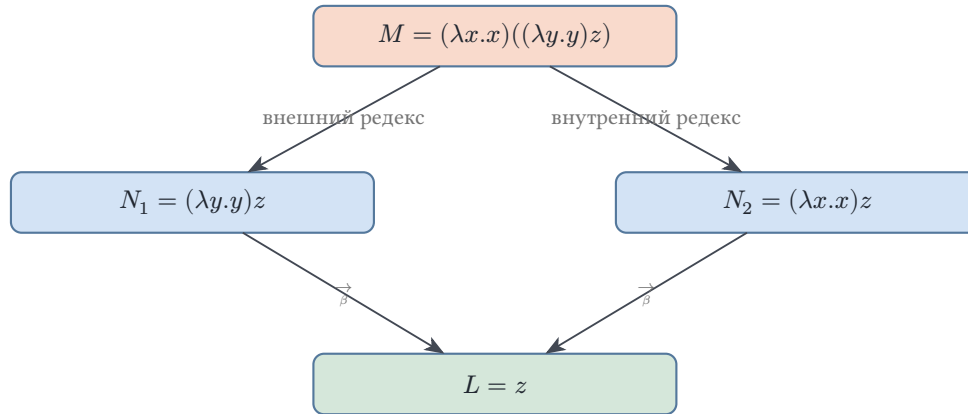
Параллельная редукция определяется индуктивно: переменная редуцируется сама в себя, аппликация и абстракция — покомпонентно, а для бета-редекса справедливо $M \succeq M' \wedge N \succeq N' \Rightarrow (\lambda x.M)N \succeq M'[x := N']$ — внешний редекс плюс внутренние шаги.

Промежуточная лемма (strip lemma): $M \xrightarrow{\beta} M_1 \wedge M \succeq M_2 \Rightarrow \exists M_3 : M_1 \succeq M_3 \wedge M_2 \xrightarrow{\beta} M_3$. Из леммы индукцией выводится свойство ромба для $\succeq$, а из него — конфлюэнтность для $\xrightarrow{\beta}$.

Технически полное доказательство громоздко, но идея прозрачна: усилить одношаговую редукцию до отношения, которое умеет сокращать несколько редексов за раз, — и ромб восстанавливается.

Рассмотрим конкретный пример: $M = (\lambda x.x)((\lambda y.y)z)$.

В M два редекса: внешний — $(\lambda x.x)(\dots)$ и внутренний — $(\lambda y.y)z$. Начнём с внешнего: $(\lambda x.x)((\lambda y.y)z) \xrightarrow{\beta} (\lambda y.y)z$, затем $(\lambda y.y)z \xrightarrow{\beta} z$. Начнём с внутреннего: $(\lambda x.x)((\lambda y.y)z) \xrightarrow{\beta} (\lambda x.x)z$, затем $(\lambda x.x)z \xrightarrow{\beta} z$. Оба пути ведут к z , и этот алмаз сходимости показан на рис. 110.



Теорема Чёрча–Россера: внешний и внутренний пути редукции сходятся к z .

Рис. 110. Алмаз Чёрча–Россера: оба порядка редукции сходятся к одному результату.

СЛЕДСТВИЕ 27.2 Единственность нормальной формы

Если терм имеет нормальную форму, она единственна (с точностью до α -конверсии). Порядок редукции не влияет на результат — можно редуцировать в любом порядке и прийти к тому же самому.

Набросок доказательства Единственность нормальной формы

Докажем, применив свойство Чёрча–Россера: любые две нормальные формы терма совпадают с точностью до α -конверсии.

Пусть терм M имеет нормальные формы N_1, N_2 . По свойству Чёрча–Россера из $M \xrightarrow{\beta} N_1, M \xrightarrow{\beta} N_2$ следует, что существует терм L , в который редуцируются обе: $N_1 \xrightarrow{\beta} L, N_2 \xrightarrow{\beta} L$. Но в нормальной форме нет редексов, поэтому обе дальше не редуцируются — их общий редукт совпадает с ними самими, то есть $N_1 \stackrel{\alpha}{=} N_2$.

При реализации λ -исчисления на компьютере это означает: мы можем выбирать любую стратегию редукции. Результат будет тем же — если он вообще существует.

Но *существование* результата от стратегии зависит.

Стратегия редукции — правило выбора следующего редекса. *Нормальная стратегия* (normal order) всегда берёт самый левый, самый внешний редекс. *Аппликативная стратегия* (applicative order) всегда берёт самый левый, самый внутренний.

ТЕОРЕМА 27.3 Стандартизации

Если терм имеет нормальную форму, то нормальная стратегия её достигает.

Набросок доказательства

Докажем индукцией по длине редукционного пути от исходного терма к нормальной форме. Доказательство опирается на лемму о стандартизации: если $M \xrightarrow{\beta} N$, то найдётся терм L такой, что M приходит к L стандартными шагами — редукциями самого левого внешнего редекса. К тому же L за конечное число шагов приходит и N . Применяя лемму к каждому шагу пути и склеивая получающиеся цепочки, собираем стандартный путь от исходного терма к нормальной форме.

Аппликативной стратегии такой гарантии нет. Терм $(\lambda x.y)\Omega$ имеет нормальную форму y : нормальная стратегия редуцирует внешний редекс и получает y за один шаг. Аппликативная стратегия сначала пытается привести аргумент Ω к нормальной форме — и навсегда остаётся в цикле.

В языках программирования это различие закрепилось как выбор между вызовом по имени (Haskell) и вызовом по значению (ML). Для вызова по имени теорема стандартизации — гарантия: пока результат существует, нормальная стратегия к нему приходит.

§ 27.3 Программирование в λ -исчислении

В бестиповом λ -исчислении нет ничего кроме функций — ни чисел, ни булевых значений, ни ветвлений — и этого достаточно.

Идея Чёрча: если λ -терм умеет только принимать аргументы и возвращать значения, то данные можно представить как *стратегию* взаимодействия с аргументами. Число — это стратегия итерации. Булево значение — стратегия выбора.

27.3.1 Булевы значения Чёрча

Булево значение делает простую вещь: выбирает одну из двух альтернатив — «если истина — то первое, иначе второе».

Представим его как функцию двух аргументов. Истина возвращает первый аргумент. Ложь возвращает второй.

ОПРЕДЕЛЕНИЕ 27.9 Булевы значения Чёрча

$\text{true} = \lambda xy.x$

$\text{false} = \lambda xy.y$

Теперь условное выражение — это просто аппликация булева значения к двум веткам:

1. $\text{true } AB \xrightarrow{\beta} A$ — истина выбирает первую ветку.
2. $\text{false } AB \xrightarrow{\beta} B$ — ложь — вторую.

Никакого специального if не требуется: булево значение «само делает выбор».

Логические операции строятся из этой же идеи:

- $\text{not} = \lambda b.b \text{ false true}$. Если b — «true», то $\text{true false true} \xrightarrow{\beta} \text{false}$. Если b — «false», то $\text{false false true} \xrightarrow{\beta} \text{true}$. Отрицание — просто перестановка аргументов местами.
- $\text{and} = \lambda ab.ab \text{ false}$. Если a — «true», возвращает b . Если a — «false», возвращает «false». Проверим: $\text{and true false} \xrightarrow{\beta} \text{true false false}$, и ещё два шага — $\text{true false false} \xrightarrow{\beta} \text{false}$.
- $\text{or} = \lambda ab.a \text{ true } b$. Если a — «true», возвращает «true». Если a — «false», возвращает b . Проверим: $\text{or false true} \xrightarrow{\beta} \text{false true true}$, и ещё два шага — $\text{false true true} \xrightarrow{\beta} \text{true}$.

Пример Вычисление and true false по шагам

$$\begin{aligned}
 \text{and true false} &= (\lambda ab.ab \text{ false}) \text{ true false} \\
 &\xrightarrow{\beta} (\lambda b. \text{true } b \text{ false}) \text{ false} \\
 &\xrightarrow{\beta} \text{true false false} \\
 &= (\lambda xy.x) \text{ false false} \\
 &\xrightarrow{\beta} (\lambda y. \text{false}) \text{ false} \\
 &\xrightarrow{\beta} \text{false}
 \end{aligned}$$

Четыре β -шага — и ответ получен. Никаких встроенных логических операций, только подстановка.

После того как кодировки проверены, про их внутреннее устройство можно забыть. Терм «not» записывается коротко — $\lambda b.b \text{ false true}$ — и не требует каждый раз разворачивать «true» и «false» до основания. Способность работать с кодированными объектами как с готовыми значениями и делает λ -исчисление похожим на настоящий язык программирования.

27.3.2 Числа Чёрча

Число n кодирует n -кратную итерацию: применить функцию f к начальному значению x ровно n раз.

1. Число 0 — применить f 0 раз, то есть вернуть x .
2. Число 1 — применить f 1 раз: $f(x)$.
3. Число 2 — применить f 2 раза: $f(f(x))$.
4. Число 3 — применить f 3 раза: $f(f(f(x)))$.
5. И так далее.

ОПРЕДЕЛЕНИЕ 27.10**Числа Чёрча**

$$\begin{aligned}
0 &= \lambda f x. x \\
1 &= \lambda f x. f x \\
2 &= \lambda f x. f(f x) \\
3 &= \lambda f x. f(f(f x)) \\
&\dots
\end{aligned}$$

Число n — это функция, которая принимает f и x и возвращает $f^n(x)$ — n -кратную итерацию f .

Теперь арифметика:

- **Последователь:** $\text{succ} = \lambda n f x. f(n f x)$. Берёт число n и применяет f ещё один раз. Проверим: $\text{succ } 0 = (\lambda n f x. f(n f x))(\lambda f x. x) \xrightarrow{\beta} \lambda f x. f x = 1$.
- **Сложение:** $\text{add} = \lambda m n f x. m f(n f x)$. Сначала n раз применяет f к x , потом ещё m раз.

$$\begin{aligned}
\text{add } 23 &\xrightarrow{\beta} \lambda f x. 2 f(3 f x) \\
&\xrightarrow{\beta} \lambda f x. 2 f(f(f(f x))) \\
&\xrightarrow{\beta} \lambda f x. f(f(f(f(f x)))) \\
&= 5
\end{aligned}$$

- **Умножение:** $\text{mult} = \lambda m n f x. m(n f) x$. Заменяет f на n -кратную итерацию f , затем применяет её m раз. Получается $m \times n$ итераций.
- **Возведение в степень:** $\text{exp} = \lambda m n. n m$. Применяет число n к числу m — n раз итерирует m , что и есть m^n . Проверим: $\text{exp } 23 = 8$.

Развернём вычисление: $\text{exp } 23 = 32$ — число 3 применяет свою функцию трижды, а в роли функции выступает число 2. Число 2, применённое к аргументу, удваивает число применений: внутренняя двойка даёт две копии x , каждая следующая — вдвое больше. Возведение в степень — *двойная итерация*: показатель степени становится числом итераций итератора.

Арифметика работает — без чисел как примитивов, без встроенных операций, на одних функциях. Число здесь — частный случай функции, а не самостоятельная сущность: функция первичнее числа.

Пример Проверка mult 23

$$\begin{aligned}
\text{mult } 23 &= (\lambda mnfx.m(nf)x)23 \\
&\xrightarrow{\beta} \lambda fx.2(3f)x \\
&\xrightarrow{\beta} \lambda fx.(\lambda x.3f(3fx))x \\
&\xrightarrow{\beta} \lambda fx.3f(3fx) \\
&\xrightarrow{\beta} \lambda fx.f(f(f(f(f(fx))))) \\
&= 6
\end{aligned}$$

Шесть итераций f — ровно 2×3 .

Замечание Числа Чёрча как итераторы

Числа Чёрча — не единственный способ закодировать натуральные числа в λ -исчислении. Существует кодирование Скотта, где число — это case-конструкция для разбора на ноль и последователь.

Но кодирование Чёрча — самое естественное с вычислительной точки зрения. Оно прямо связывает число с итерацией — и ровно этот принцип работает в циклах `for` любого языка программирования: итерация первична, число вторично.

`succ`, `add`, `mult` на числах Чёрча всегда завершаются и дают нормальную форму.

А вот `pred` — предшественник — выражается значительно сложнее. Извлечь $f^{n-1}(x)$ из $f^n(x)$, имея только функцию, — *не тривиально*. Первый работающий вариант предложил сам Клини.

Решение использует пары для накопления промежуточных значений. Пара и проекции: `pair` = $\lambda abs.sab$, `fst` = $\lambda p.p$ true, `snd` = $\lambda p.p$ false.

Начинаем с пары $(0, 0)$. На каждом шаге сдвигаем пару: $(a, b) \rightarrow (b, \text{succ } b)$. После n шагов первая компонента — $n - 1$, и её извлекает «fst».

§ 27.4 Неподвижные точки и рекурсия

Мы умеем складывать и умножать. Вернёмся к конкретной задаче — факториалу:

$$\text{fact}(n) = \begin{cases} 1 & \text{при } n = 0 \\ n \cdot \text{fact}(n - 1) & \text{при } n > 0 \end{cases}$$

Проблема: `fact` вызывает сам себя по имени. В λ -исчислении у функций нет имён. Терм $\lambda n....$ не может сослаться на «себя» — он просто анонимная функция.

Любая рекурсивная программа должна вызывать себя по имени. В обычных языках для этого есть имя функции, но в λ -исчислении имён нет — есть только переменные, абстракции и аппликации.

Идея решения — *неподвижная точка*.

Точка x называется неподвижной для функции f , если $f(x) = x$. В лямбда-исчислении: терм X такой, что $F X \stackrel{\beta}{=} X$. Неподвижная точка есть и у самых обычных функций. Если на калькуляторе жать кнопку квадратного корня раз за разом, число на дисплее в конце концов перестанет меняться. Для корня это происходит на 0 и 1: сколько ни жми — дальше не сдвинутся. Это и есть неподвижная точка: значение, на котором повторное применение функции ничего не меняет.

Если мы сможем для любой функции F построить её неподвижную точку, то сможем выразить рекурсию. Пусть F — «почти рекурсивная» функция: она принимает функцию f (которая будет играть роль рекурсивного вызова) и аргумент n , а возвращает результат. Неподвижная точка $X = F X$ — это ровно тот f , который вызывает сам себя. Сравните Y -комбинатор с термом $\Omega = (\lambda x.xx)(\lambda x.xx)$. Это то же самоприменение, только с дополнительным применением функции к результату: $xx \mapsto f(xx)$. Ω крутится вхолостую. А Y -комбинатор на каждом витке размотки применяет f — именно это превращает вечный цикл в рекурсию.

ОПРЕДЕЛЕНИЕ 27.11 Y -комбинатор Карри

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Проверим свойство Y -комбинатора: для любого F верно $Y F \stackrel{\beta}{\rightarrow} F(Y F)$.

$$\begin{aligned} Y F &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))) F \\ &\stackrel{\beta}{\rightarrow} (\lambda x.F(xx))(\lambda x.F(xx)) \\ &\stackrel{\beta}{\rightarrow} F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &= F(Y F) \end{aligned}$$

Два шага — и $Y F \stackrel{\beta}{\rightarrow} F(Y F)$: функция F , применённая к $Y F$. Можно развернуть ещё раз: $F(F(Y F))$, и так далее — сколько нужно, столько и разворачиваем. Тьюринг нашёл собственный комбинатор неподвижной точки: $\Theta = (\lambda x y.y(x y))(\lambda x y.y(x y))$. Он обладает тем же свойством: $\Theta F \stackrel{\beta}{\rightarrow} F(\Theta F)$.

Пример Факториал через Y -комбинатор

Определим «почти факториал» — функцию, которая при получении рекурсивного вызова f и аргумента n вычисляет факториал:

$$\text{Fact} = \lambda f n.(\text{isZero } n)1(\text{mult } n(f(\text{pred } n)))$$

Здесь «isZero» — лямбда-терм, проверяющий, является ли число Чёрча нулём: $\text{isZero} = \lambda n. n(\lambda x. \text{false}) \text{ true}$. pred — предшественник числа Чёрча (терм существует, но громоздок).

Тогда факториал: $\text{fact} = Y \text{ Fact}$.

Проверим $\text{fact } 2$:

$$\begin{aligned}
 \text{fact } 2 &= Y \text{ Fact } 2 \\
 &\xrightarrow{\beta} \text{Fact } (Y \text{ Fact}) 2 \\
 &\xrightarrow{\beta} (\text{isZero } 2) 1 (\text{mult } 2 (Y \text{ Fact } (\text{pred } 2))) \\
 &\xrightarrow{\beta} \text{false } 1 (\text{mult } 2 (Y \text{ Fact } 1)) \\
 &\xrightarrow{\beta} \text{mult } 2 (Y \text{ Fact } 1) \\
 &\xrightarrow{\beta} \dots \xrightarrow{\beta} 2
 \end{aligned}$$

Рекурсия без рекурсивных определений — только неподвижная точка.

Примечание Y -комбинатор и типы

Y -комбинатор невозможно типизировать в просто типизированном λ -исчислении (глава 28). Там каждое типизируемое выражение редуцируется к нормальной форме, а YF шаг за шагом воспроизводит сам себя. Рекурсия в типизированных системах требует либо явной операции неподвижной точки, либо более богатой системы типов. В бестиповом исчислении Y работает — и именно это делает бестиповое λ -исчисление Тьюринг-полным, а просто типизированное — нет.

§ 27.5 λ -исчисление и вычислимость

В главе 25 мы определили вычислимость через машины Тьюринга. Функция вычислима, если существует МТ, вычисляющая её.

Альтернативный подход: функция *лямбда-определима*, если существует лямбда-терм, который редуцируется к её значению на каждом аргументе. Точнее: функция $f : \mathbb{N}^k \mapsto \mathbb{N}$ лямбда-определима, если существует комбинатор F , такой что $F n_1 \dots n_k \xrightarrow{\beta} f(n_1, \dots, n_k)$ для всех $n_1, \dots, n_k \in \mathbb{N}$. Равенство требуется только там, где f определена. Если же f не определена на аргументе, то $F n_1 \dots n_k$ не имеет нормальной формы.

Пример Лямбда-определимость: последователь

Функция $f(n) = n + 1$ лямбда-определима: её комбинатор — последователь Чёрча из предыдущего раздела. Проверка по формуле из определения:

$$\begin{aligned} \text{succ } 4 &= (\lambda n f x. f(n f x))4 \\ &\stackrel{\beta}{\rightarrow} \lambda f x. f(4 f x) \\ &= 5. \end{aligned}$$

Числа 4 и 5 — числа Чёрча, а стрелка $\stackrel{\beta}{\rightarrow}$ — многошаговая редукция. Определение всего лишь называет комбинатором то, что мы и так строили: терм воспроизводит *таблицу значений* функции.

Тезис Чёрча (1936): класс λ -определимых функций совпадает с классом интуитивно вычислимых функций.

Тьюринг (1937) доказал, что класс λ -определимых функций в точности совпадает с классом функций, вычислимых на машине Тьюринга: две модели — одна граница вычислимости.

Доказательство эквивалентности состоит из двух половин:

1. Всякая лямбда-определимая функция вычислима на МТ: нужно написать интерпретатор лямбда-исчисления на машине Тьюринга.
2. Всякая вычислимая на МТ функция лямбда-определима: нужно закодировать конфигурацию МТ как лямбда-терм и определить функцию перехода через редукцию.

Обе половины технически громоздки, но принципиально выполнимы.

Для нас важнее следствие: всё, что можно запрограммировать на любом языке, можно выразить в λ -исчислении — три синтаксических правила, одно правило вычисления, и Тьюринг-полнота.

История повернулась *иронично*: машина Тьюринга — абстрактный математический объект — стала архитектурным прообразом компьютера с хранимой программой. А λ -исчисление — тоже абстрактный математический объект — стало ядром функциональных языков и, позже, теории типов с соответствием Карри–Ховарда (программы суть доказательства). Обе модели, рождённые в 1936-м как инструменты для доказательства неразрешимости, породили две ветви практического программирования.

Проверка λ -определимость и вычислимость

1. Почему доказательство эквивалентности λ -исчисления и машин Тьюринга распадается на две половины: интерпретатор и кодирование конфигурации?
2. Почему терм для функции, не определённой на аргументе, не имеет нормальной формы — и что это напоминает в машинах Тьюринга?

§ 27.6 * Можно ли обойтись без переменных?

В λ -исчислении переменные играют двойную роль: они обозначают аргументы и служат механизму подстановки. Но переменные — источник технических сложностей: α -конверсия, захват, условие свежести. Можно ли построить эквивалентную систему, в которой переменных нет совсем?

Моисей Шейнфинкель в 1924 году¹⁹⁷ — за восемь лет до Чёрча — изобрёл *комбинаторную логику*. В этом формализме вычисление сводится к перестановке трёх примитивных функций-комбинаторов, без переменных, без абстракции, без подстановки. Продолжил дело Шейнфинкеля Хаскелл Карри: с конца 1920-х годов он разрабатывал комбинаторную логику, стремясь построить основание математики на одних комбинаторах. Ему принадлежит и Y -комбинатор из раздела о неподвижных точках.

Три комбинатора:

- $I = \lambda x.x$ — тождество,
- $K = \lambda xy.x$ — константная функция (первый из двух),
- $S = \lambda xyz.xz(yz)$ — распределение.

Любой комбинатор можно перевести в комбинаторную логику алгоритмом *абстракции скобок* (bracket abstraction). Правила:

1. $[x]x = I$,
2. $[x]M = KM$, если x не входит свободно в M ,
3. $[x](MN) = S([x]M)([x]N)$.

Например, терм $\lambda xy.yx$ переводится так:

$$\begin{aligned} [x][y]yx &= [x](S([y]y)([y]x)) \\ &= [x](SI(Kx)) \\ &= S(K(SI))(S(KK)I) \end{aligned}$$

Результат громоздкий, но все переменные устранены. Вычисляет он то же самое, что исходный λ -терм.

Комбинаторная логика — минималистичный родственник λ -исчисления. В ней нет ни переменных, ни подстановки, ни α -конверсии — только правила редукции комбинаторов:

- $Ix \rightarrow x$
- $Kxy \rightarrow x$
- $Sxyz \rightarrow xz(yz)$

Этого достаточно для Тьюринг-полноты. Если λ -исчисление — это язык высокого уровня с автоматическим управлением памятью (переменными), то комбинаторная

¹⁹⁷M. Schönfinkel. «Über die Bausteine der mathematischen Logik». Mathematische Annalen, 1924.

логика — ассемблер без регистров: меньше понятий, но программа занимает больше места.

§ 27.7 Итоги главы

В λ -исчислении вычисление сводится к одному действию — подстановке аргумента в тело функции. Из одного правила подстановки вырастают числа Чёрча с их арифметикой и булевы значения, а Y -комбинатор добавляет рекурсию: весь репертуар программирования — от условных выражений до степеней и умножения — собирается одними функциями, без встроенных примитивов. Полнота при этом сохраняется: λ -определимые функции в точности совпадают с вычислимыми (тезис Чёрча–Тьюринга, глава 25).

Простота синтаксиса куплена ценой дисциплины. Подстановка требует различать свободные и связанные переменные, α -конверсия защищает от захвата имён, а конфлюэнтность (теорема Чёрча–Россера) гарантирует: порядок редукции на результат не влияет, и нормальная стратегия находит нормальную форму, если та существует. Каждое уточнение — ответ на конкретный сбой наивного переписывания, и вместе они делают алгебру подстановок надёжной.

Бестиповое исчисление при этом ничего не проверяет: применить число к числу ничто не запрещает, и бессмысленный терм спокойно редуцируется. Этот беспорядок и подводит к следующей главе: типы ограничат то, что можно написать, и проверка типов окажется проверкой доказательств.

Проверка Проверьте себя

1. Почему порядок β -редукции не влияет на результат?
2. Что такое захват переменной при подстановке и как его избежать?
3. Зачем нужен Y -комбинатор?
4. Что такое нормальная форма и всякий ли терм её имеет?
5. Чем свободная переменная отличается от связанной и что это значит для подстановки?
6. Что значит, что функция λ -определима, и как это связано с тезисом Чёрча–Тьюринга?

Что дальше?

Бестиповое λ -исчисление — мощный, но опасный инструмент: ничто не мешает применить число как функцию или функцию как число. $2\ \text{true}$ — синтаксически корректный терм, хотя смысла в нём нет. Программы, которые «идут не так», не отлавливаются до запуска — они либо зацикливаются, либо выдают бессмысленный результат.

В следующей главе мы введём типы — и обнаружим, что типизированные программы соответствуют математическим доказательствам. Начнём с просто типизированного лямбда-исчисления ($\lambda \rightarrow$) и увидим, что типизация — в точности логический вывод (соответствие Карри–Ховарда). Затем пройдем по лямбда-кубу — от полиморфизма до зависимых типов: $\lambda 2$, λP . Там, где доказательство есть программа, формула — тип, а проверка доказательства — проверка типов.

§ 27.8 Упражнения

Синтаксис, свободные и связанные переменные.

1. Найдите свободные переменные термов: $\lambda x.xy$, $x(\lambda x.xy)$, $\lambda ab.abc$, $(\lambda p.pq)(\lambda q.qr)$.
2. Какие из термов являются комбинаторами: $\lambda x.x$, $\lambda x.y$, $\lambda xy.x$, $x(\lambda x.x)$, $\lambda x.xyz$?
3. Сколько различных лямбда-термов можно построить, используя только переменные x и y , абстракцию и аппликацию, глубиной не более 2? (Глубина переменной — 0, а термов вида (MN) , $\lambda x.M$ — на 1 больше максимума глубин подтермов.)

Подстановка и α -конверсия.

4. Выполните α -конверсию: переименуйте связанные переменные так, чтобы никакая переменная не была одновременно свободной и связанной в одном терме. Терм: $\lambda x.(\lambda y.xy)(\lambda x.xy)$.
5. Выполните подстановку $M[x := N]$ с избеганием захвата: $M = \lambda y.xy$, $N = y$, затем $M = \lambda z.x(\lambda x.z)$, $N = z$. Покажите, какое переименование происходит во втором случае.

β -редукция и нормальные формы.

6. Редуцируйте до нормальной формы с указанием каждого β -шага: $(\lambda x.x)z$, $(\lambda xy.y)ab$, $(\lambda x.xx)(\lambda z.z)w$.
7. Редуцируйте терм $(\lambda x.xx)(\lambda x.xx)$ тремя шагами. Объясните, почему он не имеет нормальной формы.
8. Терм $(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))$ имеет нормальную форму — найдите её. Объясните, почему порядок редукции имеет значение: что будет, если сначала редуцировать аргумент? А если внешний редекс?
9. * Сформулируйте теорему Чёрча–Россера. Какое практическое следствие она имеет для вычисления на компьютере (выбор стратегии редукции)?

Булевы значения и числа Чёрча.

10. Постройте лямбда-терм хог для булевых значений Чёрча. Проверьте на всех четырёх комбинациях аргументов.

11. Постройте число Чёрча для 4 двумя способами: вручную по определению и как `succ 3`. Проверьте, что они β -эквивалентны.
12. Постройте лямбда-терм `isZero`, который возвращает «true» для нуля и «false» для любого положительного числа Чёрча. Подсказка: число n применяет функцию n раз. Если начать с «true» и применять функцию, всегда возвращающую «false», то «true» уцелеет только при $n = 0$.
13. Запишите `mult 23` и редуцируйте до нормальной формы с указанием каждого бета-шага.
14. * Постройте лямбда-терм «exp» для возведения в степень: m^n над числами Чёрча. Проверьте на `exp 23 = 8`.

Рекурсия и Y-комбинатор.

15. Выпишите определение Y-комбинатора. Покажите, что $YF \xrightarrow{\beta} F(YF)$ — выполните редукцию по шагам.
16. * Постройте лямбда-терм `pred` для предшественника числа Чёрча. Подсказка: используйте пары Чёрча:

$$\begin{aligned}\text{pair} &= \lambda \text{abs}. \text{sab} \\ \text{fst} &= \lambda p. p \text{ true} \\ \text{snd} &= \lambda p. p \text{ false}\end{aligned}$$

Начните с пары $(0, 0)$. На каждом шаге сдвигайте пару: $(a, b) \rightarrow (b, \text{succ } b)$. После n шагов первый элемент пары — $n - 1$.

17. * Используя Y-комбинатор и `pred` из предыдущей задачи, определите лямбда-терм для суммы чисел от 1 до n . Подсказка: тело рекурсии: `sum(n) = if (isZero n) 0 (add n (sum (pred n)))`.

Комбинаторная логика.

18. Выпишите три комбинатора S, K, I и их правила редукции. Покажите, что $SKKx \xrightarrow{\beta} x$ для любого x .
19. * Переведите лямбда-терм $\lambda x y. yx$ в комбинаторную логику алгоритмом абстракции скобок. Покажите каждый шаг перевода.
20. * Рассмотрим «доказательство» того, что каждый λ -терм имеет нормальную форму. Будем редуцировать терм по β , пока возможно. Каждый шаг имеет вид $(\lambda x. M)N \xrightarrow{\beta} M[x := N]$ и заменяет терм на меньший. Строгое убывание размера не может продолжаться бесконечно, поэтому процесс обязан закончиться нормальной формой. Где ошибка?
21. * Рассмотрим «доказательство» того, что β -редукция коммутативна: $M \xrightarrow{\beta} N \Rightarrow N \xrightarrow{\beta} M$. Если N получен из M заменой редекса $(\lambda x. P)Q \xrightarrow{\beta} P[x := Q]$, развер-

нём подстановку обратно — и обратный шаг готов. Раз каждый шаг редукции обратим, редукция симметрична, а значит, коммутативна. Где ошибка?

ПРОЕКТ Пошаговая редукция и сборка Y -комбинатора

Вычисление в бестиповом λ -исчислении — это переписывание: единственное действие — подстановка аргумента в тело функции (β -редукция), и из него вырастают числа Чёрча, булевы значения и рекурсия. Терм $\Omega = (\lambda x.xx)(\lambda x.xx)$ редуцируется сам в себя и не имеет нормальной формы. Комбинатор $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ — то же самоприменение с дополнительным применением f к результату. Соберите пошаговый редуктор и проследите, как одно отличие превращает расходящийся терм в рекурсию, вычисляющую факториал.

① Редуктор

Реализуйте термы λ -исчисления с подстановкой без захвата переменных: захват устраняется переименованием. Добавьте пошаговую β -редукцию нормального порядка со счётчиком шагов, лимитом глубины и печатью терма после каждого шага.

② Расходящийся терм и сборка Y

Проследите редукцию $\Omega = (\lambda x.xx)(\lambda x.xx)$: терм воспроизводит сам себя, нормальной формы нет, редуктор останавливается по лимиту шагов, а размер терма при этом постоянен. Зафиксируйте этот признак и объясните, чем он отличается от трассы растущего терма. Затем соберите комбинатор $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ и проверьте равенство $YF \xrightarrow{\beta} F(YF)$ по шагам. Реализуйте «почти факториал» $F = \lambda f.n.(\text{isZero } n)1(\text{mult } n(f(\text{pred } n)))$ — для этого понадобятся терм isZero и предшественник Клини через пары Чёрча.

③ Трассы и порядки редукции

Запустите редукцию факториала 4: трасса должна расти, накапливать умножения и свестись к числу Чёрча 24. Измерьте число шагов и максимальный размер терма и сравните с трассой Ω , которая редуцируется в себя без роста размера. Затем проверьте на редукторе, как выбранный порядок меняет трассу: терм $(\lambda x.y)\Omega$ в нормальном порядке завершается за один шаг, в аппликативном расходится. Объясните, как этот опыт согласуется с теоремой стандартизации.

④ Безымянные термы и эта-редукция

Переименование в подстановке устраняет захват переменных, но порождает книжную работу со свежими именами. Представьте термы безымянными: связанная переменная — это число — глубина её связывателя (0 — ближайший λ), а свободная остаётся по имени, и реализуйте преобразование обычного терма в безымянный и обратно. Реализуйте подстановку и β -редукцию прямо на безымянных термах: захват переменных там невозможен в принципе, его заменяют операции сдвига и подстановки индексов. Проверьте, что α -эквивалентные термы в безымянном виде совпадают, а

редукция безымянного терма согласуется с обычной. Затем добавьте η -редукцию $\lambda x.Mx \rightarrow M$ (когда x не свободна в M), выражающую экстенциональность — две функции равны, если совпадают на всех аргументах. Проверьте, что $\lambda x.fx$ редуцируется к f , а $\lambda x.xx$ не редуцируется — переменная занята, и сравните: β -редукция — вычисление, η — упрощение функции.

⑤ **Компиляция в SKI**

Три комбинатора S, K, I порождают всё λ -исчисление без переменных. Реализуйте компиляцию: $[x]x = I$, $[x]y = Ky$, $[x](MN) = S([x]M)([x]N)$. Проверьте, что скомпилированный терм редуцируется к тому же результату, что и исходный.

Итог: работающий пошаговый редуктор — именованный и безымянный — и сравнение трасс Ω и Y : самоприменение без применения функции расходится без роста размера, а с применением функции терм накапливает умножения и сводится к значению факториала.

XXVIII

Теория типов

В функцию, ожидающую число, передали строку — компилятор отклоняет программу ещё до запуска, не дав ей начать работать. *Типы* — первый барьер, который программа проходит до запуска.

В предыдущей главе (27) построено бестиповое λ -исчисление. Среди его термов есть безобидные — вроде тождественной функции $\lambda x.x$ — и есть проблемные: терм $\Omega = (\lambda x.xx)(\lambda x.xx)$ редуцируется сам в себя, и его вычисление никогда не завершается. Бестиповое исчисление не проводит различий между ними: любой терм применим к любому, поэтому $2\ \text{true}$ синтаксически корректен, хотя смысла не имеет, и Ω синтаксически корректен, хотя вычисления не завершает. Бестиповое λ -исчисление подобно ассемблеру: мощный, гибкий и совершенно безразличный к тому, что именно на нём написано.

Бестиповое исчисление не позволяет узнать это до запуска: по одному виду терма, не запуская Ω и не наблюдая его бесконечный цикл, о завершимости ничего сказать нельзя. Типы — способ дать такой ответ.

Каждому терму приписывается тип — спецификация: «на входе натуральное число, на выходе булево значение». Аппликация разрешена только тогда, когда типы согласуются. Функция типа $\text{Nat} \rightarrow \text{Bool}$ ожидает Nat . Дайте ей Bool — и программа не пройдёт проверку типов, ей даже не дадут запуститься. Типизация разводит числа, функции и булевы значения по классам: *этот* терм — функция из чисел в булевы значения, *этот* — число. Система типов следит, чтобы функция применялась только к аргументу подходящего типа.

Это похоже на систему типов в TypeScript или Rust, но проще и строже: если программа типизируется, она гарантированно завершается. Такая проверка — повседневность инженера: Rust, OCaml и TypeScript отсекают целые классы ошибок на этапе компиляции. Но типы могут больше, чем ловить несоответствие аргументов. В Idris, Coq и Lean типом записывают свойство: функция сортирует список, деление безопасно, программа завершается. Компилятор в этих системах и отклоняет бессмысленное, и проверяет утверждения.

У теории типов есть готовый аппарат: простые типы, правила типизации, деревья вывода и проверка типов. Два свойства — Subject Reduction и Strong Normalization — закрепляют правила: тип не меняется при вычислении, и типизированный терм не закикливается. Дальше типы растут в выразительности: полиморфизм, зависимые

типы, операторы над типами. На вершине этого роста стоят зависимые типы — там проверка типов становится проверкой утверждений о программе.

Идея, что тип — это утверждение, а терм — его доказательство, называется соответствием Карри–Ховарда. Соответствие работает именно с интуиционистской логикой: доказательства, кодируемые термами, конструктивны по своей природе (интуиционизм — в главе 34).

Если тип — утверждение, а терм — доказательство, то у типов две роли сразу: барьер для компилятора и гарантия для математика. *Типизируемость* становится гарантией: терм завершается, функция применяется только к подходящему аргументу, доказательство корректно. Но гарантии стоят универсальности: типизированное исчисление не выражает бесконечный цикл и потому не покрывает всё вычислимое. Для компилятора это цена, для верификатора — условие работы. Вопрос о том, что вычислимо и какой ценой, вернётся в главе 30. Что общего у проверки типов и проверки доказательств?

ИСТОРИЯ Типы против парадоксов

В 1901 году Бертран Рассел обнаружил, что наивная теория множеств разрушается одним вопросом: принадлежит ли множество всех множеств, не содержащих себя, самому себе? Любой из двух ответов приводит к противоречию, а сам вопрос заворачивается вокруг самоприменения — того же механизма, что и в нетипизируемом терме Ω . Выходом стала теория типов (1908): Рассел расслоил универсум на уровни, множество получило право собирать элементы только с уровня ниже, и вопрос о самопринадлежности потерял грамматический смысл¹⁹⁸

Чёрч перестроил расслоение на функциональной основе: простая теория типов (1940) — это λ -исчисление, в котором каждый терм несёт тип¹⁹⁹

Хаскелл Карри в 1934 году заметил, что типы комбинаторов читаются как схемы аксиом логики²⁰⁰ К тому времени Аренд Гейтинг уже формализовал интуиционистскую логику (1930) — исчисление конструктивных доказательств. Уильям Ховард в 1969 году превратил наблюдение Карри в точное соответствие «формулы как типы»: доказательство — терм, нормализация доказательства — вычисление²⁰¹

Второй этаж надстроили Жан-Ив Жирар и Джон Рейнольдс. Жирар построил полиморфное исчисление System F (глава вводит его как $\lambda 2$) в диссертации 1972 года²⁰² Джон Рейнольдс пришёл к той же системе независимо в 1974-м²⁰³

¹⁹⁸B. Russell. «Mathematical Logic as Based on the Theory of Types». American Journal of Mathematics, 1908.

¹⁹⁹A. Church. «A Formulation of the Simple Theory of Types». Journal of Symbolic Logic, 1940.

²⁰⁰H. B. Curry. «Functionality in Combinatory Logic». Proceedings of the National Academy of Sciences, 1934.

²⁰¹W. A. Howard. «The Formulae-as-Types Notion of Construction». To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980.

Попутно Жирар нашёл парадокс в первой (1971) версии теории типов Пера Мартина-Лёфа: тип всех типов воспроизводил самообъемлющие структуры, погубившие наивную теорию множеств. Ответом стала предикативная переработка (1972–1973): зависимые типы — семейства типов, индексированные термами, с конструкциями Π и Σ в роли основных²⁰⁴ Итоговое изложение Мартин-Лёф выпустил книгой в 1984 году, и именно эта система стоит за зависимыми типами в Agda и родственных языках²⁰⁵

Инженерный поворот выполнил Николаас де Брёйн: система Automath (с 1967 года) записывала математические доказательства λ -термами и проверяла их программно²⁰⁶ Три линии — вычислительная (Чёрч), логическая (Гейтинг, Ховард) и инженерная (де Брёйн) — сошлись в одной точке: расслоение, придуманное для защиты от парадокса, стало языком, на котором машина проверяет и программы, и доказательства теорем.

ОБЗОР ГЛАВЫ

В этой главе мы поставим вопрос: можно ли по виду терма узнать, что его вычисление никогда не завершится? Построим простые типы и правила типизации, научимся вести деревья вывода и проверять корректность типизации. Увидим, что Ω не типизируется и почему самоприменение упирается в противоречие. Свойства Subject Reduction и Strong Normalization объяснят, почему любая последовательность редукций типизированного терма конечна. Затем обнаружим, что тип комбинатора читается как тавтология, и соответствие Карри–Ховарда переведёт логические доказательства в λ -термы и обратно. По λ -кубу Барендрегта проследим, как растёт выразительность типов: от простых — к полиморфизму, зависимым типам и операторам над типами. В конце увидим, как пружинисты превращают теорию типов в формальную верификацию.

После этой главы вы сможете:

1. выводить типы термов простого типизированного λ -исчисления;
2. вести деревья вывода типизации и проверять их;
3. объяснять свойства Subject Reduction и Strong Normalization;
4. применять соответствие Карри–Ховарда;
5. объяснять, как зависимые типы превращают проверку типов в проверку утверждений.

²⁰²J.-Y. Girard. «Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur». 1972.

²⁰³J. C. Reynolds. «Towards a Theory of Type Structure». Colloque sur la Programmation, 1974.

²⁰⁴P. Martin-Löf. «An Intuitionistic Theory of Types: Predicative Part». Logic Colloquium „73, 1975.

²⁰⁵P. Martin-Löf. «Intuitionistic Type Theory». 1984.

²⁰⁶N. G. de Bruijn. «Automath, a Language for Mathematics». 1970.

§ 28.1 Простые типы

Определим множество допустимых типов. Тип — это классификация терма по его вычислительному поведению.

Базовые типы — это атомы: Nat (натуральные числа) и Bool (булевы значения). Мы не уточняем их внутреннюю структуру — только различаем имена. Функциональный тип $\sigma \rightarrow \tau$ описывает функцию, которая принимает аргумент типа σ и возвращает результат типа τ .

ОПРЕДЕЛЕНИЕ 28.1 Простые типы

Множество **простых типов** Type строится индуктивно:

1. Если ι — базовый тип, то $\iota \in \text{Type}$.
2. Если $\sigma, \tau \in \text{Type}$, то и функциональный тип принадлежит Type :

$$(\sigma \rightarrow \tau) \in \text{Type}$$

Никаких других типов не существует.

Пример Чтение функциональных типов

- $\text{Bool} \rightarrow \text{Bool}$ — функция из булевых в булевы: отрицание, тождественная функция.
- $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ — функция двух аргументов (каррированная): сложение, умножение. Скобки право-ассоциативны: это $\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat})$.
- $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat}$ — функция, принимающая *функцию* и возвращающая число: например, применение переданной функции к фиксированному аргументу.
- $(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool}$ — функция, принимающая унарную операцию над булевыми и булево значение, и возвращающая булево.

Приведём конкретные термы.

- $\lambda x : \text{Nat} . x + 1$ — функция из натуральных чисел в натуральные.
- Каррирование: функция двух аргументов — это функция от первого, возвращающая функцию от второго. Сложение имеет тип $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$:

$$\lambda x : \text{Nat} . \lambda y : \text{Nat} . x + y$$

- $\lambda f : \text{Nat} \rightarrow \text{Nat} . f 5$ — применение переданной функции к числу 5. Его тип: $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat}$.

Заметим параллель с логикой. Тип $\sigma \rightarrow \tau$ — та же стрелка, что и логическая импликация $\sigma \rightarrow \tau$. «Если σ , то τ » — в точности тип функции, которая по σ строит τ . Ту же стрелку можно прочесть разговорно: «дайте мне σ — верну τ ». Например, $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat}$ — «дайте мне функцию из чисел в числа, верну число». Это не случайное совпадение — к его смыслу мы вернёмся позже.

§ 28.2 Правила типизации

Мы вводим три правила, определяющих корректно типизированные термы. Запись $\Gamma \vdash M : \sigma$ читается: «в контексте Γ терм M имеет тип σ ». Контекст Γ — множество предположений о свободных переменных. Предположения имеют вид $x : \tau$.

Логическое суждение $\Gamma \vdash \sigma$ утверждает: вывод существует. Но оно не называется само доказательство. Типовая запись $\Gamma \vdash M : \sigma$ добавляет терм M . Доказательство становится объектом, который можно предъявить, проверить и запустить.

Правила — спецификация, не алгоритм. Они говорят *что* значит быть правильно типизированным. Алгоритм проверки — отдельная инженерная задача (и для $\lambda \rightarrow$ она решается за линейное время).

ОПРЕДЕЛЕНИЕ 28.2 Правила типизации $\lambda \rightarrow$

1. **var**: переменная имеет тип, приписанный ей в контексте.

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \text{var}$$

2. **app**: применение функции M к аргументу N . Если M имеет тип $\sigma \rightarrow \tau$, а N — тип σ , то результат имеет тип τ .

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \text{app}$$

3. **abs**: если тело M имеет тип τ при предположении $x : \sigma$, то λ -абстракция — функция из σ в τ .

$$\frac{(\Gamma, x : \sigma \vdash M : \tau)}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} \text{abs}$$

ЛОВУШКА Тип $\neq$ значение

Тип терма легко перепутать с его значением — с результатом вычисления. Тип — это инвариант: класс термов с общим вычислительным поведением, а не конкретный результат. Термы $2 + 3 : \text{Nat}$, $5 : \text{Nat}$ — разные, но одного типа. Тип говорит, что с термом можно делать, а не что он возвращает в этом запуске.

Каждое правило читается как шаг логического вывода: над чертой стоят посылки — условия, которые должны выполняться — а под чертой заключение, суждение о типе, которое из них следует. Три правила соответствуют трём синтаксическим формам терма: переменной, аппликации и абстракции. Разберём каждое правило на конкретных примерах.

Начнём с переменной. Если контекст содержит $x : \text{Nat}$, правило **var** выводит $\Gamma \vdash x : \text{Nat}$. Если в контексте есть $f : \text{Nat} \rightarrow \text{Bool}$, правило **var** даёт $\Gamma \vdash f : \text{Nat} \rightarrow \text{Bool}$.

Перейдём к аппликации. Пусть $f : \text{Nat} \rightarrow \text{Bool}$, а $x : \text{Nat}$. Тогда $fx : \text{Bool}$. Чтобы применить f к аргументу, тип аргумента должен совпадать с типом входа f . Если аргумент имеет тип Bool , а f ожидает Nat , правило app неприменимо. Терм $f \text{ true}$ не типизируется.

Осталось правило абстракции. Если мы знаем, что $x + 1 : \text{Nat}$ при предположении $x : \text{Nat}$, то абстракция типизируется:

$$\lambda x : \text{Nat} . x + 1 : \text{Nat} \rightarrow \text{Nat}$$

Аннотация типа при связывании переменной — это Church-типизация: тип переменной записан прямо в терме.

Примечание Стилль Чёрча и стилль Карри

Правилам типизации соответствуют две дисциплины записи термов.

1. Church-стилль: аннотации при связывателях, $\lambda x : \sigma . M$. Тип терма единствен, и проверка предъявленного типа — линейный проход по терму.
2. Curry-стилль: термы без аннотаций, $\lambda x . M$. Типы восстанавливаются автоматически, у терма бывает несколько допустимых типов, и среди них есть главный (principal) — все остальные получаются из него подстановкой.

Для $\lambda \rightarrow$ классы типизируемых термов в двух дисциплинах совпадают, различаются алгоритмы: проверка предъявленного типа против поиска типа. Поиск типов в Curry-стиле вырос в алгоритм Hindley–Milner — механизм вывода типов в ML и Haskell.

Три правила, и никаких других способов приписать тип не существует: терм корректен, если и только если существует дерево вывода, построенное по этим трём правилам.

Пример Тожественная функция

Построим вывод типа для $\lambda x : \text{Nat} . x$.

1. По правилу var : $x : \text{Nat} \vdash x : \text{Nat}$.
2. По правилу abs (применённому к предыдущей строке): $\dots \vdash \lambda x : \text{Nat} . x : \text{Nat} \rightarrow \text{Nat}$.

Тожественная функция на натуральных числах имеет тип $\text{Nat} \rightarrow \text{Nat}$.

Пример Проекция на первый аргумент

Построим вывод типа для $\lambda x : \text{Nat} . \lambda y : \text{Bool} . x$.

1. По правилу var : $x : \text{Nat}, y : \text{Bool} \vdash x : \text{Nat}$.
2. По правилу abs по y : $x : \text{Nat} \vdash \lambda y : \text{Bool} . x : \text{Bool} \rightarrow \text{Nat}$.
3. По правилу abs по x : $\dots \vdash \lambda x : \text{Nat} . \lambda y : \text{Bool} . x : \text{Nat} \rightarrow \text{Bool} \rightarrow \text{Nat}$.

Тип этой функции: $\text{Nat} \rightarrow \text{Bool} \rightarrow \text{Nat}$. Функция принимает Nat , затем Bool . Второй аргумент игнорируется: возвращается исходное число.

Визуализируем деревья вывода для этих двух примеров. Каждое дерево линейно: на каждом шаге одна посылочная ветвь, в основании аксиома var , а каждое применение правила abs добавляет одно заключение. У тождественной функции два шага: правило var и одно применение abs , и её дерево показано на рис. 111. У комбинатора K шагов три: правило var и два применения abs , и K проектирует на первый аргумент, как на рис. 112. Линейность не особый случай: каждое из трёх правил типизации имеет ровно одну посылочную ветвь, так что типовой вывод в $\lambda \rightarrow$ всегда разворачивается в цепочку.

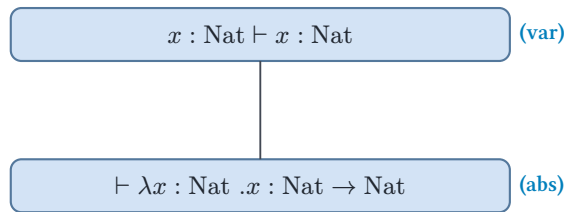


Рис. 111. Дерево вывода типа для тождественной функции.

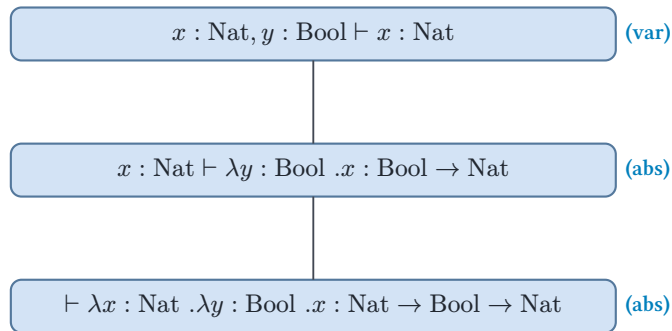


Рис. 112. Дерево вывода типа для комбинатора K .

Пример Проекция в общей форме

Тот же вывод в общей форме, с произвольными типами σ и τ .

1. По правилу var : $x : \sigma, y : \tau \vdash x : \sigma$.
2. По правилу abs по y : $x : \sigma \vdash \lambda y : \tau. x : \tau \rightarrow \sigma$.
3. По правилу abs по x : $\dots \vdash \lambda x : \sigma. \lambda y : \tau. x : \sigma \rightarrow \tau \rightarrow \sigma$.

Тип $\sigma \rightarrow \tau \rightarrow \sigma$ — аксиома минимальной логики. Это формула $A \rightarrow B \rightarrow A$, одна для всех σ и τ . Конкретные типы Nat и Bool — частный случай этой схемы.

Дерево вывода — формальный объект, а не иллюстрация: каждый узел дерева — применение одного из трёх правил типизации. Корректность типизации — это и есть существование такого дерева, а проверка типов — алгоритмический поиск дерева вывода.

Проверка Правила типизации

1. Почему $\lambda x : \text{Nat} . \lambda y : \text{Bool} . x$ типизируется, хотя переменная y в теле не используется?
2. Терм $f \text{ true}$ не типизируется, когда $f : \text{Nat} \rightarrow \text{Bool}$. Какое из трёх правил отказывает и почему?

§ 28.3 Что можно и что нельзя типизировать

Не всякий λ -терм типизируется в $\lambda \rightarrow$ — это ровно то свойство, ради которого система создавалась. Рассмотрим несколько характерных случаев.

Комбинатор $K = \lambda x . \lambda y . x$ типизируется. В Church-стиле: $\lambda x : \sigma . \lambda y : \tau . x : \sigma \rightarrow \tau \rightarrow \sigma$. Тип сообщает, что для любых конкретных типов σ и τ существует такой терм.

Сам терм монотипен (*monomorphic*): в $\lambda \rightarrow$ нет квантора по типам. σ и τ — метаварьируемые вне языка, так что K — семейство монотипных термов, по одному на пару типов. Настоящий полиморфизм появится лишь в $\lambda 2$.

Второй аргумент игнорируется, а тип $\sigma \rightarrow \tau \rightarrow \sigma$ — одна из аксиом минимальной логики высказываний. Это импликационный фрагмент без закона исключённого третьего и без *ex falso*: $A \rightarrow (B \rightarrow A)$.

Комбинатор $S = \lambda x . \lambda y . \lambda z . xz(yz)$ тоже типизируется. В Church-стиле:

$$S = \lambda x : \sigma \rightarrow \tau \rightarrow \rho . \lambda y : \sigma \rightarrow \tau . \lambda z : \sigma . xz(yz)$$

Его тип: $(\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \rho$.

Выведем этот тип по шагам. В контексте $x : \sigma \rightarrow \tau \rightarrow \rho, y : \sigma \rightarrow \tau, z : \sigma$:

1. $xz : \tau \rightarrow \rho$ (app: x к z).
2. $yz : \tau$ (app: y к z).
3. $xz(yz) : \rho$ (app: xz к yz).

Абстрагируем z, y, x — получаем указанный тип. Тип комбинатора S — вторая аксиома гильбертовского исчисления высказываний.

А теперь — терм $\lambda x . xx$. Попробуем приписать ему тип. Пусть $x : \sigma$. Для аппликации xx нужно, чтобы x одновременно имел тип $\sigma \rightarrow \tau$ (как функция) и σ (как аргумент). Получаем уравнение: $\sigma = \sigma \rightarrow \tau$.

Уравнение не имеет решения. Тип не может быть равен собственному функциональному типу — правая часть всегда структурно сложнее левой. Итак, $\lambda x . xx$ не типизируется в $\lambda \rightarrow$.

Следовательно, и $\Omega = (\lambda x . xx)(\lambda x . xx)$ не типизируется. Терм, порождающий бесконечный цикл, исключён из множества корректных программ. Самоприменение несовместимо с правилами согласования типов, и именно это его исключает.

Замечание Почему Ω не типизируется

Локальная невозможность — частный случай общего принципа: каждое правило типизации *движет* тип к меньшей структурной сложности. Тип $\sigma \rightarrow \tau$ содержит σ и τ как подвыражения, а аппликация из $\sigma \rightarrow \tau$ и σ возвращает τ — подвыражение. Вдоль цепочки редукций тип может только упрощаться. Бесконечная цепочка потребовала бы вернуться к более сложному типу, а этого правила не позволяют.

На этой интуиции построено доказательство сильной нормализации в следующем разделе (28.4): там мера сложности задаётся формально, методом редуцируемости Тейта. Полное доказательство значительно сложнее.

§ 28.4 Свойства типизированного λ -исчисления

Два свойства делают $\lambda \rightarrow$ качественно отличным от бестипового исчисления.

Первое: тип не меняется при вычислении.

ТЕОРЕМА 28.1 Subject Reduction

Если $\Gamma \vdash M : \sigma$ и $M \xrightarrow{\beta} M'$, то тип сохраняется: $\Gamma \vdash M' : \sigma$.

Тип — статическая характеристика программы. Он устанавливается до запуска и остаётся верным на каждом шаге вычисления. Тип переменной в программе ведёт себя так же: вы объявили $x : \text{int}$, и сколько бы операций ни выполнялось, x остаётся целым числом.

Набросок доказательства

Докажем лемму о подстановке индукцией по структуре терма, а из неё выведем Subject Reduction для одного бета-шага.

Лемма о подстановке: если $\Gamma, x : \tau \vdash P : \sigma$ и $\Gamma \vdash Q : \tau$, то $\Gamma \vdash P[x := Q] : \sigma$. Подстановка терма нужного типа вместо переменной сохраняет тип всего выражения.

Докажем лемму индукцией по структуре P . **Случай 1.** Если P — переменная x , то $P[x := Q] = Q$, и тип Q совпадает с типом x по предположению. **Случай 2.** Если P — другая переменная, подстановка не меняет терм, и тип сохраняется. **Случай 3.** Если P — аппликация $P_1 P_2$, то по индукции типы $P_1[x := Q]$ и $P_2[x := Q]$ совпадают с типами P_1 и P_2 , а правило типизации для аппликации гарантирует сохранение типа результата. **Случай 4.** Если P — абстракция $\lambda y. M$, рассуждаем по индукции с учётом α -конверсии для избежания захвата.

Из леммы о подстановке непосредственно следует Subject Reduction для одного шага β -редукции. Единственное правило редукции — $(\lambda x.M)N \xrightarrow{\beta} M[x := N]$, поэтому если $\lambda x.M : \sigma \rightarrow \tau$ и N имеет тип σ , то по лемме $M[x := N]$ имеет тип τ .

Subject Reduction говорит, что тип — инвариант вычисления: на каждом шаге редукции тип сохраняется. Но сам по себе инвариант не запрещает вычислению тянуться бесконечно: тип может сохраняться при бесконечной цепочке редукций.

ЛОВУШКА Тип терма $\neq$ тип результата

Тип легко принять за свойство значения: вычислить терм — и посмотреть, что получилось. Но тип приписан синтаксису: он выводится по правилам типизации, ещё до запуска, и от конкретного значения не зависит. $(\lambda x : \text{Nat} . x + 1)3 \xrightarrow{\beta} 4$. Терм изменился, тип Nat остался. Это Subject Reduction: тип сохраняется на каждом шаге редукции. Значение может меняться, тип — нет.

Второе свойство ставит барьер и против вечных вычислений: терм, типизированный в $\lambda \rightarrow$, не может заиклиться.

ТЕОРЕМА 28.2 Strong Normalization

Каждый корректно типизированный терм в $\lambda \rightarrow$ сильно нормализуем: любая последовательность β -редукций конечна и завершается нормальной формой.

Набросок доказательства

Докажем методом редуцируемости (по Тейту): класс *редуцируемых* термов задаётся индукцией по типу, и каждый типизируемый терм оказывается редуцируемым.

Терм типа Nat редуцируем, если он сильно нормализуем, а терм M типа $\sigma \rightarrow \tau$ редуцируем, если для любого редуцируемого терма $N : \sigma$ аппликация MN редуцируема.

Из определения следуют два факта. Первый факт: каждый редуцируемый терм сильно нормализуем. Докажем его индукцией по типу. На типе $\sigma \rightarrow \tau$ берём свежую переменную $x : \sigma$ (переменные редуцируемы): аппликация Mx редуцируема и сильно нормализуется по индуктивному предположению, а вместе с ней и M .

Второй факт: каждый типизируемый терм редуцируем. Он доказывается индукцией по дереву вывода типизации, где основная работа — показать, что класс редуцируемых термов замкнут относительно подстановки и аппликации.

Из этих двух фактов следует утверждение теоремы.

В $\lambda \rightarrow$ невозможно написать бесконечный цикл: какую бы стратегию редукции вы ни выбрали, вычисление всегда завершится.

Конструкция редуцируемых термов принадлежит Уильяму Тейту (1967). Мы наметили только каркас: полное доказательство замкнутости класса редуцируемых термов относительно подстановки мы здесь проводить не будем. Логическое следствие принципиально: $\lambda \rightarrow$ не Тьюринг-полон. Он вычисляет широкий класс функций — расширенные полиномы — но не все вычислимые функции.

Расширенные полиномы — функции, растущие полиномиально и собираемые из нуля, сложения, умножения и условного выражения: без циклов, без рекурсии, без самоприменения. Возведение в степень в этот класс не входит: терм $\text{exp} = \lambda m n. n m$ применяет число к числу, а простые типы такое применение *не пропускают*.

Обратная сторона гарантий — *потеря универсальности*. Рекурсия, Y-комбинатор, неподвижные точки — всё это требует выхода за пределы $\lambda \rightarrow$. Полиморфизма System F недостаточно: нужны рекурсивные типы или явный оператор неподвижной точки. И Y-комбинатор, и Ω строятся на самоприменении, а оно в простых типах не типизируется. В этом причина, по которой $\lambda \rightarrow$ не Тьюринг-полон.

Примечание Строгость и выразительность

Компромисс, знакомый каждому разработчику: чем строже система типов, тем больше ошибок она отлавливает на этапе компиляции — и тем труднее выразить некоторые корректные программы. Современные языки (Haskell, Rust, OCaml) ищут баланс: типы достаточно богаты, чтобы выразить рекурсию и полиморфизм, но достаточно строги, чтобы исключить целые классы ошибок.

§ 28.5 Соответствие Карри–Ховарда

Доказательство несёт знание об истинности утверждения и одновременно является объектом со структурой, которую можно запустить, как программу. Соответствие Карри–Ховарда показывает, какую именно структуру.

Три правила типизации $\lambda \rightarrow$ — `var`, `app`, `abs`. Три правила натурального вывода минимальной логики высказываний — аксиома, `modus ponens` и $\rightarrow$ -введение. Совпадение полное: замените типовое двоеточие на знак выводимости, стрелку типа на импликацию, терм на доказательство — получите систему логического вывода. Замените логические правила на правила построения λ -термов — получите систему типизации.

Это изоморфизм: взаимно-однозначное соответствие, сохраняющее все операции.

ОПРЕДЕЛЕНИЕ 28.3 Соответствие Карри–Ховарда

Соответствие Карри–Ховарда (Curry–Howard correspondence) — изоморфизм между теорией типов и логикой.

Теория типов	Логика
Тип σ	Высказывание σ
Терм $M : \sigma$	Доказательство M утверждения σ
$\sigma \rightarrow \tau$	Импликация $\sigma \rightarrow \tau$
$\lambda x : \sigma. M$	$\rightarrow$ -введение: предположили σ , доказали τ
MN	$\rightarrow$ -удаление (modus ponens): из $\sigma \rightarrow \tau$ и σ получаем τ
Контекст Γ	Множество предположений
Редукция терма	Нормализация доказательства
Нормальная форма	Доказательство без «обходных путей» (detours)

Каждая строка таблицы — точное структурное соответствие.

Рассмотрим три конкретных примера.

Пример Тождественная функция как тавтология $A \rightarrow A$

Терм $\lambda x : A. x : A \rightarrow A$.

В логической интерпретации: Тип $A \rightarrow A$ — формула «если A , то A ». Терм $\lambda x : A. x$ — доказательство этой формулы.

Читается это доказательство так: «Предположим A (это x). Тогда A верно (возвращаем x). Следовательно, $A \rightarrow A$ (абстрагируем x).»

Терм и доказательство говорят одно и то же — разными словами, но с одной и той же структурой.

Тождественная функция — вырожденный случай: единственное предположение переходит в вывод без изменений. Следующий пример показывает, как ведёт себя функция с двумя предположениями.

Пример Комбинатор K как аксиома $A \rightarrow B \rightarrow A$

Терм $\lambda x : A. \lambda y : B. x : A \rightarrow B \rightarrow A$.

В логической интерпретации: из A следует, что B влечёт A . Это одна из аксиом гильбертовского исчисления высказываний.

Доказательство конструктивно: «Предположим A . Предположим B . Но A у нас уже есть (это x). Игнорируем B и возвращаем A .»

Второй аргумент $y : B$ не используется в теле функции — так же как предположение B не нужно для вывода A , если A уже известно. Тип фиксирует факт

неиспользования: функция принимает B , но её результат от B не зависит. Если A уже доказано, никакая новая гипотеза не сможет этого отменить.

Комбинатор K показывает, как доказательство избавляется от ненужной посылки. Композиция — противоположный приём: посылки выстраиваются в цепочку, и вывод одной становится входом другой.

Пример Композиция как транзитивность импликации

Терм $\lambda f : A \rightarrow B. \lambda g : B \rightarrow C. \lambda x : A. g(fx)$ имеет тип $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C$.

В логической интерпретации: если из A следует B , и из B следует C , то из A следует C . Это транзитивность импликации.

Структура доказательства в точности воспроизводит структуру терма: берём $x : A$ и применяем f , получаем $fx : B$, затем применяем g и получаем $g(fx) : C$. Затем абстрагируем x, g, f — ровно в том порядке, в котором снимаются предположения в натуральном выводе.

Пример Контракция: гипотеза использована дважды

Терм $\lambda f : A \rightarrow A \rightarrow B. \lambda x : A. fxx$ типизируется. Его тип — $(A \rightarrow A \rightarrow B) \rightarrow A \rightarrow B$. Логическое прочтение: если из A дважды выводится B , то из одного A выводится B . Это правило *контракции* — допустимое использование гипотезы повторно. Переменная x встречается в терме дважды: fxx . Каждое вхождение — отдельное применение гипотезы A в доказательстве. Структура терма повторяет структуру вывода: предположили f , предположили x , применили f к x , применили результат к x ещё раз.

Теперь соберём изоморфизм целиком, на языке гильбертовского исчисления высказываний. Типы комбинаторов K и S — аксиомы: $A \rightarrow B \rightarrow A$, $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$. Аппликация — правило *modus ponens*: из терма типа $\sigma \rightarrow \tau$ и терма типа σ получаем терм типа τ . Любой λ -терм, собранный из K и S одними аппликациями, — это доказательство в гильбертовском исчислении. И обратно: у каждой теоремы минимальной логики высказываний есть терм-свидетель.

Возьмём теорему, в которой посылки переставляются: $(A \rightarrow B \rightarrow C) \rightarrow (B \rightarrow A \rightarrow C)$. Доказательство в натуральном выводе: предположим $A \rightarrow B \rightarrow C$, предположим B , предположим A . Из $A \rightarrow B \rightarrow C$ и A по *modus ponens* получаем $B \rightarrow C$. Из $B \rightarrow C$ и B снова по *modus ponens* получаем C . Снимаем предположения в обратном порядке и получаем терм $\lambda x : A \rightarrow B \rightarrow C. \lambda y : B. \lambda z : A. xzy$ — то же рассуждение, где абстракция — введение импликации, а аппликация — *modus ponens*.

Оба приёма уже встречались в главе 1. Теорема о дедукции — доказать $P \rightarrow Q$, предположив P и выведя Q . Прямое доказательство ведёт от посылок к заключению

цепочкой применений *modus ponens*. Карри–Ховард превращает эти два приёма в правила типизации: введение импликации — в абстракцию, *modus ponens* — в аппликацию. Натуральный вывод, гильбертовское исчисление и типизация — три записи одних и тех же утверждений.

Два мира, развивавшиеся независимо, сошлись в одной структуре. У этой параллели есть точная граница. Закон Пирса $((A \rightarrow B) \rightarrow A) \rightarrow A$ — классическая тавтология: если из импликации $A \rightarrow B$ следует A , то верно и само A .

УТВЕРЖДЕНИЕ 28.3 Необитаемость закона Пирса

В $\lambda \rightarrow$ не существует замкнутого терма типа $((A \rightarrow B) \rightarrow A) \rightarrow A$.

Доказательство

Докажем анализом нормальной формы: предположим обитателя, приведём его к нормальной форме и покажем, что замкнутая нормальная форма указанного типа невозможна. По теореме о сильной нормализации приведение всегда завершается, так что достаточно разобрать нормальные формы.

Замкнутая нормальная форма не может быть переменной: свободных переменных у неё нет. Она не может быть и аппликацией: голова аппликативной цепочки в нормальной форме — переменная, а свободных переменных нет. Остаётся абстракция: $M = \lambda f.N$, где $f : (A \rightarrow B) \rightarrow A$, а N — нормальная форма типа A в контексте f .

Терм N обязан быть применением fP : переменная f — единственная в контексте, её тип функциональный, и только её применение даёт тип A . Аргумент P обязан иметь тип $A \rightarrow B$. Терм P не может быть переменной f : у той тип $(A \rightarrow B) \rightarrow A$. Он не может быть и аппликацией: применение f даёт тип A , а нужен $A \rightarrow B$. Значит, P — абстракция: $P = \lambda a : A.Q$ с $Q : B$ в контексте $f, a : A$.

Но значение типа B из этого контекста не построить: a несёт тип A , применение f к чему угодно даёт тип A , и другого материала нет. Разбор случаев завершён, и ни Q , ни P , ни M не существует.

Необитаемость закона Пирса очерчивает рамку соответствия: обитаемые типы — в точности теоремы минимальной логики, и каждый обитатель является конструктивным доказательством. Классическая логика выводит эту схему через исключённое третье или двойное отрицание, а такие рассуждения не строят терм: доказательство существования без предъявления объекта в $\lambda \rightarrow$ не выражается.

Теорема о сильной нормализации $\lambda \rightarrow$ получила логическую интерпретацию: каждое доказательство в минимальной логике высказываний может быть приведено к нормальной форме — доказательству без «обходных путей» (*detours*).

Конкретный пример: $(\lambda f : A \rightarrow A. \lambda x : A. fx)(\lambda x : A. x)$. Функция ждёт функцию и применяет её к x , аргумент — тождественная функция. Введение импликации и её

немедленное удаление стоят здесь рядом, и бета-редукция *склеивает* их в один шаг: остаётся прямое доказательство без обходного пути. Это ситуации, когда формула сначала вводится через $\rightarrow$ -введение, а затем немедленно удаляется через $\rightarrow$ -удаление.

В терминах λ -исчисления это редекс: $(\lambda x : A.M)N \xrightarrow{\beta} M[x := N]$. Подстановка N вместо x устраняет обходной путь — доказывает то же утверждение более прямым способом.

Соответствие Карри–Ховарда формулируется и доказывается как математическая теорема. Этот факт изменил то, как мы понимаем доказательства, программы и связь между ними. У соответствия два рабочих направления. Доказываете теорему — построенный терм и есть программа. Пишете типизируемую программу — её тип и есть доказанная теорема. Компилятор, принимающий программу, — это верификатор, принявший доказательство.

Проверка Соответствие Карри–Ховарда

1. Что такое «обходной путь» (detour) в доказательстве, и какой β -редекс ему соответствует?
2. Почему терм композиции $\lambda f : A \rightarrow B. \lambda g : B \rightarrow C. \lambda x : A. g(fx)$ можно прочитать как доказательство транзитивности импликации?
3. Почему закон Пирса $((A \rightarrow B) \rightarrow A) \rightarrow A$ не имеет терма-обитателя и что этот факт говорит о рамках соответствия?

§ 28.6 За пределы $\lambda \rightarrow$

Соответствие Карри–Ховарда можно расширять, добавляя новые конструкции в типовую систему:

1. Добавим пары (тип-произведение $\sigma \times \tau$) — получим конъюнкцию $\sigma \wedge \tau$. Терм-свидетель: $\lambda x : \sigma. \lambda y : \tau. (x, y) : \sigma \rightarrow \tau \rightarrow \sigma \times \tau$.
2. Добавим суммы (тип-сумма, копроизведение: $\sigma + \tau$) — получим дизъюнкцию $\sigma \vee \tau$. Терм-свидетель (левая инъекция): $\lambda x : \sigma. \text{inl } x : \sigma \rightarrow \sigma + \tau$.
3. Добавим пустой тип $\perp$ — получим ложь и принцип ex falso quodlibet. Терм-свидетель: $\lambda x : \perp. \text{absurd } x : \perp \rightarrow \sigma$.

Логические имена — не произвол: пара *населена* ровно тогда, когда населены оба компонента, а это и есть конъюнкция. Инъекция «inl» населяет сумму, предъявив левый компонент: так доказывается дизъюнкция. Пустой тип $\perp$ не населён: доказательств лжи нет, и из гипотезы о нём следует что угодно (ex falso).

Но следующий шаг иного рода: разрешим типам зависеть от термов. Это приводит к зависимым типам и λ -кубу Барендрегта (1991) — системе из восьми типовых систем, различающихся тем, какие виды зависимостей в них разрешены.

Три оси куба:

1. $\lambda \rightarrow$ (базовая) — термы зависят от термов: функция принимает терм и возвращает терм.
2. $\lambda 2$ (System F, полиморфизм) — термы зависят от типов.
3. λP (зависимые типы) — типы зависят от термов. Тип $\text{Vec } n$ (вектор длины n) меняется в зависимости от значения n . Конкатенация векторов: $\text{concat} : \Pi n m : \text{Nat} . \text{Vec } n \rightarrow \text{Vec } m \rightarrow \text{Vec } (n + m)$.
4. $\lambda \omega$ (операторы над типами) — типы зависят от типов. Можно определить $\text{List} : \star \rightarrow \star$ — оператор, отображающий тип элементов в тип списка.

ОПРЕДЕЛЕНИЕ 28.4 Зависимый функциональный тип Π

Пусть A — тип, а $B(x)$ — семейство типов, по одному на каждый элемент $x : A$. **Зависимый функциональный тип** $\Pi x : A. B(x)$ населяют функции, у которых результат на каждом $a : A$ лежит в своём типе семейства: $f(a) : B(a)$. Такой тип вводится абстракцией $\lambda x : A. M$, а применение работает обычным образом: $f a : B(a)$. Если B от x не зависит, зависимость вырождается в обычную стрелку: $\Pi x : A. B = A \rightarrow B$.

ОПРЕДЕЛЕНИЕ 28.5 Зависимое произведение Σ

Пусть A — тип, а $B(x)$ — семейство типов по $x : A$. **Зависимое произведение** (зависимая пара) $\Sigma x : A. B(x)$ населяют пары (a, b) , у которых первый компонент лежит в A , а второй — в типе, подобранном по первому: $b : B(a)$. Пару собирает спаривание, а компоненты извлекают проекциями π_1 и π_2 : из $p : \Sigma x : A. B(x)$ получаем $\pi_1(p) : A$ и $\pi_2(p) : B(\pi_1(p))$. Если B от x не зависит, зависимость вырождается: $\Sigma x : A. B = A \times B$.

По соответствию Карри–Ховарда конструкциям Π и Σ достаются роли кванторов: $\Pi x : A. B(x)$ читается «для всякого x из A верно $B(x)$ », а $\Sigma x : A. B(x)$ — «существует x из A , для которого верно $B(x)$ ». Отличие от логики первого порядка в том, что квантор пробегает по элементам типа, и доказательство существования обязано предъявить пару: элемент и свидетельство для него.

Пример Зависимая пара: чётность

Утверждение « n чётно» кодируется семейством типов:

$$\text{Even } n = \Sigma k : \text{Nat} . n = 2 \cdot k$$

Элемент $\text{Even } n$ — зависимая пара: число k и свидетельство равенства $n = 2 \cdot k$. Равенство здесь — тоже тип, и населён он ровно тогда, когда равенство верно, поэтому пары существуют в точности для чётных чисел. Для $n = 4$ пара берёт $k = 2$, для $n = 5$ ни одно k равенства не даёт, и тип пуст. Отсюда честная сигнатура деления пополам:

$$\text{half} : \Pi n : \text{Nat} . \text{Even } n \rightarrow \text{Nat}$$

Функция принимает число вместе с доказательством его чётности и возвращает половину — первый компонент пары. Применить `half` к пятёрке невозможно: терм типа `Even 5` не построить.

ЛОВУШКА Тип — не булева функция

Зависимый тип легко принять за функцию, возвращающую булево значение. Тип `Even  $n$` из примера выше — не функция из чисел в `Bool`: булевозначный предикат выдаёт метку, а зависимый тип населён ровно тогда, когда утверждение верно. Населённость типа — это доказуемость утверждения. Терм типа `Even  $n$` и есть доказательство чётности числа n .

Пример Вектор: тип зависит от значения

Тип `Vec  $n$` зависит от значения n : это семейство типов, по одному на каждую длину. `Vec 0`, `Vec 1`, `Vec 2`, ... — разные типы, и смешение длин проверки не пройдёт.

Семейство задают два конструктора:

1. `nil : Vec 0` — пустой вектор нулевой длины.
2. `cons :  $\Pi n : \text{Nat} . \text{Nat} \rightarrow \text{Vec } n \rightarrow \text{Vec } (n + 1)$` — приписывает элемент спереди.

Построим вектор из двух чисел и проследим, как длина живёт в типе.

1. `nil : Vec 0`.
2. `cons 05 nil : Vec (0 + 1)`. Свернём `0 + 1` в типе: это `Vec 1`.
3. `cons 17(cons 05 nil) : Vec (1 + 1)`. Свернём `1 + 1` в типе: это `Vec 2`.

Чтобы проверить последний шаг, система типов должна *вычислить* `1 + 1` внутри типа. Проверка типов при зависимых типах — это и сопоставление синтаксиса, и частичное вычисление. В $\lambda \rightarrow$ и $\lambda 2$ так невозможно: тип там вообще не ссылается на термы-значения.

ОПРЕДЕЛЕНИЕ 28.6 System F ($\lambda 2$)

Система $\lambda 2$ (System F) расширяет $\lambda \rightarrow$ зависимостью термов от типов. Типы пополняются схемой: вместе с типом σ , содержащим типовую переменную α , разрешён тип $\forall \alpha. \sigma$. Термы пополняются двумя формами, и правил типизации два:

1. абстракция по типу $\Lambda \alpha. M$ получает тип $\forall \alpha. \sigma$, когда тело M типизируется как σ .
2. применение к типу $M[\tau]$ получает тип $\sigma[\alpha := \tau]$, когда M типизируется как $\forall \alpha. \sigma$.

Пример Полиморфная тождественная функция

Терм $\Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \rightarrow \alpha$. Подставим вместо α тип Nat . Получим тождественную функцию на числах: $\lambda x : \text{Nat}.x : \text{Nat} \rightarrow \text{Nat}$. Подставим Bool . Получим $\lambda x : \text{Bool}.x : \text{Bool} \rightarrow \text{Bool}$. Один и тот же терм служит всем типам сразу. В $\lambda \rightarrow$ каждая копия тождественной функции — отдельный терм со своим фиксированным типом.

Пример Числа Чёрча

Число 2 кодируется термом $2 = \Lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.f(fx)$. Его тип — ровно $\text{Nat} = \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. Число применяет аргумент-функцию n раз, и тип это фиксирует. Покажем один шаг редукции на конкретных типах. Применим 2 к функции $g : \text{Nat} \rightarrow \text{Nat}$ и к числу 0:

$$2g0 = (\lambda f : \text{Nat} \rightarrow \text{Nat}.\lambda x : \text{Nat}.f(fx))g0 \xrightarrow{\beta} (\lambda x : \text{Nat}.g(gx))0 \xrightarrow{\beta} g(g0).$$

Двойка применила g дважды: результат $g(g0)$ — вторая итерация. Число n применило бы g ровно n раз. Нуль — ни разу: $0 = \Lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.x$.

Примечание Сорта и виды

В системах куба два сорта. Сорт $\star$ собирает типы: $\text{Nat} : \star$ и $\text{Nat} \rightarrow \text{Bool} : \star$. Сорт $\square$ собирает виды (kinds) — типы для типов. Аксиома каркаса $\star : \square$ читается: у всякого типа есть вид. Простые типы имеют вид $\star$, а оператор над типами List — вид-стрелку $\star \rightarrow \star$. Без второго сорта нечем классифицировать выражения вроде List , которые сами типами не являются.

Разрешённые зависимости формулируются в этих сортах:

1. базовое правило всех восьми систем собирает $\Pi x : A.B$ из типа A и семейства B : термы зависят от термов;
2. $\lambda 2$ разрешает абстракцию по самим типам: $\Pi\alpha : \star.\sigma$;
3. λP разрешает семейства типов, индексированные термами: $\Pi n : \text{Nat}.\star$;
4. $\lambda\omega$ разрешает виды-стрелки: $\star \rightarrow \star$ как самостоятельный вид.

Все восемь вершин куба носят имена: $\lambda \rightarrow$, $\lambda 2$, λP , $\lambda P2$, $\lambda\omega$, $\lambda P\omega$, λC и $\text{System F}\omega$ (полиморфизм с операторами над типами). Комбинированные системы складывают оси: $\lambda P2$ — зависимые типы с полиморфизмом. $\lambda P\omega$ — зависимые типы с операторами над типами. λC — все три оси сразу.

Вершина куба — λC (Calculus of Constructions). Именно λC с индуктивными типами лежит в основе современного пруф-ассистента Coq. Lean и Agda используют схожие системы.

Современные пруф-ассистенты превращают соответствие Карри–Ховарда в рабочий инструмент формальной верификации. Проверка доказательства поручена

программе, и это меняет масштаб: формализуются целые математические теории и промышленные программы.

1. *Coq* (INRIA, 1989–) — доказана корректность компилятора *C* (CompCert), формализована теорема о четырёх красках.
2. *Lean 4* (Microsoft Research) — активно развивающееся сообщество математиков, формализующее современную математику от алгебры до теории категорий.
3. *Agda* (Chalmers) — одновременно язык программирования и пруф-ассистент с богатой системой зависимых типов.

Принцип де Брёйна (de Bruijn criterion) — архитектурный принцип, объединяющий эти системы: ядро проверки типов — маленькая, тщательно выверенная программа (несколько тысяч строк). Тактики и автоматизации могут быть сколь угодно сложными и содержать ошибки, но результат их работы (λ -терм) проверяется ядром. Если ядро приняло терм, доказательство корректно, независимо от того, как оно было получено.



Здесь доверие переезжает с человека на машину. Математик доверяет доказательству, потому что сам его провёл. Читатель — потому что проверил. С пруф-ассистентом доказательство — терм, а проверка — программа. Доверие сводится к ядру из нескольких тысяч строк — к тому, что его код верен и он честно исполняется. Так Карри–Ховард возвращается в практику: и доказательство, и его проверка становятся программами.

§ 28.7 Итоги главы

Система типов начинается как фильтр: три правила типизации $\lambda \rightarrow$ пропускают ровно те термы, чьё вычисление гарантированно завершается. Самоприменение xx сквозь фильтр не проходит — оно потребовало бы тип $\sigma = \sigma \rightarrow \tau$, то есть невозможное равенство. Subject Reduction обещает, что типизированный терм не испортится по дороге, а сильная нормализация превращает фильтр в теорему: каждый типизируемый терм сильно нормализуем, и любая последовательность редукций конечна.

Дальше выяснилось, что фильтр — это доказательство. Правила типизации *var*, *app* и *abs* совпадают с аксиомой, *modus ponens* и $\rightarrow$ -введением натурального вывода. Тип комбинатора *K* читается как аксиома $A \rightarrow B \rightarrow A$, а терм $\lambda x : A. x$ — как доказательство тавтологии, его тип — $A \rightarrow A$. Соответствие Карри–Ховарда — структурный изоморфизм между программами и доказательствами, и закон Пирса показывает его рамку: классическая тавтология остаётся без терма-свидетеля.

λ -куб Барендрегта показывает, как полиморфизм, зависимые типы и операторы над типами расширяют эту игру вплоть до Calculus of Constructions, на котором

работают Coq, Lean и Agda. В пруйф-ассистентах проверка доказательства становится проверкой типов, и доверие сводится к маленькому ядру (de Bruijn criterion).

Цена гарантий — выразительность. Типизированное исчисление покрывает лишь часть вычислимого, и именно эта цена делает систему пригодной для верификации. Дальше нас ждёт вопрос о том, чего стоит вычисление, — главы 29 и 30.

Проверка Проверьте себя

1. Почему $\Omega = (\lambda x.xx)(\lambda x.xx)$ не типизируется в $\lambda \rightarrow$? Что это говорит о самоприменении?
2. Что утверждает Subject Reduction и почему само по себе оно не запрещает бесконечный цикл?
3. Какое свойство системы типов исключает бесконечные циклы и что теряется при этом?
4. В чём соответствие Карри–Ховарда? Сопоставьте правила типизации `var`, `app` и `abs` с правилами логического вывода.
5. Почему тип комбинатора K можно прочесть как аксиому $A \rightarrow B \rightarrow A$? Что этот тип говорит о роли второго аргумента?
6. Чем $\lambda 2$ (System F) отличается от $\lambda \rightarrow$? Какой вид зависимости она добавляет?

Что дальше?

От теории типов мы движемся к вопросу об эффективности вычислений — классификации задач по требуемым ресурсам. Но прежде чем заняться классами P и NP, нам предстоит короткое отступление: в главе 29 лежат нестандартные модели арифметики, ординалы и сюрреальные числа. К сложности вычислений и классам P и NP мы вернёмся сразу после неё, в главе 30.

§ 28.8 Упражнения

Простые типы и правила типизации.

1. Постройте дерево вывода типа для $\lambda x : \text{Bool} . \lambda y : \text{Nat} . y$ в $\lambda \rightarrow$. Покажите каждый шаг и укажите применяемое правило.
2. Найдите тип терма $\lambda f : \text{Nat} \rightarrow \text{Bool} . \lambda x : \text{Nat} . f x$. Какой тавтологии минимальной логики высказываний соответствует этот тип?
3. Для каждого из следующих термов определите, типизируется ли он в $\lambda \rightarrow$ с Church-типизацией. Если да — постройте дерево вывода и приведите тип. Если нет — объясните, на каком шаге вывод терпит неудачу.
 - a) $\lambda x : \text{Nat} . \lambda f : \text{Nat} \rightarrow \text{Nat} . f(f x)$
 - b) $\lambda x : \text{Nat} . x x$

с) $\lambda f : \text{Nat} \rightarrow \text{Bool} . \lambda g : \text{Bool} \rightarrow \text{Nat} . \lambda x : \text{Nat} . g(fx)$

4. * Постройте дерево вывода для терма $\lambda x : \text{Nat} . \lambda y : \text{Nat} . x$. Сравните его с деревом для $\lambda x : \text{Nat} . \lambda y : \text{Bool} . x$. Что общего в структуре деревьев? Чем они различаются?

Нетипизируемость и самоприменение.

5. Покажите формально, что терм $\Omega = (\lambda x.xx)(\lambda x.xx)$ не типизируется в $\lambda \rightarrow$. Начните строить вывод, дойдите до уравнения на типы и продемонстрируйте противоречие.
6. * Объясните, почему ни один типизируемый в $\lambda \rightarrow$ терм не содержит подтерма вида xx . Начните строить вывод типа для xx , дойдите до уравнения на типы и продемонстрируйте, что оно неразрешимо.

Соответствие Карри–Ховарда.

7. Сопоставьте каждое из трёх правил типизации (var, app, abs) с соответствующим правилом натурального вывода. Для каждого правила объясните, почему соответствие является точным, а не приближительным.
8. Переведите следующее доказательство в λ -терм: «Предположим A . Предположим $A \rightarrow B$. Тогда по modus ponens получаем B . Следовательно, $A \rightarrow (A \rightarrow B) \rightarrow B$.» Укажите тип полученного терма.
9. * Для типа $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$ постройте терм (комбинатор S). Объясните, доказательством какой тавтологии он является. Покажите каждый шаг вывода типа.

За пределами $\lambda \rightarrow$.

10. Назовите восемь вершин λ -куба Барендрегта. Для систем $\lambda \rightarrow$, $\lambda 2$, λP , $\lambda \omega$ приведите пример типа, выразимого в этой системе, но не в $\lambda \rightarrow$.
11. * Объясните, почему теорема о сильной нормализации $\lambda \rightarrow$ означает, что каждое доказательство в минимальной логике высказываний может быть приведено к нормальной форме без обходных путей. Что такое «обходной путь» (detour) в доказательстве и как он связан с β -редексом? Приведите конкретный пример.
12. ** Тип $\text{Vec } n$ (вектор длины n) — пример зависимого типа. Объясните, почему этот тип невыразим в $\lambda \rightarrow$ и какое расширение типовой системы требуется для его выражения. Придумайте ещё один пример типа, который требует зависимых типов.

ПРОЕКТ Перебор замкнутых термов и кратчайшие доказательства

Простые типы $\lambda \rightarrow$ задаются грамматикой «атом, стрелка» и тремя правилами типизации: переменная, аппликация, абстракция. По соответствию Карри–Ховарда терм типа σ — это доказательство формулы σ : абстракция вводит

импликацию, аппликация её удаляет, поэтому длина доказательства — это размер терма. Поскольку $\lambda \rightarrow$ сильно нормализуем, всякое доказательство имеет нормальную форму, а замкнутых термов заданного размера — конечное число. Соберите детерминированный перебор замкнутых термов в нормальной форме по возрастанию размера и найдите среди них обитателей формул.

① **Проверка типов**

Реализуйте правила var, app, abs для термов с аннотациями $\lambda x : \sigma. M$. Самоприменение xx должно отвергаться с указанием позиции в терме. Объясните, почему такой терм требует неразрешимого конфликта $\sigma = \sigma \rightarrow \tau$.

② **Канонический перебор**

Соберите генератор замкнутых термов в нормальной форме по возрастанию размера на индексах де Брёйна: детерминированный порядок, без дублей, с точностью до α -эквивалентности. Продумайте, как проверить, что генератор не пропускает термы и не повторяется.

③ **Обитатели формул**

Найдите первого обитателя для каждой формулы ниже. Сверьтесь с ориентирами:

1. $\lambda x. x : A \rightarrow A$;
2. $\lambda x. \lambda y. x : A \rightarrow B \rightarrow A$;
3. $S : (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$.

Найдите обитателя перестановки аргументов $(A \rightarrow B \rightarrow C) \rightarrow (B \rightarrow A \rightarrow C)$.

④ **Минимальность**

Каждый узел терма соответствует одному применению правила вывода. Объясните, почему первый найденный по размеру обитатель — кратчайшее доказательство формулы, и выведите размер и терм.

⑤ **Отрицательный результат**

Закон Пирса $((A \rightarrow B) \rightarrow A) \rightarrow A$ не населён — необитаемость разобрана в тексте главы. Убедитесь в этом перебором замкнутых нормальных форм до размера N : сообщите число проверенных термов. Объясните, почему отсутствие обитателя до N — переборное свидетельство, а не доказательство.

Итог: работающая проверка типов $\lambda \rightarrow$, канонический перебор замкнутых нормальных форм и таблица обитателей формул с размерами, включая переборное подтверждение невыводимости закона Пирса до размера N .

XXIX

За пределами натуральных чисел

Вычисление, которое когда-нибудь завершится, делает это за конечное число шагов. Шагов может быть сколько угодно — больше, чем атомов в видимой Вселенной — но число всегда конечное. Натуральный ряд $\mathbb{N} = \{0, 1, 2, \dots\}$ — лестница таких шагов: у каждой ступени есть следующая, и зазора между ними нет. Ряд не имеет последней ступени, и привычная картина говорит: дальше ничего нет, одна бесконечность, а бесконечность — не лестница. Эта картина ошибочна: за пределами натурального ряда лежит целый мир упорядоченных бесконечностей — и у этого мира есть точная структура.

Этот мир уже появлялся в главе 7, но с одной стороны: бесконечности измерялись размером, числом элементов. Счётных множеств столько же, сколько натуральных чисел, действительных — больше, а булеан действительных — снова больше. Так поднимается ряд кардиналов:

$$\aleph_0, 2^{\aleph_0}, 2^{2^{\aleph_0}}, \dots$$

и дальше без конца. Но у бесконечного множества есть вторая сторона, которую мощность не различает, — порядок. Помимо вопроса *сколько*, есть вопрос *как*: что стоит раньше, что позже, есть ли у ряда начало, есть ли конец. Кардинал отвечает на вопрос «сколько», *ординал* — на вопрос «в каком порядке». Один и тот же размер можно выстроить по-разному: порядок — отдельное данное о множестве, не выводимое из его мощности.

В прошлой главе (28) порядок правит вычислением: каждый типизируемый терм завершается, и доказательство этого спускается по натуральной лестнице. Типы упорядочивают вычисления, но сами числа этой лестницей не ограничиваются. Георг Кантор открыл, что упорядоченный ряд продолжается и после того, как конечные числа кончились:

$$\omega, \omega + 1, \omega + 2, \dots, \omega \cdot 2, \omega^2, \omega^\omega, \dots$$

и дальше, без видимой границы. Джон фон Нейман в 1920-е годы придал этому построению точную форму: ординал — это множество всех меньших ординалов, и рецепт работает на любой ступени, конечной или бесконечной. Обычная индукция проходит только по конечным ступеням и на ω останавливается. Чтобы идти дальше, нужен предельный шаг.

Но прежде чем строить, нужно понять, почему вообще есть что строить. Аксиомы Пеано второго порядка — с индукцией по всем подмножествам — категоричны: у

них ровно одна модель, стандартный ряд. Аксиомы Пеано первого порядка (РА) категоричными не являются: индукция в них — бесконечная схема, по одному утверждению на каждую формулу языка.

У РА есть модели с элементами, большими всех натуральных чисел и неотличимыми от них средствами языка.

Это обратная сторона полноты: всякая общезначимая формула первого порядка доказуема, и эта полнота оставляет пространство для моделей, которых никто не заказывал. Логика второго порядка категорична, но платит за это: в ней нет механической проверки доказательств. Выбор давно сделан в пользу первого порядка, и цена выбора — нестандартные модели, стоящие за всеми конструкциями этой главы.

Кантор поставил вопрос: существует ли множество, в котором элементов больше, чем в натуральном ряду, и меньше, чем в вещественной прямой? Этот вопрос называли континуум-гипотезой, и обычные аксиомы на него не отвечают: Гёдель в 1940 году показал, что гипотеза с ними совместима, а Коэн в 1963-м — что совместимо и её отрицание. Эта независимость держится на той же свободе, которую логика первого порядка оставляет аксиомам.

У выхода за пределы натурального ряда есть и инженерная отдача, и самая прямая из них — доказательство завершаемости (*termination*). Чтобы показать, что рекурсивная функция не заикливается, каждой её итерации назначают меру, строго убывающую на каждом шаге. Натуральной меры хватает не всегда: для некоторых завершающихся рекурсий убывающей конечной меры не существует. Ординальная мера убывать бесконечно не может: ординалы вполне упорядочены, и убывающей цепочки без дна в них нет. На этой идее держится проверка завершаемости в системах переписывания термов.

Так сходятся два движения.

- Первое — непреднамеренное: логика первого порядка сама порождает модели, в которых натуральный ряд продолжен.
- Второе — контролируемое: ординалы, сюрреальные числа и гиперреальные числа строятся сами, решая, как далеко зайти.

Различие между ними не терминологическое: это различие между тем, что язык вынужден допускать, и тем, что конструкция способна произвести. Натуральный ряд — лишь первая ступень числовой вселенной: за ним идут ординалы, сюрреальные и гиперреальные числа. Даже у первой ступени арифметики есть рубеж: ординал ε_0 , дальше которого индукция, доступная РА, не достаёт. Что лежит за последней ступенью натурального ряда и как измерить бесконечный порядок?

ОБЗОР ГЛАВЫ

Глава выходит за пределы натурального ряда и объясняет, почему логика первого порядка оставляет для этого место. Теорема компактности покажет, откуда берутся нестандартные модели арифметики, и мы опишем их порядковую структуру: стандартный ряд, за которым плотно выстроены цепочки нестандартных элементов. Затем ординалы фон Неймана продолжат лестницу за ω : простейшие операции ординальной арифметики и предельный шаг трансфинитной индукции. Мы дойдём до рубежа ε_0 , до которого не достаёт индукция РА. Сюрреальные числа Конвея соберут в одном поле действительные числа, ординалы и бесконечно малые — один рецепт $\{L \mid R\}$ вместо нескольких числовых систем. Нестандартный анализ Робинсона замкнёт ряд: тот же приём, что породил нестандартные модели арифметики, вернёт в математику бесконечно малые на легальном основании.

После этой главы вы сможете:

1. объяснять, почему логика первого порядка оставляет нестандартные модели;
2. описывать порядковую структуру нестандартных моделей арифметики;
3. строить ординалы фон Неймана и выполнять ординальную арифметику;
4. применять трансфинитную индукцию и объяснять рубеж ε_0 ;
5. строить сюрреальные и гиперреальные числа и применять ординальные меры к завершаемости.

§ 29.1 Теорема компактности и нестандартные модели

Компактность — свойство логики первого порядка. Если у нас есть бесконечный набор условий и *каждое конечное подмножество* этих условий выполнимо, то и *весь* набор выполним одновременно — бесконечное сводится к конечному.

ТЕОРЕМА 29.1 Теорема компактности

Пусть Γ — множество предложений логики первого порядка. Если каждое конечное подмножество Γ имеет модель, то и всё Γ имеет модель.

У теоремы есть две равносильные переформулировки:

- **Контрапозиция.** Если Γ не имеет модели, то некоторое конечное подмножество Γ уже не имеет модели. (Отрицание заключения влечёт отрицание посылки.)
- **Синтаксическая версия** (через теорему о полноте Гёделя). Отсутствие модели равносильно выводу противоречия, поэтому: если из Γ выводится противоречие, то оно выводится уже из конечного числа формул.

Доказательство теоремы компактности опирается на теорему о полноте Гёделя (глава 3) и выходит за рамки этой главы. Доказательство — в упражнении ниже.

Применим компактность к арифметике. Рассмотрим язык арифметики:

Символ	Роль
0	константа
S	функция следования
+	сложение
$\cdot$	умножение
=	отношение равенства

Пусть Γ — множество всех предложений этого языка, истинных в стандартной модели $\mathbb{N}$.

В частности, Γ содержит все аксиомы PA.

Идея доказательства — ввести новую константу c и объявить, что она больше любого натурального числа. Само по себе это требование не противоречиво: любое конечное число таких условий выполнимо — достаточно выбрать c достаточно большим. А компактность гарантирует, что и весь бесконечный набор условий выполним одновременно.

Добавим к языку новую константу c и рассмотрим множество предложений:

$$\Gamma^* = \Gamma \cup \{c > 0, c > S(0), c > S(S(0)), \dots\}$$

Эти условия утверждают, что c больше любого натурального числа.

ТЕОРЕМА 29.2 Существование нестандартной модели арифметики

Существует модель PA первого порядка, содержащая элемент, больший любого натурального числа $0, 1, 2, \dots$

Доказательство

Докажем построением с помощью теоремы компактности.

Рассмотрим язык арифметики, расширенный новой константой c , и множество предложений

$$\Gamma^* = \Gamma \cup \{c > 0, c > 1, c > 2, \dots\},$$

где Γ — множество всех предложений, истинных в стандартной модели $\mathbb{N}$. Сюда входят все аксиомы PA.

Любое конечное подмножество Γ^* имеет модель: в нём упоминается лишь конечное число утверждений вида

$$c > n_1, \dots, c > n_k.$$

Достаточно положить $c = \max(n_1, \dots, n_k) + 1$, и в стандартной модели $\mathbb{N}$ все эти утверждения истинны.

По теореме компактности всё Γ^* имеет модель, и в этой модели есть элемент c^M , больший любого $n \in \mathbb{N}$.

Модель M содержит стандартный натуральный ряд $\{0, 1, 2, \dots\}$, но к нему не сводится.

Эта модель называется *нестандартной моделью арифметики*. Она содержит «бесконечно большие» элементы, но множество истинных предложений первого порядка (замкнутых формул) у неё такое же, как у стандартной модели.

Замечание Почему это не противоречит индукции?

Схема индукции в РА говорит: если свойство P выполнено для 0 и выполнено $P(n) \rightarrow P(n+1)$, то P выполнено для всех n . Казалось бы, свойство « n — стандартное натуральное число» удовлетворяет посылке, и индукция должна была бы исключить нестандартные элементы.

Но свойство « n стандартно» *невыразимо* в языке арифметики первого порядка. Схема индукции применяется только к свойствам, определяемым формулами языка РА. Никакая формула первого порядка не выделяет в точности стандартные элементы: любое выразимое свойство, истинное для всех стандартных n , истинно и для некоторого нестандартного элемента (это ровно то, что гарантирует компактность). Именно поэтому они существуют.

Пример Overspill на конкретной формуле

Возьмём нестандартный элемент c . Формула $\varphi(x) = x < c$ истинна для каждого стандартного числа: элемент c больше всех них. По принципу overspill (переполнение) она истинна и для некоторого нестандартного элемента. Таким элементом служит предшественник $c - 1$: он тоже нестандартен и меньше c . У формулы $x < c + 1$ нестандартным свидетелем служит сам c . Формула, охватывающая все стандартные числа, захватывает и нестандартные по соседству: границу между частями модели не провести выразимым свойством.

Посмотрим на порядковую структуру счётной нестандартной модели арифметики.

Стандартная часть — натуральный ряд $0, 1, 2, \dots$ порядкового типа ω . Это привычные числа: у каждого есть последователь, и ряд тянется вверх без конца, но начинается с нуля. За ней идёт нестандартная часть, и устроена она совсем иначе.

Возьмём любой нестандартный элемент c . Он не одинок: вместе с ним в модель входит целая цепочка, устроенная как целые числа:

$$\dots, c - 2, c - 1, c, c + 1, c + 2, \dots$$

— бесконечная в обе стороны. У c есть предшественник $c - 1$, а у того — свой предшественник, и так вниз без конца. Вверх цепочка устроена так же: последователь, последователь последователя, и так далее. В цепочке нет ни первого, ни последнего элемента: она не начинается и не кончается. Стандартная часть таким свойством не обладает: натуральный ряд начинается с нуля.

Все нестандартные элементы разбиты на такие цепочки, и между любыми двумя из них лежит ещё одна, а между ними — ещё одна, и так далее. Порядок цепочек плотен, как порядок рациональных чисел.

Плотность видна из самой модели: если a и b лежат в разных цепочках, разность $b - a$ нестандартна. Деление с остатком выразимо в языке арифметики, поэтому в модели существует *середина*: a плюс половина разности, округлённая вниз. Она лежит строго между a и b и принадлежит третьей цепочке, и тот же приём продолжается без конца.

Любая счётная нестандартная модель РА имеет порядковый тип

$$\omega + (\mathbb{Q} \times \mathbb{Z})$$

Сначала идёт стандартная часть ω . Затем — цепочки, устроенные как $\mathbb{Z}$, между любыми двумя из которых вклинивается ещё одна, так что порядок плотный, как у $\mathbb{Q}$.

Структура нестандартной модели показывает обратную сторону полноты логики первого порядка — аксиомы Пеано задают алгебраические операции, но не фиксируют порядковый тип, и нестандартные элементы неисключимы.

Пример Элемент c

В нестандартной модели есть элемент c , больший всех стандартных натуральных чисел. Тогда следующие элементы — тоже нестандартные, и все они попарно различны:

- $c + 1$
- $c + 2$
- $c + c = 2c$
- $c \cdot c = c^2$

У c есть предшественник $c - 1$, и дальше вниз цепочка тянется бесконечно, но никогда не достигает стандартной части. Вся цепочка устроена как $\mathbb{Z}$ и «висит» над ω .

Пример Порядок нестандартных элементов

Возьмём нестандартный элемент c , больший всех стандартных натуральных чисел. Тогда в модели выполнено $c < c + c < c \cdot c$, и каждое неравенство строгое.

Неравенство $c + c > c$ следует из $c > 0$. Разность $(c + c) - c = c$ положительна.

Неравенство $c \cdot c > c + c$ следует из равенства

$$c \cdot c - (c + c) = c \cdot (c - 2).$$

Нестандартный c больше не только нуля, но и двойки, поэтому $c - 2 > 0$ и произведение $c \cdot c$ лежит выше суммы.

Оба элемента, $c + c$ и $c \cdot c$, нестандартны и больше c , а c больше всех стандартных чисел.

Проверка Нестандартные модели арифметики

1. Почему у нестандартного элемента c всегда есть предшественник и последователь, а у стандартного 0 — нет?
2. Почему порядок цепочек нестандартных элементов плотный, и какое знакомое множество упорядочено так же?

§ 29.2 Ординалы за пределами ω

Нестандартные модели — *непреднамеренное* расширение, навязанное ограничениями логики первого порядка. Ординалы — расширение *контролируемое*: мы сами решаем, как далеко зайти. Это математика, а не физика — ω не спрятан в природе, и аксиома о нём не становится правдоподобнее от того, как хорошо она объясняет наблюдения. Мы объявляем, что бесконечное множество существует, — и следим лишь за тем, чтобы аксиомы не вели к противоречию. Так аксиома бесконечности творит ω из ничего — одним объявлением, без дальнейших построений.

Натуральные числа по фон Нейману построены в главе 22: $0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, и каждое число оказывается множеством всех меньших. Ординалы продолжают тот же рецепт туда, где натуральные числа кончаются.

ОПРЕДЕЛЕНИЕ 29.1 Ординал фон Неймана

Множество α называется **ординалом фон Неймана**, если α транзитивно и вполне упорядочено отношением принадлежности.

Транзитивность означает: $x \in \alpha$ влечёт $x \subseteq \alpha$, поэтому элемент элемента ординала сам является его элементом. Каждый элемент ординала сам является ординалом, а сам ординал собирает все меньшие ординалы: принадлежность здесь работает как порядок, и запись $\beta < \alpha$ означает $\beta \in \alpha$. Предел всех конечных ординалов — $\omega = \{0, 1, 2, \dots\}$.

ω — первый ординал, который не является натуральным числом. Необычно в нём то, что у него нет предшественника: ω не равен $\alpha + 1$ ни для какого ординала. Ноль не имеет предшественника среди натуральных чисел, и ω повторяет эту особенность на следующем рубеже. Аксиомы Пеано требуют, чтобы элемент без предшественника был внутри натурального ряда единственным, но не запрещают других упорядоченных множеств с таким элементом.

После ω идут:

$$\begin{aligned}\omega + 1 &= \omega \cup \{\omega\} = \{0, 1, 2, \dots, \omega\}, \\ \omega + 2 &= (\omega + 1) \cup \{\omega + 1\} = \{0, 1, 2, \dots, \omega, \omega + 1\}, \\ &\dots\end{aligned}$$

ЛОВУШКА $\omega + 1 \neq \omega$ как ординалы

Мощности равны: $|\omega + 1| = |\omega|$. Биекцию задаёт отображение

$$\omega \mapsto 0, n \mapsto n + 1.$$

Но ординал — это порядковый тип, и биекции недостаточно: нужен *порядковый* изоморфизм, сохраняющий порядок. У $\omega + 1$ есть наибольший элемент ω , а у ω его нет. А изоморфизм обязан сохранять наличие наибольшего элемента. Поэтому $\omega + 1 \neq \omega$ как ординалы, хотя их мощности как кардиналов равны.

Картинка, которую стоит удерживать: ординал — это длина очереди, а не число людей в ней. Пока очередь конечна, две меры неразличимы: в очереди из трёх человек третий ждёт ровно двоих. Пусть бесконечная очередь замкнётся — человек из самого начала выходит и встаёт в конец. Людей не прибавилось, но у очереди появился последний: тот, кому теперь предстоит ждать бесконечно многих. Так из неизменного количества рождается новый порядковый тип — $\omega + 1$.

Предел цепочки

$$\omega, \omega + 1, \omega + 2, \dots$$

— ординал $\omega \cdot 2$. Он равен $\omega + \omega$ и выглядит так:

$$\omega \cdot 2 = \{0, 1, 2, \dots, \omega, \omega + 1, \omega + 2, \dots\}$$

Дальше идут $\omega \cdot 2 + 1, \omega \cdot 2 + 2, \dots$ — цепочка снова не имеет конца. У неё нет последнего элемента, поэтому ни один член вида $\omega \cdot 2 + n$ не может быть её пределом. Предел — супремум всех $\omega \cdot 2 + n$. Этот супремум и есть ординал $\omega \cdot 3$.

Здесь появляется механизм, который будет повторяться на каждой следующей ступени. Ординал не обязан быть «следующим за предыдущим» в смысле $\alpha + 1$. Если у нас есть бесконечная цепочка ординалов, в которой каждый следующий больше предыдущего и у цепочки нет конца, то её супремум — снова ординал, и он больше каждого элемента цепочки. Такой ординал называется *предельным*: у него нет предшественника. Он не является $\alpha + 1$ ни для какого ординала. С предельным ординалом мы уже встречались: это ω , предел конечных чисел. Теперь предельный шаг становится приёмом, который можно применять снова и снова.

ЛОВУШКА Предел ординалов $\neq$ супремум мощности

Предел последовательности ординалов легко принять за «мощность, к которой они стремятся» — это ошибка. Предельный ординал — супремум в порядковом

смысле: наименьший ординал, больший каждого члена последовательности. ω — предел конечных чисел. Мощности всех конечных n при этом конечны. Предельный шаг определяется порядком, а не мощностью.

Вот ступени лестницы и идея предела, которая стоит за каждой:

- $\omega \cdot 3$ — предел цепочки $\omega \cdot 2, \omega \cdot 2 + 1, \omega \cdot 2 + 2, \dots$. К $\omega \cdot 2$ мы добавляем ещё одну копию ряда. Она прибавляется сразу целиком — одним супремумом, без шагов.
- $\omega \cdot 4, \omega \cdot 5, \dots$ — и снова конечные прибавки, и снова предел. Предел этой цепочки — $\omega \cdot \omega = \omega^2$. Здесь новая идея: бесконечным становится количество копий ω .
- За ω^2 следует $\omega^3, \omega^4, \dots$. Предел этой последовательности — ω^ω . Здесь новая идея: бесконечным становится показатель степени.
- За ω^ω идёт башня: $\omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \dots$. Высота башни — конечное число n , и его можно брать сколь угодно большим. Предел всей башни — ε_0 . Это ординал, для которого $\omega^{\varepsilon_0} = \varepsilon_0$. Здесь новая идея: экспонента $\alpha \rightarrow \omega^\alpha$ достигает неподвижной точки.

Каждая новая ступень — новый способ взять предел. Мы начали с цепочки без конца и взяли её супремум — получили $\omega \cdot 3$. Потом бесконечным стало количество копий — получили ω^2 . Потом бесконечным стал показатель — получили ω^ω . Потом бесконечной стала сама башня — получили ε_0 . На каждом шаге мы спрашивали, что является пределом уже построенного, и новый знак возникал из ответа.

Первый ординал после ω , который нельзя получить из меньших ординалов сложением, умножением и возведением в степень, — это ε_0 . Причина — в уравнении $\omega^{\varepsilon_0} = \varepsilon_0$. Возведение ω в степень любого меньшего ординала даёт снова ординал. Он снова меньше ε_0 . Каждая конечная башня лежит ниже своего предела. Операции не могут перепрыгнуть этот рубеж.

Неподвижную точку удобно ощутить как ловушку: сколько ни добавляй этажей к башне $\omega^{\omega^{\dots}}$, ничего не меняется — башня уже бесконечна, и ещё один этаж не отличим от уже построенного. Выбраться из ловушки можно только одним способом — прибавить единицу к самому ε_0 : за ним начинается следующая башня, и лестница продолжается.

И это только начало: РА первого порядка не в состоянии доказать *трансфинитную индукцию* вплоть до ε_0 . За ним идут ещё большие ординалы.

Расширение натурального ряда ординалами показано на рис. 113: натуральные числа $0, 1, 2, \dots$ сменяются первым бесконечным ординалом ω . За ним идут $\omega + 1, \omega + 2, \dots$, затем $\omega \cdot 2$ и ω^2 .

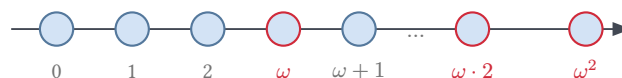


Рис. 113. Ординалы как расширение натурального ряда.

29.2.1 Ординальная арифметика

ОПРЕДЕЛЕНИЕ 29.2 Ординальные операции

Сложение, умножение и возведение в степень ординалов задаются рекурсией по правому аргументу:

- $\alpha + 0 = \alpha$. $\alpha + (\beta + 1) = (\alpha + \beta) + 1$. Для предельного λ сумма $\alpha + \lambda$ определяется как $\sup_{\beta < \lambda} (\alpha + \beta)$.
- $\alpha \cdot 0 = 0$. $\alpha \cdot (\beta + 1) = \alpha \cdot \beta + \alpha$. Для предельного λ произведение $\alpha \cdot \lambda$ определяется как $\sup_{\beta < \lambda} (\alpha \cdot \beta)$.
- $\alpha^0 = 1$. $\alpha^{\beta+1} = \alpha^\beta \cdot \alpha$. Для предельного λ степень α^λ определяется как $\sup_{\beta < \lambda} \alpha^\beta$.

Предельные случаи делают все три операции непрерывными по правому аргументу: значение на предельном ординале — супремум значений на всех меньших. Ординальная арифметика отличается от кардинальной и от обычной арифметики натуральных чисел. Отличие видно уже на сложении и умножении: $a + b = b + a$ перестаёт быть общим законом. То же с умножением: $a \cdot b = b \cdot a$ — тоже не всегда верно. Для конечных чисел перестановка ничего не меняет, для ординалов — меняет результат.

ТЕОРЕМА 29.3 Некоммутативность ординальной арифметики

Сложение и умножение ординалов не коммутативны:

1. $1 + \omega = \omega$.
2. Но $\omega + 1 > \omega$.
3. $2 \cdot \omega = \omega$.
4. Но $\omega \cdot 2 = \omega + \omega > \omega$.

Набросок доказательства

Докажем каждое из четырёх утверждений отдельно.

Для каждого утверждения опишем соответствующий порядок и предъявим изоморфизм с натуральным рядом либо укажем собственный начальный сегмент. Начнём со сложения: $1 + \omega$ — это ординал, полученный добавлением одного элемента *перед* натуральным рядом, и ряд $* < 0 < 1 < 2 < \dots$ изоморфен ω . Изоморфизм задаёт отображение

$$* \mapsto 0, n \mapsto n + 1.$$

Напротив, $\omega + 1$ — натуральный ряд с дополнительным элементом *после* всех натуральных, поэтому в нём есть наибольший элемент, а в ω — нет. Следовательно, $\omega + 1 > \omega$.

Обратимся к умножению: $2 \cdot \omega$ — это ω копий ординала $2 = \{0, 1\}$, упорядоченных лексикографически — сначала по номеру копии, затем по элементу. Изоморфизм с ω даёт отображение

$$(i, 0) \mapsto 2i, (i, 1) \mapsto 2i + 1.$$

Наконец, $\omega \cdot 2 = \omega + \omega$ — это две копии ω подряд, и первая копия является собственным начальным сегментом, поэтому $\omega \cdot 2 > \omega$.

ЛОВУШКА Порядковая сумма $\neq$ кардинальная

Ординалы хочется складывать и умножать как обычные числа — и это ошибка. Ординальное сложение некоммутативно: $\omega + 1 \neq 1 + \omega$. Ведь $1 + \omega = \omega$. А $\omega + 1 > \omega$.

Ординальное умножение ведёт себя так же: $2 \cdot \omega = \omega$. Но $\omega \cdot 2 \neq 2 \cdot \omega$. Коммутативность кардинальной арифметики на ординалы не переносится.

Пример Ординальные операции

- ω — порядковый тип натурального ряда. Это ряд $0, 1, 2, 3, \dots$
- $\omega + 1$ — натуральные числа и ещё один элемент после всех. Ряд выглядит как $0, 1, 2, \dots, a$.
- $1 + \omega = \omega$ — один элемент перед натуральным рядом не меняет порядкового типа.
- $\omega + \omega = \omega \cdot 2$. Это две копии $\mathbb{N}$ подряд.
- ω^2 — это ряд из копий. Копий ровно ω . Каждая копия — это ω . Пары (i, j) упорядочены сначала по первой компоненте, затем по второй.

Пример Степени: множитель слева и справа

Асимметрия сложения и умножения переносится на степени. Умножение на ω слева: $\omega \cdot \omega^\omega = \omega^1 \cdot \omega^\omega = \omega^{1+\omega}$. Прибавление единицы к показателю слева ничего не меняет: $1 + \omega = \omega$. Получаем $\omega \cdot \omega^\omega = \omega^\omega$. Умножение на ω справа: $\omega^\omega \cdot \omega = \omega^\omega \cdot \omega^1 = \omega^{\omega+1}$. Здесь единица прибавляется к показателю справа: $\omega + 1 > \omega$. Поэтому $\omega^{\omega+1} > \omega^\omega$. Множитель ω слева оставляет ω^ω на месте, справа — увеличивает.

Некоммутативность ординальной арифметики — прямое следствие того, что у ω нет предшественника: добавление элемента *перед* рядом сохраняет порядковый тип, а добавление *после* создаёт наибольший элемент. Левый множитель задаёт размер одной копии, правый — число копий. В конечном случае разницы нет. В бесконечном — два разных понятия, совпадающих для натуральных чисел и расходящихся для ординалов.

29.2.2 Трансфинитная индукция

Трансфинитная индукция обобщает обычную математическую индукцию на все ординалы. В обычной индукции два шага: база и переход. База — это $n = 0$. Переход — это $n \rightarrow n + 1$. В трансфинитной добавляется третий — предельный шаг.

ОПРЕДЕЛЕНИЕ 29.3 Трансфинитная индукция

Пусть $P(\alpha)$ — свойство ординалов и κ — ординал. Трансфинитная индукция до κ проводится тремя шагами:

- **База.** $P(0)$ верно.
- **Переход.** Для каждого α из $P(\alpha)$ следует $P(\alpha + 1)$.
- **Предельный шаг.** Для каждого предельного $\lambda < \kappa$ из $P(\beta)$ при всех $\beta < \lambda$ следует $P(\lambda)$.

Принцип утверждает: если все три шага проведены, то $P(\alpha)$ верно для каждого $\alpha < \kappa$.

Конечные ординалы проходятся обычной индукцией: $P(0), P(1), P(2), \dots$. На ординале ω обычная индукция останавливается: сколько бы конечных шагов ни сделать, $P(\omega)$ из них не следует. Предельный шаг закрывает разрыв: он позволяет заключить $P(\omega)$ из справедливости $P(n)$ при всех конечных n . Без этого шага границу между конечным и бесконечным не перейти.

Пример Ассоциативность ординального сложения

Чтобы доказать ассоциативность ординального сложения

$$(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma),$$

фиксируют α, β . Индукцию проводят по γ .

- **База:** $\gamma = 0$. Тогда $(\alpha + \beta) + 0 = \alpha + \beta = \alpha + (\beta + 0)$.
- **Переход:** $\gamma \rightarrow \gamma + 1$. Если верно для γ , то

$$(\alpha + \beta) + (\gamma + 1) = ((\alpha + \beta) + \gamma) + 1 = (\alpha + (\beta + \gamma)) + 1 = \alpha + ((\beta + \gamma) + 1) = \alpha + (\beta + (\gamma + 1)).$$

- **Предельный шаг:** для предельного γ ,

$$(\alpha + \beta) + \gamma = \sup_{\delta < \gamma} ((\alpha + \beta) + \delta) = \sup_{\delta < \gamma} (\alpha + (\beta + \delta)) = \alpha + \sup_{\delta < \gamma} (\beta + \delta) = \alpha + (\beta + \gamma),$$

где первый и последний переходы — определение ординального сложения на предельном шаге, второй — индуктивное предположение, третий — непрерывность сложения по правому аргументу ($\alpha + \sup X = \sup_{x \in X} (\alpha + x)$ для любого множества ординалов X).

В теории алгоритмов трансфинитная индукция применяется для доказательства завершаемости систем переписывания термов. Механизм работает так:

1. Строится ординальная мера — функция из термов в ординалы.
2. При каждом шаге переписывания мера строго убывает.

3. Поскольку ординалы вполне упорядочены, бесконечная убывающая цепочка невозможна.
4. Следовательно, система гарантированно завершается.²⁰⁷

29.2.3 * Теорема Гудстейна

Тот же приём с ординальной мерой решает чисто арифметическую задачу, для которой убывающей конечной меры не существует. Гудстейн в 1944 году определил последовательность, элементы которой долго растут и которая при этом гарантированно доходит до нуля.

ОПРЕДЕЛЕНИЕ 29.4 Наследственная запись

Пусть $b \geq 2$. Числа $0, 1, \dots, b-1$ записываются как сами числа. Число $n \geq b$ записывается в виде $n = c_1 b^{k_1} + \dots + c_m b^{k_m}$, где $1 \leq c_i < b$, показатели строго убывают и каждый показатель k_i снова записан в наследственной записи по основанию b (*hereditary notation*).

Последовательность Гудстейна строится по правилу: элемент записывается наследственно по текущему основанию b , каждое вхождение b заменяется на $b+1$, из результата вычитается единица, и полученное число становится следующим элементом. Начальным элементом служит любое натуральное число.

Пример Последовательность Гудстейна для $n = 4$

Первый элемент: $4 = 2^2$ в наследственной записи по основанию 2. Замена $2 \rightarrow 3$ даёт $3^3 = 27$, вычитание единицы даёт 26. Дальше: $26 = 2 \cdot 3^2 + 2 \cdot 3 + 2$, замена основания даёт $2 \cdot 4^2 + 2 \cdot 4 + 2 = 42$, и вычитание даёт 41. Следующие элементы: 60, 83, 109, 139, и элементы растут без видимой тенденции к спаду.

ТЕОРЕМА 29.4 Теорема Гудстейна

Всякая последовательность Гудстейна достигает нуля за конечное число шагов.

Набросок доказательства

Докажем через строго убывающую ординальную меру. Мера элемента — его наследственная запись по текущему основанию b , в которой каждое вхождение b заменено на ω . Для $4 = 2^2$ мера равна ω^ω , для $26 = 2 \cdot 3^2 + 2 \cdot 3 + 2$ мера равна $2 \cdot \omega^2 + 2 \cdot \omega + 2$, и обе лежат ниже ε_0 . Замена основания b на $b+1$ меру не меняет: на месте основания всё равно оказывается ω . Вычитание единицы меру

²⁰⁷Тот же приём — трансфинитную индукцию до ε_0 — впервые применил Герхард Генцен: он доказал непротиворечивость арифметики (Г. Генцен. «Die Widerspruchsfreiheit der reinen Zahlentheorie». Mathematische Annalen, 1936). По второй теореме Гёделя (глава 26) арифметика не может доказать собственную непротиворечивость. Доказательство Генцена потому и опирается на индукцию до ε_0 , что она выходит за пределы того, что арифметика способна доказать.

уменьшает: в последней ненулевой позиции либо уменьшается на единицу коэффициент, либо его место занимают меньшие степени. Точнее: вместо единичного коэффициента перед ω^k появляется сумма коэффициентов $b - 1$ при меньших степенях, и такая сумма меньше ω^k . Мера строго убывает на каждом шаге, а бесконечной строго убывающей цепочки ординалов не существует, поэтому последовательность доходит до нуля.

Завершение при этом растянуто далеко за пределы видимого: последовательность из примера выше доходит до нуля за $3 \cdot 2^{402653211} - 3$ шагов, и полный прогон недоступен никакому компьютеру. Теорема Гудстейна недоказуема в арифметике Пеано: Кирби и Пэрис в 1982 году показали, что завершаемость всех последовательностей Гудстейна из аксиом PA не выводится.²⁰⁸ Причина та же, что и у Генцена: ординальные меры живут ниже ε_0 , и завершаемость доказывается трансфинитной индукцией до ε_0 . Это тот же рубеж, на котором Генцен построил доказательство непротиворечивости арифметики (сноска выше).

Проверка Ординалы за пределами ω

1. Почему ε_0 нельзя получить из меньших ординалов сложением, умножением и возведением в степень, хотя ω^ω получить можно?
2. Как ординальная мера позволяет доказать, что система переписывания термов завершается?
3. Как ординальная мера доказывает завершаемость последовательностей Гудстейна и какое свойство ординалов в этом доказательстве используется?

§ 29.3 Сюрреальные числа

В 1974 году Дональд Кнут опубликовал небольшую книгу «Surreal Numbers: How Two Ex-Students Turned On to Pure Mathematics and Found Total Happiness». Книга была написана в форме диалога: двое молодых людей находят на необитаемом острове камень с надписью «В начале всего была пара $\{\mid\}$ » — и шаг за шагом восстанавливают теорию, придуманную Джоном Хортоном Конвеем.

Конвей заметил, что свойства чисел можно вывести из одного рекурсивного определения — понятия игры. Игра — это пара множеств игр: $\{L \mid R\}$. Оба множества состоят из уже определённых игр. Число — это игра, в которой оба множества состоят из чисел. Каждый элемент L меньше каждого элемента R .

²⁰⁸G. Goodstein. «On the Restricted Ordinal Theorem». Journal of Symbolic Logic, 1944; L. Kirby, J. Paris. «Accessible Independence Results for Peano Arithmetic». Bulletin of the London Mathematical Society, 1982.

ОПРЕДЕЛЕНИЕ 29.5 Сюрреальное число

Сюрреальное число — это пара множеств сюрреальных чисел

$$\{L \mid R\},$$

где $\forall x \in L, \forall y \in R : x < y$. Множество L называется левым множеством, а R — правым.

Условие $\forall x \in L, \forall y \in R : x < y$ гарантирует, что левое множество целиком лежит ниже правого, и пара читается как *зазор* между ними. Левые опции — нижние оценки числа, правые — верхние.

Определение рекурсивно и на первый взгляд парадоксально: мы определяем числа через множества чисел, которые ещё не определены. Но конструкция начинается с пустого множества: в нулевой день нет ещё ни одного числа, и единственная возможная пара — $\{\emptyset \mid \emptyset\}$.

Числа рождаются по дням:

- **День 0.** Единственное число: $0 = \{\mid\}$ (оба множества пусты).
- **День 1.** Используя 0, строим:
 - $1 = \{0 \mid\}$ — число, большее нуля.
 - $-1 = \{\mid 0\}$ — число, меньшее нуля.
- **День 2.** Используя $\{-1, 0, 1\}$, строим:
 - $2 = \{1 \mid\}$.
 - $\frac{1}{2} = \{0 \mid 1\}$ — между нулём и единицей.
 - $-\frac{1}{2} = \{-1 \mid 0\}$.
 - $-2 = \{\mid -1\}$.

С каждым днём доступных элементов становится больше, и рождается всё больше чисел. К дню n рождены все числа вида

$$\frac{a}{2^{n-1}}.$$

Здесь a — целое число, причём $|a| \leq 2^{n-1}$. Уже ко второму дню рождены числа 2, −2. После всех конечных дней — после ω дней — рождаются все действительные числа. Сам Кант описывал этот переход как рождение вселенной чисел: день за днём рождаются лишь двоичные дроби, и полнота действительных чисел не просматривается — а после всех конечных дней происходит взрыв, и действительные числа появляются сразу все.

Но на этом конструкция не останавливается. В день ω появляются новые числа:

- $\omega = \{0, 1, 2, \dots \mid\}$ — число, большее всех натуральных.
- $\varepsilon = \{0 \mid 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots\}$ — положительное число, меньшее всех положительных действительных (бесконечно малое).

Каждое новое число собирается из уже рождённых: элементы его левого и правого множеств — числа предыдущих дней, и каждое ребро дерева ведёт от такого элемента к новорождённому. Уровни дерева — дни рождения: конечные дни приносят двоичные дроби, а день ω — первые бесконечное ω и бесконечно малое ε . Порядок рождения от пустой пары $\{\mid\}$ до ω и ε показан на рис. 114.

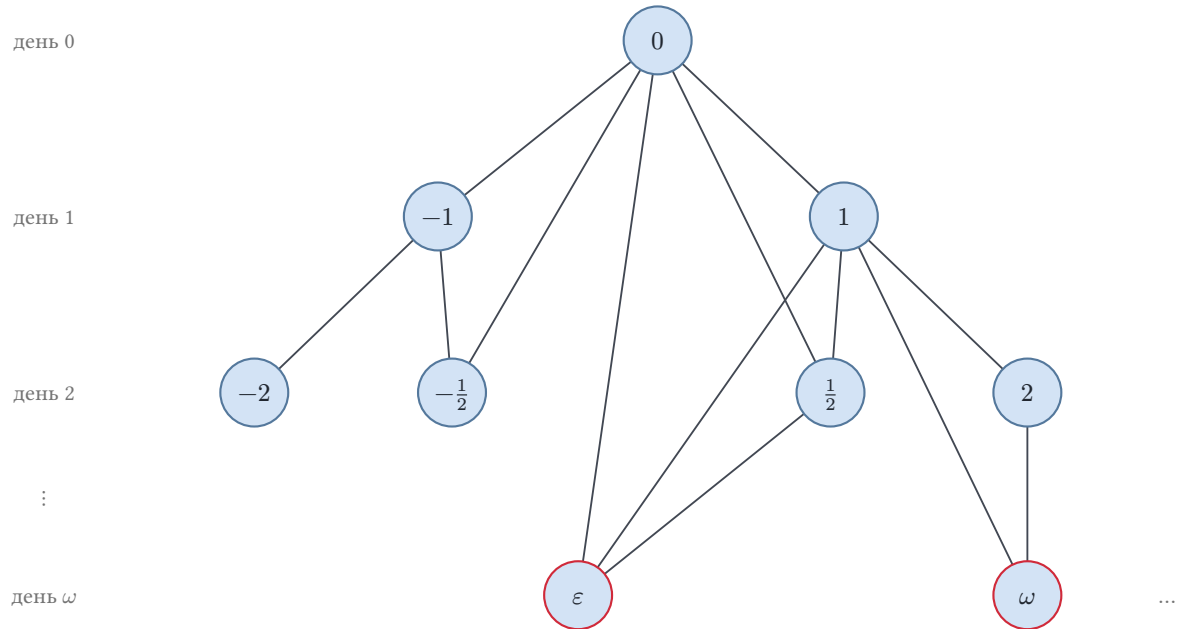


Рис. 114. Дерево рождения сюрреальных чисел.

Бесконечно малое (*infinitesimal*) удобно увидеть как щепку, брошенную в зазор, о котором мы не подозревали. Число $\pi + \varepsilon$ таково, что

$$\pi < \pi + \varepsilon < x$$

для любого действительного $x > \pi$. Такая щепка лежит в каждом зазоре между действительными числами.

В день $\omega + 1$ появляются новые числа. Среди них $\omega + 1, \omega - 1, \frac{\varepsilon}{2}$, и так далее.

Пример Половина бесконечно малого

Число $\frac{\varepsilon}{2}$ строится как пара $\{0 \mid 1\}$. Число $\frac{\varepsilon}{2}$ строится по тому же рецепту. Левое множество — 0, правое — сам ε .

$$\frac{\varepsilon}{2} = \{0 \mid \varepsilon\}.$$

Число $\frac{\varepsilon}{2}$ положительно: $0 < \frac{\varepsilon}{2}$. Оно меньше ε : в правом множестве стоит ровно ε . Итак, $0 < \frac{\varepsilon}{2} < \varepsilon$: половина бесконечно малого снова бесконечно мало. Тот же шаг повторяется: $\frac{\varepsilon}{4} = \{0 \mid \frac{\varepsilon}{2}\}$, и так до любой двоичной доли.

Сюрреальные числа образуют *поле*: их можно складывать, вычитать, умножать и делить (на ненулевое). Сложение определяется рекурсивно через опции. $x + y = \{L_x + y, x + L_y \mid R_x + y, x + R_y\}$, где L_x — левые опции, R_x — правые опции того

же числа. Для y — аналогично. Запись $L_x + y$ — сокращение для множества всех сумм вида

$$l + y,$$

где l пробегает все левые опции.

Проверим определение на простейшем примере. $1 + 1 = \{0 + 1, 1 + 0\} = \{1\} = 2$. Это потому, что $2 = \{1\}$ — число, родившееся на второй день. Сумма определяется чисто из структуры опций, без обращения к готовому ответу: мы вычисляем результат из определения.

Вычитание, умножение и деление определяются так же — рекурсивно, через опции. Все аксиомы поля проверяются индукцией по дню рождения числа: проверка механическая, но утомительная, и мы не будем воспроизводить её здесь.²⁰⁹

Сюрреальные числа содержат в себе:

- Все действительные числа — как сечение $\{\text{рациональные меньше } x \mid \text{рациональные больше } x\}$.
- Все ординалы — как $\{\text{все меньшие ординалы}\}$.
- Все бесконечно малые — как $\{0 \mid 1, \frac{1}{2}, \frac{1}{4}, \dots\}$.
- Комбинации: $\omega + 1, \omega - 1, \frac{\omega}{2}, \sqrt{\omega}$, и многое другое.

ИСТОРИЯ Как писались «Сюрреальные числа»

Дональд Кнут провёл неделю в отеле в Осло, работая над книгой о сюрреальных числах. Он писал её от руки, в форме диалога между двумя персонажами — Элис и Биллом — которые находят древний текст и шаг за шагом открывают конструкцию Конвея.

Кнут сознательно выбрал диалогическую форму, чтобы передать процесс математического открытия. Читатель не получает готовую теорию — он проходит путь вместе с персонажами, от недоумения к озарению. Название «Surreal Numbers» — игра слов: surreal (сюрреалистический) + real (действительный), числа, которые «более реальны, чем реальные».

Джон Конвей придумал сюрреальные числа в конце 1960-х, занимаясь анализом комбинаторных игр (включая знаменитую игру «Ним»). Он заметил, что игры естественным образом образуют абелеву группу относительно суммы, и что некоторые игры ведут себя как числа. Конвей не публиковал теорию, и она стала известна благодаря книге Кнута²¹⁰ и позже — монографии Конвея²¹¹.

²⁰⁹Полное изложение — в монографии Конвея «On Numbers and Games». 1976.

²¹⁰D. E. Knuth. «Surreal Numbers». 1974.

²¹¹J. H. Conway. «On Numbers and Games». 1976.

Замечание Сюрреальные vs нестандартные числа

Нестандартные модели арифметики и сюрреальные числа — два способа выйти за пределы $\mathbb{N}$, и они принципиально различны.

Нестандартная модель — *непреднамеренное* расширение: она возникает потому, что логика первого порядка допускает модели, не изоморфные стандартной. Нестандартный элемент «притворяется» натуральным числом: свойство «быть стандартным» невыразимо. Ни одна формула первого порядка не выделяет в точности множество стандартных элементов. Если формула истинна для всех стандартных n , то по свойству *overspill* она истинна и для некоторого нестандартного элемента. Отдельные элементы формулы различают: $x = 0$ истинно ровно для нуля. Но выделить один элемент — не то же самое, что выделить весь класс стандартных элементов, и именно класс невыразим. Порядково же нестандартный элемент больше всех стандартных натуральных.

Сюрреальные числа — *преднамеренное* расширение: мы сами строим их, шаг за шагом, сознательно расширяя числовую систему. ω в сюрреальных числах — ординал. Он не удовлетворяет аксиомам Пеано: например, он больше любого натурального числа и не получается из 0 конечным числом применений функции последователя.

Первый путь — логический: что *может* быть моделью, если смотреть через язык первого порядка. Второй путь — алгебраический: что можно *построить*, не ограничивая себя аксиомами Пеано.



Сюрреальные числа собирают в одном поле и действительные числа, и ординалы, и бесконечно малые. Один рецепт — пара множеств $\{L \mid R\}$ — порождает всю числовую прямую, а заодно и то, что за её пределами. Это напоминает другой рецепт из главы о конструкциях чисел (22): и там, и там одно определение, применённое к самому себе, вырастает в целую иерархию. Рекурсия строит числа, которые сами служат материалом для следующих.

§ 29.4 Нестандартный анализ

Нестандартные модели арифметики — частный случай более общего явления: логика первого порядка допускает модели, не изоморфные стандартной, для любой бесконечной структуры.

Нестандартный анализ Абрахама Робинсона (1961) применяет эту идею к действительным числам.

Робинсон построил расширение действительных чисел — поле гиперреальных чисел ${}^*\mathbb{R}$ — в котором есть:

- Бесконечно большие числа (больше любого действительного).
- Бесконечно малые (меньше любого положительного действительного, но больше нуля).
- Каждое действительное число окружено «облаком» гиперреальных, бесконечно близких к нему (монадой).

Все теоремы, верные в $\mathbb{R}$, верны и в расширении. Расширение обозначается ${}^*\mathbb{R}$. Они переносятся через принцип переноса (transfer principle): истинность предложений первого порядка при переходе к расширению не меняется.

Пример Что переносит принцип переноса

Коммутативность сложения $x + y = y + x$ — предложение первого порядка: оно говорит только о числах и их операциях. Поэтому она верна и в гиперреальном расширении: гиперреальные числа складываются в любом порядке. Свойство «быть стандартным» формулой первого порядка *не выражается* — ровно как в арифметике. Нестандартные элементы существуют именно потому, что их наличие не запрещено ни одним предложением первого порядка.

Примечание Три инструмента нестандартного анализа

Аппарат раздела собирается из трёх инструментов. Первый — принцип переноса: предложение первого порядка, истинное в $\mathbb{R}$, истинно и в ${}^*\mathbb{R}$, и наоборот. Второй — стандартная часть: каждое конечное гиперреальное число бесконечно близко ровно к одному действительному числу $\text{st}(x)$. Конечность означает $|x| < n$ для некоторого действительного n , и производные в примерах ниже вычисляются как st от отношения приращений. Третий — overspill: формулы первого порядка не отделяют стандартные числа от нестандартных. Свойство «быть бесконечно малым» в ${}^*\mathbb{R}$ формулой не выделяется: квантор по натуральным числам пробегает и нестандартную часть ${}^*\mathbb{N}$, и требование $|x| < \frac{1}{n}$ для всех таких n оставляет только $x = 0$. Само поле ${}^*\mathbb{R}$ строится той же теоремой компактности, что и нестандартная модель арифметики: к истинной теории $\mathbb{R}$ добавляется новая константа с аксиомами $c > n$ (упражнение в конце главы).

Но инфинитезимальные, изгнанные из анализа в XIX веке при его арифметизации (Вейерштрасс, Дедекинд), возвращаются на легальном основании.

Определение предела через ε, δ :

$$\forall \varepsilon > 0, \exists \delta > 0, \dots$$

заменяется прямым утверждением: если $x \approx a$, то $f(x) \approx L$.

Пример Производная через бесконечно малые

Пусть x — стандартное действительное число. Пусть ε — бесконечно малое. Рассмотрим функцию $f(x) = x^2$. Приращение в точке x :

$$f(x + \varepsilon) - f(x) = (x + \varepsilon)^2 - x^2 = 2x\varepsilon + \varepsilon^2.$$

Делим на ε :

$$\frac{f(x + \varepsilon) - f(x)}{\varepsilon} = 2x + \varepsilon.$$

Число $2x + \varepsilon \approx 2x$. Стандартная часть результата — производная: $f'(x) = 2x$.

Пример Производная x^3

Тот же приём для $f(x) = x^3$. Приращение: $(x + \varepsilon)^3 - x^3 = 3x^2\varepsilon + 3x\varepsilon^2 + \varepsilon^3$.

Делим на ε :

$$\frac{f(x + \varepsilon) - f(x)}{\varepsilon} = 3x^2 + 3x\varepsilon + \varepsilon^2.$$

Слагаемые $3x\varepsilon$ и ε^2 бесконечно малы. Стандартная часть результата — $3x^2$.
Производная: $f'(x) = 3x^2$.

Нестандартный анализ не доказывает новых теорем об $\mathbb{R}$: всё, что доказуемо в нём, доказуемо и в стандартном — это следует из принципа переноса. Но он предлагает *другой язык* для доказательств — часто более естественный и короткий. То, что в стандартном анализе требует трёх кванторов и двух эпсилон-дельт, в нестандартном формулируется как одно отношение бесконечной близости.

ИСТОРИЯ Изгнание и возвращение инфинитезимальей

Ньютон и Лейбниц использовали бесконечно малые величины свободно: производная — это отношение двух бесконечно малых, интеграл — сумма бесконечно многих бесконечно малых слагаемых. Беркли в 1734 году высмеял инфинитезимальей как «призраки умерших величин» (ghosts of departed quantities). Логическая несостоятельность состояла в том, что бесконечно малая не равна нулю (иначе на неё нельзя делить), но в конце доказательства её отбрасывают как ноль.

Коши и Вейерштрасс в XIX веке заменили инфинитезимальей эпсилон-дельта-формализмом. Бесконечно малые были «изгнаны» из математики на полтора столетия.

Абрахам Робинсон в 1961 году вернул им легальный статус, построив нестандартный анализ на основе теории моделей. Инфинитезималь — элемент нестандартной модели действительных чисел. Парадокс Беркли разрешается: инфинитезималь не является действительным числом, и правила действий с ней определяются принципом переноса.

§ 29.5 Итоги главы

Всё началось с одной неприятности: логика первого порядка не фиксирует стандартную модель арифметики. Теорема компактности превращает эту неприятность в теорему — у РА есть модели с элементами, лежащими выше всего натурального ряда, и средствами языка их от чисел не отличить. Порядковая структура таких моделей поддаётся описанию: стандартный ряд, за которым плотно выстроены цепочки нестандартных элементов.

Дальше мы взяли расширение под контроль и стали строить сами. Ординалы фон Неймана продолжили ряд за ω с некоммутативной арифметикой и трансфинитной индукцией, а сюрреальные числа Конвея собрали в одном поле действительные числа, ординалы и бесконечно малые — один рецепт вместо нескольких числовых систем. Общий принцип прост: на каждой ступени спрашивать, что является пределом уже построенного, и добавлять его.

Так появились $\omega \cdot 2, \omega^\omega, \varepsilon_0$ — рубеж, до которого не достаёт индукция РА. Трансфинитная индукция добавляет к базе и переходу предельный шаг, и утверждение охватывает все ординалы сразу. Нестандартный анализ Робинсона вернул инфинитезимальм легальный статус: парадокс Беркли разрешился, когда инфинитезималь получила точное место в нестандартной модели и правила действий из принципа переноса.

Различие между нестандартными моделями и сюрреальными числами — различие между языком описания и языком построения: логика первого порядка показывает, где кончается описание, а конструкции ординалов и сюрреальных чисел — что за этой границей можно строить. С этим аппаратом мы возвращаемся к вычислимому: классификация задач по ресурсам, классы сложности и вопрос о соотношении классов P и NP — в главе 30.

Проверка Проверьте себя

1. Почему РА первого порядка имеет нестандартные модели, а РА второго порядка — нет?
2. Почему схема индукции не исключает нестандартные элементы? Какое свойство натуральных чисел невыразимо в языке первого порядка?
3. Чем ординальная арифметика отличается от кардинальной? Приведите пример некоммутативности.
4. Что такое предельный ординал и какой новый шаг появляется в трансфинитной индукции?
5. Что такое сюрреальное число и какие числовые системы оно вбирает в себя?
6. Чем сюрреальные числа отличаются от нестандартных моделей? В каком смысле одно расширение «преднамеренное», а другое — нет?

Что дальше?

Мы вышли за пределы натурального ряда — в нестандартные модели, ординалы и сюрреальные числа. Теперь мы возвращаемся к вопросу о пределах вычислимого. В главе 30 мы изучим классы сложности и иерархию вычислительной сложности. Коснёмся и вопроса о соотношении классов P и NP .

§ 29.6 Упражнения

Теорема компактности и нестандартные модели.

1. Докажите теорему компактности для логики первого порядка из теоремы о полноте Гёделя. Подсказка: если Γ не имеет модели, из него выводится противоречие. Вывод конечен, поэтому противоречие выводится из конечного подмножества Γ .
2. Покажите, что порядковый тип счётной нестандартной модели PA есть $\omega + (\mathbb{Q} \times \mathbb{Z})$. Подсказка: воспользуйтесь тем, что любые два счётных плотных линейно упорядоченных множества без концов изоморфны (теорема Кантора о плотных порядках).
3. * Объясните, почему схема индукции в PA первого порядка не исключает нестандартные элементы. Какое свойство натуральных чисел невыразимо в языке первого порядка и почему это принципиально?

Ординалы и ординальная арифметика.

4. Докажите, что $1 + \omega = \omega$. Постройте явный порядковый изоморфизм между

$$\{*\} \cup \{0, 1, 2, \dots\}$$

и

$$\{0, 1, 2, \dots\}.$$

Здесь $*$ предшествует всем натуральным числам.

5. Докажите, что $\omega + 1 > \omega$. Покажите, что в $\omega + 1$ есть наибольший элемент. В ω — нет. Следовательно, они не изоморфны как порядки.
6. Докажите, что $2 \cdot \omega = \omega$.
7. Покажите, что $\omega \cdot 2 > \omega$. Опишите порядковые типы обоих ординалов.
8. * Докажите ассоциативность ординального сложения $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$. Проведите трансфинитную индукцию по γ . Отдельно разберите предельный шаг.
9. Выпишите наследственную запись числа 5 по основанию 2 и вычислите первые пять элементов последовательности Гудстейна, начавшейся с 5. Для каждого

элемента выпишите наследственную запись по текущему основанию. Проверьте на первых трёх элементах, что ординальная мера строго убывает.

Сюрреальные числа.

10. Вычислите следующие сюрреальные числа:

- a) $\{1 \mid\}$
- b) $\{-1 \mid 0\}$
- c) $\{-1 \mid 1\}$

Какие действительные числа они представляют?

11. Покажите, что в сюрреальных числах $\varepsilon = \{0 \mid 1, \frac{1}{2}, \frac{1}{4}, \dots\}$ является бесконечно малым. То есть $0 < \varepsilon < \frac{1}{n}$. Это верно для любого натурального n .
12. * Объясните структурное различие между нестандартными моделями арифметики и сюрреальными числами. В каком смысле одно расширение является «непреднамеренным», а другое — «преднамеренным»? Почему ω в сюрреальных числах не удовлетворяет аксиомам Пеано?

Нестандартный анализ.

13. * Объясните, как построить нестандартную модель действительных чисел (${}^*\mathbb{R}$), опираясь на теорему о компактности. Набросайте схему: добавьте константу c . Добавьте аксиомы $c > n$. Их должно быть по одной на каждое натуральное n . Примените компактность к конечным подтеориям.
14. * Рассмотрим «доказательство» того, что все ординалы счётны. База: 0 счётен. Если все ординалы, меньшие α , счётны, то α — либо последователь $\beta + 1$ счётного β , либо супремум счётного множества счётных ординалов, поэтому α счётен. По трансфинитной индукции заключаем, что все ординалы счётны. Где ошибка? Подсказка: множество всех счётных ординалов само является ординалом — его называют ω_1 . Сколько слагаемых в объединении $\omega_1 = \bigcup_{\beta < \omega_1} \beta$?

ПРОЕКТ Ординальная завершаемость: последовательности Гудстейна

Инструмент проекта вычисляет последовательности Гудстейна и ординальные меры их элементов: по натуральному числу он печатает наследственную запись, шаги последовательности и меры в нормальной форме Кантора, проверяя строгое убывание мер. Убывание меры — это доказательство завершаемости, и инструмент ведёт его механически на любом заданном горизонте шагов.

① Наследственная запись

Представьте число n наследственной записью по основанию b : коэффициенты меньше b , а показатели — снова наследственные записи. Реализуйте парсер и печать записи и проверьте обратимость: разбор печати возвращает исходное число для всех n до 10^4 и оснований от 2 до 10.

② Шаг Гудстейна

Реализуйте шаг на самой записи, без вычисления числа: подъём основания $b \rightarrow b + 1$ во всех показателях и вычитание единицы из последней ненулевой позиции. Вычислите первые элементы последовательностей для n от 0 до 10 и сверьте с ручным счётом: последовательность для 5 начинается с 5, 27, 255, 467, 775.

③ **Ординальная мера**

Представьте ординалы ниже ε_0 в нормальной форме Кантора (*Cantor normal form*): список пар (коэффициент, степень), где степень — снова такой список. Реализуйте сравнение ординалов и вычитание единицы из положительного ординала, затем меру элемента: наследственная запись по основанию b с заменой b на ω .

④ **Проверка убывания**

Для каждого n от 0 до 100 проследите первые 1000 шагов последовательности и проверьте инвариант: мера каждого элемента строго меньше меры предыдущего. Для $n = 4$ напечатайте меры первых десяти элементов и покажите на них главное: элементы растут, меры убывают.

⑤ **Рубеж ε_0**

Печатайте меры в нормальной форме Кантора и следите за башнями степеней: высота башни в мере любого элемента не превышает высоты башни у начального элемента. В отчёте объясните, почему доказательство завершаемости ведётся трансфинитной индукцией до ε_0 и почему индукция PA, до этого рубежа не доходящая, не может его повторить.

Итог: инструмент, который по натуральному числу строит последовательность Гудстейна, печатает элементы и их ординальные меры и проверяет строгое убывание мер на заданном горизонте шагов.

XXX

Сложность и NP-полнота

Глава 26 установила границу вычислимого: существуют задачи, которые не решает никакой алгоритм, и это доказано. Разрешимость отвечает на вопрос «можно ли решить в принципе?» Естественная реакция — считать разрешимые задачи решёнными: раз программа существует, проблема закрыта. Следующий вопрос — «как быстро можно решить?»

Задача коммивояжёра: даны n городов и попарные расстояния. Найти кратчайший маршрут, проходящий через каждый город ровно один раз. Алгоритм полного перебора просматривает все $n!$ маршрутов. Алгоритм корректен и завершается для любого n . Для $n = 100$ число маршрутов превышает количество атомов в наблюдаемой Вселенной. Алгоритм *существует*, но ответа не дожидаться.

Коммивояжёр не одинок. Расписание: можно ли выполнить все работы к сроку, если каждая требует своих ресурсов? Карта: можно ли покрасить страны тремя цветами так, чтобы соседние различались? Сумма подмножества: есть ли набор чисел с заданной суммой? Для каждой из этих задач лучший известный алгоритм остаётся экспоненциальным — и полвека поисков не дали полиномиального.

Интуиция подсказывает: проблема в экспоненциальном переборе. Значит, нужно определить, какие задачи решаются быстро, а какие — нет. Быстро — это за полиномиальное время: число шагов растёт как n^k для некоторой константы k . Класс таких задач называется P. Выбор полинома — не произвол: полиномы замкнуты относительно композиции, и класс P не зависит от модели вычислений.

Полиномиальный рост n^2 — повседневность, а экспоненциальный 2^n уже при небольших n делает полный перебор неисполнимым. Экспоненту можно прочувствовать через развилки: одна развилка — две дороги, двадцать — больше миллиона, а триста — больше, чем атомов в наблюдаемой Вселенной.

Есть задачи, для которых быстрого алгоритма не видно. Коммивояжёра, раскраску и рюкзак (*knapsack*) объединяет общее свойство: предложенное решение можно *проверить* за полиномиальное время. Маршрут — сложить n чисел и сравнить с эталоном, раскраску — пройти по границам и убедиться, что соседи различаются, набор грузов — сложить веса и сравнить с целевым числом. Пример из быта — sudoku, где решение ищут часами, а предъявленную расстановку проверяют за минуты. Класс задач, в которых решение проверяется быстро, называется NP. Найти же оптимальный маршрут среди всех возможных — совсем другая трудность.

Здесь возникает главный вопрос теории сложности: совпадает ли класс задач, решаемых быстро, с классом задач, проверяемых быстро? Если $P = NP$, то творческий акт поиска решения принципиально не сложнее верификации. По образной формуле Скотта Ааронсона: всякий, кто способен оценить симфонию, был бы Моцартом, а всякий, кто следит за рассуждением шаг за шагом, — Гауссом. Если $P \neq NP$ — а большинство теоретиков склоняется к этому — то существуют задачи, для которых проверка ответа *неизмеримо* легче его нахождения. Вопрос открыт полвека и остаётся главной нерешённой проблемой computer science.

Эталоном трудности станет SAT — задача, уже разобранный в главе 14: верификация схем из главы 11 свелась именно к ней. Если бы SAT решалась быстро, то $P = NP$ — в этом теорема Кука–Левина и смысл первой NP-полной задачи.

Время — не единственный ресурс вычисления. В главе 11 глубина была временем, а размер — площадью. В теории сложности время и память — два ресурса, а случайность — третий, и каждому соответствует свой спектр классов.

NP-полнота не означает безнадёжности. Она утверждает трудность в худшем случае. Реальные экземпляры (*instances*) — расписания, схемы, сети — часто проще, и на этой разнице строится инженерия NP-трудных задач. Как измерить трудность задачи, если алгоритм для неё существует, но ответа не дожидаться?

ИСТОРИЯ Открытая проблема: как вопрос о сложности изменил математику

Юрис Хартманис и Ричард Стернс в 1965 году в статье «On the Computational Complexity of Algorithms»²¹² впервые сделали ресурс вычисления предметом математического исследования. Они ввели классы сложности и доказали теорему об иерархии по времени: машине, которой разрешено больше времени, доступно больше задач. Так за шесть лет до Кука возникла систематическая теория сложности.

Стивен Кук в 1971 году опубликовал статью «The Complexity of Theorem-Proving Procedures»²¹³, где доказал, что задача выполнимости булевой формулы (SAT) NP-полна. Любой язык из NP полиномиально сводится к SAT. Кук ввёл само понятие NP-полноты как инструмент классификации задач по их вычислительной трудности.

Леонид Левин в 1973 году, работая в Москве независимо от Кука, пришёл к тому же результату, но шёл от переборных задач. Он искал «универсальные переборные задачи», самые трудные в своём классе, к которым сводятся все остальные переборные. Теорема Кука–Левина дала первый естественный пример NP-полной задачи и открыла путь к доказательству NP-полноты сотен других задач.

²¹²J. Hartmanis, R. E. Stearns. «On the Computational Complexity of Algorithms». Transactions of the American Mathematical Society, 1965.

²¹³S. A. Cook. «The Complexity of Theorem-Proving Procedures». Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 1971.

Ричард Карп опубликовал «Reducibility Among Combinatorial Problems»²¹⁴. Статья содержала 21 NP-полную задачу и цепочки полиномиальных сведений между ними. Эта работа показала, что NP-полнота — общее свойство десятков вычислительных проблем: клика, вершинное покрытие, гамильтонов цикл, разбиение, рюкзак — все они полиномиально эквивалентны SAT. Майкл Гари и Дэвид Джонсон опубликовали «Computers and Intractability: A Guide to the Theory of NP-Completeness»²¹⁵. Книга Гари и Джонсона — каталог сотен NP-полных задач и систематическая техника доказательства NP-полноты через сведения — стала настольной для теоретиков computer science.

В 2000 году Clay Mathematics Institute включил проблему P против NP в число семи задач тысячелетия с премией в миллион долларов. Полвека спустя вопрос Кука остаётся открытым: эквивалентно ли нахождение решения его проверке. Ответа нет ни в одну сторону.

ОБЗОР ГЛАВЫ

Начнём с коммивояжёра: для ста городов маршрутов больше, чем атомов во Вселенной, и полный перебор не оставляет шансов. Увидим, что значит «быстро»: класс P, устойчивый к смене модели вычислений, и класс NP, где решение проверяется быстро. Главный вопрос теории — совпадает ли P с NP — требует формального аппарата: полиномиальных сведений и теоремы Кука–Левина, дающей первую NP-полную задачу — SAT. Цепочки сведений распространяют NP-полноту на клику и вершинное покрытие, а карта цепочек наметит маршруты дальше — к гамильтонову циклу и рюкзаку. Практикам нужны стратегии для NP-трудных задач: точные алгоритмы, приближения, эвристики и параметризованная сложность. В конце мы выйдем за пределы P и NP: память порождает PSPACE, случайность — BPP, а чередование кванторов — полиномиальную иерархию.

После этой главы вы сможете:

1. определять классы P и NP и объяснять, почему полиномиальное время — критерий эффективности;
2. проверять решение за полиномиальное время и формулировать вопрос $P = NP$;
3. сводить задачи полиномиальными сведениями и строить простые сведения;
4. объяснять теорему Кука–Левина и пользоваться SAT как эталоном NP-полноты;
5. различать время, память и случайность как ресурсы и объяснять иерархии классов.

²¹⁴R. M. Karp. «Reducibility Among Combinatorial Problems». 1972.

²¹⁵M. R. Garey, D. S. Johnson. «Computers and Intractability: A Guide to the Theory of NP-Completeness». 1979.

§ 30.1 Класс P

Неформально, P — это класс задач, для которых существует *эффективный* алгоритм. Формально, «эффективный» означает «полиномиальный»: время работы ограничено полиномом от размера входа.

У выбора полинома есть практические оговорки:

- Алгоритм за время $O(n^{100})$ формально принадлежит P, но на практике неприемлем.
- Экспоненциальный алгоритм за $O(1.0001^n)$ может работать быстро для умеренных n .
- Алгоритм за $O(2^n)$ перестает работать уже при небольших n .

Полиномиальное время — лучший из известных формальных критериев эффективности. На практике большинство полиномиальных алгоритмов имеют небольшую степень (обычно $O(n)$, $O(n^2)$ или $O(n^3)$). Задачи, для которых известен лишь экспоненциальный алгоритм, как правило, трудны.

ОПРЕДЕЛЕНИЕ 30.1 Класс P

Классом P называется класс языков (задач разрешения, decision problems), разрешимых детерминированной машиной Тьюринга за полиномиальное время:

$$P = \bigcup_{k \geq 1} \text{TIME}(n^k).$$

Здесь $\text{TIME}(n^k)$ — класс языков, разрешимых машиной Тьюринга за время $O(n^k)$.

Класс P устойчив к изменению модели вычислений: одноленточная и многоленточная машины Тьюринга и RAM-машина меняют время работы не более чем на полиномиальный фактор, поэтому принадлежность классу P при этом сохраняется.

Пример Задачи из P

К классу P принадлежат:

- Сортировка ($O(n \log n)$), поиск в ширину и глубину.
- Кратчайшие пути (алгоритм Дейкстры), минимальное остовное дерево (Прим, Крускал).
- 2-раскраска (проверка двудольности), эйлеров цикл.
- Проверка планарности графа.
- Линейное программирование (алгоритм эллипсоидов, interior-point methods).
- Проверка простоты числа (алгоритм AKS, 2002 — прорыв: детерминированный полиномиальный).
- Разбор контекстно-свободных грамматик (алгоритм CYK, $O(n^3)$).

Многие из этих задач долгое время имели статус «неизвестно, в P ли они». Линейное программирование попало в P только в 1979 году (алгоритм эллипсоидов Хачияна). Проверка простоты попала в P в 2002 году. Наше знание о классе P меняется — новые алгоритмы перемещают в P задачи, ранее считавшиеся более сложными.

Замечание Почему P — это полином, а не что-то другое?

Определение P через полиномиальное время — не единственный мыслимый выбор. Можно было бы определить эффективность через линейное время или через $O(n \log n)$. Выбор полинома обоснован двумя структурными свойствами.

Первое — замкнутость относительно композиции. Процедура, использующая другой алгоритм как подпрограмму, должна оставаться в том же классе эффективности. Полином от полинома есть полином. Класс $O(n \log n)$ этим свойством не обладает: композиция двух $O(n \log n)$ -алгоритмов даёт $O(n \log^2 n)$ — мы покидаем класс. Полином — минимальный класс, замкнутый относительно операции «использовать как подпрограмму».

Второе — инвариантность относительно модели вычислений. Переход от одноленточной машины Тьюринга к многоленточной или RAM-машине меняет время работы не более чем на полиномиальный фактор. Задача, полиномиальная на одной модели, полиномиальна на всех. Свойство «быть линейным» такой инвариантностью не обладает.

Эти два свойства вместе делают выбор полинома математической необходимостью. Полиномиальное время — минимальный класс, одновременно замкнутый относительно композиции и устойчивый относительно смены модели вычислений.

§ 30.2 Класс NP

NP расшифровывается как Nondeterministic Polynomial time — и, вопреки распространённой ошибке, не как Not Polynomial. Это класс задач, решение которых *проверяется* за полиномиальное время при наличии *сертификата* (доказательства, свидетеля).

ОПРЕДЕЛЕНИЕ 30.2 Класс NP

Классом NP называется класс языков L , для которых существует полиномиальный верификатор V и полином p , такие что:

$$w \in L \leftrightarrow \exists c : |c| \leq p(|w|) \text{ и } V(w, c) \text{ принимает.}$$

Неформально: «угадай сертификат, проверь за полиномиальное время». Верификатор V — детерминированный полиномиальный алгоритм.

Эквивалентное определение: языки, разрешимые *недетерминированной* машиной Тьюринга за полиномиальное время (отсюда и название класса).

Примечание Почему через верификатор?

Почему NP определяют через верификатор — «угадай сертификат, проверь за полином» — а не через недетерминированную машину, как сказано в названии? Определение через верификатор ближе к интуиции: большинство практических NP-задач (коммивояжёр, раскраска, SAT) естественно формулируются именно как «найти решение» и «проверить решение». Недетерминированная МТ — эквивалентная, но менее наглядная формулировка: «угадывание» сертификата соответствует выбору перехода в каждой недетерминированной развилке. Альтернативное определение — через экзистенциальную характеристику: $L \in \text{NP}$, если $w \in L$ равносильно

$$\exists c : R(w, c)$$

для полиномиально разрешимого предиката R . Это определение подчёркивает логическую природу класса и напрямую связывает NP с теоретико-доказательной сложностью.

Пример Задачи из NP

- **SAT**: сертификат — выполняющее присваивание переменных. Проверка: подставить присваивание в формулу и вычислить её значение.
- **Гамильтонов цикл**: сертификат — сам цикл. Проверка: пройти по циклу и убедиться, что каждое ребро существует и вершины не повторяются.
- **Вершинное покрытие** размера $\leq k$: сертификат — множество вершин. Проверка: убедиться, что размер $\leq k$ и каждое ребро инцидентно покрытию.
- **Клика** размера $\geq k$: сертификат — множество из k вершин. Проверка: убедиться, что между ними существуют все попарные рёбра (их $\frac{k(k-1)}{2}$).
- **k-раскраска**: сертификат — назначение цветов вершинам. Проверка: для каждого ребра цвета его концов различны.
- **Сумма подмножества**: сертификат — подмножество чисел. Проверка: сложить числа и сравнить с целевой суммой.

$$P \subseteq \text{NP}$$

Включение P в NP следует из определения: если задача решается за полиномиальное время, то в качестве сертификата подойдёт пустая строка, а верификатор выполнит алгоритм решения.

Вопрос о равенстве

$$P = \text{NP}$$

— одна из семи задач тысячелетия (Clay Mathematics Institute, премия 1,000,000). Большинство экспертов полагают, что $P \neq NP$ ²¹⁶, но доказательства нет ни для одной из сторон.

Класс NP, как и P, замкнут относительно основных операций. Пересечение, объединение, конкатенация и замыкание Клини не выводят за пределы NP.

УТВЕРЖДЕНИЕ 30.1 Замкнутость NP

Пусть $L_1, L_2 \in NP$. Тогда:

1. $L_1 \cap L_2 \in NP$.
2. $L_1 \cup L_2 \in NP$.
3. $L_1 L_2 \in NP$ (конкатенация).
4. $L_1^* \in NP$ (звезда Клини).

Набросок доказательства

Докажем построением: для каждой из операций предъявим верификатор, собирающийся из верификаторов для L_1 и L_2 .

Пересечение и объединение: верификатор для $L_1 \cap L_2$ запускает верификаторы V_1 и V_2 на одном сертификате и принимает, если оба приняли. Для объединения — если хотя бы один принял.

Конкатенация: вход w недетерминированно разбивается на две части: $w = uv$. Верификатор для L_1 проверяет часть u . Верификатор для L_2 проверяет часть v .

Звезда Клини: вход w недетерминированно разбивается на непустые фрагменты: $w = w_1 w_2 \dots w_k$ (число фрагментов не превышает $|w|$, так как каждый фрагмент непуст). Для каждого фрагмента w_i вызывается верификатор L_1 . Если все фрагменты принадлежат L_1 , вход принимается.

Эта замкнутость — не техническая мелочь. Она означает, что NP-задачи можно комбинировать: если вы умеете проверять сертификаты для L_1, L_2 , вы автоматически умеете проверять сертификаты для их пересечения или конкатенации. Класс NP *алгебраически устойчив* — свойство, которое мы ожидаем от разумного определения эффективной проверки.

Вопрос P против NP не поддаётся решению полвека, и одна из причин — барьер релятивизации. Бейкер, Гилл и Соловей в 1975 году доказали: стандартный метод теории сложности, диагонализация, не может разрешить этот вопрос²¹⁷. А именно: существуют оракулы A и B , такие что $P^A = NP^A$, но $P^B \neq NP^B$.

²¹⁶Опрос Gasarch 2012 года показал, что около 80% респондентов верят в $P \neq NP$. Около 9% — в $P = NP$. Остальные затруднились с ответом или считают проблему независимой. Повторный опрос 2019 года дал схожие результаты.

²¹⁷T. Baker, J. Gill, R. Solovay. «Relativizations of the P=?NP Question». SIAM Journal on Computing, 1975.

Оракульная машина Тьюринга получает дополнительную ленту-оракул: за один шаг она записывает слово и узнаёт, принадлежит ли оно языку A . Классы P^A и NP^A — это P и NP для машин с таким оракулом. Диагонализация *релятивизируется*: её доказательства лишь симулируют машины как чёрные ящики, не заглядывая в их устройство, поэтому тот же аргумент работает с любым оракулом O на месте ленты. Разрешила бы диагонализация вопрос P против NP , она установила бы $P^O \neq NP^O$ для всех O , что противоречит оракулу A с $P^A = NP^A$. Значит, доказательство потребует методов, выходящих за пределы диагонализации.

Мы определили классы P и NP как две характеристики разрешимых задач по требуемым ресурсам. Но сказать «эта задача в NP » — значит сказать о ней довольно мало. В NP лежат и тривиальные задачи из P (вроде сортировки), и предположительно трудные (коммивояжёр, SAT). Чтобы различать задачи внутри NP , нам нужно понятие сравнительной трудности. Мы не умеем доказывать нижние оценки для задач из NP (для этого пришлось бы разделить P и NP), но мы умеем доказывать, что одна задача не легче другой. Инструмент для этого — полиномиальное сведение, прямой наследник map-one сведения (сведения «многие-в-одно») из теории вычислимости.

Если мы умеем эффективно преобразовывать входы задачи A во входы задачи B , то B не может быть легче A . А если к B сводится любая задача из NP , то B — NP -трудна. Этот раздел — основа современной теории сложности: иерархия трудности внутри NP , построенная на сведениях.

Проверка Классы P и NP

1. Почему в качестве эталона эффективности выбран полином, а не $O(n \log n)$ или линейное время?
2. Почему включение $P \subseteq NP$ доказывается столь коротко, хотя интуитивно «найти решение» труднее, чем «проверить»?

§ 30.3 Полиномиальные сведения и NP-полнота

Сравнивать задачи по трудности позволяют *сведения* — тот же приём, что и в теории вычислимости. Идея сведения знакома из быта — две непохожие задачи иногда оказываются одной задачей. В игре «числовой скрэбл» двое по очереди выбирают числа от 1 до 9, и выигрывает тот, кто первым соберёт три числа с суммой 15. Если расположить числа в магическом квадрате, игра превращается в крестики-нолики: каждая тройка с суммой 15 — это линия на доске. Стратегия, выигрывающая в крестики-нолики, переносится в числовой скрэбл без изменений.

Сведение $A \leq_p B$ означает: «если бы мы умели эффективно решать B , то смогли бы эффективно решить и A ». Технически: любой вход задачи A можно за полиномиальное время преобразовать во вход задачи B , так что ответ сохраняется («да» переходит в «да», «нет» — в «нет»). Преобразование не обязано быть хитрым —

достаточно, чтобы оно было корректным и быстрым. Если такое преобразование существует, то A полиномиально не труднее B : решив B за полиномиальное время, мы автоматически решили и A за полиномиальное время. А именно:

1. преобразуем вход A во вход B (полиномиальное время),
2. решаем B (полиномиальное время),
3. возвращаем ответ.

Композиция двух полиномов — полином.

Это прямой наследник many-one сведения из теории вычислимости (глава 26). Разница — в ограничении на время. Преобразование должно работать за полиномиальное время, а не просто быть вычислимым. Это ограничение — принципиальное: оно гарантирует, что эффективное решение B даёт эффективное решение A , а не просто теоретическую разрешимость.

ОПРЕДЕЛЕНИЕ 30.3 Полиномиальное сведение (*polynomial-time reduction*)

Язык A называется **полиномиально сводимым** к языку B , обозначается $A \leq_p B$, если существует вычислимая за полиномиальное время функция f , такая что

$$w \in A \leftrightarrow f(w) \in B.$$

ЛОВУШКА Сведение: размер выхода полиномиален

Полиномиальное сведение легко считать бесплатным по размеру — это ошибка. Сведение — вычислимая за полиномиальное время функция, и размер её *выхода* тоже должен быть полиномиален относительно входа. Экспоненциальный вывод за полиномиальное время невозможен: сама запись результата занимает время, сравнимое с его длиной. Для NP-полноты это существенно: сведение не должно раздувать экземпляр.

УТВЕРЖДЕНИЕ 30.2 Замкнутость относительно сводимости

- Если $A \leq_p B$ и $B \in P$, то $A \in P$.
- Если $A \leq_p B$ и $B \in NP$, то $A \in NP$.

Доказательство

Докажем оба утверждения построением: для $A \in P$ соберём полиномиальный алгоритм, а для $A \in NP$ — полиномиальный верификатор.

Для первого утверждения: по определению сводимости существует полиномиально вычислимая функция f , такая что $w \in A$ тогда и только тогда, когда $f(w) \in B$. Поскольку $B \in P$, существует полиномиальный алгоритм, решающий B . Тогда алгоритм для A : вычислить $f(w)$ за полиномиальное время и

подать результат на вход алгоритму для B . Композиция двух полиномов — полином, следовательно $A \in P$.

Для второго утверждения: если $B \in NP$, то существует полиномиальный верификатор $V_B(y, c)$, где y — вход, c — сертификат. Определим верификатор для A : $V_A(w, c) = V_B(f(w), c)$. Если $w \in A$, то $f(w) \in B$, и существует сертификат c для $f(w)$, который работает и для w . Верификатор V_A полиномиален как композиция полиномиальной f и полиномиального V_B .

Полиномиальное сведение позволяет определить «самые трудные» задачи в NP.

ОПРЕДЕЛЕНИЕ 30.4 NP-трудная задача

Задача B называется **NP-трудной**, если любая задача из NP полиномиально сводится к B : $\forall A \in NP, A \leq_p B$.

ОПРЕДЕЛЕНИЕ 30.5 NP-полная задача

Задача B называется **NP-полной**, если $B \in NP$ и B является NP-трудной.

ЛОВУШКА NP-полнота $\neq$ NP-трудность

NP-полную и NP-трудную задачи часто считают одним и тем же — это ошибка. NP-трудная задача — та, к которой сводится любая задача из NP, но сама она может лежать вне NP. NP-полная — NP-трудная и лежащая в NP. Проблема остановки NP-трудна, но не в NP: она вообще неразрешима. NP-полнота — «самая трудная в NP». NP-трудность — «не легче любой NP-задачи».

УТВЕРЖДЕНИЕ 30.3 Значимость NP-полноты

Если хотя бы одна NP-полная задача принадлежит P, то $P = NP$. Если хотя бы одна NP-полная задача не принадлежит P, то $P \neq NP$ и ни одна NP-полная задача не принадлежит P.

Доказательство

Докажем в обе стороны.

Сначала предположим, что B — NP-полная задача и $B \in P$. Для любой $A \in NP$ по определению NP-полноты $A \leq_p B$. По предыдущему утверждению о замкнутости, из $B \in P$ и $A \leq_p B$ следует $A \in P$. Таким образом, $NP \subseteq P$, а поскольку $NP \supseteq P$ всегда верно, получаем $P = NP$.

Обратно: если $P \neq NP$, то никакая NP-полная задача не может принадлежать P — иначе по первой части мы получили бы $P = NP$.

Таким образом, все NP-полные задачи образуют единый класс эквивалентности: либо все они решаются эффективно, либо ни одна. Подобно ветеринару, признавшему в пуделе и сенбернаре один вид, теоретик видит за непохожими задачами одну: лекарство, работающее для одной из них, сработает для всех.

30.3.1 Теорема Кука–Левина

Существование NP-полных задач доказывается — их предъявляет теорема Кука–Левина. Теорема Кука (1971) и независимо Левина (1973) доказала, что SAT NP-полна — исторически первая NP-полная задача и концептуально отправная точка. (Подробное введение в SAT, кодирование задач в КНФ и устройство SAT-решателей — в главе 14.)

ТЕОРЕМА 30.4 Теорема Кука–Левина²¹⁸

SAT NP-полна.

Набросок доказательства

Докажем в две части: сначала покажем, что SAT принадлежит NP, затем — что SAT NP-трудна.

SAT принадлежит NP: сертификат — выполняющее присваивание переменных, а проверка сводится к подстановке и вычислению формулы за линейное время.

Докажем NP-трудность: пусть $L \in \text{NP}$ и M — недетерминированная МТ, принимающая L за время $p(n)$. Построим КНФ-формулу φ , кодирующую таблицу вычисления M на входе w — как на рис. 76 из главы 14.

Переменные формулы — это $x_{t,p,s}$: в момент времени t в позиции p ленты записан символ s . Второй набор — $y_{t,i,q}$: в момент времени t головка находится в позиции i , а состояние — q .

Дизъюнкты кодируют корректную начальную конфигурацию (вход w в начальных позициях). Они также кодируют корректность переходов: для каждого момента t и позиции p содержимое ленты, состояние и положение головки должны изменяться в соответствии с δ . Наконец, они кодируют достижение принимающего состояния к моменту $p(n)$.

Размер формулы φ : $O(p(|w|)^2)$. φ выполнима тогда и только тогда, когда существует принимающее вычисление M на w .

Таблица вычисления МТ — это матрица размера $p(n) \times p(n)$, где каждая ячейка зависит только от трёх ячеек предыдущей строки. Именно эта локальность и делает

²¹⁸S. A. Cook. «The Complexity of Theorem-Proving Procedures». STOC, 1971. Л. А. Левин. «Универсальные задачи перебора». Проблемы передачи информации, 1973. Независимо и одновременно установили NP-полноту SAT. На Западе она чаще известна как теорема Кука, в России — как теорема Кука–Левина.

кодирование возможным: корректность всего вычисления выражается конъюнкцией дизъюнктов фиксированного размера, а не единой формулой, описывающей вычисление целиком.

Остальные сотни NP-полных задач доказываются цепочками сведений от SAT — одного универсального представителя достаточно. Любая задача из NP кодируется как SAT-формула. Именно на этой идее работают современные SAT-решатели, справляющиеся с миллионами переменных на структурированных промышленных примерах.

Замечание Почему теорема Кука–Левина открыла путь к доказательству NP-полноты?

До теоремы Кука–Левина NP был классом, определённым через недетерминированные машины Тьюринга — абстрактно и неудобно для построения конкретных доказательств NP-трудности. После неё появился представитель — SAT — к которому можно сводить другие задачи.

Доказательство Кука показывает, что SAT *выражает* саму структуру недетерминированного вычисления. Таблица вычисления МТ, локальные переходы, принимающее состояние — всё переводится в дизъюнкты фиксированного размера. SAT становится языком, на котором можно сформулировать любую задачу из NP.

SAT — точка контакта между абстрактным определением NP и конкретными комбинаторными задачами. Если SAT решается за полиномиальное время — P и NP совпадают.

Сведение — это перевод задачи на другой язык. Как перевод текста сохраняет смысл, меняя форму, так полиномиальное сведение переводит задачу A в задачу B , сохраняя ответ и добавляя не более чем полиномиальное искажение размера. Если B решается быстро, то и A решается быстро — через перевод и решение B : переводим вход, решаем, возвращаем ответ. NP-полные задачи — универсальные приёмники таких переводов: к ним сводится любая задача из NP.

30.3.2 Зоопарк NP-полных задач

После доказательства NP-полноты SAT цепочки полиномиальных сведений установили NP-полноту сотен задач из самых разных областей. Название этого раздела — не метафора: в теории сложности существует настоящий «зоопарк» — Complexity Zoo²¹⁹ — постоянно пополняемый онлайн-каталог классов сложности, начатый Скоттом Ааронсоном и поддерживаемый сообществом теоретиков.

²¹⁹Scott Aaronson, Greg Kuperberg, Christopher Granade. Complexity Zoo — live wiki-каталог классов сложности, более 550 классов на октябрь 2024 года: <https://complexityzoo.net>. Для начинающих — Petting Zoo, 17 основных классов: https://complexityzoo.net/Petting_Zoo.

К 2024 году в нём собрано более 550 классов — от P и NP до экзотических обитателей вроде SBP, QMA(2) и BQP/qpoly. Наш «зоопарк» в этой главе скромнее: несколько десятков NP-полных задач, сгруппированных по областям.

УТВЕРЖДЕНИЕ 30.5 Стандартные цепочки сведений

Классический маршрут доказательства NP-полноты:

$$\text{SAT} \leq_p \text{3-SAT} \leq_p \text{Vertex Cover} \leq_p \text{Clique} \leq_p \text{Independent Set}.$$

Другая ветвь:

$$\text{3-SAT} \leq_p \text{Subset Sum} \leq_p \text{Partition} \leq_p \text{Knapsack}.$$

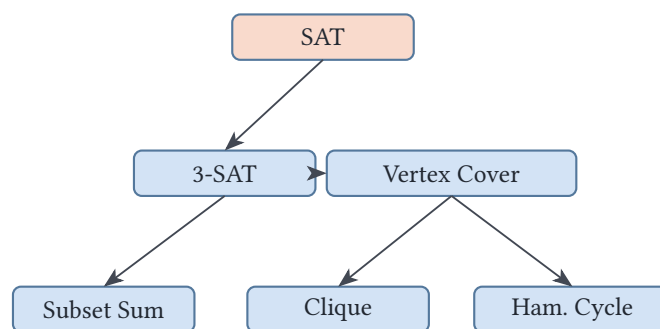
Ещё одна:

$$\text{3-SAT} \leq_p \text{3-Coloring} \leq_p \text{Vertex Cover} \leq_p \text{Hamiltonian Cycle} \leq_p \text{TSP}.$$

Переход $\text{SAT} \leq_p \text{3-SAT}$ расщепляет длинные дизъюнкты с помощью вспомогательных переменных: дизъюнкт из m литералов заменяется на $m - 1$ троек со свежими переменными.

Каждое отдельное сведение строится по одному образцу. Суть маршрута в том, что, дойдя от SAT до задачи B , мы автоматически доказали NP-трудность B . Доказательства остальных переходов опущены: каждый — стандартная конструкция, и ниже подробно разобраны три её образца: сведение IND к CLIQUE, 3-SAT к вершинному покрытию и 3-SAT к клике.

Рис. 115 собирает эти цепочки в единую карту. Каждая стрелка — полиномиальное сведение, а SAT (вверху) — исторически первая NP-полная задача, от которой расходятся все остальные доказательства NP-трудности.



Каждая стрелка: $\leq_p$ (полиномиальное сведение)

Рис. 115. Канонические цепочки полиномиальных сведений.

Прежде чем разбирать технически насыщенное сведение, рассмотрим пример, который укладывается в одну идею.

Пример Сведение IND к CLIQUE

Независимое множество (Independent Set, IND): даны граф G и число k , существует ли множество из k попарно несмежных вершин? **Клика** (CLIQUE): даны граф G и число k , существует ли множество из k попарно смежных вершин?

Это — одна и та же задача, вывернутая наизнанку. Возьмём дополнение графа $\bar{G}$: ребро между вершинами есть в $\bar{G}$ тогда и только тогда, когда его нет в G . Множество вершин, попарно не связанных рёбрами в G , становится множеством вершин, попарно связанных рёбрами в $\bar{G}$. Независимое множество размера k в G — это клика размера k в $\bar{G}$.

Сведение: $f(G, k) = (\bar{G}, k)$. Построение дополнения графа требует $O(|V|^2)$ времени (по матрице смежности), так что f полиномиальна. Корректность: $(G, k) \in \text{IND} \leftrightarrow (\bar{G}, k) \in \text{CLIQUE}$.

Сведение дополнением графа — прозрачное: задача превращается в себя, лишь рёбра меняются на не-рёбра. Технические сведения устроены иначе: на каждый фрагмент входа строится свой гаджет, и весь граф собирается из этих деталей.

Пример Сведение: 3-SAT к вершинному покрытию

Дана 3-КНФ формула φ с m дизъюнктами от n переменных. Построим граф G и число k так, чтобы φ была выполнима тогда и только тогда, когда в G есть вершинное покрытие размера $\leq k$.

Конструкция графа:

- Для каждой переменной x_i создаём ребро-гаджет из двух вершин: x_i и $\bar{x}_i$.
- Для каждого дизъюнкта $(l_1 \vee l_2 \vee l_3)$ создаём треугольник из трёх вершин с метками l_1, l_2, l_3 .
- Соединяем каждую вершину треугольника-дизъюнкта с соответствующей литеральной вершиной в переменном гаджете.
- Положим $k = n + 2m$.

Корректность:

- Прямое направление (φ выполнима $\Rightarrow$ вершинное покрытие размера $\leq k$): из каждого переменного ребра берём истинный литерал (n вершин). В каждом треугольнике-дизъюнкте оставляем непокрытой вершину, соответствующую истинному литералу (она покрыта со стороны переменного гаджета), и берём две оставшиеся вершины ($2m$ вершин).
- Обратное направление (покрытие размера $\leq k \Rightarrow \varphi$ выполнима): для покрытия каждого переменного ребра нужно не менее n вершин, для каждого треугольника-дизъюнкта — не менее $2m$. Итого получается минимум $n + 2m$ вершин. Следовательно, минимальное покрытие содержит ровно одну вершину из каждого переменного ребра (задающую присваивание). Оно со-

держит ровно две из каждого треугольника-дизъюнкта (третья должна быть покрыта со стороны переменного гаджета).

Размер графа: $2n + 3m$ вершин, $k = n + 2m$.

Пример Сведение: 3-SAT к клике

Дана 3-КНФ формула φ с m дизъюнктами. Построим граф G так, чтобы φ была выполнима тогда и только тогда, когда в G есть клика размера m .

Конструкция графа:

- Для каждого дизъюнкта создаём три вершины, по одной на литерал.
- Вершины одного дизъюнкта рёбрами не соединяем.
- Две вершины из разных дизъюнктов соединяем ребром, если их литералы не противоположны: пара $x, \bar{x}$ не связывается.

Корректность:

- Прямое направление (φ выполнима $\Rightarrow$ клика размера m): выполняющее присваивание выбирает в каждом дизъюнкте один истинный литерал. Выбранные m вершин попарно соединены: два истинных литерала не могут быть противоположны.
- Обратное направление (клика размера $m \Rightarrow \varphi$ выполнима): внутри одного дизъюнкта рёбер нет, поэтому из каждого в клику попадает не более одной вершины. В клике m вершин, значит, взята ровно одна из каждого дизъюнкта. Противоположных литералов в клике нет: ребро между ними запрещено конструкцией. Присваивание, делающее выбранные литералы истинными, непротиворечиво и выполняет каждый дизъюнкт.

Граф содержит $3m$ вершин. Число рёбер не превосходит $O(m^2)$, и построение полиномиально.

Основные NP-полные задачи по категориям:

- **Логика:** SAT, 3-SAT, Max-2-SAT, выполнимость схемы.
- **Графы:** клика, независимое множество, вершинное покрытие, гамильтонов цикл, TSP, 3-раскраска.
- **Числа:** сумма подмножества, разбиение, рюкзак, целочисленное линейное программирование.
- **Расписания:** Job Shop Scheduling, планирование с ограничениями.
- **Головоломки:** sudoku, сапёр («Minesweeper»), тетрис. Обобщённый Super Mario Bros. — PSPACE-полон²²⁰ — то есть потенциально сложнее любой NP-полной задачи.

²²⁰E. D. Demaine, G. Viglietta, A. Williams. «Super Mario Bros. Is Harder/Easier Than We Thought». 2016. — обобщённая версия игры PSPACE-полна, а не NP-полна.

Замечание Почему не все NP-задачи — либо лёгкие, либо NP-полные?

После знакомства с NP-полнотой возникает соблазн разделить все задачи из NP на две категории: лёгкие (P) и NP-полные. Интуиция подсказывает: если задача в NP, то она либо решается быстро, либо к ней сводится SAT. Эта дихотомия была бы удобной — но она ложна.

Ричард Ладнер в 1975 году доказал: если $P \neq NP$, то существуют задачи, не лежащие ни в P, ни среди NP-полных — задачи *промежуточной* сложности. $P \neq NP \Rightarrow NP \setminus (P \cup NPC) \neq \emptyset$, где NPC — класс NP-полных задач.

Доказательство Ладнера конструктивно: оно предъявляет искусственный язык, построенный как «SAT с дырками». Язык совпадает с SAT на длинах из специально выбранного множества, а между выбранными длинами оставлены экспоненциальные промежутки, на которых язык пуст.

Почему язык не лежит в P. Длины выбираются перечислением всех полиномиальных машин, и для каждой из них находится длина, на которой она ошибётся на каком-то SAT-входе. Будь язык разреши за полиномиальное время, его машина встретила бы в перечислении и была бы уличена в ошибке.

Почему язык не NP-полон. Полиномиальное сведение SAT к языку переводит вход длины n в строку длины не более n^c для некоторой константы c . Достаточно оставить промежуток шире, чем n^c , чтобы сведение «промахивалось» мимо всех длин, на которых язык совпадает с SAT.

Схема промежутков: длины $n_1 < n_2 < n_3 < \dots$ выбираются так, что $n_{i+1} = 2^{n_i}$. Скачок от n_i к 2^{n_i} превосходит любой полином n^c от n_i при достаточно больших n_i , поэтому сведение не попадает в «рабочие» длины, где язык непуст.

Естественные кандидаты на роль NP-промежуточных задач — факторизация целых чисел и задача об изоморфизме графов. Для них не найдено ни полиномиальных алгоритмов, ни доказательства NP-полноты, и есть косвенные свидетельства (структурные теоремы об интерактивных доказательствах), что они не являются NP-полными. Если $P \neq NP$, то NP содержит бесконечную иерархию: от P через задачи промежуточной трудности до NP-полных.

§ 30.4 Стратегии решения NP-трудных задач

Итак, для NP-полных задач полиномиальный алгоритм маловероятен. Но NP-полные задачи решают каждый день:

- TSP решается точно для тысяч городов (Concorde).
- SAT-решатели справляются с формулами, содержащими миллионы переменных.
- Планировщики в логистике, компиляторы, верификаторы оборудования — все они ежедневно решают NP-трудные задачи на практике.

Теория NP-полноты утверждает трудность в *худшем случае*: полиномиального алгоритма для *всех* случаев не существует. Но конкретные прикладные экземпляры почти всегда имеют структуру, которую можно использовать, — малую древесную ширину, разреженность ограничений, иерархическую декомпозицию.



NP-полнота говорит о худшем случае, а жизнь происходит в среднем. Рюкзак NP-полон, но упаковка багажа не занимает вечность: реальные экземпляры непохожи на те, что строят в доказательствах NP-трудности. Обратное тоже верно: если задача формально в P, но константа в полиноме велика, практика об этом не узнает. Сложность — граница в принципе. Инженерия — то, что сходится на конкретных данных.

Когда точное решение практически недостижимо, инженер меняет цель: вместо оптимального решения он ищет достаточно хорошее. Отказ от оптимальности — осознанный выбор, открывающий путь к решению. В распоряжении практика четыре семейства стратегий:

- Точные алгоритмы с экспоненциальным временем, но с малой константой в экспоненте (branch-and-bound).
- Приближённые алгоритмы с гарантированной точностью.
- Эвристики без гарантий, но с хорошим эмпирическим поведением.
- Параметризованные алгоритмы (FPT), экспоненциальные только по малому параметру.

Первая стратегия — не сдаваться и искать точное решение, используя структурные свойства задачи для отсеечения заведомо бесперспективных ветвей перебора.

Стратегия	Гарантия	Пример
Точные алгоритмы	Оптимальный ответ, экспонента в худшем случае	Метод ветвей и границ, SAT-решатели (CDCL)
Приближённые алгоритмы	Ответ не хуже оптимального в c раз	2-приближение вершинного покрытия
Эвристики	Без гарантий, хорошее поведение на практике	Локальный поиск, имитация отжига
Параметризованные (FPT)	Точный ответ, экспонента только по параметру k	$O(2^k \cdot n)$ для вершинного покрытия

Примечание Точные алгоритмы

Метод ветвей и границ (branch and bound) и поиск с возвратом экспоненциальны в худшем случае, но на практике часто работают быстро благодаря отсечениям. TSP для тысяч городов решается точно branch-and-cut методами (Concorde) за разумное время на реальных дорожных графах. **SAT-решатели** (CDCL, conflict-driven clause learning) справляются с задачами на миллионы переменных и

десятки миллионов дизъюнктов, несмотря на NP-полноту SAT. Это работает на структурированных индустриальных примерах. Устройство CDCL — распространение единичного дизъюнкта, эвристика VSIDS, обучение на конфликтах — разобрано в главе 14.

Пример 2-приближение для вершинного покрытия

Оптимизационная версия задачи: дан граф G , требуется найти вершинное покрытие наименьшего размера. Алгоритм: пока остаются непокрытые рёбра, выбрать любое из них (u, v) , добавить в покрытие *оба* конца u и v и удалить все рёбра, инцидентные u или v .

Каждая итерация удаляет хотя бы одно ребро, поэтому итераций не больше $|E|$, а время работы составляет $O(|E|)$. Результат — корректное покрытие: ребро удалялось, только когда один из его концов уже попал в покрытие.

Гарантирован коэффициент 2: выбранные рёбра попарно не имеют общих концов и образуют паросочетание M . Любое покрытие обязано содержать хотя бы один конец каждого ребра паросочетания, поэтому размер минимального покрытия OPT не меньше $|M|$. Алгоритм берёт оба конца каждого выбранного ребра, и его покрытие насчитывает $2|M|$ вершин — не более $2 \cdot \text{OPT}$.

Коэффициент 2 достигается: на графе из k рёбер без общих концов алгоритм добавит в покрытие все $2k$ вершин, хотя минимум равен k .

Для некоторых задач (общий TSP) не существует приближения ни с каким константным коэффициентом, если только $P \neq NP$.

Причина — в сведении от гамильтонова цикла: рёбрам графа назначают расстояние 1, остальным парам городов — константу, заметно большую $c \cdot n$. Если бы алгоритм гарантировал маршрут не длиннее c оптимальных, он по длине ответа *отличал бы* граф с гамильтоновым циклом от графа без него — и решал бы NP-полную задачу за полиномиальное время.

Замечание Полиномиальные острова

Ещё одна стратегия — искать разрешимые подклассы задачи. NP-полнота вершинного покрытия — свойство задачи на *произвольных* графах. На двудольных графах задача решается за полиномиальное время. Размер минимального вершинного покрытия равен размеру максимального паросочетания (теорема Кёнига, глава 9). Паросочетание ищется сведением к потоку, а само покрытие строится по минимальному разрезу той же сетевой модели.

У NP-полных задач часто есть такие «полиномиальные острова»: сужения, на которых структура ограничивает сложность. При этом даже на островах остаётся граница: найти паросочетание — полиномиально, а *подсчитать* все паросочетания — #P-полная задача (класс #P вводится ниже).

Когда ни один из структурированных подходов не срабатывает, остаются эвристики.

Точные алгоритмы, приближения и эвристики — арсенал практика. Параметризованная сложность добавляет в этот арсенал ещё один инструмент: экспоненциальный по малому параметру, полиномиальный по размеру входа.

Замечание Сложность и криптография

Современная криптография с открытым ключом опирается на *предположительную* трудность конкретных задач. Если $P = NP$, то любой полиномиально проверяемый сертификат можно найти за полиномиальное время — и односторонних функций не существует. Таким образом, $P \neq NP$ — необходимое условие существования криптографии. Но не достаточное: NP-полнота гарантирует трудность лишь в худшем случае. Криптографии же нужна *трудность в среднем случае*: случайно сгенерированный экземпляр с высокой вероятностью должен быть труден.

Факторизация целых чисел и задача дискретного логарифмирования лежат в $NP \cap coNP$ ($L \in coNP$, если дополнение $\bar{L} \in NP$) и предположительно не являются NP-полными. Иначе полиномиальная иерархия (см. раздел 30.8) схлопнулась бы до первого уровня. Квантовый алгоритм Шора (1994) решает обе задачи за полиномиальное время на квантовом компьютере, что мотивирует переход к постквантовой криптографии на задачах теории решёток. Но сам принцип остаётся неизменным: безопасность стоит на вычислительной трудности математической задачи.

§ 30.5 Параметризованная сложность

Экспоненциальный алгоритм — не всегда приговор. $O(2^n)$ для $n = 100$ — астрономическое число. Но что, если экспонента зависит от малого параметра k , а от размера входа n — лишь полиномиально?

ОПРЕДЕЛЕНИЕ 30.6 FPT — Fixed-Parameter Tractable

Задача с параметром k (выделенным аспектом входа) называется **параметрически разрешимой** (FPT), если она разрешима за время

$$f(k) \cdot n^{O(1)},$$

где f — произвольная вычислимая функция, а полином от n не зависит от k .

Суть: экспонента «загнана» в параметр k , а по размеру входа n алгоритм полиномиален. Если k мало — алгоритм быстр независимо от n .

Пример Что такое «малый параметр»

Поиск клики размера k перебором всех подмножеств k вершин занимает $O(n^k)$ — это не FPT, ведь k в показателе степени.

Поиск вершинного покрытия размера k решается за $O(2^k \cdot n)$ — экспонента есть, но она зависит только от k , а не от размера графа. Для графа из миллиона вершин и $k = 10$ это $2^{10} \cdot 10^6 \approx 10^9$ операций — секунды, а не годы.

Разница принципиальна. n^k при $k = 10$ и $n = 10^6$ — это 10^{60} , недостижимо ни для какого компьютера.

30.5.1 Вершинное покрытие: от экспоненты к FPT

Задача о вершинном покрытии: даны граф G и число k , существует ли множество из $\leq k$ вершин, покрывающее все рёбра? Задача NP-полна. Полный перебор всех подмножеств размера k даёт $O(n^k)$ — это не FPT, так как k в показателе степени n .

FPT-алгоритм использует технику *ограниченного дерева поиска* (bounded search tree):

- Выбрать непокрытое ребро (u, v) .
- Либо u входит в покрытие, либо v (или оба).
- Рекурсивно исследовать обе ветви. Глубина рекурсии не превышает k .

Время работы: $O(2^k \cdot n)$.

Посмотрим, как дерево поиска работает на звезде с центром s и листьями: ищем покрытие размера 1. Ветвь «взять центр» сразу покрывает все рёбра. Ветвь «взять лист» оставляет непокрытыми остальные рёбра и *исчерпывает* бюджет. Ответ находится за два рекурсивных вызова, хотя полный перебор подмножеств перебирал бы каждый лист. Улучшенные алгоритмы снижают базу экспоненты. Лучший известный на 2025 год результат — $O^*(1.25284^k)$ (Harris and Narayanaswamy, 2022)²²¹.

Для $k = 10$ и $n = 1000$: $1.25284^{10} \approx 10$, и даже с полиномиальным множителем по n , скрытым в нотации O^* , это практически мгновенно. FPT превращает NP-полную задачу в практически разрешимую для экземпляров с малым значением параметра.

30.5.2 Древесная ширина и теорема Курселя

Многие NP-трудные задачи на графах становятся полиномиальными (и даже линейными) на графах с ограниченной константой *древесной ширины* (treewidth).

Древесная ширина — это мера того, насколько граф похож на дерево. Дерево имеет древесную ширину 1. Графы, моделирующие дорожные сети, электрические схемы и исходный код программ (control-flow graphs), часто имеют небольшую древесную ширину — именно потому, что они структурно близки к деревьям.

²²¹D. G. Harris, N. S. Narayanaswamy. «A faster algorithm for Vertex Cover parameterized by solution size». arXiv:2205.08022, 2022, STACS 2024.

Пример Древесная ширина простых графов

Дерево имеет древесную ширину 1, и обратно: граф древесной ширины 1 — это лес. Цикл имеет древесную ширину 2: он устроен как дерево с одной лишней связью. Полный граф из n вершин имеет древесную ширину $n - 1$: в нём каждая вершина связана со всеми остальными. Дорожная сеть близка к дереву потому, что она *росла как дерево*: магистрали ветвятся, а пересечений немного.

ТЕОРЕМА 30.6 Теорема Курселя

Любое свойство графа, выразимое в монадической логике второго порядка (MSO_2), разрешимо за линейное время на графах с ограниченной константой древесной ширины.²²² В MSO_2 выразимы k -раскрашиваемость, наличие гамильтонова цикла, вершинное покрытие, связность — словом, большинство NP-трудных графовых задач.

Доказательство выходит за рамки главы: оно переводит формулу в эквивалентный древесный автомат, обходящий декомпозицию графа. Константа в $O(n)$, однако, зависит от размера формулы и может быть астрономической. Теорема Курселя утверждает принципиальную разрешимость, но практические алгоритмы используют динамическое программирование по древесной декомпозиции с существенно лучшими константами.

Параметризованная сложность смещает фокус с размера входа на структурный параметр: вся экспоненциальная сложность сосредоточена в k , а по n время полиномиально.

Проверка Параметризованная сложность

1. Почему в FPT-алгоритме для вершинного покрытия достаточно ветвиться по концам непокрытого ребра, и почему глубина перебора ограничена k ?
2. Почему графы дорожных сетей и контроль-потоки программ обычно имеют малую древесную ширину, и какое следствие это имеет для NP-трудных задач?

§ 30.6 Ядро задачи

Перед запуском ветвления или решателя экземпляр стоит почистить. Для вершинного покрытия чистка напрашивается сама собой: изолированная вершина не покрывает ни одного ребра, а вершина со ста соседями при бюджете $k = 10$ попадает в покрытие независимо от дальнейшего поиска. Промышленные решатели начинают именно с такой чистки: SAT-решатель до поиска распространяет единич-

²²²B. Courcelle. «The Monadic Second-Order Logic of Graphs I: Recognizable Sets of Finite Graphs». Information and Computation, 1990.

ные дизъюнкты, а решатель линейного программирования запускает *presolve*-фазу. Прежде чем платить $O(2^k \cdot n)$, стоит уменьшить сам множитель n .

Разрозненные приёмы при этом складываются в каркас с гарантией: для многих задач несколько простых правил сжимают *любой* экземпляр до размера, зависящего только от параметра, — будь во входе миллион вершин или десять, после преобразования размер будет один и тот же.

ОПРЕДЕЛЕНИЕ 30.7 Ядро

Алгоритм называется **ядром** (kernel) параметризованной задачи, если он за полиномиальное от размера входа время переводит каждый экземпляр (x, k) в эквивалентный экземпляр (x', k') размера

$$|x'| + k' \leq g(k),$$

где g — вычислимая функция одного только параметра.

Эквивалентность означает совпадение ответов: (x, k) даёт ответ «да» тогда и только тогда, когда (x', k') даёт ответ «да». Параметр k' может отличаться от исходного и обычно оказывается меньше.

Если g — полином от k , ядро называется полиномиальным. Тогда предобработка даёт гарантию, которую не портит рост входа: как бы ни был огромен экземпляр, после чистки остаётся не больше $g(k)$ данных, и удалённое не могло повлиять на ответ.

Ядро проще всего увидеть на знакомом материале — вершинном покрытии: в главе оно появлялось уже трижды, в сведении из 3-SAT, в 2-приближении и в ветвлении за $O(2^k \cdot n)$.

Пример Ядро для вершинного покрытия

Даны граф G и бюджет k , существует ли покрытие из $\leq k$ вершин? Работают два правила.

- **Правило степени 0:** изолированная вершина удаляется. Она не покрывает ни одного ребра, и её присутствие не влияет ни на ответ, ни на бюджет.
- **Правило высокой степени:** если $\deg(v) > k$, вершина v попадает в покрытие, бюджет уменьшается на единицу, а v с инцидентными рёбрами удаляется. Правило безопасно: если v вне покрытия, каждый его сосед обязан лежать в покрытии, иначе их общее ребро останется непокрытым. Соседей при этом больше k , и бюджет уже исчерпан — вершина со степенью выше k принадлежит *каждому* покрытию размера $\leq k$.

Правила применяют до исчерпания: удаление вершины меняет степени её соседей, и сработавшее правило открывает дорогу следующему. Когда ни одно

правило больше не применимо, максимальная степень оставшегося графа не превышает текущего бюджета k' , а изолированных вершин нет.

УТВЕРЖДЕНИЕ 30.7 Размер ядра

Исчерпывающее применение двух правил либо выносит ответ «нет» — когда рёбер осталось больше k'^2 — либо оставляет экземпляр с $\leq k'^2$ рёбрами и $\leq 2k'^2$ вершинами, где k' — оставшийся бюджет. Поскольку $k' \leq k$, в терминах исходного параметра ядро насчитывает не более k^2 рёбер и $2k^2$ вершин — это полиномиальное ядро.

Доказательство

Докажем подсчётом рёбер.

Рёбра. Пусть после предобработки граф G' с бюджетом k' имеет вершинное покрытие C размера $\leq k'$. Каждое ребро инцидентно хотя бы одной вершине покрытия, а каждая вершина G' инцидентна не более чем k' рёбрам: максимальная степень не превышает бюджета. Поэтому $|E'| \leq |C| \cdot k' \leq k'^2$. Если же рёбер больше k'^2 , покрытия размера $\leq k'$ не существует, и ответ «нет» известен уже после чистки. В этом случае ядро выдаёт тривиальный «нет»-пример: $k' + 1$ попарно не пересекающихся рёбер при бюджете k' — каждое из них требует собственной вершины, и бюджета не хватает.

Вершины. Изолированных вершин в G' нет, поэтому каждая вершина инцидентна хотя бы одному ребру, а ребро инцидентно не более чем двум вершинам. Отсюда $|V'| \leq 2|E'| \leq 2k'^2$.

Бюджет в ходе правил только уменьшался, поэтому $k' \leq k$, и в терминах исходного параметра ядро насчитывает не более $2k^2$ вершин и k^2 рёбер.

Полиномиальное ядро даёт FPT-алгоритм без всякой изобретательности. Предобработка стоит полином от размера входа. Дальше ядро решается прямым перебором: данных в нём не больше $g(k)$, поэтому перебор рассматривает не более $2^{g(k)}$ кандидатов, а проверка каждого кандидата полиномиальна. Итог укладывается в $f(k) \cdot n^{O(1)}$ — в определение FPT. Для вершинного покрытия слепой перебор даже не нужен: ветвление из предыдущего раздела на ядре из $2k^2$ вершин работает за $O(2^k \cdot k^2)$, и полиномиальный множитель n ужался до k^2 . Предобработка при этом полезна и сама по себе: даже когда экспонента по k выходит за пределы достижимого, очищенный экземпляр проще для любого следующего инструмента — перебора, решателя, приближения.

Замечание Обратное направление

Верно и обратное: всякая FPT-задача имеет ядро, пусть с функцией размера, повторяющей время работы FPT-алгоритма, — вычислимой, но практически бес-

полезной. Обе части вместе дают точную характеристику: параметризованная задача лежит в FPT тогда и только тогда, когда у неё есть ядро. Полиномиальность ядра — отдельное и гораздо более сильное требование. У задачи о длинном пути (существует ли путь длины $\geq k$) FPT-алгоритмы есть, а полиномиального ядра, по современным представлениям, нет: его существование схлопнуло бы полиномиальную иерархию (раздел 30.8) — сценарий, считающийся столь же неправдоподобным, как $P = NP$.

§ 30.7 Классы сложности за пределами NP

Классы P и NP — лишь начало таксономии сложности. За ними простирается богатый спектр классов, упорядоченных по вычислительным ресурсам. Рис. 116 показывает основные классы и известные соотношения между ними: от P до EXP, с открытыми вопросами на каждой промежуточной границе.

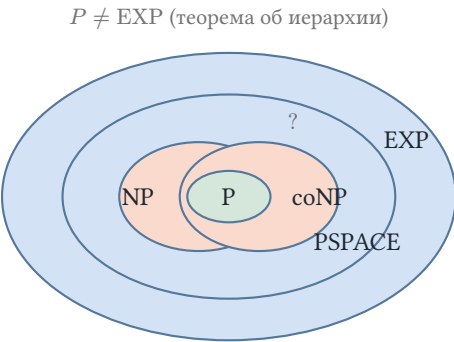


Рис. 116. Основные классы сложности.

УТВЕРЖДЕНИЕ 30.8 Спектр классов сложности			
Класс	Ограничение ресурса	Каноническая задача	Статус
P	Полиномиальное время	Кратчайший путь, сортировка, MST	Поддаётся эффективному решению
NP	Недетерминированное полиномиальное время	SAT, TSP, клика	Открыт: $P = NP$?
coNP	Дополнение NP	UNSAT, тавтология	Открыт: $NP = coNP$?
PSPACE	Полиномиальная память	QBF, обобщённые игры	$NP \subseteq PSPACE$
EXP	Экспоненциальное время	Обобщённые шахматы, го	Открыт: $PSPACE \subset EXP$?
#P	Подсчёт количества решений	#SAT, подсчёт паросочетаний	Сложнее NP

Известные соотношения между основными классами образуют цепочку включений.

УТВЕРЖДЕНИЕ 30.9 Иерархия сложности

$$P \subseteq NP \subseteq PSPACE \subseteq EXP.$$

Благодаря теореме об иерархии по времени $P \subset EXP$, хотя бы одно из включений в цепочке строгое.

ТЕОРЕМА 30.10 Теорема об иерархии по времени²²³

Для любой конструируемой по времени функции $t(n) \geq n \log n$, вычислимой за время $O(t(n))$,

$$TIME(o(t(n))) \subsetneq TIME(t(n) \log t(n)).$$

Существует язык, разрешимый за время $t(n) \log t(n)$, но не разрешимый за время $o(t(n))$: машина с бюджетом $t(n) \log t(n)$ решает задачи, недоступные машинам, работающим асимптотически быстрее.

Набросок доказательства

Докажем диагонализацией в ресурсной форме: машина с большим временным бюджетом имитирует машины с меньшим и отличается от каждой на её собственном входе. Пронумеруем все машины Тьюринга: $M_1, M_2, \dots$. Машина D на входе w длины n симулирует M_n на w в течение $t(n)$ шагов и выносит противоположный ответ: если M_n приняла, D отвергает, иначе принимает. Симуляция чужой машины стоит логарифмическую накидку, поэтому D укладывается в время $O(t(n) \log t(n))$. Если машина M_i работает за время $o(t(n))$, то уже при больших i на любом входе длины i она останавливается в пределах $t(i)$ шагов — и D выносит противоположный ответ. Значит, язык машины D не разрешается ни одной машиной с временем $o(t(n))$, а сам разрешается за $t(n) \log t(n)$.

Следствие: $P \subsetneq EXP$. Возьмём $t(n) = 2^n$: теорема даёт язык, разрешимый за время $2^n \cdot n$, но не за время $o(2^n)$. Всякий полином n^k есть $o(2^n)$, поэтому этот язык не лежит ни в одном классе $TIME(n^k)$, а объединение $TIME(n^k)$ по всем k — это в точности P .

Для памяти верен аналог: для конструируемой по памяти функции $s(n) \geq \log n$ выполнено $SPACE(o(s(n))) \subsetneq SPACE(s(n))$. Теорема об иерархии по памяти даёт строгие ступени по всей лестнице памяти, вплоть до $PSPACE \subsetneq EXPSPACE$, где $EXPSPACE$ — языки, требующие экспоненциальной памяти.

²²³J. Hartmanis, R. E. Stearns. «On the Computational Complexity of Algorithms». Transactions of the American Mathematical Society, 1965.

Какое именно включение строгое в цепочке $P \subseteq NP \subseteq PSPACE \subseteq EXP$ — остаётся открытым вопросом уже более полувека.

Спектр классов не исчерпывается этой цепочкой.

ОПРЕДЕЛЕНИЕ 30.8 Класс coNP

Классом coNP называется класс языков, дополнения которых лежат в NP:

$$\text{coNP} = \{L : \bar{L} \in \text{NP}\}.$$

Равносильная формулировка: $L \in \text{coNP}$, если существует полиномиально вычислимый предикат R и полином p , такие что $w \in L$ тогда и только тогда, когда каждая строка c длины не более $p(|w|)$ удовлетворяет $R(w, c)$.

Например, UNSAT (проверка невыполнимости булевой формулы) лежит в coNP: сертификат невыполнимости устроен сложнее, чем сертификат выполнимости. Сертификат выполнимости — выполняющее присваивание, которое легко проверить. Для невыполнимости же *отсутствие* присваивания не предъявляется конечным объектом, который можно проверить за полиномиальное время. Открыт вопрос: $NP = \text{coNP}$? Если нет, то NP не замкнут относительно дополнения — асимметрия между «найти пример» и «доказать отсутствие примеров» не случайна.

Класс #P (counting P) — задачи *подсчёта* количества решений, а не просто проверки существования. #SAT (#P-полная) сложнее SAT (NP-полная): зная количество выполняющих присваиваний, можно узнать и существование, но не наоборот. Таксономия сложности простирается и на подсчёт — это отдельное измерение, не сводимое к решению.

Рассмотрим подробнее два ресурса за пределами временной сложности: память (PSPACE) и случайность (BPP).

30.7.1 Сложность по памяти: PSPACE

ОПРЕДЕЛЕНИЕ 30.9 PSPACE

PSPACE: языки, разрешимые детерминированной МТ с полиномиальной памятью. Время не ограничено: экспоненциальное время допустимо, если память остаётся полиномиальной.

ОПРЕДЕЛЕНИЕ 30.10 NPSPACE

NPSPACE: языки, разрешимые недетерминированной МТ с полиномиальной памятью.

Известно: $NP \subseteq PSPACE$ (перебор всех сертификатов требует времени, но пространство можно переиспользовать).

Пример Память против времени

Задача достижимости в ориентированном графе: существует ли путь из s в t ? Поиск в глубину тратит $O(V + E)$ времени и $O(V)$ памяти — полиномиально всё. Но есть задачи, где время и память расходятся. Перебор 2^n присваиваний булевой формулы требует экспоненциального времени, а памяти — лишь $O(n)$, потому что присваивания можно генерировать по одному, не храня их все.

Именно такие задачи — экспоненциальное время при полиномиальной памяти — и составляют PSPACE. Классический представитель — QBF: истинность формулы с чередующимися кванторами $\forall x_1 \exists x_2 \forall x_3 \dots Q x_n \varphi(x_1, \dots, x_n)$.

Вопрос, добавляет ли недетерминизм силы, для времени остаётся открытым — это проблема P vs NP. Для полиномиальной памяти тот же вопрос решён, и ответ даёт теорема Сэвича.

ТЕОРЕМА 30.11 Теорема Сэвича²²⁴

$\text{NPSPACE} = \text{PSPACE}$. Недетерминизм не добавляет мощности для полиномиальной памяти, в отличие от времени (где вопрос P vs NP открыт).

Набросок доказательства

Докажем построением: сведём проверку достижимости принимающей конфигурации к рекурсивному делению пополам по числу шагов.

Недетерминированное вычисление с памятью $s(n)$ моделируется детерминированной МТ с памятью $O(s(n)^2)$. Проверим, достижима ли принимающая конфигурация из начальной за $2^{O(s(n))}$ шагов. Это делается рекурсивным делением пополам (divide and conquer). Конфигурация B достижима из A за k шагов, если существует промежуточная конфигурация C , достижимая из A за $\frac{k}{2}$ шагов, из которой B достижима за $\frac{k}{2}$ шагов. Перебор C использует ту же память рекурсивно, а глубина рекурсии — $\log 2^{O(s(n))} = O(s(n))$, отсюда квадрат памяти.



Теорема Сэвича: $\text{NPSPACE} = \text{PSPACE}$. Для времени аналогичный вопрос — P vs NP — открыт. Доказательство Сэвича переиспользует память: проверили ветвь недетерминированного вычисления — откатились, перезаписали ячейки, проверили другую. Время не переиспользуешь: каждый шаг вычисления необратимо потрачен.

Каноническая PSPACE-полная задача — QBF (Quantified Boolean Formula): дана булева формула с кванторами $\forall x_1 \exists x_2 \forall x_3 \dots Q x_n \varphi(x_1, \dots, x_n)$, истинна ли она? В отличие

²²⁴W. J. Savitch. «Relationships between nondeterministic and deterministic tape complexities». Journal of Computer and System Sciences, 1970.

от SAT, где все переменные экзистенциально квантифицированы ($\exists x_1 \dots \exists x_n$), в QBF кванторы чередуются — и это чередование поднимает задачу из NP в PSPACE.

Чередование кванторов превращает вопрос в *игру двух игроков*: $\exists$ выбирает значения своих переменных, $\forall$ — своих, и истинность формулы означает победу первого. Перебор всех стратегий требует экспоненциального времени, но лишь полиномиальной памяти.

PSPACE-полнота распространяется на многие игровые задачи — например, Sokoban (толкание ящиков). В таких играх пространство полиномиально (доска), а время — экспоненциально (число возможных последовательностей ходов). Обобщённые шахматы и го ещё труднее — они EXPTIME-полны, и таблица выше относит их к EXP.

Проверка Сложность по памяти

1. Почему память и время ведут себя по-разному: $\text{NPSPACE} = \text{PSPACE}$, но вопрос $P = NP$ открыт?
2. Почему чередование кванторов в QBF поднимает задачу из NP в PSPACE?

30.7.2 Вероятностные вычисления: BPP

Ещё одно расширение классической парадигмы — вероятностные алгоритмы. Машина Тьюринга может подбрасывать монетку и выбирать переход в зависимости от результата. Такая машина может ошибаться, но вероятность ошибки должна быть ограничена (обычно $\frac{1}{3}$, с возможностью уменьшить повторением).

ОПРЕДЕЛЕНИЕ 30.11 BPP

BPP (Bounded-error Probabilistic Polynomial time): языки, для которых существует полиномиальная вероятностная МТ, ошибающаяся с вероятностью $\leq \frac{1}{3}$ на каждом входе.

Ошибка возможна в обе стороны:

1. ложное принятие: $w \notin L$, а МТ приняла;
2. ложное отвергание: $w \in L$, а МТ отвергла.

RP — односторонний вариант: ложное принятие исключено, ложное отвергание остаётся.

ОПРЕДЕЛЕНИЕ 30.12 RP

RP (Randomized Polynomial time): языки, для которых вероятностная МТ никогда не принимает ошибочно ($w \notin L$ всегда отвергает), но может отвергнуть ошибочно с вероятностью $\leq \frac{1}{2}$. То есть $w \in L$ принимается с вероятностью $\geq \frac{1}{2}$.

ОПРЕДЕЛЕНИЕ 30.13 **coRP**

coRP — дополнение RP: $L \in \text{coRP}$, если $\bar{L} \in \text{RP}$. Машина никогда не отвергает ошибочно ($w \in L$ всегда принимает), но может принять ошибочно с вероятностью $\leq \frac{1}{2}$.

ZPP ужесточает требование до нуля ошибок, оставляя лишь неопределённость: машина либо отвечает верно, либо сообщает «не знаю».

ОПРЕДЕЛЕНИЕ 30.14 **ZPP**

ZPP (Zero-error Probabilistic Polynomial time): языки, для которых МТ всегда даёт правильный ответ, но *ожидаемое* время работы полиномиально.

Известно:

$$P \subseteq \text{ZPP} \subseteq \text{RP} \subseteq \text{BPP} \subseteq \text{PSPACE}.$$

Цепочка — лестница ослабленных гарантий: ZPP не ошибается никогда, RP и coRP ошибаются только в одну сторону, BPP — в обе. Включение $\text{BPP} \subseteq \text{PSPACE}$ держится на переборе: все бросания монетки можно перечислить, храня по одному текущему исходу. Детерминированный перебор ветвей укладывается в полиномиальную память.

Вероятность ошибки можно уменьшить повторением. Запустим BPP-алгоритм k раз на одном входе и примем ответ большинством голосов. Независимость запусков делает вероятность ошибки большинства экспоненциально малой: не более $e^{-2k(\frac{1}{2}-\frac{1}{3})^2} = e^{-\frac{k}{18}}$ (неравенство Хёффдинга, ср. примечание ниже). Так ценой полиномиального множителя во времени ошибка становится сколь угодно малой.

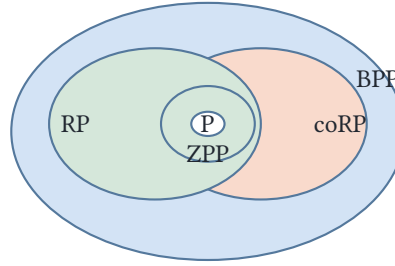
Пример Вероятностные алгоритмы

В проверке простоты Миллера–Рабина для простого p тест всегда отвечает «простое». Для составного n с вероятностью не меньше $\frac{3}{4}$ — «составное». Ошибка односторонняя (ложное «простое» возможно, ложное «составное» — нет): для языка «число составное» это RP, а для «число простое» — coRP.

Повторение теста k раз снижает вероятность ошибки до 4^{-k} : вероятностные классы становятся практичными именно благодаря амплификации.

Класс ZPP отличается тем, что ошибки нет вовсе: алгоритм либо выдаёт ответ, либо честно сообщает «не знаю», но работает полиномиальное время в ожидании.

Соотношение между ZPP, RP и coRP точнее, чем простая цепочка включений. ZPP — это в точности пересечение RP и coRP: безошибочный вероятностный алгоритм есть то же самое, что алгоритм с односторонней ошибкой в каждую сторону.



$$P \subseteq ZPP = RP \cap \text{coRP} \subseteq \text{BPP}$$

Рис. 117. ZPP, RP и coRP.

УТВЕРЖДЕНИЕ 30.12 **ZPP и $RP \cap \text{coRP}$** **$ZPP = RP \cap \text{coRP}$.****Набросок доказательства**

Докажем равенство двумя включениями: сначала $ZPP \subseteq RP \cap \text{coRP}$, затем обратное включение.

$ZPP \subseteq RP \cap \text{coRP}$: докажем первое включение. Включение $ZPP \subseteq \text{coRP}$ симметрично.

ZPP-машина *всегда корректна*, когда возвращает 0 или 1, а символ ? она возвращает лишь тогда, когда не успела закончить в отведённое время. Ожидаемое время работы полиномиально, так что найдётся полиномиальная граница $p(n)$ с $E[T] \leq p(n)$. Запустим машину, обрывая её после $2p(n)$ шагов. По неравенству Маркова $P(T \geq c \cdot E[T]) \leq \frac{1}{c}$, и при $c = 2$ она успевает закончить с вероятностью не менее $\frac{1}{2}$. Константа 2 — минимальное c с условием $1 - \frac{1}{c} \geq \frac{1}{2}$, то есть ровно тот порог, что даёт допустимую для RP вероятность успеха. Если машина не успела, отвечаем 0.

Правильный ответ	Машина вернула	Закключение
$w \in L$	1	Приняли верно
$w \in L$	?	Ответили 0 — ложное отвер- гание, для RP допустимо
$w \notin L$	0	Отвергли верно
$w \notin L$	?	Ответили 0 — ошибки нет, ведь 0 корректен

Мост между свойствами «всегда корректен» и «возвращает ?» — обрыв Las-Vegas машины на рубеже $2p(n)$ шагов. Обрыв сохраняет корректность ответа, а неопределённость ? превращается в ответ 0, допустимый для одностороннего класса RP.

$RP \cap \text{coRP} \subseteq ZPP$: докажем обратное включение. Пусть p_1 — RP-алгоритм (нет ложных принятий). Пусть p_2 — coRP-алгоритм (нет ложных отверганий).

Построим ZPP-алгоритм: если p_2 вернул 0 — ответ 0 (ошибки быть не может: coRP не отвергает ложно). Если p_1 вернул 1 — ответ 1 (ошибки быть не может: RP не принимает ложно). Иначе — ?. Вероятность получить ? не превышает $\frac{1}{2}$. Матожидание числа попыток до получения определённого ответа равно 2. Следовательно, ожидаемое время работы полиномиально.

Примечание Почему в определении BPP именно $\frac{1}{3}$?

Константа $\frac{1}{3}$ в определении BPP произвольна. Вероятность ошибки можно заменить на любую константу из $(0, \frac{1}{2})$. Иначе говоря, вероятность успеха — на любую из $(\frac{1}{2}, 1)$. Вероятность ошибки можно сделать экспоненциально малой — $\frac{1}{2^{q(|x|)}}$ для любого полинома q — ценой полиномиального числа повторений.

Запустим вероятностный алгоритм n раз и примем решение большинством голосов. Неравенство Чернова (концентрация биномиального распределения) гарантирует, что вероятность того, что большинство запусков ошибётся, экспоненциально убывает с ростом n :

$$P(\text{ошибка большинства}) \leq e^{-2n(p-\frac{1}{2})^2},$$

где p — вероятность правильного ответа в одном запуске. При $p = \frac{2}{3}$ и $n = 18q(|x|) \ln 2$ получаем вероятность ошибки не более $\frac{1}{2^{q(|x|)}}$.

Это свойство — **усиление вероятности** (amplification) — существенно для вероятностных алгоритмов: цена ошибки экспоненциально мала, а цена проверки — всего лишь полиномиальный множитель.

Главный открытый вопрос вероятностных вычислений: можно ли полностью избавиться от случайности? Формально — верно ли, что $\text{BPP} = \text{P}$? Большинство теоретиков полагают, что да — любой вероятностный алгоритм можно избавиться от случайности ценой полиномиального замедления. Но доказательство требует либо сильных теоретико-сложностных предположений (существование трудных булевых функций), либо прорыва в технике псевдослучайных генераторов.

Вероятностные алгоритмы — быстрая сортировка со случайным выбором опорного элемента, тест Миллера–Рабина для простоты, min-cut алгоритм Каргера. Они работают быстрее и проще своих детерминированных аналогов²²⁵. Неизвестно, можно ли полностью избавиться их от случайности, не потеряв в эффективности.

Проверка Вероятностные вычисления

1. Чем RP отличается от coRP, и почему $\text{ZPP} = \text{RP} \cap \text{coRP}$?
2. Почему повторение запусков делает вероятность ошибки в BPP экспоненциально малой?

²²⁵S. Arora, B. Barak. «Computational Complexity: A Modern Approach». 2009. Главы 7 (randomized computation), 20 (derandomization)

§ 30.8 * Полиномиальная иерархия

Класс NP допускает логическую характеристику:

$$L \in \text{NP} \leftrightarrow \exists x : R(w, x),$$

где R — полиномиально разрешимый предикат, а x пробегает сертификаты полиномиальной длины. coNP получается заменой квантора на универсальный:

$$L \in \text{coNP} \leftrightarrow \forall x : R(w, x).$$

Естественное обобщение — добавить ещё один квантор:

$$L \in \Sigma_2 \leftrightarrow \exists x_1 \forall x_2 : R(w, x_1, x_2),$$

$$L \in \Pi_2 \leftrightarrow \forall x_1 \exists x_2 : R(w, x_1, x_2).$$

Продолжая, получаем полиномиальную иерархию.

ОПРЕДЕЛЕНИЕ 30.15 Полиномиальная иерархия

Для каждого $k \geq 0$:

- Σ_k : языки, определяемые формулой $\exists x_1 \forall x_2 \exists x_3 \dots Q_k x_k : R(w, x_1, \dots, x_k)$, где $R \in P$, а кванторы чередуются k раз (первый — $\exists$).
- $\Pi_k = \text{co-}\Sigma_k$: то же с первым квантором $\forall$.

$$\text{PH} = \bigcup_{k \geq 0} \Sigma_k = \bigcup_{k \geq 0} \Pi_k.$$

Базовые случаи: $\Sigma_0 = \Pi_0 = P$, $\Sigma_1 = \text{NP}$, $\Pi_1 = \text{coNP}$.

Пример Задача из Σ_2

Существует ли булева формула размера $\leq t$, эквивалентная данной формуле φ ?

Ответ «да» можно проверить: предъявляем формулу ψ размера $\leq t$. Убеждаемся, что $\psi \equiv \varphi$. Но эквивалентность — это *универсальный* квантор: $\psi(a) = \varphi(a)$ для всех присваиваний a . Итак, задача описывается двумя чередующимися кванторами: $\exists \psi, \forall a$. Это и есть формат Σ_2 .

Общий принцип: каждый дополнительный квантор поднимает задачу на следующий уровень иерархии.

Структурное свойство: если $\Sigma_k = \Pi_k$ для некоторого k , то PH схлопывается до уровня k . В частности, $P = \text{NP}$ влечёт схлопывание PH до P . Равенство $\text{NP} = \text{coNP}$ влечёт схлопывание до первого уровня.

Открытый вопрос: бесконечна ли PH — то есть верно ли, что $\Sigma_k \subset \Sigma_{k+1}$ для всех k ? Большинство теоретиков полагают, что да, но доказательство отсутствует — как и для $P \neq \text{NP}$, частным случаем которого при $k = 0$ этот вопрос и есть.

Если NP выражает трудность поиска решения (один экзистенциальный квантор), то P выражает трудность игры двух игроков с полной информацией: каждый дополнительный квантор соответствует ходу. Проблема P против NP — нулевой уровень вопроса о том, насколько сложнее становится задача с каждым новым кванторным чередованием.

§ 30.9 Итоги главы

Главным открытым остаётся вопрос P против NP: эквивалентно ли нахождение решения его проверке. Классы P и NP развели быстрое решение и быструю проверку, полиномиальные сведения дали способ сравнивать задачи, а теорема Кука–Левина предъявила первую NP-полную задачу — SAT. Полиномиальное время выбрано как критерий эффективности не случайно: класс P замкнут относительно композиции и устойчив к смене модели вычислений.

С тех пор NP-полнота распространяется цепочками сведений. За SAT последовали клика, вершинное покрытие, гамильтонов цикл, разбиение, рюкзак — и сотни задач из логики, графов и оптимизации образовали один класс эквивалентности. 3-SAT NP-полна, а 2-SAT — нет, и именно длина дизъюнкта решает это различие.

Дальше выяснилось, что трудность измерима и структурирована. Параметризованная сложность загоняет экспоненту в малый параметр k , оставляя полиномиальную зависимость от размера входа и превращая NP-трудные задачи в FPT. По памяти и вероятности выстраиваются классы PSPACE, BPP, RP и ZPP, связанные теоремой Сэвича и равенством $ZPP = RP \cap \text{coRP}$. Полиномиальная иерархия выражает игру двух игроков, а не один экзистенциальный квантор. Теорема Ладнера показывает, что даже при $P \neq NP$ между классами есть промежуточные ступени.

Инженерный вывод — в том, что NP-полнота описывает худший случай, а практика живёт в структуре конкретных экземпляров. Точные алгоритмы с отсечениями, приближения с гарантиями, эвристики и FPT складываются в набор стратегий решения. Дальше тот же взгляд переносится с задач на программы: автоматическое доказательство свойств кода — тема абстрактной интерпретации (глава 31).

Проверка Проверьте себя

1. Чем NP-полная задача отличается от NP-трудной?
2. Почему 3-SAT NP-полна, а 2-SAT нет? Что это говорит о роли длины дизъюнкта?
3. Что было бы, если бы $P = NP$? Какие последствия для поиска решений и для криптографии?
4. Почему NP-полные задачи образуют единый класс эквивалентности? Что изменится, если одна NP-полная задача окажется в P?

5. Что такое FPT и чем параметризованная сложность меняет взгляд на NP-трудные задачи?
6. Зачем нужны полиномиальные сведения, если мы не умеем доказывать нижние оценки? Что они позволяют утверждать о сравнительной трудности задач?

Что дальше?

Теория сложности классифицирует задачи по вычислительным ресурсам, устанавливая границы между классами P, NP, PSPACE и далее. Всюду в этой классификации работает чёткая бинарная логика: задача либо в P, либо нет. Сертификат либо корректен, либо нет. Сведение либо сохраняет ответ, либо бесполезно. Сама структура определения классов сложности опирается на классическую теорию множеств, где элемент либо принадлежит множеству, либо нет.

Но классификация задач — не конечная цель. Задачи нужно решать быстро, а программы — ещё и проверять: доказать, что программа не упадёт, не разделит на ноль, не выйдет за границы массива. Точная проверка свойств программ в общем случае неразрешима (глава 26) — однако индустрия проверяет программы каждый день. Секрет — в аппроксимации: вместо точного доказательства машина вычисляет корректное приближение, и его достаточно для практики. В следующей главе (31) мы рассматриваем абстрактную интерпретацию — автоматическое доказательство свойств программ через аппроксимацию множеств состояний.

§ 30.10 Упражнения

Классы P и NP.

1. Объясните, почему $P \subseteq NP \subseteq PSPACE$. Какие включения являются строгими (доказано), а какие остаются открытыми проблемами?
2. Приведите три примера задач из P и три примера задач из NP, не упомянутых в тексте главы. Для каждой NP-задачи опишите сертификат и процедуру его проверки.
3. Покажите, что задача 2-SAT лежит в P. Почему из этого не следует, что 3-SAT лежит в P?
4. * Докажите, что если $L_1, L_2 \in NP$, то $L_1 \cup L_2 \in NP$ и $L_1 \cap L_2 \in NP$. Опишите верификаторы для объединения и пересечения.

Полиномиальные сведения и NP-полнота.

5. Докажите, что задача о независимом множестве NP-полна. Сведите к ней задачу о вершинном покрытии и используйте связь: I — независимое множество тогда и только тогда, когда $V \setminus I$ — вершинное покрытие.

6. Сведите задачу о клике к задаче о вершинном покрытии. Докажите корректность сведения.
7. Докажите, что класс coNP замкнут относительно полиномиального сведения: если $A \leq_p B$ и $B \in \text{coNP}$, то $A \in \text{coNP}$.
8. * Покажите, что задача проверки тавтологичности булевой формулы (TAUTOLOGY) coNP-полна. Указание: рассмотрите связь с SAT.

От NP-полноты к практике.

9. Докажите, что если $P = NP$, то существует полиномиальный алгоритм не только для проверки существования гамильтонова цикла, но и для его *построения*. Подсказка: используйте самосведение — последовательно удаляйте рёбра и проверяйте, остался ли гамильтонов цикл.
10. Докажите, что язык всех выполнимых хорновских формул лежит в P. Подсказка: алгоритм распространения меток (unit propagation) для хорновских формул работает за линейное время.
11. * Докажите, что если существует разреженный NP-полный язык (язык, в котором число строк длины n ограничено полиномом от n), то $P = NP$. Это результат Мэхэни (1982) — приведите схему доказательства.

За пределами NP.

12. * Докажите, что TQBF (задача истинности квантифицированных булевых формул) PSPACE-полна.
13. ** Объясните, почему вероятность ошибки в BPP можно уменьшить полиномиальным числом повторений — с $\frac{1}{3}$ до 2^{-n} . Сформулируйте неравенство Чернова и покажите, как оно даёт экспоненциальное убывание вероятности ошибки большинства голосов.
14. * Рассмотрим «доказательство» включения $NP \subseteq P$. Пусть $L \in NP$ — язык с полиномиальным верификатором V . То есть $w \in L \leftrightarrow \exists c : V(w, c) = 1$ для некоторого полиномиального по длине сертификата c . Чтобы решить задачу, достаточно найти сертификат или убедиться, что его нет, — то есть обратить процедуру проверки. Обращение полиномиального алгоритма полиномиально, поэтому $NP \subseteq P$. Где ошибка?
15. * Рассмотрим «доказательство» включения $NP \subseteq \text{coNP}$. Пусть $L \in NP$ — язык с полиномиальным верификатором V . То есть $w \in L \leftrightarrow \exists c : V(w, c) = 1$. Определим верификатор дополнения $V'(w, c) = \neg V(w, c)$. Для $w \notin L$ верификатор V отвергает любой сертификат. Значит, $V(w, c) = 0$ для всех c . Тогда V' принимает w с любым сертификатом, поэтому $\bar{L} \in NP$. Итак, $L \in \text{coNP}$. Где ошибка? Подсказка: вспомните определение coNP: нужен верификатор с $w \in \bar{L} \leftrightarrow \exists c : V'(w, c) = 1$. Верно ли обратное направление?

ПРОЕКТ Сведение NP-трудной задачи к SAT

Классы P и NP задаются через *быстрое решение* и *быструю проверку*. Теорема Кука–Левина делает SAT универсальной NP-полной задачей, а цепочки сведений переносят трудность между задачами: вершинное покрытие, клика, независимое множество. Соберите работающий конвейер: граф и число k сводятся явной конструкцией гаджетов к КНФ, формула решается решателем DPLL. Найденное присваивание проверяется верификатором, а разрыв между «найти» и «проверить» измеряется таблицей времён.

① Верификатор

Реализуйте КНФ-представление формулы: литералы и присваивание. Верификатор за один проход по дизъюнктам должен проверять, что присваивание выполняет формулу, — сертификат и его проверка за линейное время.

② Сведение

Сведите вершинное покрытие размера $\leq k$ к SAT: для каждого ребра добавьте дизъюнкт «хотя бы один конец в покрытии». Ограничение «не более k выбранных вершин» — кардинальный гаджет (последовательный счётчик). Дизъюнкты выписывайте явно.

③ Решатель

Реализуйте DPLL с unit-пропагацией и чистыми литералами. Ведите статистику решений, пропагаций и конфликтов. Решите формулу сведённого экземпляра и извлеките покрытие из присваивания.

④ Измерение разрыва

Измерьте разрыв между «найти» и «проверить»: таблица времён и размеров по n . Продумайте, как устроить эксперимент, чтобы отделить рост времени поиска от линейной проверки.

⑤ Независимое множество

Сведите IND к CLIQUE через дополнение графа: независимое множество размера k в G — это клика размера k в дополнении $\overline{G}$. Проверьте на примерах, что сведение сохраняет ответ.

Итог: работающий конвейер: граф и число k сводятся явной конструкцией гаджетов к КНФ, формула решается решателем DPLL, найденное покрытие проверяется верификатором. Таблица времён отделяет «проверить» от «найти».

XXXI

Абстрактная интерпретация

Программы ошибаются, и последствия бывают дорогими: деление на ноль останавливает программу, выход за границы массива портит данные, разыменование нулевого указателя роняет сервис. Хочется знать заранее, что программа не упадёт и не сломается, — до того, как она попадёт в продакшен.

Наивный способ проверить — запустить программу и посмотреть, что будет. Программа, которая делит на введённое число:

```
read x
y := 10 / x
print y
```

Пусть она прошла тесты на трёх входах: $x = 5$, $x = -2$, $x = 7$. Все три запуска успешны — и всё равно ничего не доказано. На четвёртом входе, $x = 0$, программа упадёт с ошибкой деления на ноль. Тестирование показывает наличие ошибок, но никогда не доказывает их отсутствия: ошибка может прятаться на любом следующем входе. Грузоподъёмность моста вычисляют по чертежу, из статике и механики материалов, а не гоняя по мосту всё более тяжёлые грузовики, пока он не рухнет. Программная инженерия долго была устроена по второму способу: каждый тест — грузовик, и предел программы узнают по моменту её падения.

Перебрать все входы невозможно. Если вход — 64-битное целое, возможных значений 2^{64} — больше, чем секунд в возрасте Вселенной. А для программы, читающей список чисел, входов бесконечно много. Значит, доказывать свойство нужно для *всех* запусков сразу, не выполняя программу ни на одном из них.

Предыдущая глава 30 измеряла сложность задач: как быстро вычисляется то, что вычислимо. Ответы были асимптотическими — классы сложности, NP-полнота.

Вопрос другой — можно ли доказать, что конкретная программа ведёт себя правильно на всех входах сразу. Точного автоматического ответа не существует: свойства произвольной программы неразрешимы (глава 26).

Остаётся приближённый ответ — и его оказывается достаточно. Разрешается третий ответ, «не знаю», — и задача перестаёт быть безнадёжной. Простейший корректный анализ отвечает «не знаю» на любой вопрос: он не ошибается, но и не сообщает ничего полезного. Вся дальнейшая работа — ужимать этот «не знаю» до точного ответа там, где это возможно.

Аппарат для приближённого ответа уже наполовину построен. В главе 8 знаковая решётка появилась как предвестник: супремум сливал информацию из ветвей условного оператора, неподвижная точка (*fixed point*) — результат анализа цикла. Тогда это был взгляд в будущее. Теперь он становится главным.

Идея проста — заменить точные значения их грубыми обобщениями. Чтобы доказать «знаменатель не равен нулю», не нужно знать, какое именно значение принимает x , — достаточно знать, что x не бывает нулём. Вместо числа остаётся его *знак*: положительное, отрицательное или ноль. К ним добавляются два особых значения — «неизвестно» и «недостижимо». Таких обобщений конечное число, и программа «выполняется» на них за конечное число шагов.

Абстрактное выполнение *корректно* аппроксимирует настоящее: всё, что может произойти в реальной программе, учтено в абстрактной. Если абстрактный анализ показал, что деление на ноль невозможно, — это доказано для всех запусков. Платой за корректность служит потеря точности: абстрактное выполнение может сообщить о возможной ошибке там, где на деле всё в порядке. Знак не различает 5 и -3 , поэтому про сумму $5 + (-3)$ домен знаков скажет «неизвестно». Это размен, на котором построена вся область: за конечный автоматический анализ платят неточностью, но не теряют ни одной реальной ошибки.

Наивная интуиция подсказывает: чтобы узнать, упадёт ли программа, её надо запустить. Запуск показывает один путь из многих. Рассуждение на абстракции покрывает все пути сразу — и не требует ни одного запуска. Точность при этом теряется, корректность — никогда. Тот же принцип — рассуждать о неточном — в главе 35 приведёт к нечётким множествам. Как ужать ответ «не знаю» до точного, не потеряв ни одной реальной ошибки?

ИСТОРИЯ Рождение идеи: от Флойда до Кузо

Первый шаг к автоматическому доказательству свойств программ — метод индуктивных утверждений Роберта Флойда (1967) и логика Хоара (1969). Идея: разметить программу утверждениями, доказать, что каждое утверждение сохраняется переходом, и вывести, что итоговое свойство выполняется для всех запусков. Метод красив, но требует, чтобы инварианты циклов придумывал человек.

В 1977 году Патрик Кузо и Радия Кузо (Patrick Cousot и Radhia Cousot) опубликовали статью «Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints». Работа показала, что статические анализаторы можно строить единообразно: как вычисление неподвижных точек операторов на решётках. Инвариант цикла больше не нужно изобретать — он вычисляется автоматически. Если точная неподвижная точка недостижима, Кузо предложили приближать её операторами расширения (*widening*) и сужения (*narrowing*). Так абстрактная интерпретация стала общим каркасом, объединившим разрозненные анализы потоков данных: распростра-

нение констант, анализ интервалов, анализ указателей — всё это оказалось частными случаями одной схемы. Почти полвека каркас Кузо остаётся основой индустриального анализа программ. Из знаков и интервалов домены выросли в полиэдры (Cousot и Halbwachs, 1978) и октагоны (Miné, 2001) — каждый под свою задачу. На этой теории работают оптимизирующие компиляторы, статические анализаторы и верификация систем, где ошибка стоит дороже любой неточности.

ОБЗОР ГЛАВЫ

В этой главе мы займёмся знаковой абстракцией: заменим числа знаками и посмотрим, как на них «выполняется» программа. Связка Галуа свяжет абстрактные значения с множествами конкретных состояний, а переносящие функции — с условием корректности. Познакомимся с доменами, которые решают реальные задачи: интервалами, константами, вычетами и их произведениями. Разберём циклы: теорема Кнастера–Тарского объяснит, почему неподвижная точка существует, а widening заставит расходящуюся итерацию сойтись. Вернёмся к цене аппроксимации: ложные срабатывания неизбежны, пропуски ошибок — нет. Закончим там, где абстрактная интерпретация работает в производстве: компиляторы, статические анализаторы, верификация полётного софта.

После этой главы вы сможете:

1. объяснять, почему тестирование не доказывает корректность;
2. строить абстрактные домены — знаки, интервалы, константы — и выполнять программу на них;
3. применять связки Галуа и переносящие функции;
4. находить неподвижные точки и пользоваться widening и narrowing;
5. объяснять, чем абстрактная интерпретация отличается от верификации Хоара.

§ 31.1 Знаки вместо чисел

Пусть значения переменных — целые числа. Заменим каждое число его *знаком*: положительное, отрицательное или ноль. Вместо «переменная равна 5» скажем «переменная положительна».

ОПРЕДЕЛЕНИЕ 31.1 Домен знаков

Домен знаков состоит из пяти элементов:

- $\top$ — «знак неизвестен» (значение может быть любым),
- $+$ — «значение положительно»,
- 0 — «значение равно нулю»,
- $-$ — «значение отрицательно»,

- $\perp$ — «значение невозможно» (переменная здесь не определена).

Пять элементов выстраиваются в решётку: чем ниже стоит элемент, тем точнее он описывает состояние программы. Вид этой решётки — на рис. 118.

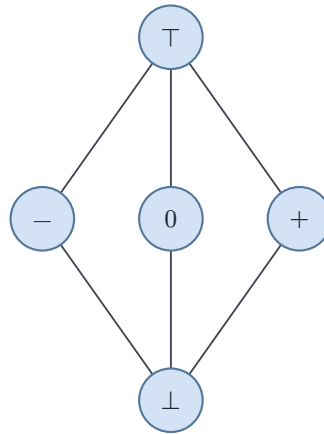


Рис. 118. Решётка домена знаков: $\perp \preceq \{-, 0, +\} \preceq T$.

ОПРЕДЕЛЕНИЕ 31.2 Информационный порядок

Говорим, что $a \preceq b$ (« a не грубее b »), если любой конкретный смысл a содержится в конкретном смысле b .

Например, $+ \preceq T$, поскольку «положительное» точнее, чем «любое», и $\perp \preceq +$, поскольку «недостижимо» точнее, чем «положительное».

Примечание Почему порядок «наоборот»

В обычной математике $\leq$ означает «меньше или равно». Здесь $\preceq$ означает «не грубее»: элемент ниже по порядку несёт больше информации. Это *информационный* порядок, стандартная условность теории решёток: чем ниже спускаемся, тем точнее знаем состояние программы.

Каждое *абстрактное значение* описывает множество конкретных значений. Значение $+$ описывает все положительные числа. Значение 0 — только ноль. Значение T — все целые числа.

Пусть $\mathcal{C}$ — множество конкретных состояний программы (например, все пары значений переменных). А D — абстрактный домен. Между ними действуют две функции: конкретизация и абстракция.

ОПРЕДЕЛЕНИЕ 31.3 Конкретизация

Конкретизация — это отображение $\gamma : D \rightarrow \mathcal{P}(\mathcal{C})$, которое абстрактному значению сопоставляет множество конкретных состояний, которые оно представляет.

ОПРЕДЕЛЕНИЕ 31.4 Абстракция

Абстракция — это отображение $\alpha : \mathcal{P}(\mathcal{C}) \rightarrow D$, которое множеству конкретных состояний сопоставляет наиболее точное абстрактное значение, его покрывающее:

$$\alpha(X) = \sqcap \{d \in D \mid X \subseteq \gamma(d)\}.$$

Для домена знаков: $\alpha(\{5, 7\}) = +$, и $\gamma(+) = \{x \in \mathbb{Z} \mid x > 0\}$.

Замечание Связка Галуа

Пара (α, γ) образует *связку Галуа*: для любых $X \subseteq \mathcal{C}$ и $d \in D$

$$\alpha(X) \preceq d \Leftrightarrow X \subseteq \gamma(d).$$

Обе стороны утверждают одно: абстрактное значение d описывает всё множество X . Абстракция даёт *наиболее точное* описание, конкретизация — *все* значения, которые d представляет. Это согласование делает анализ корректным по построению: абстрактный результат никогда не теряет реальные поведения.

ОПРЕДЕЛЕНИЕ 31.5 Абстрактный домен

Абстрактный домен — это решётка $(D, \preceq, \perp, \top, \sqcup, \sqcap)$, где:

- $\preceq$ — информационный порядок («не грубее»),
- $\perp$ — наименьший элемент («недостижимо»),
- $\top$ — наибольший элемент (все значения),
- $\sqcup$ — объединение (*join*): $a \sqcup b$ описывает всё, что описывают a и b ,
- $\sqcap$ — пересечение (*meet*): $a \sqcap b$ — наиболее точное значение, совместимое с обоими.

Объединение соответствует слиянию веток программы, пересечение — усилению требования.

Пример Объединение и пересечение на домене знаков

Слияние двух веток: в одной $x = +$, в другой $x = -$. После слияния $x = + \sqcup - = \top$: какая ветка выполнялась, зависит от условия, и знак неизвестен. Ветка, помеченная как недостижимая, не добавляет информации: $\perp \sqcup + = +$.

Пересечение уточняет знание: $+ \sqcap \top = +$ — из «любого знака» остаётся «положительное». Если требования несовместимы, пересечение пусто: $- \sqcap + = \perp$.

Примечание Абстракция состояний, не только значений

Для программы с n переменными абстрактное состояние — вектор из n абстрактных значений. Например, $(-, +, 0, \dots)$. Порядок поэлементный: $a \preceq b$,

если каждый $a_i \preceq b_i$. Так конечный текст программы описывается конечным множеством абстрактных состояний — и вычисление над ним заведомо останавливается.

§ 31.2 Вычисления в абстрактном мире

Абстрактные значения описывают множества состояний, но анализ должен идти по программе: каждому оператору нужен двойник, работающий с абстрактными значениями. Построим его от конкретной семантики оператора — и перенесём эту механику в абстрактный мир.

Конкретная семантика оператора превращает множество состояний в множество состояний. Например, оператор $x := x + 1$ переводит множество значений в сдвинутое множество. То есть $X + 1 = \{v + 1 \mid v \in X\}$. Абстрактная семантика — та же операция, но на абстрактных значениях.

ОПРЕДЕЛЕНИЕ 31.6 Переносящая функция

Переносящая функция — это отображение $\llbracket s \rrbracket^\sharp : D \rightarrow D$, которое вычисляет абстрактное состояние после оператора s по абстрактному состоянию d .

Переносящая функция обязана быть *корректна*: она не имеет права терять ни одно реальное поведение.

ОПРЕДЕЛЕНИЕ 31.7 Корректность переноса

Переносящая функция $\llbracket s \rrbracket^\sharp$ **корректна** (не пропускает реальных поведений), если для любого абстрактного состояния d

$$\llbracket s \rrbracket(\gamma(d)) \subseteq \gamma(\llbracket s \rrbracket^\sharp(d)),$$

где $\llbracket s \rrbracket$ — конкретная семантика оператора s на множествах состояний.

Всё, что реально может произойти после s в состоянии из d , покрывается абстрактным результатом. Перенос никогда не теряет настоящие поведения — только добавляет невозможные.

Дополнительно перенос предполагается *монотонным*: из $d_1 \preceq d_2$ следует $\llbracket s \rrbracket^\sharp(d_1) \preceq \llbracket s \rrbracket^\sharp(d_2)$. Без монотонности итерации по циклу могут не сходиться.

Пример Присваивание в домене знаков

Пусть $x := x + 1$. Абстрактный результат:

- $+$ $\rightarrow$ $+$: положительное плюс единица — положительное,
- $-$ $\rightarrow$ $\top$: отрицательное плюс единица может стать нулём,

- $0 \rightarrow +$,
- $\top \rightarrow \top, \perp \rightarrow \perp$.

Правило $- \rightarrow \top$ — честная цена абстракции: знак не различает -1 и -5 , поэтому не видит, что $-1 + 1 = 0$. Перенос корректен: реальный результат покрывается абстрактным $\top$.

Примечание

Домен знаков — это тип с четырьмя значениями и таблицей операций:

```
// Домен знаков: минус, ноль, плюс и "не знаю" (top).
enum Sign { Neg, Zero, Pos, Top }

// Сложение знаков: плюс ⊕ минус = Top, x ⊕ 0 = x.
fn add(a: Sign, b: Sign) -> Sign {
  match (a, b) {
    (Sign::Zero, x) | (x, Sign::Zero) => x,
    (Sign::Neg, Sign::Neg) => Sign::Neg,
    (Sign::Pos, Sign::Pos) => Sign::Pos,
    _ => Sign::Top, // разные знаки могут дать всё что угодно
  }
}
```

Значения $+$, $-$, 0 , $\top$ — это варианты типа `Sign`, а таблица сложения — функция `add`. Константы и интервалы — соседние домены, построенные по той же схеме.

Пример Сложение в домене знаков

Операция $z := x + y$ требует абстрактной таблицы сложения знаков. Правила:

- $+\boxplus+=+$, $-\boxplus-=-$,
- $x\boxplus 0 = x$ для любого знака x ,
- $+\boxplus-=\top$: положительное плюс отрицательное может быть и положительным, и нулём, и отрицательным,
- $\top\boxplus x = \top$ при $x \neq \perp$.
- $\perp\boxplus x = \perp$. В частности, $\top\boxplus\perp=\perp$.

Правило $+\boxplus-=\top$ — источник неточности: знак не хранит величин. Он не видит, что $7 + (-4) = 3 > 0$. Корректность сохраняется: абстрактный результат $\top$ покрывает реальный.

Пример Выражение $(x \cdot y) + z$ в домене знаков

Пусть $x = -, y = +, z = -$. Сначала умножение: $-\boxtimes+=-$. Затем сложение: $-\boxplus-= -$. Абстрактный результат —: выражение отрицательно. Конкретный пример: $(-3) \cdot 5 + (-2) = -17$, и знак совпал. Из таблицы умножения знаков понадобилась одна строка: минус на плюс — минус. Точность не потеряна ни на одном шаге: знаки согласованы, и $\boxplus$ не выродился в $\top$.

Примечание Как выводить переносящие функции

Переносящая функция выводится из конкретной семантики. Для оператора $z := x + y$ конкретный результат — множество $X + Y = \{u + v \mid u \in \gamma(x), v \in \gamma(y)\}$. Абстрактный результат — самая точная абстракция этого множества: $\alpha(X + Y)$. Поэтому переносящая функция $\llbracket z := x + y \rrbracket^\#(d) = \alpha(\{u + v \mid u \in \gamma(d_x), v \in \gamma(d_y)\})$ корректна по построению. Именно так строятся таблицы абстрактных операций: перебрали все пары знаков, посчитали конкретное множество, абстрагировали.

Проверка Вычисления в абстрактном мире

1. Почему правило $- \rightarrow +$ для $x := x + 1$ нарушило бы корректность, а $- \rightarrow \top$ — нет?
2. Как устроен универсальный рецепт построения корректной переносящей функции из конкретной семантики?
3. Зачем переносу монотонность?

§ 31.3 Домены на практике

Знаковая решётка — простейший, но не единственный домен. Разные вопросы о программе требуют разных доменов.

31.3.1 Интервальный домен

Программа суммирует элементы массива:

```
s := 0
for i in 0..n do
  s := s + a[i]
```

Индекс i никогда не должен выйти за границы массива. Чтобы доказать это, не нужно знать, какое именно значение принимает i на каждом шаге: достаточно знать, что i лежит в отрезке $[0, n - 1]$. Отсюда и *интервальный домен*: значение переменной — отрезок $[l, h]$, а не отдельное число.

ОПРЕДЕЛЕНИЕ 31.8 Интервальный домен

Абстрактное значение — интервал $[l, h]$ (либо особый элемент $\perp$), где $l, h \in \mathbb{Z} \cup \{-\infty, +\infty\}$. Конкретизация — все целые числа из интервала $[l, h]$. $\top = [-\infty, +\infty]$. $\perp$ — отдельный элемент «пусто».

- Порядок по включению: $[a, b] \preceq [c, d]$ тогда и только тогда, когда $[a, b] \subseteq [c, d]$.
- Объединение $\sqcup$ — выпуклая оболочка (*convex hull*): $[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$.

- Пересечение $\sqcap$ — наиболее точный интервал $[\max(a, c), \min(b, d)]$ (пусто, если нижняя граница больше верхней).

Интервальный домен теряет связи между переменными. Если в каждой точке программы $y = x - 1$, точное множество пар лежит на прямой. Например, точки $(2, 1), (3, 2), (5, 4)$ — три такие пары. Интервальный анализ видит лишь прямоугольник $x \in [2, 5], y \in [1, 4]$ и не замечает, что y определён значением x . Связи между переменными возвращают более богатые домены — октагоны и полиэдры, упомянутые в начале главы.

Интервалы хороши для доказательства границ: индексов массивов, счётчиков, буферов. Сложение интервалов поэлементное: $[a, b] + [c, d] = [a + c, b + d]$. Умножение (для конечных границ) — $[a, b] \cdot [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$. Если граница бесконечна, мы для простоты полагаем результат $\top$ (стандартный интервальный домен разбирает бесконечные случаи точнее).

Пример Выход за границы массива

Пусть в программе $i \in [0, 9]$, а массив имеет 10 элементов. Переносящая функция обращения к элементу $A[i]$ проверяет, что индекс в границах: $i \in [0, 9]$. Если интервал i выходит за границы — например, $[0, 10]$ — анализатор сообщает о возможном выходе за границы. Сообщение может оказаться ложным (переполнения нет), но реальный выход за границы он не пропустит: корректность гарантирует отсутствие ложноотрицательных.

31.3.2 Домен констант

ОПРЕДЕЛЕНИЕ 31.9 Домен констант

Абстрактное значение — либо конкретное целое c , либо особое значение $\top$ («не константа»). Ещё одно особое значение — $\perp$ («недостижимо»). Порядок: $\perp \preceq c \preceq \top$ для всех целых c .

Домен констант доказывает равенства: после присваиваний $x := 3, y := x + 4$ выполняется равенство $y = 7$. Компиляторы используют его для *распространения констант*: подставить в код вычисленное значение и упростить выражение.

Одно правило стоит особняком: умножение на ноль убивает неопределённость, $0 \cdot \top = 0$. Ноль, умноженный на любое конкретное значение, даёт ноль, поэтому результат известен точно даже при $\top$ -операнде. Например, для $t := 0, v := \top, z := t \cdot v$ анализ даёт $z = 0$, хотя v неизвестно.

Пример Распространение констант

Программа:

```

x := 3
y := x + 4
z := y * 2

```

Анализ констант: $x = 3, y = 3 + 4 = 7, z = 14$. Компилятор выполнит замену $z := y * 2 \rightarrow z := 14$ и, возможно, удалит лишние присваивания. Это и есть применение абстрактной интерпретации в оптимизирующих компиляторах.

31.3.3 Домен вычетов

ОПРЕДЕЛЕНИЕ 31.10 Домен вычетов (*congruence domain*)

Абстрактное значение — класс вычетов $c \bmod m$: все числа, сравнимые с c по модулю m . Сложение: $(c_1 \bmod m_1) \boxplus (c_2 \bmod m_2) = (c_1 + c_2) \bmod \gcd(m_1, m_2)$. Умножение: $(c_1 \bmod m_1) \boxtimes (c_2 \bmod m_2) = (c_1 c_2) \bmod \gcd(m_1, m_2)$.

Класс вычетов $0 \bmod 2$ — чётные числа, класс $1 \bmod 2$ — нечётные. Домен вычетов отслеживает остатки по модулю и позволяет выводиться свойства делимости: например, $x^2 + x$ чётно. Он сочетается с интервальным доменом в произведениях.

Пример Домен вычетов выводит делимость

Пусть анализ знает, что $x \equiv 3 \bmod 4$. Посчитаем $x^2 + x$ правилами домена. Умножение: $(3 \bmod 4) \boxtimes (3 \bmod 4) = 9 \bmod \gcd(4, 4) = 9 \bmod 4 = 1 \bmod 4$. Сложение: $(1 \bmod 4) \boxplus (3 \bmod 4) = 4 \bmod 4 = 0 \bmod 4$. Вывод: $x^2 + x$ делится на 4, а значит, и на 2. Проверка на конкретном числе: $x = 7, 7^2 + 7 = 56 = 14 \cdot 4$.

31.3.4 Произведение доменов

ОПРЕДЕЛЕНИЕ 31.11 Произведение доменов

Произведение доменов — домен, абстрактное состояние которого — пара (d_1, d_2) , где d_i — значение в соответствующем домене.

Порядок поэлементный, объединение покомпонентное: $(a_1, a_2) \sqcup (b_1, b_2) = (a_1 \sqcup b_1, a_2 \sqcup b_2)$.

Чтобы получать сразу несколько свойств, домены объединяют в произведение. Произведение знаков и констант даёт и знак, и точное значение (когда оно есть). Произведение интервалов и вычетов — и границы, и чётность. Домены не обязаны быть независимыми: *редуцированное произведение* переиспользует информацию одного домена для уточнения другого.

Пример Произведение знаков и констант

Программа:

```

x := 5
y := x - 10
z := x + y

```

В произведении знаков и констант состояние — пара (d_1, d_2) : знак и константа. После первого присваивания: $x = (+, 5)$. После второго: $y = (-, -5)$. После третьего присваивания знак равен $+ \boxplus - = \top$. Значение: $5 + (-5) = 0$. Произведение даёт $z = (\top, 0)$: знак неизвестен, значение — ноль в точности. Домен знаков в одиночку сказал бы $\top$, домен констант — 0. Произведение хранит оба ответа сразу, и каждый из них пригодится своему анализу.

§ 31.4 Неподвижная точка и расходимость

Простая последовательность операторов переносится естественно: идём по программе, применяя переносящие функции. С условными операторами объединяем ветки через $\sqcup$. Сложность — циклы: тело цикла может выполняться сколько угодно раз.

ОПРЕДЕЛЕНИЕ 31.12 Перенос условий

Условие переносится как **фильтр**: абстрактное состояние пересекается с интервалом, заданным условием. Ветка «да» ограничивается самим условием, ветка «нет» — его дополнением.

Например, для условия $i < n$ интервал i в ветке «да» становится $[l, h] \rightarrow [l, \min(h, n - 1)]$ (при известной константе n). Если $l > n - 1 \perp$: ветка «да» недостижима. Ветка «нет» получает пересечение с дополнением: $[l, h] \sqcap [n, +\infty]$. Если n неизвестна (её интервал — $\top$), фильтр ничего не отсекает. Верхняя граница не меняется: $\min(h, +\infty) = h$.

Рассмотрим цикл с телом B . Пусть E — состояние в голове цикла при входе. Определим $F(d) = E \sqcup \llbracket B \rrbracket^\sharp(d)$: объединение состояния на входе и результата одного прохода тела из d . Состояние после любого числа проходов — наименьшая неподвижная точка F :

$$\text{lfp}(F) = \sqcup_{k \geq 0} F^k(\perp).$$

Смысл: последовательность $F^k(\perp)$ монотонно накапливает состояния, достижимые в голове цикла после всё большего числа проходов тела. Объединив все $F^k(\perp)$, получаем состояние после любого числа проходов. В главе 8 мы доказали (теорема

Кнастера–Тарского), что такая неподвижная точка существует и что монотонная функция на конечной решётке достигает её за конечное число шагов.

ТЕОРЕМА 31.1 Итерация Клини сходится на конечных доменах

Если F монотонна и домен конечен, последовательность итераций $d_0 = \perp$, $d_{k+1} = F(d_k)$ достигает $\text{lfp}(F)$ за конечное число шагов.

Доказательство

Докажем, что последовательность итераций монотонна и потому в конечной решётке стабилизируется на наименьшей неподвижной точке.

Монотонность даёт $d_0 \preceq d_1 \preceq d_2 \preceq \dots$: последовательность не убывает по информационному порядку. В конечной решётке любая неубывающая последовательность стабилизируется. С какого-то момента $d_{k+1} = d_k$, и тогда $F(d_k) = d_k$ — неподвижная точка.

Она наименьшая: любая неподвижная точка F содержит $\perp = d_0$, а потому содержит и все последующие d_k .

ТЕОРЕМА 31.2 Корректность абстрактной итерации (*soundness*)

Пусть переносающая функция тела цикла корректна в смысле определения выше. Тогда $\gamma(d_k)$ содержит все конкретные состояния, достижимые в голове цикла менее чем за k проходов тела. В частности, всякое состояние, достижимое в голове цикла, лежит в $\gamma(\text{lfp}(F))$.

Доказательство

Докажем индукцией по k .

Конкретный аналог итерации задаётся рекуррентой $X_0 = \emptyset$ и $X_{k+1} = \gamma(E) \cup \llbracket B \rrbracket(X_k)$, где $\llbracket B \rrbracket$ — конкретная семантика тела на множествах состояний. Множество X_k — это в точности состояния в голове цикла, достижимые менее чем за k проходов тела: вначале пусто, каждый шаг добавляет входные состояния и результат ещё одного прохода тела.

База. $X_0 = \emptyset = \gamma(\perp)$: конкретизация $\perp$ пуста по определению.

Шаг. Пусть $X_k \subseteq \gamma(d_k)$. Конкретная семантика монотонна по включению, поэтому $\llbracket B \rrbracket(X_k) \subseteq \llbracket B \rrbracket(\gamma(d_k))$. Корректность переноса даёт $\llbracket B \rrbracket(\gamma(d_k)) \subseteq \gamma(\llbracket B \rrbracket^\#(d_k))$. Объединение $E \sqcup \llbracket B \rrbracket^\#(d_k)$ описывает всё, что описывают его слагаемые, поэтому

$$\begin{aligned}
X_{k+1} &= \gamma(E) \cup \llbracket B \rrbracket(X_k) \\
&\subseteq \gamma(E) \cup \gamma(\llbracket B \rrbracket^\#(d_k)) \\
&\subseteq \gamma(E \sqcup \llbracket B \rrbracket^\#(d_k)) = \gamma(d_{k+1}).
\end{aligned}$$

Осталось перейти к пределу. Каждое достижимое состояние лежит в некотором X_k , то есть в $\gamma(d_k)$, а из $d_k \preceq \text{lfp}(F)$ и монотонности γ следует $\gamma(d_k) \subseteq \gamma(\text{lfp}(F))$.

Домен знаков конечен — циклы в нём анализируются перебором. Интервальный домен бесконечен: интервалы вида $[0, n]$ с ростом n никогда не стабилизируются. Именно здесь наивная итерация ломается.

Пример Расходящаяся итерация

Рассмотрим цикл

```

i := 0
while i < n do
  i := i + 1

```

Интервальный анализ состояния i в голове цикла даёт последовательность $[0, 0], [0, 1], [0, 2], [0, 3], \dots$. Каждая итерация расширяет интервал на единицу. Процесс никогда не остановится: наивной итерацией неподвижную точку не получить.

Условие $i < n$ здесь не помогает: значение n неизвестно (его интервал — $\top$), поэтому фильтр условия ничего не отсекает (см. определение переноса условий выше).

Пример Сходящаяся итерация: цикл с известной границей

Тот же цикл с известной границей:

```

i := 0
while i < 10 do
  i := i + 1

```

Условие $i < 10$ фильтрует состояние в голове цикла: перед телом интервал пересекается с $[0, 9]$. Итерация $d_{k+1} = F(d_k)$ от $\perp$:

k	d_k
0	$\perp$
1	$[0, 0]$
2	$[0, 1]$
3	$[0, 2]$
4	$[0, 3]$

k	d_k
...	...
10	$[0, 9]$
11	$[0, 9]$

На десятом шаге интервал перестал расти: $d_{10} = d_{11} = [0, 9]$ — неподвижная точка. Верхняя граница восстановлена фильтром условия: она равна 9. Итерация сошлась, потому что граница 10 известна заранее.

Ответ Кузо — оператор *расширения* (widening), который заставляет итерацию сойтись.

§ 31.5 Как заставить итерацию сойтись (widening)

ОПРЕДЕЛЕНИЕ 31.13 Оператор расширения

Бинарная операция $\nabla : D \times D \rightarrow D$, для которой:

- $a \sqcup b \preceq a \nabla b$ (результат не менее груб, чем объединение),
- для любой возрастающей последовательности $d_0 \preceq d_1 \preceq \dots$ последовательность $w_0 = d_0, w_{k+1} = w_k \nabla d_{k+1}$ стабилизируется за конечное число шагов.

Оператор ∇ «ускоряет» рост, сбрасывая нестабильные компоненты в $\top$.

Пример Widening для интервалов

Для интервалов классический widening:

$$[l_1, h_1] \nabla [l_2, h_2] = [l, h], l = \begin{cases} -\infty & \text{если } l_2 < l_1 \\ l_1 & \text{иначе} \end{cases}, h = \begin{cases} +\infty & \text{если } h_2 > h_1 \\ h_1 & \text{иначе} \end{cases}.$$

В примере со счётчиком: $[0, 0] \nabla [0, 1] = [0, +\infty]$: верхняя граница выросла — сбрасываем её в бесконечность. Следующая итерация: $[0, +\infty] \nabla [0, +\infty] = [0, +\infty]$ — стабилизация. Прыжок widening к бесконечности, обходящий лестницу из $[0, 1], [0, 2], [0, 3], \dots$, показан на рис. 119.

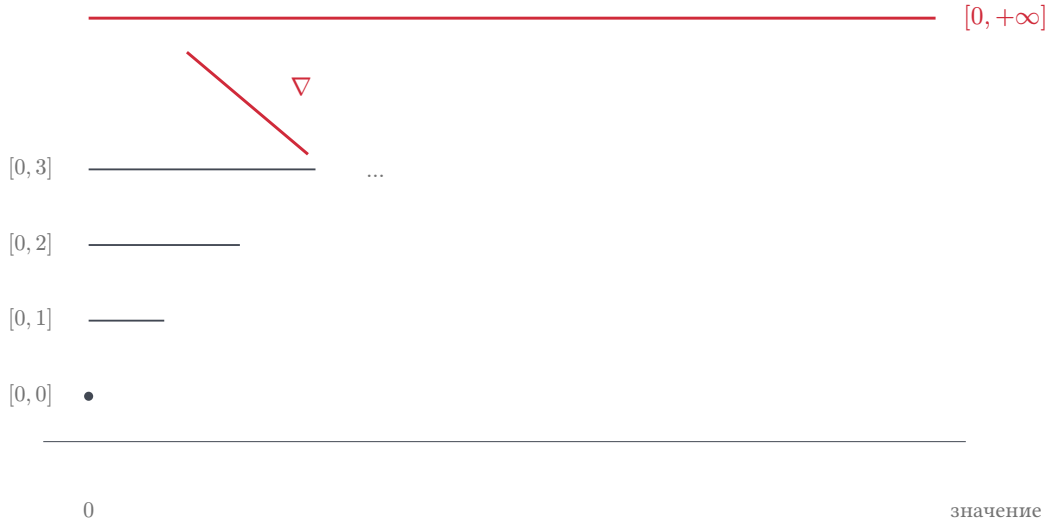


Рис. 119. Widening для цикла со счётчиком.

АЛГОРИТМ Вычисление неподвижной точки с widening

На каждой итерации в голове цикла применяем F с widening:

1. $w_0 = \perp$.
2. $w_{k+1} = w_k \nabla F(w_k)$.
3. Повторяем, пока $w_{k+1} \neq w_k$.

По свойству widening последовательность стабилизируется за конечное число шагов. Результат — корректная (но, возможно, грубая) неподвижная точка.

Результат widening корректен по той же индукции: остановившееся значение w удовлетворяет $F(w) \preceq w$, и монотонность F даёт $d_k \preceq w$ для всех k . Значит, $\gamma(w)$ покрывает все реальные состояния цикла, хотя и грубее наименьшей неподвижной точки.

Widening гарантирует сходимость, но платит точностью: интервал $[0, +\infty]$ в примере со счётчиком говорит лишь « i неотрицательно», теряя верхнюю границу. Чтобы вернуть точность, после widening применяют оператор сужения (narrowing).

ОПРЕДЕЛЕНИЕ 31.14 Оператор сужения

Бинарная операция $\triangle$, для которой $a \triangle b \preceq a$: результат не грубее первого операнда.

Повторяя сужение из уже достигнутой неподвижной точки, получаем более точную неподвижную точку. В примере со счётчиком сужение идёт по условию цикла. При известной константе n фильтр $i < n$ (см. перенос условий) уточняет интервал $[0, +\infty] \rightarrow [0, n - 1]$: верхняя граница восстановлена из условия.

Пример Widening и narrowing: цикл со счётчиком

Цикл $i := 0; \text{while } i < 100 \text{ do } i := i + 1$ анализируем в два приёма. Фаза widening: $w_0 = \perp$, затем $w_{k+1} = w_k \nabla F(w_k)$.

$$w_1 = [0, 0], w_2 = [0, +\infty], w_3 = [0, +\infty].$$

Уже на втором шаге верхняя граница сброшена в бесконечность, и итерация стабилизировалась. Фаза narrowing: из неподвижной точки $[0, +\infty]$ уточняем результат по условию цикла. Фильтр $i < 100$ даёт $[0, 99]$. Повторное сужение ничего не меняет: $[0, 99]$ — неподвижная точка. Две фазы вместе дали точный ответ там, где ни widening, ни наивная итерация по отдельности не справились.

Схема «widening для сходимости, narrowing для точности» — стандартный рецепт анализа циклов в интервальном домене.

Замечание Почему widening вообще работает

В бесконечной решётке интервалов нет никакой гарантии сходимости наивной итерации. Widening вводит *искусственную* сходимость: как только граница начала расти, он объявляет её бесконечной. Это грубо, но корректно: результат всё ещё покрывает все реальные поведения. Точность теряется контролируемо — ровно в той мере, в какой этого требует остановка.

Проверка Widening и narrowing

1. Почему наивная итерация сходится в домене знаков, но расходится в интервальном?
2. Что widening делает с растущей границей и что возвращает narrowing?
3. Почему результат widening остаётся корректной аппроксимацией, хотя он грубее наименьшей неподвижной точки?

§ 31.6 Цена аппроксимации

Абстрактный анализ корректен, но не точен. Слово «корректен» здесь означает: анализатор не пропускает реальные ошибки. Слово «не точен» означает: он может сообщить об ошибке, которой на деле нет.

ОПРЕДЕЛЕНИЕ 31.15 Ложные срабатывания

Если анализатор сообщает о возможной ошибке в точке, где ошибки не бывает, — это **ложное срабатывание** (false positive).

Корректный анализатор допускает ложные срабатывания, но не допускает пропусков: если ошибка реально возможна, анализатор обязательно её сообщит.

Анализатор не различает реальную ошибку и ложную тревогу. Абстракция рисует вокруг исполнений оболочку, и если оболочка не задевает «запретную» область, ошибка доказанно невозможна. Если оболочка область задевает, перед анализатором два неразличимых случая: либо внутри области действительно попадает исполнение, либо не попадает ни одного. Абстрактная картина в обоих случаях одинакова, и сообщить об ошибке нужно в обоих: замалчивание во втором было бы пропущенной реальной ошибкой.

ЛОВУШКА **Корректный не значит точный**

«Анализатор не нашёл деление на ноль» означает «деление на ноль доказано невозможным» — но только если анализатор корректен. Несложный (но некорректный) анализатор, который ничего не анализирует, тоже никогда не сообщит об ошибке. Ценность анализа — именно в гарантии отсутствия пропусков.

Плата — ложные срабатывания: инженер вручную разбирает сообщения и выясняет, какие из них реальны. Искусство анализа — уметь отличать реальные срабатывания от ложных, не пожертвовав корректностью. Масштаб платы оценивают цифрами: на полётном софте Airbus в сотни тысяч строк C коммерческие анализаторы выдают тысячи предупреждений, которые инженеры разбирают вручную. Astrée спроектировали так, чтобы довести ложные срабатывания до нуля, — за счёт доменов, повторяющих структуру полётного софта.

Пример **Точность и домен**

Рассмотрим программу $x := 6, y := -7, z := x + y$. Домен знаков даёт для неё $z = \top$: знак суммы не вычисляется. В действительности $z = -1$ — определённое число. Домен констант посчитал бы $z = -1$ точно. Домен интервалов — $[-1, -1]$.

Выбор домена — компромисс между точностью, стоимостью и тем, какое свойство нужно доказать. Корректность при этом не зависит от выбора: любой корректный домен не пропустит ошибку.

§ 31.7 Применения абстрактной интерпретации

Абстрактная интерпретация — не абстрактная теория: на ней держится индустрия статического анализа.

31.7.1 Компиляторы

Оптимизирующие компиляторы (GCC, LLVM) внутри — анализаторы потоков данных. Распространение констант, анализ диапазонов значений, анализ указателей, удаление мёртвого кода — всё это частные случаи абстрактной интерпретации. Компилятор доказывает свойства программы, чтобы безопасно выполнить оптимизацию:

- заменить выражение константой,
- убрать неисполняемую ветку,
- развернуть цикл.

31.7.2 Верификация полётного софта

Статический анализатор *Astrée*, развитый в ENS в 2000-х, доказал отсутствие ошибок времени выполнения (деление на ноль, переполнение, выход за границы) в полётном софте самолётов Airbus. Эти системы летают с результатами *Astrée*, потому что анализ корректен: доказательство отсутствия ошибок переживает сертификацию. Домен октагонов, разработанный Антуаном Мине для *Astrée*, — классический пример домена, спроектированного под конкретную задачу.

31.7.3 Промышленные анализаторы

Polyspace (MathWorks) использует абстрактную интерпретацию для верификации встраиваемого софта. Frama-C — открытая платформа анализа кода на C, ядро которой — интервальный анализ на абстрактной интерпретации. Абстрактная интерпретация — стандартный инструмент индустрии.

31.7.4 Генерация инвариантов для верификации

Доказательство в логике Хоара требует инвариантов циклов (тройки Хоара — в главе 32). Абстрактная интерпретация вычисляет инварианты автоматически: неподвижная точка переносающей функции цикла и есть инвариант. Вот как абстрактная интерпретация усиливает ручную верификацию: человек доказывает общую структуру, а инварианты циклов машина находит сама.

ИСТОРИЯ *Astrée*: доказательство для самолёта

В 2003 году команда под руководством Патрика Кузо и его соавторов начала разработку *Astrée* — анализатора, нацеленного на полётный софт Airbus. Позднее *Astrée* доказала отсутствие ошибок времени выполнения в программе управления полётом Airbus A340 и A380 — системе из десятков тысяч строк C. Корректность анализа стала юридическим аргументом: сертификационные органы приняли машинное доказательство отсутствия ошибок. История *Astrée* — пример того, как теория решёток и неподвижных точек из научной статьи превратилась в стандарт авиационной безопасности.

§ 31.8 Символьное исполнение

Абстрактная интерпретация жертвует точностью ради корректности: анализатор может сообщить о ложной ошибке. Символьное исполнение — противоположный полюс: оно перебирает пути точно, ценой стоимости.

Вместо конкретного значения переменная получает *символьное выражение* — формулу от входных параметров. Состояние вычисления — это стек, память и *условие пути*: накопленные ограничения на входы. При ветвлении состояние *разветвляется*: условие добавляется к условию пути одной ветки, его отрицание — другой. Решатель SAT/SMT (SAT с теориями, например с арифметикой) проверяет, выполнимо ли условие пути. Невыполнимое условие делает путь недостижимым, и состояние отбрасывается. Для выполнимого условия решатель возвращает модель — конкретные значения входных переменных, на которых условие истинно. Так по каждому достижимому пути собирается конкретный набор входов, который этот путь выполняет.

Пример Одна программа, два пути

Программа: $x > 5 \rightarrow y := x + 1$, иначе $y := 2x$.

После ветвления символьное исполнение ведёт два состояния. Первое: $x > 5, y = x + 1$. Второе: $x \leq 5, y = 2x$.

Из первого условия решатель выдаёт конкретный вход, например $x = 6$. Из второго — например $x = 0$. Так символьное исполнение перечисляет все пути и для каждого находит конкретный пример.

Противоречие в условии пути иногда видно сразу: путь с условиями $x > 5, x \leq 5$ недостижим при любых входах. Каждый путь в дереве исполнения — класс эквивалентности входов: все входы, которые ведут программу по одной и той же последовательности операторов.

Символьное исполнение лежит в основе генерации тестов, поиска уязвимостей и подтверждения отчётов статических анализаторов. Оно точно там, где абстрактная интерпретация аппроксимирует, но платит за это: число путей растёт экспоненциально, а завершение программы и достижимость конкретной ветки неразрешимы в общем случае (глава 26). Поэтому на практике два подхода работают вместе: консервативный анализ находит подозрительные места, символьное исполнение подтверждает или опровергает их по конкретным путям.

§ 31.9 Итоги главы

Вопрос о том, можно ли доказать свойство программы для всех запусков сразу, соединил две знакомые темы — неразрешимость (глава 26) и приближение. Вместо множества конкретных состояний анализ работает с маленькой решёткой — знаки, интервалы — решётки с информационным порядком приходят из главы 8, а связь абстрактного с конкретным задаёт пара α, γ (связка Галуа). Анализ потока данных IFDS из той же главы — дистрибутивный частный случай этой схемы. Переносимые функции выводятся из конкретной семантики по единому рецепту — посчитать

конкретное множество и абстрагировать его — поэтому корректность анализа гарантирована по построению, а не доказывается отдельно.

Дальше из этой схемы вырастает практика. Циклы анализируются неподвижными точками: итерация от $\perp$ сходится к наименьшей неподвижной точке, а widening и narrowing ускоряют сходимость ценой контролируемой грубости. Корректность при этом сохраняется: анализатор может сообщить о ложной ошибке, но пропустить реальную он права не имеет — и именно это свойство сделало Astrée юридическим аргументом при сертификации полётного софта Airbus. Символьное исполнение стоит на другом полюсе того же спектра: оно перебирает пути точно и платит за это стоимостью.

Аппроксимация с гарантиями работает там, где точное доказательство недостижимо, и это делает абстрактную интерпретацию повседневным инструментом компиляторов и верификаторов. Model checking из главы 33 проверяет конечные системы точно, а бесконечные пространства состояний остаются за абстрактной аппроксимацией. Конечные автоматы из главы 23 находят в анализе свою роль: автомат — удобный абстрактный домен для описания последовательностей событий в программе. Дальше курс снимает последнее ограничение этого языка: нечёткие множества (глава 35) допускают частичную принадлежность.

Проверка Проверьте себя

1. Чем элемент решётки отличается от множества конкретных состояний, и зачем нужны оба?
2. Почему корректность переноса — это не то же самое, что точность?
3. Как связаны абстракция α и конкретизация γ ?
4. Почему произведение доменов повышает точность, а не просто склеивает ответы?
5. В каком смысле widening заведомо неточен, и почему это приемлемо?
6. Почему ложные срабатывания неизбежны, а пропуски ошибок — нет?

Что дальше?

Абстрактная интерпретация всё время аппроксимирует: вместо точного значения — знак, вместо множества — интервал. Сам язык аппроксимации построен на чёткой бинарной принадлежности: значение либо в интервале, либо нет. Следующая и последняя глава снимает это ограничение: нечёткие множества (глава 35) допускают частичную принадлежность, и мы пересматриваем понятие множества, с которого начали курс, — теперь с позиции, вооружённой опытом логики, алгебры и вычислений.

§ 31.10 Упражнения

Абстракция и переносящие функции.

1. Что такое переносная функция? Почему она должна быть корректной?
2. Сформулируйте условие корректности переноса словами и формулой.
3. Постройте абстрактную таблицу умножения для домена знаков. Проверьте корректность каждой строки.
4. * Объясните, почему $+ \boxplus - = \top$ в домене знаков. Почему это корректно, но неточно? Приведите конкретный пример двух чисел, для которых домен знаков теряет точность при сложении.

Домены на практике.

5. Чем интервальный домен отличается от знакового? Какое свойство программы доказывает каждый из них?
6. Для программы

```
x := -2
y := x * x
if y > 0 then z := 1 else z := 0
```

проведите анализ в домене знаков. Что скажет домен констант?

7. Для цикла $i := 0$. `While  $i < 100$  do  $i := i + 1$` проведите интервальный анализ вручную. Считайте, что условие цикла фильтрует состояние (перенос условий: пересечение с $[0, 99]$). Сколько шагов нужны наивной итерации без widening? Что даёт widening?
8. * Домен знаков с операцией $+ \boxplus - = \top$ теряет точность. Предложите домен, который различает «строго положительное» и «неотрицательное», и постройте таблицу сложения.

Widening и неподвижные точки.

9. Пусть F — монотонная функция на конечной решётке. Докажите, что наивная итерация из $\perp$ достигает неподвижной точки.
10. Покажите, что интервальный widening корректен: $[l_1, h_1] \sqcup [l_2, h_2] \preceq [l_1, h_1] \nabla [l_2, h_2]$.
11. * Спроектируйте абстрактный домен для анализа чётности ($c \bmod 2$). Постройте переносные функции для присваивания, сложения, умножения и условного оператора. Проверьте корректность.
12. ** В главе 8 неподвижная точка цикла вычислялась на конечной знаковой решётке перебором. Объясните, почему тот же подход не работает для интервалов, и как widening это исправляет. Программа вычисляет факториал: что докажет интервальный анализ с widening и почему $\top$ для результата при больших входных — честный ответ, а не дефект анализа?

13. * Пусть конкретный домен — множество целых чисел $\mathbb{Z}$ (одна переменная), а конкретизация знаков задана так: $\gamma(\perp) = \emptyset$, $\gamma(-) = \{x \in \mathbb{Z} \mid x < 0\}$, $\gamma(0) = \{0\}$, $\gamma(+) = \{x \in \mathbb{Z} \mid x > 0\}$, $\gamma(\top) = \mathbb{Z}$. Вычислите $\alpha(\{-7, 3, 0\})$, $\alpha(\{2, 4\})$, $\alpha(\emptyset)$ и $\alpha(\{5, -5\})$. Проверьте условие связки Галуа $\alpha(X) \preceq d \Leftrightarrow X \subseteq \gamma(d)$ для $X = \{-1, 4\}$ при $d = \top$ и $d = +$. Докажите, что пара (α, γ) всегда образует связку Галуа.

ПРОЕКТ Каркас абстрактных доменов

Соберите работающий каркас абстрактной интерпретации: решётку с информационным порядком, монотонный трансформер и итерацию к наименьшей неподвижной точке. Подключите к нему два домена — интервалы и знаки — мини-язык программ с `assert` и конкретный интерпретатор, чтобы поверять анализ реальными запусками. Соберите его из связки Галуа, переноса функций, теоремы Кнастера–Тарского и операторов расширения и сужения. Новый домен подключается к одному и тому же анализу.

① Каркас решётки и итерация

Реализуйте абстрактный домен как структуру с порядком «не грубее», элементами $\perp$ и $\top$, объединением $\sqcup$ и пересечением $\sqcap$. Задайте абстрактное состояние как отображение из переменных в абстрактные значения и продумайте, что означает состояние $\perp$ и как отличить недостижимую ветку от живой. Реализуйте итерацию к наименьшей неподвижной точке монотонного трансформера с фазами `widening` и `narrowing` и проверьте на каждом домене аксиомы решётки и монотонность трансформера.

② Интервальный домен

Реализуйте отрезок $[l, h]$ с бесконечными границами: сложение и вычитание поэлементные, умножение по произведениям границ, деление по угловым частным. Продумайте, что возвращает деление на интервал, содержащий ноль: обрушение в $\perp$ делает трансформер немонотонным, и итерация к неподвижной точке начинает осциллировать. Верните $\top$ с отдельным предупреждением и объясните, почему корректность при этом сохраняется.

Реализуйте перенос условий и `widening` по Кузо: растущая граница сбрасывается в бесконечность, и итерация сходится там, где наивная расходится. Продумайте, когда `widening` теряет точность безвозвратно и как `narrowing` по условию цикла возвращает потерянную границу. Что даёт перенос условия цикла в голову цикла, и почему без него `widening` останавливается на грубой неподвижной точке?

③ Домен знаков

Реализуйте знак как битовую маску классов «отрицательное», «ноль», «положительное» и постройте таблицы арифметики по представителям классов: переберите все пары представителей, посчитайте конкретное множество, абстрагируйте. Продумайте, почему представителей классов доста-

точно для точной таблицы, и проверьте таблицы перебором конкретных пар, включая усечение целочисленного деления. Реализуйте перенос условий и проверку «definitely» для ассертов перебором пар знаков.

④ **Мини-язык и анализ**

Соберите мини-язык — присваивания, if, while, assert, чтение входа — построчный парсер и конкретный интерпретатор, который собирает наблюдения и реальные ошибки. Напишите общий анализатор, параметризованный доменом: слияние веток объединением, цикл через неподвижную точку, перенос условий, предупреждения о делении на возможно-ноль, чтении возможно неинициализированной переменной и возможно нарушенном ассерте. Продумайте, почему отсутствующая переменная в состоянии означает $\top$, а не $\perp$, и почему состояния и предупреждения тел циклов стоит записывать только от финального инварианта.

⑤ **Проверка и сравнение доменов**

Напишите детерминированный генератор случайных программ: циклы с гарантированным завершением и случайные входы для интерпретатора. Сверьте анализ с прогонами: каждое реально достижимое значение лежит в конкретизации γ абстрактного состояния той же точки, а каждая реальная ошибка предупреждена на том же операторе. Продумайте, что будет, если нарушится одно из двух условий, и почему случайные программы не доказывают корректность, но надёжно ловят её потери.

Затем соберите корпус программ: циклы со счётчиком, деление на вход, безопасное деление в цикле, дизъюнктивные значения, чтение неинициализированной переменной, реальные и ложные нарушения ассертов. Для каждой программы сведите в таблицу предупреждения и ложные срабатывания обоих доменов с эталоном из конкретного интерпретатора. Объясните, какие программы доказывает каждый домен и где они дополняют друг друга: интервалы видят границы, знаки — точные множества знаков.

Итог: работающий каркас абстрактной интерпретации с двумя доменами: цикл со счётчиком и известной границей получает точный инвариант и точный выход, widening сходится там, где наивная итерация расходится, narrowing возвращает точность, а рандомизированная проверка и таблица сравнения доменов на корпусе подтверждают корректность и показывают разницу в точности.

XXXII

Формальная верификация

Программа прошла миллион тестов — и всё равно может упасть на миллион первом. Миллион входов — это лишь конечное подмножество бесконечного множества возможных запусков, и ни один из них не говорит о следующем. Свойство вида «программа не упадёт ни на одном входе» — это утверждение обо всех элементах бесконечного множества. Один контрпример его опровергает, но ни одно конечное число примеров не подтверждает.

Инженерная история ведёт счёт таким поражениям. Лучевая терапевтическая машина Therac-25 в 1985–1987 годах стала источником как минимум шести аварий с передозировкой облучения: состояние гонки проявлялось только при быстром вводе параметров оператором и не воспроизводилось на большинстве запусков. Ракета Ariane 5 в 1996 году погибла через 37 секунд после старта: 64-битная скорость полёта не поместилась в 16-битную переменную бортовой системы, и необработанное переполнение отключило инерциальную систему. Обе программы были испытаны — на тех входах, которые испытатели сумели придумать.

У доказательства для всех запусков есть и положительная сторона. Компилятор CompCert для существенного подмножества языка C сопровождается машинно-проверенным доказательством: каждое преобразование исходного текста в объектный код сохраняет семантику, и гарантия действует на всех входах, а не на выборке. Микроядро seL4 снабжено доказательством корректности вплоть до машинного кода и используется в критических системах — от автономных дронов до спутников.

«Программа корректна» — фраза, которая сама по себе ничего не означает. Программа не содержит в себе требования к самой себе, корректность — это отношение между текстом и требованием к нему. Прежде чем доказывать, требование нужно записать — превратить невнятное «работает правильно» в точное утверждение. Тройка Хоара и есть такая запись: если перед оператором выполнено предусловие, то после его завершения выполнено постусловие. Предусловие и постусловие вместе образуют контракт: тройка описывает то, что программа *обещает*.

В главе 1 есть инструмент для рассуждений обо всех элементах множества сразу — математическая индукция. Программа — конечный текст из операторов, и тот же приём применим к ней. Доказательство по структуре программы покрывает все её запуски так же, как индукция покрывает все натуральные числа. Трудность — циклы: число шагов заранее неизвестно и у каждого запуска своё. Индукция решает и эту задачу, если найти в цикле то, что остаётся неизменным: *инвариант*.

Найти инвариант — значит разгадать, что во время всех этих изменений состояния программы остаётся прежним.

Доказательство тройками — это вывод: из предусловия выводится постусловие применением правил к каждому оператору. Доказательство становится механическим: инструменты верификации превращают размеченную программу в условия верификации и передают их SMT-солверам главы 15 вроде Z3, CVC5, Yices и Alt-Ergo. Человек придумывает инварианты, машина проверяет следствия.

Тестирование при этом не исчезает: падающий тест — это контрпример к утверждению о корректности. Тестирование и доказательство в инженерной практике работают вместе. В главе 31 был другой ответ на тот же вопрос: верификация как абстракция — выполнить программу на грубых обобщениях состояний, где инвариант вычисляют ценой приближения. А поведение систем во времени — слова «всегда», «когда-нибудь», «на следующем шаге» — станет предметом главы 33.

Как доказать корректность программы для всех её запусков, не выполнив её ни на одном?

ИСТОРИЯ От индуктивных утверждений к аксиоматической семантике

Связь программ с логикой устанавливалась в четыре шага:

- Роберт Флойд, 1967 год: состояниям программы приписываются индуктивные утверждения, доказываемые индукцией по структуре вычисления²²⁶.
- Тони Хоар, 1969 год: та же идея оформляется как аксиоматическая семантика — правила вывода для каждого оператора, включая аксиому присваивания и правило цикла²²⁷.
- Эдсгер Дейкстра: рассуждение разворачивается в обратную сторону — исчисление слабейших предусловий выводит предусловие из постусловия и становится основой автоматической генерации условий верификации²²⁸.
- Стивен Кук, 1978 год: относительная полнота — всякая верная тройка выводима в правилах Хоара²²⁹.

Глава 31 уже упоминала индуктивные утверждения Флойда. Здесь они становятся рабочим инструментом верификации.

ОБЗОР ГЛАВЫ

Начнём с троек Хоара: контракт «предусловие — оператор — постусловие» как формула о программе — и полный набор правил вывода, от аксиомы присваивания до

²²⁶R. W. Floyd. «Assigning Meanings to Programs». 1967.

²²⁷C. A. R. Hoare. «An Axiomatic Basis for Computer Programming». Communications of the ACM, 1969.

²²⁸E. W. Dijkstra. «A Discipline of Programming». Prentice-Hall, 1976.

²²⁹S. A. Cook. «Soundness and Completeness of an Axiom System for Program Verification». SIAM Journal on Computing, 1978.

правил для условного оператора и цикла. Завершит раздел метатеория: доказательство того, что всякая выводимая тройка верна, и замечание о полноте правил. Затем — инварианты: как их угадывать, как доказательство сохранения превращается в индукцию и как вариант цикла добавляет к частичной корректности завершение. В конце — путь от аннотированной программы к SMT-запросу: условия верификации, критерий корректности через их общезначимость, разбор верификации обмена в Dafny и роль контрпримеров.

После этой главы вы сможете:

1. записывать контракты программы тройками Хоара;
2. доказывать корректность операторов правилами вывода и объяснять, почему выводимой тройке можно доверять;
3. находить инварианты циклов, доказывать их сохранение и терминируемость вариантом;
4. выписывать условия верификации размеченной программы и формулировать критерий их общезначимости;
5. сводить проверку контракта к SMT-запросу солверу.

§ 32.1 Тройки и правила вывода

Методы доказательств этой главы — прямое рассуждение, контрапозиция, индукция — находят прямое применение в программировании. Утверждение о программе — это формула, а проверка программы — доказательство этой формулы. Заодно различие тестирования и доказательства, о котором шла речь во введении, станет здесь точным.

32.1.1 Утверждения и тройки Хоара

ОПРЕДЕЛЕНИЕ 32.1 Ассерт

Логическая формула, размещённая в коде, называется **ассертом**, если она должна быть истинна в этой точке выполнения.

Ассерты документируют ожидания и отлавливают ошибки во время выполнения. В логике главы 1 «утверждение» — синоним высказывания. Чтобы не смешивать два смысла, кодовую проверку назовём ассертом.

ОПРЕДЕЛЕНИЕ 32.2 Тройка Хоара

Тройка

$$\{P\} S \{Q\}$$

называется **тройкой Хоара**, если в каждом запуске, в котором предусловие P истинно перед выполнением оператора S и оператор завершается, постусловие Q истинно после завершения.

Оговорка о завершении фиксирует *частичную* корректность: тройка не требует, чтобы оператор вообще остановился. Тотальную корректность — завершение плюс постусловие — рассмотрим в разделе о терминируемости. Три части тройки — предусловие, оператор и постусловие — связаны, как контракт: перед оператором программа обязана удовлетворить P , а после него обещает Q , как показано на рис. 120.



Рис. 120. Контракт: предусловие, оператор, постусловие.

Пример Корректная тройка Хоара

$$\{x = 5\}x := x + 1\{x = 6\}$$

Здесь x равно 5 до присваивания и x равно 6 после.

Пример Тройка Хоара с условным оператором

$$\{x \geq 0\} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \{y = |x|\}$$

Если вход неотрицателен, то после условного оператора выполнено $y = |x|$ — при $x > 0$ ветка «then» записывает в y значение x , при $x = 0$ ветка «else» записывает $-x$, и тогда $y = -x = 0 = |x|$. Случай $x < 0$ исключён предусловием.

Замечание Частичная наблюдаемость тройки

Тройка Хоара говорит только о пред- и постусловии: она фиксирует, что постусловие выполнено *после* выполнения оператора S , но не описывает, что происходит *во время* выполнения. Промежуточные состояния, через которые прошёл оператор, тройкой не контролируются.

- Предусловие кодирует предположения о состоянии программы.
- Постусловие кодирует гарантии.

Вместе они образуют контракт: если вызывающая сторона удовлетворяет P , процедура обеспечивает Q .

32.1.2 Аксиома присваивания и прямой проход

Аксиома присваивания — единственное правило, связывающее присваивание с логикой.

ОПРЕДЕЛЕНИЕ 32.3 Аксиома присваивания

Тройка

$$\{P[x := E]\} x := E \{P\}$$

верна для любого постусловия. Здесь $P[x := E]$ — формула P , в которой каждое свободное вхождение x заменено выражением E .

Примечание Тонкость подстановки

Подстановка $P[x := E]$ заменяет все свободные вхождения x одновременно, а выражение E вычисляется в *старом* состоянии, до присваивания. Поэтому аксиома читается «назад»: из постусловия она восстанавливает предусловие.

Прямой проход — рассуждение от предусловия к постусловию, в направлении, противоположном чтению аксиомы. Выполнив присваивание $x := E$, мы знаем, что новое значение x равно значению выражения E , вычисленному до присваивания. Подставляя это новое значение в известные утверждения, мы продвигаем их вперёд по программе, пока не получим постусловие. Точный обратный инструмент — *слабейшее предусловие* по аксиоме, которое восстанавливает предусловие из постусловия.

Пример Прямой проход: удвоение числа

Программа $y := 2 * x$ удваивает число из x и записывает результат в y , и тройка Хоара записывает это как

$$\{x = a\} y := 2 * x \{y = 2a\}$$

.

При прямом проходе предусловие даёт $x = a$, а после присваивания мы знаем $y = 2 * x$, откуда $y = 2 * a = 2a$. При обратном проходе из постусловия $y = 2a$ аксиома присваивания восстанавливает предусловие $(y = 2a)[y := 2 * x]$, то есть $2 * x = 2a$, и при $x = a$ оно выполнено, так что два направления сошлись. Числовой прогон при $a = 3, x = 3$ подтверждает это: после $y := 2 * x$ получаем $y = 6 = 2 * 3$.

ОПРЕДЕЛЕНИЕ 32.4 Правило композиции

Из тройки

$$\{P\} S_1 \{R\}$$

и тройки

$$\{R\} S_2 \{Q\}$$

следует тройка

$$\{P\} S_1 \circ S_2 \{Q\}$$

. Промежуточное условие R склеивает два оператора в последовательность.

Правило читается как склеивание: R — единственная связь между S_1 и S_2 , через которую проходит рассуждение, как на рис. 121.

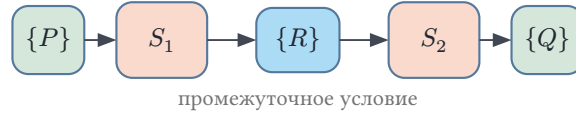


Рис. 121. Композиция двух операторов с промежуточным условием.

Пример Композиция в действии

Докажем тройку

$$\{x = 3\} x := x + 1 \circ x := x \cdot 2 \{x = 8\}$$

, идя от постусловия назад.

Для второго присваивания из постусловия $x = 8$ получаем промежуточное условие $R = (x = 8)[x := x \cdot 2]$, то есть $x \cdot 2 = 8$, откуда $x = 4$. Для первого присваивания аксиома из $x = 4$ даёт предусловие $(x = 4)[x := x + 1]$, то есть $x + 1 = 4$, откуда $x = 3$.

Промежуточное условие $R \equiv x = 4$ склеивает операторы по правилу композиции, и тройка доказана.

ОПРЕДЕЛЕНИЕ 32.5 Правило следствия

Из тройки

$$\{P'\} S \{Q'\}$$

, из посылки $P \rightarrow P'$ и посылки $Q' \rightarrow Q$ следует тройка

$$\{P\} S \{Q\}$$

.

Предусловие можно усилить, а постусловие ослабить. Если тройка верна для более широкого предусловия P' , она верна и для более узкого P , а из более сильного постусловия Q' следует более слабое Q .

Пример Усилить предусловие, ослабить постусловие

Доказана тройка

$$\{x = 5\} x := x + 1 \{x = 6\}$$

. Усилим предусловие: добавим факт, не касающийся x .

$$\{x = 5 \wedge y = 0\} x := x + 1 \{x = 6\}$$

. Тройка остаётся верной: лишний факт присваивание не трогает.

Ослабим постусловие: из $x = 6 \Rightarrow x \geq 5$, и тройка

$$\{x = 5\} x := x + 1 \{x \geq 5\}$$

тоже верна.

ОПРЕДЕЛЕНИЕ 32.6 Слабейшее предусловие

Самая широкая формула, для которой верна тройка

$$\{P\} S \{Q\}$$

, называется **слабейшим предусловием** и обозначается $\text{wp}(S, Q)$. Всякая формула P , для которой эта тройка верна, влечёт $\text{wp}(S, Q)$.

Для присваивания $\text{wp}(x := E, Q) = Q[x := E]$. Для последовательности $\text{wp}(S_1 ; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$. Для условного оператора $\text{wp}(\text{if } B \text{ then } S_1 \text{ else } S_2, Q) = (B \wedge \text{wp}(S_1, Q)) \vee (\neg B \wedge \text{wp}(S_2, Q))$.

Обратный проход от постусловия к предусловию, вычисляющий wp , — стандартный способ генерации условий верификации.

Пример Слабейшее предусловие условного оператора

Программа $\text{if } x > 0 \text{ then } y := 1 \text{ else } y := 0$ с постусловием $y = 1$ проверяется по формуле wp для условного оператора. Считаем ветки подстановкой: $\text{wp}(y := 1, y = 1) = (1 = 1)$ — всегда истинно, а $\text{wp}(y := 0, y = 1) = (0 = 1)$ — всегда ложно. Собираем формулу целиком:

$$(x > 0 \wedge (1 = 1)) \vee (x \leq 0 \wedge (0 = 1)).$$

Первая скобка упрощается до $x > 0$, вторая всегда ложна, поэтому слабейшее предусловие — $x > 0$. При $x = 0$ оно нарушено, и программа действительно выдаёт $y = 0$. Усиленное предусловие $x > 3$ тоже корректно, но отсекает входы, на которых программа справляется.

32.1.3 Правила для условного оператора и цикла

ОПРЕДЕЛЕНИЕ 32.7 Правило условного оператора

Из тройки

и тройки

$$\{P \wedge B\} S_1 \{Q\}$$

следует тройка

$$\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{Q\}$$

Условие B разводит вычисление на две ветки, и обе должны привести к одному постусловию Q , как показано на рис. 122.

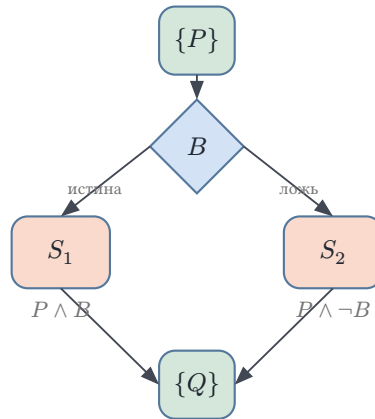


Рис. 122. Две ветки условного оператора.

Вернёмся к примеру с модулем: правило условного оператора разбирает его на две ветки, и обе должны сойтись к постусловию $y = |x|$.

Ветка если. Предусловие $x \geq 0 \wedge x > 0$ влечёт $x = |x|$, поэтому после присваивания $y := x$ получаем $y = |x|$.

Ветка иначе. Предусловие $x \geq 0 \wedge \neg(x > 0)$ влечёт $x = 0$, поэтому после присваивания $y := -x$ получаем $y = -0 = 0 = |0| = |x|$.

По правилу условного оператора тройка

$$\{x \geq 0\} \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \{y = |x|\}$$

доказана.

ОПРЕДЕЛЕНИЕ 32.8 Правило цикла

Из тройки

$$\{I \wedge B\} S \{I\}$$

следует тройка

$$\{I\} \text{ while } B \text{ do } S \{I \wedge \neg B\}$$

. Это частичная корректность — завершение цикла не гарантировано.

Формула I в правиле цикла — инвариант: он выполняется перед каждой проверкой условия и сохраняется телом. Где взять инвариант и что от него требуется — предмет следующего раздела.

Проверка Правила вывода

1. Какие две посылки требует правило условного оператора и что должно быть общим у обеих веток?
2. Почему заключение правила цикла содержит $I \wedge \neg B$, хотя тело цикла выполняется только при истинном B ?
3. Правило следствия усиливает предусловие и ослабляет постусловие — почему это не портит уже доказанную тройку?

32.1.4 Корректность и полнота правил

Правила собраны в исчисление, и перед тем как на него опираться, нужно доказать метатеорему: механика вывода не порождает лжи.

ТЕОРЕМА 32.1 Корректность правил Хоара

Всякая тройка, выводимая правилами присваивания, композиции, следствия, условного оператора и цикла, верна: в каждом запуске, где предусловие истинно и программа завершается, истинно постусловие.

Доказательство

Докажем индукцией по дереву вывода: каждая аксиома даёт верную тройку, а каждое правило из верных посылок строит верное заключение. Верность тройки

$$\{P\} S \{Q\}$$

— её определение: в каждом запуске с истинным P и завершившимся S истинно Q .

База: присваивание. Пусть состояние σ удовлетворяет $P[x := E]$. Присваивание переводит σ в состояние σ' , где x равно значению выражения E в σ , а остальные переменные не изменились. По определению подстановки формула P истинна в σ' тогда и только тогда, когда $P[x := E]$ истинна в σ : все вхождения x в P при вычислении в σ' получают значение выражения E , вычисленное в σ . Значит, постусловие P истинно после присваивания, и аксиома даёт верную тройку.

Композиция. Пусть посылки

$$\{P\} S_1 \{R\}$$

и

$$\{R\} S_2 \{Q\}$$

верны по индуктивному предположению. Рассмотрим запуск $S_1 \circ S_2$ из состояния с истинным P : первая часть завершается в состоянии с истинным R , и это состояние служит началом запуска второй части, которая завершается с истинным Q .

Условный оператор. Пусть посылки

$$\{P \wedge B\} S_1 \{Q\}$$

и

$$\{P \wedge \neg B\} S_2 \{Q\}$$

верны. Всякий запуск начинается с проверки B : при истинном B начальное состояние удовлетворяет $P \wedge B$, и ветка S_1 по индуктивному предположению приводит к Q . При ложном — то же рассуждение для ветки S_2 . Обе ветки сходятся к Q .

Цикл. Пусть посылка

$$\{I \wedge B\} S \{I\}$$

верна. Покажем индукцией по числу выполненных итераций, что перед каждой проверкой условия истинно I . База — вход в цикл: I истинно по предположению о запуске. Шаг — итерация начинается в состоянии с истинным $I \wedge B$ и по индуктивному предположению о теле завершается с истинным I . Если цикл завершается после n итераций, последняя проверка происходит в состоянии с истинным I и ложным B , то есть с истинным $I \wedge \neg B$.

Следствие. Посылки $P \rightarrow P'$ и $Q' \rightarrow Q$ — чистые формулы без операторов программы: они истинны во всех состояниях. Запуск из состояния с истинным P есть запуск из состояния с истинным P' , и для него по индуктивному предположению после завершения верно Q' , откуда верно и Q .

Во всех случаях заключение верно, и индукция по дереву вывода завершает доказательство.

Корректность правил доказана индукцией, и выбор метода предопределён устройством самих правил. Дерево вывода конечно: каждое применение правила строит заключение из конечного числа посылок, и дерево растёт из конечного числа аксиом. Семантика тройки говорит обо всех запусках — о бесконечном множестве траекторий состояния. Индукция остаётся мостом между конечным рассуждением

и бесконечным утверждением: в главе 1 она заменяла перебор натуральных чисел, здесь — перебор деревьев вывода.

Внутри случая цикла спрятана вторая индукция. Правило цикла берёт в посылке сохранение инварианта одним телом и выдаёт заключение обо всех итерациях сразу. Промежуточный шаг — индукция по числу итераций — упакован в саму фигуру вывода. Применение правила обменивает одну индукцию на другую: дерево вывода уменьшается на узел, а работа смещается к поиску инварианта — единственной части рассуждения, которую механика вывода не берёт на себя.

За корректностью стоит пара свойств из главы 2: доказуемость влечёт истинность ($\vdash \subseteq \models$), а истинность доказуемость ($\models \subseteq \vdash$). Корректность правил Хоара — левое включение в новой одежде. Пустая система правил тоже корректна, поэтому интерес начинается с правого включения — выводит ли исчисление всё истинное.

Примечание Полнота правил Хоара

Правое включение для троек Хоара доказал Стивен Кук: система правил *относительно полна*. Всякая верная тройка выводима, если язык ассертов достаточно выразителен, чтобы записывать нужные инварианты и слабейшие предположения. Оговорка о выразительности — не мелочь: для бедного языка формул исчисление может быть полным и бессодержательным одновременно, ведь в нём просто нечего доказывать. Практическое следствие: если тройка верна, но доказательство не идёт, правила вывода не виноваты — недостаёт либо инварианта, либо выразительного ассерта.

§ 32.2 Инварианты и тотальная корректность

Правило цикла поставило формулу I в посылку и оставило открытым вопрос, где её взять. Инвариант — ремесло: определение фиксирует, каким он обязан быть, угадывание требует практики, а доказательство сохранения каждый раз превращается в индукцию. Замыкает раздел терминируемость — недостающая половина корректности.

32.2.1 Инварианты циклов

ОПРЕДЕЛЕНИЕ 32.9 Инвариант цикла

Формула I называется **инвариантом** цикла $\text{while } B \text{ do } S$, если выполнены три условия:

1. тело цикла сохраняет I , то есть верна тройка

$$\{I \wedge B\} S \{I\}$$

;

2. $I \wedge \neg B$ влечёт постусловие;

3. I выполняется перед первой итерацией.

Инвариант живёт в узле цикла: он выполняется до проверки условия, тело его сохраняет, а выход приводит к постусловию, как показано на рис. 123.

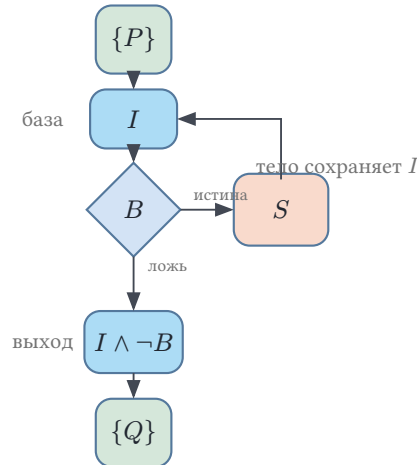


Рис. 123. Цикл с инвариантом: база, сохранение, выход.

Это база индукции, отдельная посылка доказательства. Из первого условия правило цикла даёт тройку

$$\{I\} \text{ while } B \text{ do } S \{I \wedge \neg B\}$$

. Вторая посылка превращает $I \wedge \neg B$ в постусловие правилом следствия. Инвариантов у цикла много, и большинство бесполезны: формула `true` сохраняется любым телом, но на выходе не оставляет ничего. Полезный инвариант достаточно широк, чтобы тело его сохраняло, и достаточно точен, чтобы $I \wedge \neg B$ давало постусловие.

Примечание Как угадать инвариант

Инвариант начинают с постусловия и ослабляют, пока его не начнёт сохранять тело цикла. В линейном поиске постусловие « x найден или $i = n$ » слишком точное для середины итерации: к позиции i ещё не обращались. Ослабляем до «все позиции до i не содержат x »: тело проверяет ровно текущую позицию, и свойство сохраняется. К ослабленному постусловию добавляют границы вроде $0 \leq i \leq n$. Это защищает чтение $A[i]$ от выхода за пределы массива.

Пример Инвариант цикла: линейный поиск

Линейный поиск ищет $x \in A[0..n-1]$ и возвращает n , если элемент не найден.

```

i := 0
while i < n and A[i] != x do
  i := i + 1
  
```

Инвариант. $I \equiv 0 \leq i \leq n \wedge \forall j \in \{0, \dots, i-1\} A[j] \neq x$ — все позиции до i проверены и не содержат x . Докажем индукцией по числу итераций, что I — инвариант цикла.

База. Перед циклом $i = 0$, диапазон $\{0, \dots, -1\}$ пуст, и формула $\forall j \in \emptyset$ истинна вакуумно, так что I выполнено.

Шаг. Пусть инвариант I и условие цикла $i < n \wedge A[i] \neq x$ истинны, тело увеличивает счётчик на единицу, и надо показать $I[i := i + 1]$. Условие цикла даёт $i + 1 \leq n$ и $A[i] \neq x$, а по инварианту I все позиции до i не содержат x , поэтому все позиции до $i + 1$ не содержат x , и инвариант сохраняется.

Выход. Из инварианта I и отрицания условия цикла $\neg(i < n \wedge A[i] \neq x) \equiv i \geq n \vee A[i] = x$ получаем постусловие, и с учётом границы $i \leq n \wedge i \geq n \Rightarrow i = n$ случай $i \geq n$ означает, что элемент не найден, а случай $A[i] = x$ означает, что элемент найден.

Шаг и выход используют правило следствия: предусловие усиливается условием цикла, постусловие ослабляется до желаемого.

Пример Инвариант цикла: накопление суммы

Программа суммирует элементы массива:

```
s := 0
i := 0
while i < n do
  s := s + a[i]
  i := i + 1
```

Инвариант. $I \equiv 0 \leq i \leq n \wedge s = \sum_{j=0}^{i-1} a_j$ — s хранит сумму первых i элементов.

Докажем индукцией по числу итераций, что I — инвариант цикла. **База.** Перед циклом $i = 0$, сумма по пустому диапазону равна 0, и $s = 0$. **Шаг.** Тело добавляет к сумме элемент a_i и увеличивает i на единицу, поэтому инвариант снова выполнен. **Выход.** При $i = n$ сумма равна $\sum_{j=0}^{n-1} a_j$, то есть накоплен весь массив.

32.2.2 Верификация программ через индукцию

Доказательство того, что инвариант сохраняется телом цикла, — это шаг индукции. База индукции — инвариант, установленный перед первой итерацией. Шаг — переход: если I истинно перед произвольной итерацией, то тело оставляет его истинным перед следующей. Принцип математической индукции из главы 1 закрывает все итерации сразу: для любого их числа инвариант остаётся на месте.

Разборы линейного поиска и накопления суммы устроены одинаково, хотя программы разные: устанавливаем инвариант, доказываем сохранение, извлекаем по-

стусловие из выхода. Смена программы меняет формулу I , но скелет рассуждения — база, сохранение, выход — повторяется от цикла к циклу. Правило цикла упаковывает этот скелет в одну фигуру вывода: посылка сохранения — шаг индукции, заключение — вывод для всех итераций. Именно так инструменты верификации доказывают корректность программ. Развернуть цикл на конечное число итераций можно и без инварианта: раскрутка на k шагов даёт верное утверждение о первых k итерациях и ничего — обо всех остальных. Раскрутка — родственник тестирования: как и тест, она проверяет конечную выборку запусков, и как тест, она не закрывает вопрос. Инвариант — то, что превращает конечную раскрутку в утверждение обо всех итерациях. Найти его — содержательная часть доказательства, не поддающаяся автоматизации.

Проверка Верификация через индукцию

1. Какие три условия делают формулу I инвариантом цикла, и какое из них является шагом индукции?
2. Что в доказательстве линейного поиска играет роль базы индукции?
3. Почему сохранение инварианта доказывается для одного шага тела, а выводится для любого числа итераций?

32.2.3 Терминируемость

Частичная корректность не требует завершения: тройка

$$\{P\} S \{Q\}$$

утверждает лишь, что если S завершается, то Q верно. Тотальная корректность требует и завершения, и постусловия.

ОПРЕДЕЛЕНИЕ 32.10 Тотальная тройка

Тройка $[P] S [Q]$ называется **тотальной**, если из истинности P перед выполнением следует, что S завершается и после него верно Q .

Завершение цикла доказывают через *вариант цикла* (loop variant) — функцию f от состояния программы со значениями во вполне упорядоченном множестве (обычно натуральных числах), которая строго убывает на каждой итерации. Вариант — это прогресс, каждая итерация приближает состояние к выходу, тогда как инвариант — сохранение. Вполне упорядоченное множество не содержит бесконечно убывающих цепочек, поэтому цикл не может выполняться вечно — рано или поздно вариант достигает минимального значения, и условие выхода срабатывает.

ОПРЕДЕЛЕНИЕ 32.11 Правило цикла с вариантом

Из тройки

$$\{I \wedge B \wedge f = z\} S \{I \wedge f < z\}$$

следует $[I]$ while B do S $[I \wedge \neg B]$. Здесь f — вариант цикла, а z — свежая переменная, фиксирующая значение f до итерации.

Для тотальной корректности правило цикла усиливают вариантом. На каждой итерации вариант уменьшается, и во вполне упорядоченном множестве цепочка значений обрывается, как показано на рис. 124.

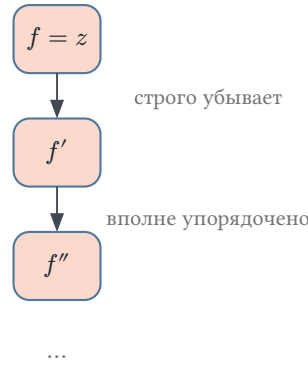


Рис. 124. Вариант убывает: бесконечная цепочка невозможна.

Пример Вариант цикла

Программа while $x > 0$ do $x := x - 1$ завершается. Вариант $f = x$. На каждой итерации x уменьшается на 1. Натуральные числа вполне упорядочены, поэтому бесконечная убывающая цепочка $x > x - 1 > x - 2 > \dots$ невозможна.

§ 32.3 От аннотаций к SMT-солверу

Разметка предусловиями, постусловиями и инвариантами превращает доказательство корректности в конечный список формул. Условие верификации — атом этого списка: формула об одном фрагменте программы, и её проверка не требует запускать программу.

ОПРЕДЕЛЕНИЕ 32.12 Условие верификации

Формула вида $A \Rightarrow \text{wp}(S, A')$, где A — ассерт перед фрагментом S и A' — ассерт после него, называется **условием верификации** фрагмента S . Для цикла while B do S с инвариантом I инструмент порождает три условия: базу $P \Rightarrow I$, сохранение $I \wedge B \Rightarrow \text{wp}(S, I)$ и выход $I \wedge \neg B \Rightarrow Q$.

ТЕОРЕМА 32.2 Критерий корректности

Аннотированная программа с контрактом

$$\{P\} S \{Q\}$$

корректна — в смысле частичной корректности — тогда и только тогда, когда все её условия верификации общезначимы.

Набросок доказательства

Докажем индукцией по структуре размеченной программы: у каждого узла — своё правило вывода, и осталось проверить, что условия верификации совпадают с посылками правил.

В одну сторону: пусть все условия верификации общезначимы. Для фрагмента S между соседними ассертами A и A' общезначимость $A \Rightarrow \text{wp}(S, A')$ по правилу следствия даёт тройку

$$\{A\} S \{A'\}$$

: определения wp для присваивания, композиции и условного оператора в точности повторяют соответствующие правила вывода. Для цикла три условия — база, сохранение и выход — служат посылками правила цикла и правила следствия. Правило композиции склеивает фрагменты в цепочку, и на вершине остаётся тройка контракта.

В обратную сторону: пусть программа корректна. По индукции формула $\text{wp}(S, A')$ истинна ровно в тех состояниях, из которых фрагмент S завершается с истинным A' . Условие $A \Rightarrow \text{wp}(S, A')$ истинно во всех состояниях: иначе корректность нарушалась бы на состоянии, где A истинно, а S уводит вычисление от A' .

АЛГОРИТМ Верификация размеченной программы

1. Программист записывает предусловия, постусловия и инварианты циклов.
2. Инструмент генерирует условия верификации (verification conditions) — логические формулы, истинность которых гарантирует корректность программы.
3. SMT-солверы главы 15 — Z3, CVC5, Yices, Alt-Ergo — проверяют эти формулы на языке SMT-LIB, решая, выполнимо ли отрицание каждой из них.

Среди промышленных инструментов верификации:

- Dafny²³⁰ — верифицирует аннотированные программы на собственном языке.
- Why3²³¹ — портал, передающий условия верификации разным солверам.
- Frama-C²³² — анализ кода на языке C.

Теперь пройдем весь путь до конца — от программы к условию верификации, SMT-LIB-запросу и ответу солвера.

²³⁰K. R. M. Leino. «Dafny: An Automatic Program Verifier for Functional Correctness». LPAR 2010 (LNCS 6355).

²³¹J.-C. Filliâtre, A. Paskevich. «Why3 — Where Programs Meet Provers». ESOP 2013.

²³²CEA LIST. «Frama-C User Manual». 2008–2025.

Пример Условие верификации для трёх строк

Программа:

```
x := 1
y := x + 1
assert y == 2
```

Инструмент верификации превращает программу в формулу: если выполнены оба присваивания, то выполнено и постусловие. Условие верификации имеет вид

$$(x = 1) \wedge (y = x + 1) \Rightarrow (y = 2).$$

Формула общезначима, и её отрицание невыполнимо. Эту формулу инструмент отправляет SMT-солверу — Z3, CVC5 или Alt-Ergo — и солвер отвечает, что контрпримеров нет.

Пример От программы к условию верификации и SMT

Программа обменивает значения двух переменных:

```
t := x
x := y
y := t
```

Докажем тройку

$$\{x = a \wedge y = b\} t := x ; x := y ; y := t \{x = b \wedge y = a\}$$

. Здесь a, b — исходные значения x и y . Вычислим слабое предусловие обратным проходом. Для последнего присваивания $\text{wp}(y := t, x = b \wedge y = a) = x = b \wedge t = a$. Для второго присваивания $\text{wp}(x := y, x = b \wedge t = a) = y = b \wedge t = a$. Для первого присваивания $\text{wp}(t := x, y = b \wedge t = a) = y = b \wedge x = a$. Слабейшее предусловие — $x = a \wedge y = b$, в точности предусловие тройки, поэтому программа корректна.

Проверим вывод машиной. Каждое присваивание — это равенство между свежей константой и старым значением, и условие верификации говорит, что всякий раз, когда выполнены предусловие и все равенства переходов, выполнено и постусловие. Отрицание условия передаём солверу на языке SMT-LIB из главы 15:

```
(declare-const x0 Int)
(declare-const y0 Int)
(declare-const a Int)
(declare-const b Int)
(declare-const t0 Int)
```

```
(declare-const x1 Int)
(declare-const y1 Int)

(assert (not
  (=> (and (= x0 a) (= y0 b)
    (= t0 x0) (= x1 y0) (= y1 t0))
    (and (= x1 b) (= y1 a)))))
(check-sat)
```

Солвер отвечает `unsat` — отрицание условия невыполнимо, значит, само условие общезначимо и тройка верна. Тройка корректна тогда и только тогда, когда отрицание её условия верификации невыполнимо. Если бы солвер ответил `sat`, он предъявил бы модель — контрпример, на котором тройка нарушается.

Пример Условия верификации цикла: три формулы

Программа суммирует числа от 0 до n :

```
s := 0
i := 0
while i <= n do
  s := s + i
  i := i + 1
```

Контракт:

$$\{n \geq 0\} S \left\{ s = \frac{n \cdot (n + 1)}{2} \right\}$$

, инвариант $I \equiv 0 \leq i \leq n + 1 \wedge s = \frac{i \cdot (i - 1)}{2}$. Граница $n + 1$ в инварианте оправдана условием цикла: при $i = n$ тело выполняется ещё раз, и счётчик доходит до $n + 1$. Инструмент размечает три точки — вход в программу, вершину цикла и выход — и порождает по условию верификации на каждую.

База: инвариант установлен до первой итерации. Обратный проход по инициализации даёт $\text{wp}(s := 0; i := 0, I) \equiv 0 \leq 0 \leq n + 1 \wedge 0 = \frac{0 \cdot (0 - 1)}{2}$, и условие принимает вид

$$n \geq 0 \Rightarrow (0 \leq n + 1 \wedge 0 = 0).$$

Сохранение: тело оставляет инвариант, если условие впустило итерацию. Обратный проход по телу даёт $\text{wp}(s := s + i; i := i + 1, I) \equiv 0 \leq i + 1 \leq n + 1 \wedge s + i = \frac{i \cdot (i + 1)}{2}$, и условие принимает вид

$$\begin{aligned} & \left(0 \leq i \leq n + 1 \wedge s = \frac{i \cdot (i - 1)}{2} \wedge i \leq n \right) \\ \Rightarrow & \left(0 \leq i + 1 \leq n + 1 \wedge s + i = \frac{i \cdot (i + 1)}{2} \right). \end{aligned}$$

Выход: инвариант вместе с отрицанием условия цикла даёт постусловие. Условие принимает вид

$$\left(0 \leq i \leq n + 1 \wedge s = \frac{i \cdot (i - 1)}{2} \wedge i > n\right) \Rightarrow \left(s = \frac{n \cdot (n + 1)}{2}\right).$$

Каждая формула общезначима по школьной арифметике. В сохранении к равенству инварианта прибавляется i : из $s = \frac{i \cdot (i - 1)}{2}$ следует $s + i = \frac{i \cdot (i + 1)}{2}$. В выходе неравенства $i \leq n + 1$ и $i > n$ дают $i = n + 1$, и подстановка в инвариант даёт постусловие. Все три формулы уходят солверу — как и в примере с обменом, солвер отвечает `unsat` на отрицание каждой.

Промышленные инструменты прячут выписанные формулы, но делают ровно это. Верификатор `Dafny` принимает программу на собственном языке с аннотациями-контрактами и проверяет метод целиком. Программа обмена из примера выше записывается так:

```
method Swap(x: int, y: int) returns (u: int, v: int)
  ensures u == y && v == x
{
  var t := x;
  u := y;
  v := t;
}
```

Контракт состоит из единственного постусловия `ensures`, предусловие пусто, и любой вход допустим. `Dafny` переводит метод в промежуточный язык `Boogie`, `Boogie` строит условия верификации — ту же цепочку слабейших предусловий, что выписана в примере выше, — и передаёт их SMT-солверу `Z3`. Солвер отвечает, что все условия общезначимы, и верификатор принимает метод: тело не содержит ни одной аннотации, потому что цепочка присваиваний покрывается подстановками без инвариантов.

Достаточно испортить контракт, чтобы увидеть обратную сторону. Заменяем постусловие на неверное:

```
method BadSwap(x: int, y: int) returns (u: int, v: int)
  ensures u == x && v == y
{
  var t := x;
  u := y;
  v := t;
}
```

Условие верификации $(x = a \wedge y = b) \Rightarrow (y = a \wedge x = b)$ не общезначимо: при $a \neq b$ посылка истинна, а заключение ложно. Солвер возвращает выполнимое отрицание вместе с моделью — $x = 0$, $y = 1$ — и верификатор печатает сообщение об ошибке с этими значениями в роли контрпримера. Путь от ручного SMT-LIB-запро-

са к автоматическому вердикту — один и тот же; различие лишь в том, кто строит формулы. Why3 идёт дальше по той же дороге: одна аннотированная программа, несколько солверов — и независимый вердикт каждого видно рядом.

32.3.1 Контрпримеры и тестирование

Пример Неверная тройка

Тройка

$$\{x = 5\} \ x := x + 1 \ \{x = 7\}$$

ложна. Присваивание увеличивает x на единицу, поэтому из предусловия $x = 5$ получается $x = 6$, а не 7. Условие верификации

$$(x = 5) \Rightarrow (x + 1 = 7)$$

не общезначимо — при $x = 5$ посылка истинна, а заключение $x + 1 = 7$ даёт $6 = 7$. Его отрицание выполнимо, и солвер предъявляет эту модель как контрпример.

Падающий тест — это контрпример к утверждению «программа корректна на этом входе». Тестирование может продемонстрировать наличие ошибок, но никогда их отсутствие — точно так же, как один контрпример опровергает $\forall$, но никакое количество примеров его не доказывает.

Тестирование — это поиск контрпримера, доказательство — демонстрация того, что контрпримеров не существует. Тестирование находит лёгкие ошибки, доказательство — глубокие, и в практике нужны оба. Контрпример от солвера и падающий тест — один и тот же объект в двух ролях: формула указывает вход, тест оформляет его в исполняемый прогон. Поэтому доказательство и тестирование не соперничают: проваленное доказательство дарит тестам точку, в которой ошибка живёт.

§ 32.4 Итоги главы

Тройка Хоара записывает корректность программы как контракт — предусловие, оператор, постусловие — и тем самым сводит программирование к логике главы 1. Доказательство по структуре программы заменяет перебор всех запусков. Правила вывода — аксиома присваивания, композиция, следствие, правила для условного оператора и цикла — переносят рассуждение с операторов на формулы, а инвариант превращает шаг цикла в шаг индукции. Аксиома присваивания читается в обратную сторону — из постусловия она восстанавливает предусловие.

Слабейшее предусловие даёт точный обратный инструмент — из постусловия оно восстанавливает самое широкое предусловие, для которого контракт ещё верен. Корректность правил гарантирует, что выводимая тройка верна, а критерий через условия верификации сводит корректность программы к общезначимости конеч-

ного списка формул. Правила при этом полны относительно языка ассертов: если верная тройка не выводится, недостаёт инварианта или выразительного ассерта, а не правила вывода. Верификация разделяет труд между человеком и машиной — инварианты придумывает человек, а условия верификации проверяют SMT-солверы главы 15 на языке SMT-LIB.

Частичная корректность говорит о том, что верно после завершения, тотальная — что оператор вообще завершается, и здесь работает вариант цикла. Тестирование остаётся поиском контрпримера, а доказательство — демонстрацией того, что контрпримеров нет, и это различие, начатое в самой первой главе, становится здесь точным.

Проверка Проверьте себя

1. Почему тройка Хоара — это частичная корректность, и что добавляет вариант цикла для тотальной?
2. Как аксиома присваивания читается «назад», и что такое слабейшее предусловие?
3. Как инструмент верификации превращает размеченную программу в условие верификации, и что отвечает SMT-солвер корректной программе?
4. Что гарантирует корректность правил Хоара, и какая индукция работает внутри случая цикла?
5. Какие три условия верификации инструмент порождает для цикла с инвариантом?

Что дальше?

Тройки Хоара доказывают корректность отдельной программы, размечая её предусловиями и инвариантами. В главе 33 появятся модальные и темпоральные операторы, чтобы рассуждать о системах во времени и проверять их автоматически. Абстрактная интерпретация из главы 31 — другой ответ на тот же вопрос. Она не требует инвариантов от человека, но платит за это приближением и возможными ложными срабатываниями.

§ 32.5 Упражнения

Тройки Хоара, инварианты, корректность.

1. Запишите тройку Хоара для программы, удваивающей число, и проверьте её прямым проходом.
2. Что такое инвариант цикла, и как он используется в доказательстве корректности?
3. Чем тестирование отличается от доказательства корректности? Какую роль играет контрпример, и почему тест не может доказать отсутствие ошибок?
4. Правилom следствия докажите тройку

$$\{x = 3\} x := x + 1 \{x > 3\}$$

. Исходите из тройки

$$\{x = 3\} x := x + 1 \{x = 4\}$$

.

5. Проверьте по правилу условного оператора тройку

$$\{x \geq 0\} \text{ if } x > 0 \text{ then } y := 1 \text{ else } y := 0 \{y \geq 0\}$$

.

6. Для цикла `while i < n do i := i + 1` покажите, что $0 \leq i \leq n$ — инвариант. Проверьте базу, шаг и выход и объясните, почему при выходе $i = n$.

7. Найдите слабое предусловие $\text{wp}(x := x + 1, x = 5)$.

8. Для программы `y := 2 * x` выпишите условие верификации и SMT-LIB-запрос, проверяющий тройку. Предусловие: $x = 2$. Постусловие: $y = 4$. Что ответит солвер?

9. Приведите вариант цикла для `while x < n do x := x + 1` (n фиксировано) и объясните, почему цикл завершается.

10. Для программы `s := 0; i := 0; while i < n do (s := s + i; i := i + 1)` докажите постусловие $s = \frac{n(n-1)}{2}$, подобрав инвариант цикла. Проверьте базу, сохранение и выход.

11. * Найдите слабое предусловие для программы `if x >= 0 then y := x else y := -x` и постусловия $y = |x|$, и докажите тройку правилом условного оператора.

ПРОЕКТ Различение настоящей ошибки и слишком сильного инварианта

Постройте инструмент верификации программ мини-языка тройками Хоара: инварианты придумывает человек, а условия верификации проверяет SMT-солвер. Доведите до доказательства три программы — линейный поиск, бинарный поиск, алгоритм Евклида — и разберитесь в главном практическом факте: один и тот же контрпример означает *два разных события* — либо программа сломана, либо инвариант слишком силен, и солвер опроверг именно его, а не программу. Различение «чини инвариант» и «чини программу» — критерий, который нельзя автоматизировать.

① Слабейшее предусловие

Реализуйте обратный проход по операторам: $\text{wp}(x := E, Q) = Q[x := E]$, подстановка для чтений массива через условный терм, разбор `if` на ветки, для `while` — три условия верификации (база, сохранение, выход). Отрицание каждого условия печатайте в SMT-LIB.

② Солвер

Подключите Z3: на выполнении отрицания солвер предъявляет модель — значения переменных, чтений массива и выражений программы, то есть

конкретный контрпример. Продумайте, как перевести модель в человеко-читаемое сообщение об ошибке.

③ *Цикл уточнения*

Доведите три программы — линейный поиск, бинарный поиск, алгоритм Евклида — с удалёнными инвариантами до доказательства по циклу «угадал -> контрпример -> добавил клаузу». Записывайте в журнал, какой контрпример породил какую клаузу.

④ *Ловушка*

Проверьте бинарный поиск с переполнением $\text{mid} = (\text{lo} + \text{hi}) / 2$: в математических целых программа доказывается, а в 32-битной арифметике сумма переполняется, и контрпример не лечится правкой инварианта. Примените критерий «чини инвариант» против «чини программу» на достижимом состоянии: если недостающая клауза ложна в достижимом состоянии, чинить нужно программу.

⑤ *Траектория*

Для каждой программы соберите журнал цепочки «инвариант -> контрпример -> поправка» и итоговое доказательство всех условий верификации.

Итог: инструмент верификации мини-языка с выдачей SMT-LIB и контрпримерами от Z3, три доказанные программы с журналом уточнения, где видно, какой контрпример породил какую клаузу, и явный критерий, отличающий слабый инвариант от сломанной программы.

XXXIII

Модальная логика

Глава 32 доказала корректность программ тройками Хоара: что верно до и после оператора. Но есть свойства, которые не видны ни на входе, ни на выходе: «запрос рано или поздно получит ответ», «никогда не наступит взаимная блокировка». Они говорят о самом *процессе*: о том, как система ведёт себя во времени. Но прежде — вопрос постарше: что вообще значит «утверждение истинно»?

Вот два утверждения: «Завтра в 14:00 пойдёт дождь» и $2 + 2 = 4$. Оба претендуют на истину, но истина у них разной природы: дождь мог бы и не пойти, тогда как арифметика другой быть не может. «Завтра пойдёт дождь» — истина *случайная*, она держится на капризе обстоятельств. $2 + 2 = 4$ — истина *необходимая*, не бывает мира, где это не так. Как будто слово «истинно» прячет внутри два разных смысла, а логика из первых глав слепо их смешивает.

Для двух видов истины понадобятся два оператора: $\Box$ читается «необходимо», а $\Diamond$ — «возможно». Их смысл улавливает картина возможных миров: каждый мир — одна из мыслимых картин реальности. Необходимо истинно то, что верно в каждом мире, возможно — то, что верно хотя бы в одном. В одном мире идёт дождь, в другом — нет, но $2 + 2 = 4$ истинно в каждом. Трудность в том, какие миры брать в расчёт. Для физика — те, где действуют наши законы природы: быстрее света не разогнаться никому. Для логика — все непротиворечивые картины, даже те, где физика другая. «Невозможно» в устах физика и логика — разные слова, и пока вопрос без ответа, формальной логики не построить.

Выход — придать мирам структуру достижимости. Из каждого мира достижимы лишь некоторые другие, и операторы $\Box$, $\Diamond$ проверяют формулу по достижимым мирам. Свойства самой достижимости отражаются в аксиомах: каждой аксиоме отвечает свойство отношения. Одна и та же пара операторов описывает знание, убеждение, обязанность и время, и каждое чтение диктует свои аксиомы. Знание истинно, а убеждение может быть ложным — поэтому аксиома истинности допустима для первого и запрещена для второго.

Если миры — моменты времени, а достижимость — течение времени, модальная логика становится темпоральной: $\Box$ — «всегда», а $\Diamond$ — «когда-нибудь». И это ровно тот инструмент, которого не хватало: свойства «запрос получит ответ» и «не будет взаимной блокировки» получают точный смысл. Именно их автоматически проверяет model checking. Как из загадки о завтрашнем дне вырастает инструмент, проверяющий поведение систем во времени?

ИСТОРИЯ От морского сражения до семантики Крипке

Различение случайной и необходимой истины старше формальной логики: Аристотель в трактате «De Interpretatione» на примере завтрашнего морского сражения первым заметил, что у высказываний о будущем есть своя логика. Необходимое влечёт возможное, но обратное неверно, при этом операторы выражаются друг через друга: «необходимо» значит «невозможно не», а «возможно» — «не необходимо не». Лейбниц предложил картину, определившую всю дальнейшую историю: необходимо истинно то, что верно «во всех возможных мирах». Слово «мир» здесь — метафора: *способ, каким мог бы быть мир*. Двести лет картина оставалась метафорой: не было ответа, какие миры считать и как по ним проверять формулы. Первую попытку формализации предпринял Карнап²³³: мир — полное описание того, какие атомарные утверждения истинны, что-то вроде строки таблицы истинности. «Необходимо» значило: истинно во всех таких описаниях. Попытка споткнулась о вложенные модальности: формула «возможно необходимо p » ничем не отличалась от «необходимо p ». Причина кроется в плоскости самой картины: все миры в ней равноправны, между ними нет никакой структуры, и вложенным модальностям не за что зацепиться. Структура появилась в 1959 году в статье о полноте модальной логики, написанной Крипке ещё школьником²³⁴: из данного мира видны лишь те миры, что *достижимы* из него. Вложенные модальности получили за что зацепиться: из мира видно, какие миры достижимы, из них — какие дальше, и так строится лестница вложений.

ОБЗОР ГЛАВЫ

Фреймы и операторы: что такое «мир», достижимость и оценка, как определяется истинность формул $\Box p$, $\Diamond p$. Затем мы установим прямое соответствие: свойства отношения достижимости записываются аксиомами и наоборот. Рефлексивность — это $\Box p \rightarrow p$, транзитивность — это $\Box p \rightarrow \Box \Box p$. Из аксиом соберутся именованные системы D, T, B, S_4, S_5 , мы научимся их выбирать и сложим в единую картину — куб модальностей.

Потом мы перейдём к задачам модальной логики: выполнимости и общезначимости, алгоритмам и солверам, приложениям вне model checking. Дальше мы рассмотрим другие чтения той же пары операторов: знание, убеждение, обязанность. Затем время становится модальностью: темпоральные логики LTL и CTL. В конце главы мы обратимся к model checking на практике: как свойства систем проверяются автоматически.

После этой главы вы сможете:

²³³R. Carnap. «Meaning and Necessity: A Study in Semantics and Modal Logic». 1947.

²³⁴S. A. Kripke. «A Completeness Theorem in Modal Logic». Journal of Symbolic Logic, 1959.

1. строить фреймы Крипке и вычислять истинность модальных формул;
2. связывать свойства отношения достижимости с аксиомами;
3. различать именованные системы D, T, B, S_4, S_5 и выбирать нужную;
4. читать модальности как знание, убеждение, обязанность и время;
5. объяснять model checking и проверку свойств вида «никогда не наступит блокировка».

§ 33.1 Фреймы и операторы

Модальная логика — логика *миров* и их взаимной достижимости. Формула может быть истинна в одном мире и ложна в другом, а операторы $\Box, \Diamond$ говорят о том, что верно во всех или в некотором достижимом мире.

Формулы строятся из *атомарных утверждений* — элементарных высказываний вроде «идёт дождь» или «процесс в критической секции». Атом не разбирается на части: он просто истинен или ложен в каждом мире, а формулы собираются из атомов связками и операторами, как в главе 1. Набор всех атомов обозначают AP (от англ. *atomic propositions*).

ОПРЕДЕЛЕНИЕ 33.1 Фрейм

Фрейм — пара $F = (W, R)$, где:

- W — непустое множество **миров** (точек, ситуаций).
- $R \subseteq W \times W$ — отношение **достижимости**. Запись $R(u, v)$ читается «из мира u виден мир v ».

Фрейм описывает только геометрию: какие миры из каких видны. Например, во фрейме с $W = \{w_1, w_2\}, R = \{(w_1, w_2)\}$ из w_1 виден w_2 , а из w_2 — ничего. Что в этих мирах истинно, фрейм не говорит — за это отвечает оценка.

ОПРЕДЕЛЕНИЕ 33.2 Модель и оценка

Оценка приписывает каждому атому множество миров, где он истинен: $V : AP \rightarrow \mathcal{P}(W)$. Запись $w \in V(p)$ означает «атом p истинен в мире w ». **Модель** — тройка $M = (W, R, V)$: фрейм плюс оценка. Модель **построена на фрейме F** , если она имеет вид (W, R, V) для некоторой оценки V .

В примере выше оценка может решить, что p истинен в w_2 и ложен в w_1 , а q истинен в обоих. Одна и та же геометрия фрейма допускает много оценок, и каждая даёт свою модель.

ОПРЕДЕЛЕНИЕ 33.3 Истинность модальной формулы

Отношение $M, w \models A$ («формула A истинна в мире w модели M ») определяется индукцией по построению формулы.

База и булевы связки:

- атом: $M, w \models p$ тогда и только тогда, когда $w \in V(p)$.
- отрицание: $M, w \models \neg B$ тогда и только тогда, когда $M, w \not\models B$.
- конъюнкция: $M, w \models B \wedge C$ тогда и только тогда, когда истинны обе формулы (дизъюнкция и импликация — как обычно).

Модальные случаи — единственное, что отличает эту логику от пропозициональной:

- необходимость: $M, w \models \Box B$ тогда и только тогда, когда $M, w' \models B$ для *всех* w' с $R(w, w')$.
- возможность: $M, w \models \Diamond B$ тогда и только тогда, когда $M, w' \models B$ для *некоторого* w' с $R(w, w')$.

Формула A истинна в модели M , если $\forall w \in W : M, w \models A$.

$\Box$ читается «необходимо», а $\Diamond$ — «возможно». Формула $\Box p$ означает «необходимо p », а $\Diamond p$ — «возможно p ».

Операторы двойственны — каждый выражается через другой с отрицанием:

$$\Box A = \neg \Diamond \neg A,$$

$$\Diamond A = \neg \Box \neg A.$$

В самом деле, «необходимо» значит «невозможно не», а «возможно» — «не необходимо не».

Пример Считаем истинность вручную

Пусть $W = \{w_1, w_2\}$. Из w_1 достижимы оба мира, из w_2 — только w_2 . Атом p истинен только в w_2 , атом q — в обоих. Тогда:

- $M, w_1 \models \Diamond p$: из w_1 виден w_2 , где p истинен.
- $M, w_1 \not\models \Box p$: из w_1 виден и w_1 , где p ложен.
- $M, w_1 \models \Box \Diamond p$: в каждом достижимом из w_1 мире (в w_1 и в w_2) формула $\Diamond p$ истинна — из обоих виден w_2 .

Последний пункт — первый пример *вложенной* модальности: оценка $\Diamond p$ внутри $\Box$ смотрит на достижимость «второго уровня».

Пример Три мира и $\Box(p \rightarrow q)$

Миры w_0, w_1, w_2 . Достижимость: $w_0 \rightarrow w_1, w_0 \rightarrow w_2, w_1 \rightarrow w_2, w_2 \rightarrow w_2$. Оценка: $V(p) = \{w_2\}, V(q) = \{w_1, w_2\}$. Проверим формулу $\Box(p \rightarrow q)$ в мире w_0 .

1. Из w_0 видны w_1 и w_2 .
2. В w_1 атом p ложен, поэтому импликация $p \rightarrow q$ истинна.
3. В w_2 истинны оба атома, и $p \rightarrow q$ истинна.
4. Импликация истинна во всех достижимых мирах: $M, w_0 \models \Box(p \rightarrow q)$.

Для сравнения, $\Box p$ в w_0 ложна: в w_1 атом p не выполнен. А $\Diamond(p \wedge q)$ истинна: свидетель — мир w_2 .

Замечание Пустая достижимость

Если из мира w не достижим ни один мир, то $M, w \models \Box B$ выполнено *вакуумно* — нет ни одного мира, опровергающего B . А $M, w \models \Diamond B$ ложно — нет ни одного мира, подтверждающего B .

Это тонкость с пустыми мирами, которую устраняет свойство серийности (ниже): требование, чтобы у каждого мира был хотя бы один достижимый.

Формула A валидна во фрейме F ($F \models A$), если она истинна в каждой модели на F . Валидность на уровне фреймов — основное понятие: аксиома соответствует целому классу фреймов. Квантирование по всем оценкам существенно: на уровне отдельной модели соответствие теряется. Пример — модель с одним миром w , пустым R и оценкой $V(p) = \{w\}$. Тогда $M, w \models \Box p$ вакуумно и $M, w \models p$, поэтому схема $\Box p \rightarrow p$ истинна в модели, хотя R не рефлексивно.

Чтобы из формул строить доказательства, нужна система вывода. Минимальная нормальная модальная логика K задаётся одной аксиомой и одним правилом:

$$\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q).$$

- аксиома K читается так: «если необходимо $p \rightarrow q$, то из необходимости p следует необходимость q ».
- правило необходимости: из выводимости A выводится $\Box A$ (доказанное утверждение необходимо).

Нормальными называют логики, замкнутые относительно правила необходимости и содержащие аксиому K . Сама K — наименьшая из них, все остальные получаются добавлением аксиом. Читается аксиома K на примере: если необходимо, что дождь влечёт мокрую землю, и необходимо, что идёт дождь, то необходимо, что земля мокрая.

§ 33.2 Свойства отношения и аксиомы

Главное содержание модальной логики — соответствие между свойствами отношения достижимости и аксиомами. Каждая аксиома ровно выражает одно свойство R , но на уровне фреймов.

ТЕОРЕМА 33.1 Соответствие свойства и аксиомы

Схема в правой колонке валидна во фрейме F тогда и только тогда, когда отношение R фрейма обладает свойством из левой колонки:

- *Серийность* ($\forall u \exists v : R(u, v)$) — у каждого мира есть хотя бы один достижимый. Соответствует аксиоме D : $\Box p \rightarrow \Diamond p$.
- *Рефлексивность* ($\forall w : R(w, w)$) — каждый мир виден из самого себя. Соответствует аксиоме T : $\Box p \rightarrow p$.
- *Симметричность* ($R(u, v) \rightarrow R(v, u)$) — достижимость взаимна. Соответствует аксиоме B : $p \rightarrow \Box \Diamond p$.
- *Транзитивность* ($R(u, v) \wedge R(v, w) \rightarrow R(u, w)$) — два шага склеиваются в один. Соответствует аксиоме 4 : $\Box p \rightarrow \Box \Box p$.
- *Евклидовость* ($R(w, u) \wedge R(w, v) \rightarrow R(u, v)$) — если из w видны u и v , то u и v видят друг друга. Соответствует аксиоме 5 : $\Diamond p \rightarrow \Box \Diamond p$.

Доказательство

Докажем соответствие в обе стороны для рефлексивности с аксиомой T и для симметричности с аксиомой B , остальные аксиомы проверяются аналогичной конструкцией.

Аксиома T . Покажем сначала, что из рефлексивности R следует схема T . Пусть R рефлексивно, и в модели на F истинно $\Box p$ в мире w . По рефлексивности выполнено $R(w, w)$, поэтому из истинности $\Box p$ в w следует истинность p в w , и схема $\Box p \rightarrow p$ истинна.

Обратно, пусть R не рефлексивно: есть мир w с $\neg R(w, w)$. Возьмём оценку $V(p) = W \setminus \{w\}$, при которой атом p истинен всюду, кроме w . Тогда $M, w \models \Box p$: все достижимые миры удовлетворяют p (их может не быть вовсе). Но $M, w \not\models p$, так что схема T ложна.

Аксиома B . Покажем, что из симметричности R следует схема $p \rightarrow \Box \Diamond p$. Пусть R симметрично, и в мире w модели на F истинно p . Для любого мира v с $R(w, v)$ симметричность даёт $R(v, w)$, поэтому в v истинно $\Diamond p$: среди достижимых из v миров есть w . Значит, $\Box \Diamond p$ истинно в w , и схема B истинна в w .

Обратно, пусть R не симметрично: есть миры w, v с $R(w, v)$ и $\neg R(v, w)$. Возьмём оценку $V(p) = \{w\}$. Тогда $M, w \models p$. Но $M, w \not\models \Box \Diamond p$: мир v достижим из w , а $\Diamond p$ в v ложна, потому что единственный мир с истинным p — это w , и $\neg R(v, w)$.

Пример Рефлексивность и аксиома T

Фрейм с мирами w_0, w_1 и рефлексивной достижимостью: $R = \{(w_0, w_0), (w_1, w_1)\}$. Проверим аксиому T — схему $\Box p \rightarrow p$ — в мире w_0 .

При оценке $V(p) = \{w_1\}$ формула $\Box p$ в w_0 ложна: достижимый из w_0 мир w_0 сам опровергает p . Импликация $\Box p \rightarrow p$ истинна: посылка ложна. При оценке $V(p) = \{w_0, w_1\}$ формула $\Box p$ в w_0 истинна, и p истинна там же — импликация снова истинна.

Рефлексивность работает в общем случае: мир w достижим из себя, и посылка $\Box p$ влечёт p в w . Без петли (w_0, w_0) схема опровергается: при оценке $V(p) = \{w_1\}$ посылка $\Box p$ в w_0 истинна (единственный достижимый мир w_1 выполняет p), а заключение p ложно.

Пример Транзитивность и аксиома 4

Фрейм с мирами w_0, w_1, w_2 и транзитивным отношением: $R = \{(w_0, w_1), (w_1, w_2), (w_0, w_2)\}$. Оценка $V(p) = \{w_1, w_2\}$: атом p ложен только в w_0 . В w_0 формула $\Box p$ истинна: из w_0 видны w_1 и w_2 , в обоих p выполнен. И $\Box\Box p$ истинна в w_0 : из w_1 виден w_2 с p , из w_2 не видно ничего, и $\Box p$ там истинна вакуумно. Схема $\Box p \rightarrow \Box\Box p$ в w_0 выполнена.

Уберём переход $w_0 \rightarrow w_2$: отношение $R = \{(w_0, w_1), (w_1, w_2)\}$ больше не транзитивно. Та же оценка $V(p) = \{w_1, w_2\}$. В w_0 формула $\Box p$ по-прежнему истинна: из w_0 виден только w_1 , и в нём p выполнен. Но $\Box\Box p$ в w_0 ложна: из w_1 виден w_2 , где p не выполнен, значит, $\Box p$ в w_1 ложна. Схема аксиомы 4 опровергнута ровно там, где два шага не склеились в один.

Каждая строка теоремы — это уже отдельная система: K плюс аксиома D — система D (логика серийных фреймов), K плюс T — система T (логика рефлексивных фреймов), а добавив к T аксиому B , 4 или 5, получаем системы B , S_4 и S_5 . Так системы собираются из аксиом как из кубиков, и следующий раздел складывает эти кубики в одну картину.

Проверка Свойства отношения и аксиомы

1. Сформулируйте соответствие: какое свойство отношения достижимости выражает аксиома B и почему это не рефлексивность?
2. Почему из одного только соответствия аксиомы свойствам не следует, что система с большим числом аксиом всегда сильнее? Приведите пример свойства, которое влечёт другое, но не выражается более слабой аксиомой.
3. В доказательстве соответствия для B использовалась оценка $V(p) = \{w\}$. Почему она работает и что показал бы переход $R(v, w)$, если бы он был?

§ 33.3 Модальные системы и куб модальностей

Модальная система — это логика, заданная аксиомами: берём K и добавляем схемы. Чем больше аксиом, тем сильнее логика — тем больше формул в ней выводимо. Систем бесконечно много: аксиом можно добавлять самыми разными способами, и почти каждая добавка даёт новую логику. Ниже — избранные системы, которые заслужили собственные имена, но существуют и другие, и часть из них мы увидим на кубе.

Система	Добавка к K	Класс фреймов	Типичное чтение
K	—	все фреймы	минимальная логика
D	аксиома D	серийные	обязанность, убеждение
T	аксиома T	рефлексивные	необходимость, знание
B	$T + B$	рефлексивные и симметричные	редко сама по себе
S_4	$T + 4$	рефлексивные и транзитивные (предпорядки)	знание, время
S_5	$T + 5$	отношения эквивалентности	полное знание

S_5 допускает несколько равносильных заданий. Это $K + T + 5$, $K + T + B + 4$, $K + T + B + 4 + 5$ — все они дают одну и ту же логику.

Разберём каждую систему отдельно: что она добавляет к K , каким фреймам соответствует и где её читают.

33.3.1 Система K : минимум

K — наименьшая нормальная логика: она есть в любой другой, и её аксиома — это просто правило переноса необходимости через импликацию. Аксиома K называется *аксиомой распределения*: $\Box$ распределяется по импликации. Без неё необходимость вела бы себя как произвольный одноместный оператор без внутренней логики.

Примечание Полнота K

Формула валидна во всех фреймах тогда и только тогда, когда она выводима в K . Это классическая теорема полноты модальной логики. Тот же метод даёт полноту именованных систем относительно своих классов фреймов. T полна относительно рефлексивных фреймов, S_4 — относительно предпорядков, S_5 — относительно отношений эквивалентности.

33.3.2 Система D : серийность

Добавив к K аксиому D ($\Box p \rightarrow \Diamond p$), получаем логику серийных фреймов: у каждого мира есть хотя бы один достижимый. Это устраняет вакуумную истину $\Box$: если что-то необходимо, то найдётся мир, где оно истинно. Деонтическое чтение: $Op \rightarrow Pr$ — «если p обязательно, то p разрешено». Обязанность без разрешения — деспотия.

33.3.3 Система T : рефлексивность

T — это K плюс аксиома T ($\Box p \rightarrow p$): необходимое истинно. Ей соответствуют рефлексивные фреймы: каждый мир виден из самого себя. Алетическое чтение: если p необходимо, то p просто истинно — « $2 + 2 = 4$ » необходимо, и $2 + 2 = 4$. Эпистемическое чтение: если агент знает, что p , то p истинно — знание не бывает ложным.

33.3.4 Система B : симметричность

B (в честь Брауэра) — это T плюс аксиома B ($p \rightarrow \Box \Diamond p$): если p истинно, то необходимо, что p возможно. Ей соответствуют рефлексивные симметричные фреймы. Сама по себе система B встречается редко, но это важная ступень: вместе с транзитивностью она даёт S_5 .

33.3.5 Система S_4 : транзитивность

S_4 — это T плюс аксиома 4 ($\Box p \rightarrow \Box \Box p$), и ей соответствуют рефлексивные транзитивные фреймы, то есть *предпорядки*. Эпистемическое чтение: $Kp \rightarrow KKp$ — *позитивная интроспекция*: «если знаю, то знаю, что знаю». Временное чтение: «всегда всегда p » — то же, что «всегда p »: цепочка времени не уводит в сторону.

33.3.6 Система S_5 : евклидовость

S_5 — вершина куба: T плюс аксиома 5 ($\Diamond p \rightarrow \Box \Diamond p$). Название — по аналогии с аксиомой Евклида «равные третьему равны между собой»: если из w видны u и v , то u и v видят друг друга. Для отношения эквивалентности евклидовость выполняется автоматически. Равносильное задание — $T + B + 4$, и ей соответствуют ровно отношения эквивалентности: мир раскладывается на классы эквивалентности, и $\Box$ с $\Diamond$ становятся кванторами по одному классу. Эпистемическое чтение добавляет *негативную интроспекцию*: $\neg Kp \rightarrow K\neg Kp$ — «если не знаю, то знаю, что не знаю». S_5 — стандартная логика знания: агент полностью прозрачен для самого себя.

Включения между системами образуют картину, которую называют *кубом модальностей*. Он показан на рис. 125.

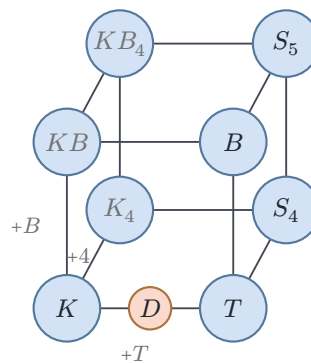


Рис. 125. Куб модальностей: системы, порождённые аксиомами T , B , 4.

Читается куб так: каждая вершина — система K плюс своё подмножество аксиом $\{T, B, 4\}$. Три оси — три аксиомы, поэтому вершин $2^3 = 8$. Ребро вверх — добавление одной аксиомы, так что все включения на кубе — настоящие и соседние системы отличаются ровно одной аксиомой.

Известные системы занимают четыре вершины: T , B (это $T + B$), S_4 (это $T + 4$) и S_5 (это $T + B + 4$ — вспомните равносильное задание выше). Серые вершины KB , K_4 , KB_4 — такие же нормальные логики, просто без привычных имён. Точка D стоит на ребре $K \rightarrow T$: аксиома D выводима из T (рефлексивность влечёт серийность), поэтому $D \subset T$, а сама D — не ось куба.

Куб — карта известных систем, а не их полный перечень. Аксиомы T , B , 4 — не единственные «переключатели»: аксиома 5 дала бы четвёртую ось и куб размерности 4 с $2^4 = 16$ вершинами. Систем и вовсе бесконечно много: между S_4 и S_5 лежат промежуточные логики $S_{4.2}$, $S_{4.3}$ и их обобщения.

Замечание Всё ли описывается фреймами

Соответствие «аксиома — свойство отношения» работает не для всякой аксиомы. Есть нормальные логики, которым не соответствует никакой класс фреймов (их называют *неканоническими*), — но их построение требует аппарата далеко за пределами этого курса. Кроме того, бывают *не-нормальные* логики — без правила необходимости или без аксиомы K . Таковы, например, ранние системы Льюиса S_1 , S_2 , S_3 . Большинство логик, которые встречаются на практике, — нормальные и фреймовые.

33.3.7 Как выбрать систему

Выбор системы — это выбор аксиом, а значит — выбор свойств достижимости и чтения операторов. Алгоритм выбора прост:

1. Решите, что означают $\Box$, $\Diamond$ в вашей задаче: необходимость, знание, убеждение, обязанность, время.
2. Сформулируйте принципы, которые для этого чтения обязаны выполняться.
3. Соберите систему из соответствующих аксиом.

Примеры готовых сборок:

- *Знание*: истинность (T) и позитивная интроспекция (4) дают S_4 . Полное знание (5) — S_5 .
- *Убеждение*: не обязано быть истинным, но непротиворечиво (D), с интроспекциями (4, 5) — логика $KD45$.
- *Обязанность*: D (обязан — значит разрешено), без T : обязанность не гарантирует факт.
- *Время*: транзитивность (4) как минимум. Линейность или ветвление будущего — темпоральные логики ниже.

Чем больше аксиом, тем сильнее система: больше выводимых формул, но меньше моделей. Выбор — это компромисс между силой логики и адекватностью чтению.

Проверка Выбор модальной системы

1. Почему в кубе модальностей аксиома D стоит на ребре $K \rightarrow T$, а не образует отдельную вершину? Что это говорит о соотношении D и T ?
2. Для чтения «обязанность» нужна аксиома D , но не T . Объясните, почему: какое свойство обязанности выражает D и почему T было бы ошибочно?
3. В чём разница между знанием в S_4 и в S_5 и какая интроспекция добавляется на пути от S_4 к S_5 ?

§ 33.4 Задачи модальной логики

У каждой логики есть свои задачи, вокруг которых «всё крутится»: в пропозициональной это SAT (глава 14), в логике первого порядка — выполнимость и общезначимость (глава 3). В модальной логике задач три:

- *Выполнимость в модели*: есть ли мир w , где $M, w \models A$? В конечной модели это решается обходом за полиномиальное время — этим занимается model checking (ниже).
- *Валидность во фрейме*: выполняется ли A во всех моделях на фрейме F ?
- *Общезначимость в системе*: валидна ли A во всех фреймах класса, заданного системой (например, во всех рефлексивных)?

Сложность общезначимости и выполнимости — посередине между пропозициональной и первопорядковой: для K, T, S_4, S_5 задача выполнимости PSPACE-полна²³⁵. Это сложнее NP-полного SAT, но всё ещё разрешимо, тогда как логика первого порядка неразрешима вообще.

Канонический PSPACE-полный представитель — задача QBF об истинности булевых формул с чередующимися кванторами, разобранный в главе 30. Родство прямое: $\Box$ и $\Diamond$ — кванторы по достижимым мирам, и вложенные модальности порождают чередование кванторов.

Алгоритмы для модальной логики:

- *Табличный метод (tableau)*: пытаемся построить мир, где формула истинна, применяя правила к каждой подформуле. Модальные правила создают новые миры: $\Diamond A$ требует свежий достижимый мир с A . $\Box A$ требует A во всех уже известных достижимых мирах. Для нормальных логик достаточно конечных моделей (свойство конечной модели), поэтому поиск всегда завершается.
- *Перевод в логику первого порядка*: семантика Крипке записывается формулами первого порядка (мир — элемент, R — отношение, $\Box$ — квантор по достижимым).

²³⁵R. E. Ladner. «The Computational Complexity of Provability in Systems of Modal Propositional Logic». SIAM Journal on Computing, 1977.

Для K перевод сохраняет выполнимость, и солвер может перепоручить задачу первопорядковому решателю.

- *Символьные методы* (BDD) — для темпорального model checking, о них ниже.

Пример Модальная формула как формула первого порядка

Формула $\Box\Diamond p$ в мире w переводится как

$$\forall u : R(w, u) \rightarrow \exists v : (R(u, v) \wedge p(v)).$$

Читается: из любого мира, достижимого из w , достижим мир, где p истинно. Свободная переменная w — тот мир, в котором формулу проверяют. Символы R и p становятся обычными предикатами, и задача о модальной формуле становится задачей первого порядка.

Солверы и инструменты:

- Табличные и переводящие: LoTREC, Spartacus (перевод в первый порядок + SPASS), KSP, InKreSAT, MleanCoP.
- Model checkers: NuSMV, SPIN — про них в конце главы.
- Доказательные ассистенты: модальные логики формализованы в Coq и Lean.

Приложения модальной логики выходят далеко за пределы model checking:

- *Эпистемическая логика*: знание и убеждение в мультиагентных системах. Логика BAN²³⁶ анализирует протоколы аутентификации: что именно знает каждый участник после обмена сообщениями.
- *Деонтическая логика*: нормы, обязанности и разрешения в праве и контрактах.
- *Логика доказуемости (GL)*: $\Box p$ читается «формула p доказуема в арифметике». Аксиома Лёба формализует вторую теорему Гёделя о неполноте²³⁷.
- *Темпоральные базы данных*: стандарт SQL:2011²³⁸ хранит периоды времени, а запросы «когда было истинно» — это темпоральные операторы.

§ 33.5 Знание, обязанность, убеждение

Пара $\Box, \Diamond$ — не только «необходимо» и «возможно»: достаточно переопределить, что считать «мирами» и «достижимостью», и та же рамка получает новое содержательное чтение. Схоластика различала необходимые и случайные истины, Кант — аналитические и синтетические. Модальная логика делает такие различия формальными.

²³⁶M. Burrows, M. Abadi, R. Needham. «A Logic of Authentication». ACM Transactions on Computer Systems, 1990.

²³⁷M. H. Löb. «Solution of a Problem of Leon Henkin». Journal of Symbolic Logic, 1955; K. Gödel. «Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme». Monatshefte für Mathematik und Physik, 1931.

²³⁸ISO/IEC 9075:2011. «Information technology — Database languages — SQL». 2011.

Аксиомы предыдущего раздела — точный логический отпечаток каждого чтения:

- **Алетические модальности** — необходимость и возможность. Философский источник — Лейбниц: необходимо истинно то, что верно во всех *возможных мирах*. Аксиома T обязательна: необходимое истинно.
- **Эпистемические модальности** — знание: Kp читается «агент знает, что p ». Аксиома T ($Kp \rightarrow p$) — знание истинно. Этим знание отличается от убеждения. Аксиома 4 ($KKp \rightarrow Kp$) — *позитивная интроспекция*: «если знаю, то знаю, что знаю». Аксиома 5 ($\neg Kp \rightarrow K\neg Kp$) — *негативная*: «если не знаю, то знаю, что не знаю». Обе описывают рационального агента.
- **Деонтические модальности** — обязанность и разрешение: Op (« p обязательно»). Pp — « p разрешено». Операторы двойственны, как $\Box$ и $\Diamond$: $Op \equiv \neg P\neg p$ — «обязанность» значит «не разрешено обратное». Вместо T действует D : $Op \rightarrow Pp$ (если обязан, то разрешено). Но $Op \rightarrow p$ неверно: обязанность не гарантирует факт.
- **Доксастические модальности** — убеждение: Bp («агент полагает, что p »). Выполняется D , но не T : убеждения могут быть ложными.

Пример Чтения в деле

Формула $\Box p \rightarrow p$ истинна при чтении «знание»: если агент знает, что идёт дождь, то дождь идёт. При чтении «обязанность» та же формула абсурдна: из «водитель обязан остановиться» не следует, что машина стоит. При чтении «убеждение» она тоже запрещена: можно искренне верить в ложное.

Темпоральная логика — ещё одно чтение той же пары: $\Box$ — «всегда», $\Diamond$ — «когда-нибудь», а «миры» — моменты времени. Именно это чтение доведено до инженерного инструмента — model checking — и разбирается в следующих разделах.

Проверка Знание, обязанность, убеждение

1. Одна и та же формула $\Box p \rightarrow p$ истинна при чтении «знание», но абсурдна при «обязанность». Какую аксиому она выражает в каждом чтении и почему выбор аксиомы меняет её смысл?
2. Чем знание отличается от убеждения формально? Какая аксиома запрещает убеждению быть истинным?
3. Почему для обязанности вместо T действует D , и что случилось бы с системой, если бы мы ошибочно добавили T к деонтическому чтению?

§ 33.6 Время как модальность

Для реактивных систем «мир» — это *состояние* программы или аппаратуры, а «достижимость» — один шаг выполнения. Модель Крипке — это модальная модель, в которой вместо оценки V на мирах используется разметка состояний. Это одно и то же, записанное по-разному: $V(p) = \{s \mid p \in L(s)\}$.

ОПРЕДЕЛЕНИЕ 33.4 Модель Крипке

Модель Крипке — тройка $M = (S, R, L)$, где:

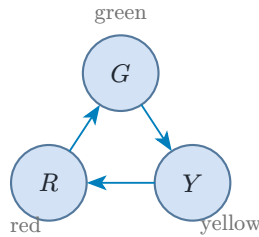
- S — конечное множество **состояний**.
- $R \subseteq S \times S$ — отношение **переходов** (обычно серийное: у каждого состояния есть хотя бы один следующий шаг).
- $L : S \rightarrow \mathcal{P}(\text{AP})$ — **разметка**: каждому состоянию приписано множество атомарных утверждений (тот самый набор AP), истинных в нём.

- Состояние — снимок системы (значения всех переменных).
- Переход — один шаг выполнения.
- Разметка — что истинно в этом состоянии.

Путь (path) — бесконечная последовательность состояний $\pi = s_0, s_1, s_2, \dots$, в которой каждый шаг подчинён условию $(s_i, s_{i+1}) \in R$ для всех i , то есть движение по переходам идёт без остановки.

Пример Модель Крипке: светофор

Состояния: G (зелёный), Y (жёлтый), R (красный). Переходы: $G \rightarrow Y \rightarrow R \rightarrow G$ (цикл длины 3).



Атомарные утверждения: green, yellow, red. Разметка: $L(G) = \{\text{green}\}$, $L(Y) = \{\text{yellow}\}$, $L(R) = \{\text{red}\}$. Свойство «после зелёного рано или поздно красный» записывается формулой $\Box(\text{green} \rightarrow \Diamond \text{red})$ — «всегда, если зелёный, то когда-нибудь красный».

33.6.1 LTL: линейная темпоральная логика

LTL (Linear Temporal Logic) рассматривает *один путь* за раз: формула истинна или ложна на бесконечной последовательности состояний.

ОПРЕДЕЛЕНИЕ 33.5 Синтаксис LTL

Формулы LTL:

- Атомарное утверждение $p \in \text{AP}$ — формула.
- Если φ, ψ — формулы, то $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi$ — формулы.
- Если φ, ψ — формулы, то $\bigcirc \varphi$ — формула (« φ в следующем состоянии»).
- Если φ, ψ — формулы, то $\varphi U \psi$ — формула (« φ до тех пор, пока ψ »).

Оператор U (until, «до тех пор, пока») — основной темпоральный оператор: $\varphi U \psi$ — «сначала φ , и в какой-то момент наступает ψ ». Остальные — производные:

$$\Diamond \varphi = \text{true } U \varphi,$$

$$\Box \varphi = \neg \Diamond \neg \varphi.$$

$\Diamond \varphi$ — «когда-нибудь φ » (в будущем, включая сейчас), а $\Box \varphi$ — «всегда φ » (во всех состояниях пути).

ОПРЕДЕЛЕНИЕ 33.6 Семантика LTL

Пусть $\pi = s_0, s_1, s_2, \dots$ — путь, а π^i — его хвост с s_i . Запись $\pi \models \varphi$ означает «формула φ истинна на пути π ». Истинность задаётся индукцией по структуре формулы:

$$\pi \models p \leftrightarrow p \in L(s_0),$$

$$\pi \models \neg \varphi \leftrightarrow \pi \not\models \varphi,$$

$$\pi \models \varphi \wedge \psi \leftrightarrow \pi \models \varphi \wedge \pi \models \psi,$$

$$\pi \models \varphi \vee \psi \leftrightarrow \pi \models \varphi \vee \pi \models \psi,$$

$$\pi \models \bigcirc \varphi \leftrightarrow \pi^1 \models \varphi,$$

$$\pi \models \varphi U \psi \leftrightarrow \exists j : (\pi^j \models \psi \wedge \forall i < j : \pi^i \models \varphi).$$

Пример Свойства в LTL

Светофор: $\Box(\text{green} \rightarrow \Diamond \text{red})$ — «после зелёного когда-нибудь красный». Последовательность фаз: $\Box(\text{green} \rightarrow \bigcirc \text{yellow})$ — «после зелёного сразу жёлтый» (на каждом пути светофора это верно). Взаимная блокировка: $\Box \neg(\text{deadlock}_1 \wedge \text{deadlock}_2)$ — «никогда не будет блокировки». Ответ на запрос: $\Box(\text{request} \rightarrow \Diamond \text{response})$ — «каждый запрос получает ответ».

Пример Проверка LTL-формулы на пути

Путь светофора: $\pi = G, Y, R, G, Y, R, \dots$ — бесконечный цикл фаз. Проверим формулу $\Box(\text{green} \rightarrow \bigcirc \text{yellow})$. В состоянии G следующий шаг — Y , импликация истинна. В состояниях Y и R посылка ложна, импликация истинна вакуумно. Формула истинна на всём пути.

Проверим формулу $\Box(\text{green} \rightarrow \Diamond \text{red})$. Из G через два шага наступает R , импликация истинна. В состояниях Y и R посылка ложна. Формула тоже истинна на пути.

А формула $\Box \bigcirc \text{red}$ ложна: в состоянии G следующий шаг — Y , а не R .

Полезны две эквивалентности-развёртки (они же — неподвижные точки):

$$\Box\varphi = \varphi \wedge \bigcirc\Box\varphi,$$

$$\Diamond\varphi = \varphi \vee \bigcirc\Diamond\varphi.$$

«Всегда» значит «сейчас и всегда в следующем состоянии», а «когда-нибудь» — «сейчас или когда-нибудь в следующем».

Для model checking модель снабжается начальным состоянием s_0 . LTL-формула выполняется на M , если она выполняется на *всех* путях из s_0 : $M \models \varphi$.

33.6.2 CTL: темпоральная логика ветвящегося времени

CTL (Computation Tree Logic) рассматривает *все* пути сразу: кванторы пути A («на всех путях») и E («существует путь») предшествуют темпоральному оператору.

ОПРЕДЕЛЕНИЕ 33.7 Синтаксис CTL

Формулы CTL:

- Атомарное утверждение — формула.
- Булевы комбинации формул — формулы.
- Если φ, ψ — формулы, то $A \bigcirc \varphi, E \bigcirc \varphi, A(\varphi U \psi), E(\varphi U \psi)$ — формулы.

Производные операторы:

- $A \Diamond \varphi \equiv A(\text{true } U \varphi)$ («на всех путях когда-нибудь»).
- $E \Diamond \varphi \equiv E(\text{true } U \varphi)$ («существует путь, где когда-нибудь»).
- $A \Box \varphi \equiv \neg E \Diamond \neg \varphi$.
- $E \Box \varphi \equiv \neg A \Diamond \neg \varphi$.

ОПРЕДЕЛЕНИЕ 33.8 Семантика CTL

Формула CTL оценивается в *состоянии* модели, и пути входят в оценку только через кванторы A и E . Отношение $M, s \models \varphi$ определяется индукцией по формуле. Клаузы для $\bigcirc$:

$$M, s \models A \bigcirc \varphi \leftrightarrow \forall s' \in R(s) : M, s' \models \varphi,$$

$$M, s \models E \bigcirc \varphi \leftrightarrow \exists s' \in R(s) : M, s' \models \varphi.$$

Для U :

- $M, s \models A(\varphi U \psi)$ — на *всех* путях, начинающихся в s , выполняется $\varphi U \psi$ (сначала φ , затем ψ).
- $M, s \models E(\varphi U \psi)$ — существует путь из s , на котором выполняется $\varphi U \psi$.

A читается «во всех возможных будущих», E — «в некотором возможном будущем». Например, $A \bigcirc \text{resp}$ — «в любом следующем состоянии — ответ», а $E \bigcirc \text{resp}$ — «есть хотя бы один следующий шаг с ответом». Так же читаются пары с $\Box$: $A \Box \varphi$ — «на всех путях из состояния φ всегда истинна», $E \Box \varphi$ — «существует путь, на котором φ всегда истинна».

Различие между A и E : $A\Diamond \text{ resp } E\Diamond$ — «ответ гарантирован при любом ходе системы», а $E\Diamond \text{ resp } A\Diamond$ — «существует ход, приводящий к ответу». Первую формулу проверяет model checking как свойство системы, вторую — как возможность.

Пример LTL против CTL

Свойство «всегда возможно перезапуск» — $A\Box E\Diamond \text{ restart}$: в каждом состоянии любого пути существует продолжение, ведущее к перезапуску. Это типично CTL-свойство: LTL его не выражает, потому что LTL не умеет говорить «существует путь» внутри пути. Обратно, LTL выражает $\Box\Diamond p \rightarrow \Box\Diamond q$ — справедливость (fairness). «Если p встречается бесконечно часто, то и q встречается бесконечно часто» — это CTL не выражает. Выразительные силы LTL и CTL несравнимы: есть свойства, выразимые только в одной из логик.

§ 33.7 Алгоритмы model checking

Проверка, выполняется ли CTL-формула в модели, — это разметка состояний подформулами: от атомарных к составным, снизу вверх. Ключ к операторам — неподвижные точки.

Неподвижная точка — множество X , которое не меняется при некотором преобразовании. Для $E(\varphi U \psi)$ преобразование имеет вид $X \rightarrow \psi \vee (\varphi \wedge E \circ X)$, и наименьшее множество, остающееся на месте, — это в точности $E(\varphi U \psi)$. Для $E\Box\varphi$ берётся, наоборот, наибольшая неподвижная точка.

ОПРЕДЕЛЕНИЕ 33.9 Неподвижные точки CTL

- Для $E(\varphi U \psi)$ — наименьшая неподвижная точка:

$$E(\varphi U \psi) \equiv \psi \vee (\varphi \wedge E \circ E(\varphi U \psi)).$$

- Для $E\Box\varphi$ — наибольшая:

$$E\Box\varphi \equiv \varphi \wedge E \circ E\Box\varphi.$$

- Для $A(\varphi U \psi)$:

$$A(\varphi U \psi) \equiv \psi \vee (\varphi \wedge A \circ A(\varphi U \psi)).$$

Пример Labeling-алгоритм для CTL

Проверим $E(\varphi U \psi)$ в модели $M = (S, R, L)$. Обозначим $\text{pre}_\exists(X) = \{s \in S : \exists s' \in R(s), s' \in X\}$ — состояния, из которых одним шагом можно попасть в X («пре-образ по существованию»). Алгоритм:

1. $Y := \{s : s \models \psi\}$.
2. Повторять, пока не стабилизируется: $Y := Y \cup (\{s : s \models \varphi\} \cap \text{pre}_\exists(Y))$.

Полученное Y — в точности состояния, где истинно $E(\varphi U \psi)$. Для $E\Box\varphi$ — двойственно: $Y := \{s : s \models \varphi\}$. Затем $Y := Y \cap \text{pre}_{\exists}(Y)$ до стабилизации.

Для $A\Box\varphi$ используется пре-образ «по всем преемникам»: $\text{pre}_{\forall}(X) = \{s : \forall s' \in R(s), s' \in X\}$.

Свойства живости ($\Diamond\varphi, \varphi U \psi$) — наименьшие неподвижные точки, а свойства безопасности ($\Box\varphi$) — наибольшие. Различие отражает природу: безопасность опровергается конечным префиксом («вот здесь оба процесса в критической секции»), живость — только бесконечным путём («запрос висит вечно, ответа нет»).

Пример Labeling на светофоре

Проверим в модели светофора формулу $A\Box E\Diamond \text{red}$: «на любом пути всегда есть возможность дойти до красного». Начнём с labeling для $E\Diamond \text{red} = E(\text{true} U \text{red})$ — наименьшей неподвижной точки. Начальное множество — состояния с red, то есть $\{R\}$.

Шаг	Множество Y
0	$\{R\}$
1	$\{Y, R\}$
2	$\{G, Y, R\}$
3	$\{G, Y, R\}$

На первом шаге добавляется Y , из которого за один шаг достигается R , а на втором — G , из которого за один шаг достигается Y . Далее множество не растёт: $E\Diamond \text{red}$ истинна во всех трёх состояниях. Тогда $A\Box E\Diamond \text{red}$ истинна: $E\Diamond \text{red}$ выполняется в каждом состоянии каждого пути. Формула подтверждает свойство светофора — из любого состояния красный достигим.

33.7.1 LTL через автоматы Бюхи

Разметка через неподвижные точки работает для CTL, потому что CTL-формула оценивается в состоянии. LTL-формула оценивается на целом пути, и свести проверку к разметке состояний напрямую не получается. Стандартный путь — три шага:

1. Построить автомат Бюхи $A_{\neg\varphi}$ для отрицания свойства: он принимает ровно те бесконечные пути, на которых $\neg\varphi$ истинна.
2. Построить синхронное произведение $M \times A_{\neg\varphi}$ модели и автомата: состояния — пары из состояния модели и состояния автомата, согласованные по разметке, переходы — одновременный шаг обоих.
3. Искать в произведении путь, бесконечно часто попадающий в принимающие состояния.

Если такого пути нет — свойство выполняется. Если есть — это контрпример: конкретное ошибочное поведение системы. Искомый путь всегда имеет вид лассо

— префикс плюс повторяющийся цикл, — поэтому его поиск сводится к проверке достижимости цикла в принимающей компоненте.

§ 33.8 Model checking на практике

Model checking — метод автоматической верификации конечных систем. Программа или аппаратный модуль моделируется как размеченная система переходов. Это граф, где вершины — состояния, рёбра — переходы, а вершины размечены утверждениями о значениях переменных.

Практическая трудность — взрыв пространства состояний: n булевых переменных дают 2^n состояний. 40 переменных — уже около 10^{12} состояний, а реальные системы имеют тысячи переменных. Современные model checker'ы (NuSMV, SPIN) справляются с 10^{20} и более состояниями.

Способы борьбы с взрывом делятся на два лагеря:

- **Символьные** (NuSMV): множества состояний представляются булевыми формулами в виде BDD (Binary Decision Diagrams) — компактных ациклических графов, кодирующих булевы функции. Операции над множествами (объединение, пересечение, проверка пустоты) выполняются как операции над BDD, а общие подструктуры формул разделяются, поэтому графы остаются маленькими.²³⁹
- **Явные** (SPIN): состояния перебираются по одному, а взрыв обходится двумя приёмами — редукцией частичного порядка (независимые переходы разных процессов можно переставлять, проверяется один представитель каждого класса эквивалентных путей) и битовым хэшированием (пройденные состояния кладутся в битовую таблицу по хэшу, память крошечная, но коллизии могут заставить пропустить состояние).²⁴⁰

Место model checking среди методов верификации видно при сравнении с тройками Хоара из главы 32. Тройки Хоара — логика *состояний*: предусловие, постусловие, инвариант. Модели Крипке — логика *поведения*: последовательности состояний, темпоральные операторы. Первые удобны для функций и циклов, вторые — для реактивных систем: протоколов, аппаратуры, распределённых алгоритмов. Абстрактная интерпретация (глава 31) соединяет оба мира: она вычисляет множества состояний, в которых свойство гарантированно выполняется.

Проверка Model checking на практике

1. Чем тройки Хоара отличаются от моделей Крипке и почему первые удобны для функций, а вторые — для реактивных систем?
2. Объясните взрыв пространства состояний на примере n булевых переменных и назовите два способа борьбы с ним.

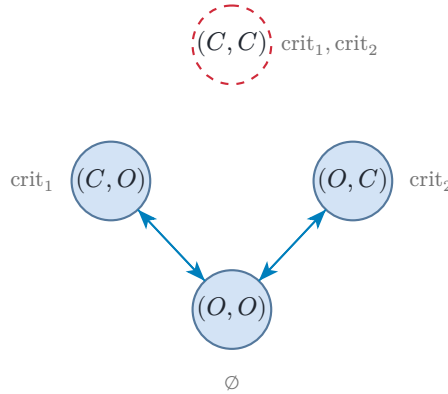
²³⁹Е. М. Clarke, O. Grumberg, D. Peled. «Model Checking». 1999.

²⁴⁰C. Baier, J.-P. Katoen. «Principles of Model Checking». 2008.

3. Как LTL-свойство проверяется через автоматы Бюхи и что даёт синхронное произведение модели с автоматом отрицания?

Пример Взаимное исключение

Два процесса конкурируют за критическую секцию. Состояние — пара положений процессов: O (вне секции) или C (в критической секции). Атомарные утверждения: $\text{crit}_1, \text{crit}_2$ («процесс i в критической секции»).



Требование взаимного исключения — $A \Box \neg(\text{crit}_1 \wedge \text{crit}_2)$: на всех путях никогда не бывает, что оба процесса в критической секции одновременно. Model checker строит пространство состояний из всех комбинаций переменных процессов и проверяет формулу. Состояние (C, C) — то самое, где $\text{crit}_1 \wedge \text{crit}_2$ истинно, и в корректной модели оно недостижимо. Если же проверка находит путь в него, контрпример предъявляется как диагностика гонки.

Пример Безопасность: labeling для $A \Box \neg(\text{crit}_1 \wedge \text{crit}_2)$

Состояния модели взаимного исключения: OO, OC, CO, CC . Первый символ — положение первого процесса, второй — второго: O вне секции, C в критической секции. Переходы: $OO \rightarrow OC, OO \rightarrow CO, OC \rightarrow OO, CO \rightarrow OO$. Labeling для наибольшей неподвижной точки $A \Box \varphi$. Здесь $\varphi = \neg(\text{crit}_1 \wedge \text{crit}_2)$.

1. Начало: $Y = \{OO, OC, CO\}$ — состояния, где φ истинна. Состояние CC в Y не входит: в нём оба процесса в критической секции.
2. Шаг: $Y := Y \cap \text{pre}_\vee(Y)$. Из $\emptyset$ оба перехода ведут в Y . Из OC и CO — по одному переходу, тоже в Y . Состояние CC остаётся вне Y независимо от пре-образа.
3. Повторный шаг ничего не меняет: получена наибольшая неподвижная точка.

Итак, формула $A \Box \neg(\text{crit}_1 \wedge \text{crit}_2)$ истинна в состояниях OO, OC, CO . В начальном состоянии $\emptyset$ взаимное исключение доказано, а состояние CC нарушало бы свойство, но оно недостижимо из $\emptyset$: контрпримера нет.

§ 33.9 Итоги главы

Слово «истинно» прячет два разных смысла — случайный и необходимый. Ответ Крипке — миры и достижимость: $\Box p$ истинно, когда p верно во всех достижимых мирах, а $\Diamond p$ — когда хотя бы в одном. Фрейм задаёт геометрию достижимости, оценка — что истинно в каждом мире, и на этой паре строится вся истинность модальных формул.

Свойства отношения достижимости записываются аксиомами: серийность, рефлексивность, симметричность, транзитивность и евклидовость порождают логики D, T, B, S_4, S_5 . Их включения складываются в куб модальностей с вершиной S_5 — логикой отношений эквивалентности. Выбор системы — это выбор чтения операторов: знание, убеждение, обязанность или время диктуют разные аксиомы.

Вторая половина превратила метафору в инженерный инструмент: миры стали состояниями системы, достижимость — шагом выполнения, а модальности — словами «всегда» и «когда-нибудь». Model checking проверяет такие свойства автоматически: CTL — разметкой через неподвижные точки, LTL — автоматами Бюхи, а взрыв пространства состояний сдерживается символьными представлениями вроде BDD.

Плата за выразительность — сложность: выполнимость модальной логики PSPACE-полна, посередине между SAT и логикой первого порядка. Всё это время истина оставалась классической — два значения, привязанные к миру, — и именно это допущение следующая глава поставит под вопрос.

Проверка Проверьте себя

1. Что такое мир Крипке и отношение достижимости, и как они задают истинность $\Box p$ и $\Diamond p$?
2. Какие свойства отношения достижимости записываются аксиомами T , 4 и B ?
3. Почему S_5 — логика отношений эквивалентности, и что это значит для чтения модальностей?
4. Чем CTL отличается от LTL в model checking, и какими автоматами проверяется каждая?
5. Почему выполнимость модальной логики PSPACE-полна, а не NP-полна, как классический SAT?

Что дальше?

Семейство модальностей шире двух рассмотренных здесь. Динамическая логика программ делает модальностью саму программу: $[\alpha]\varphi$ читается «после выполнения α всегда верно φ », и хоаровские тройки из верификации — её синтаксический частный случай. Эпистемическая логика делает модальностью знание: $K_a\varphi$ — «агент a знает φ », и на ней стоят протоколы с общим знанием в криптографии и распределённых системах. Мю-исчисление добавляет к модальностям наименьшие

и наибольшие неподвижные точки — связку с абстрактной интерпретацией — и выразительно закрывает весь модель-чекинг. Модальная логика описывает миры и достижимость, а темпоральная логика — её частный случай, где «миры» — моменты времени. В главе 34 мы сменим классическую семантику на конструктивную, где истина — предъявленное доказательство.

§ 33.10 Упражнения

Фреймы и системы аксиом.

1. Постройте фрейм из трёх миров, на котором $\Box p \rightarrow p$ (аксиома T) не валидна. Какое свойство отношения R нарушено?
2. Какое свойство отношения соответствует аксиоме D : $\Box p \rightarrow \Diamond p$? Почему серийность устраняет вакуумную истину $\Box$?
3. Выпишите все семь включений между системами K, D, T, B, S_4, S_5 . Подсказка: две цепочки сходятся в S_5 .
4. Какая из систем D, B, S_4, S_5 — логика отношений эквивалентности, а какая — логика частичных предпорядков?
5. * Назовите все восемь вершин куба с осями T, B . 4. Какие из них вам уже знакомы, а какие — нет?
6. * Аксиома B ($p \rightarrow \Box \Diamond p$) валидна на симметричных фреймах. Постройте фрейм, где она не валидна, и объясните, какое свойство отношения нарушено.

Темпоральные логики.

7. Постройте модель Крипке для светофора с тремя состояниями. Запишите в LTL: «после зелёного когда-нибудь красный».
8. Запишите в LTL свойство «запрос всегда рано или поздно получает ответ». В чём отличие $\Box(\text{request} \rightarrow \Diamond \text{response})$ от $\Box \Diamond \text{response}$?
9. * Чем CTL-формула $A \Box E \Diamond \text{restart}$ отличается от любой LTL-формулы? Почему LTL не может выразить «существует путь»?

Алгоритмы и сложность.

10. Проверьте labeling-алгоритмом, где истинно $E(\text{yellow } U \text{ red})$ в светофоре.
11. * Почему выполнимость в модальной логике сложнее пропозициональной (NP-полной), но проще первопорядковой (неразрешимой)?
12. * Почему свойства живости — наименьшие неподвижные точки, а безопасности — наибольшие? Свяжите с тем, чем опровергается каждое из них.
13. * Опишите три шага model checking LTL через автоматы Бюхи. Что означает путь в синхронном произведении?

14. * Проверьте табличным методом (tableau) для системы K : формула $\Box(p \wedge q) \rightarrow (\Box p \wedge \Box q)$ общезначима, а множество формул $\Diamond p$ и $\Box \neg p$ невыполнимо. Для первого постройте таблицу для отрицания формулы и покажите, что все ветви замкнуты. Для второго постройте таблицу и покажите, что единственная ветвь замкнута. Для каждого шага указывайте применяемое правило: пропозициональные правила, двойственность $\neg \Box A \equiv \Diamond \neg A$, правило для $\Diamond$ (новый мир) и правило для $\Box$ (наследование в существующие миры).

ПРОЕКТ Проверка свойств безопасности и живости

Постройте STL-чекер и проверьте на нём свойства безопасности и живости трёх систем. Model checking отвечает на вопрос, существует ли путь, где система входит в нежелательное состояние: оба процесса в критической секции, запрос без ответа, вечное ожидание. Свойства безопасности опровергаются конечным путём (оба процесса внутри), свойства живости — только бесконечным (запрос висит вечно), и эта разница — суть, а не деталь: перебор конечных путей вечный голод никогда не обнаружит, тогда как проверка пустоты производства его находит. Соберите инструмент из разметки через неподвижные точки и проверки пустоты производства.

① Неподвижные точки

Реализуйте разметку состояний через неподвижные точки для STL. Для $E(\varphi U \psi)$ возьмите наименьшую неподвижную точку. Для $E\Box\varphi$ — наибольшую. Добавьте пре-образы $\text{pre}_\exists$, $\text{pre}_\forall$.

② Трассы из неподвижных точек

Вердикт «ложно» — половина ответа: чекер должен предъявить трассу, демонстрирующую нарушение. Извлеките контрпример из слоёв итерации неподвижных точек. Для $A\Box p$ (на всех путях всегда) — кратчайший путь к состоянию без p . Для $A\bigcirc p$ (на всех путях на следующем шаге) — один шаг к нарушению. Для $A\Diamond p$ (на всех путях когда-нибудь) — лассо, избегающее p навсегда. Механизм: наименьшая неподвижная точка строится слоями — номер слоя, на котором состояние вошло в множество, это его ранг, и кратчайший путь к цели извлекается спуском по рангам. В наибольшей неподвижной точке лассо ищется как цикл внутри итогового множества: у каждого состояния этого множества есть преемник внутри него, а конечность множества даёт повтор — префикс плюс цикл. Для $A(\varphi U \psi)$ нарушение — либо путь, где φ перестаёт выполняться раньше ψ , либо лассо без ψ . Для операторов с экзистенциальной семантикой извлекайте свидетелей: для $E(\varphi U \psi)$ — путь к ψ через φ , для $E\Box\varphi$ — лассо внутри φ . Проверьте трассы на системах главы. Безопасность $A\Box\neg(\text{crit}_1 \wedge \text{crit}_2)$ на ошибочном взаимном исключении даёт конечный путь в критическую секцию. Живость $A\Diamond \text{response}$ при голоде — вечное лассо без ответа.

③ Системы

Соберите три системы: светофор, взаимное исключение двух процессов и «запрос/ответ» с вечно голодным запросом.

④ *Ключевой эксперимент*

Для свойства живости постройте автомат Бюхи для отрицания вручную. Разберите случаи $\Box p$, $\Diamond p$. Покажите, что контрпример — бесконечное лассо в произведении. Конечный перебор его не видит, тогда как проверка пустоты его видит.

⑤ *Два случая*

Разделите два случая на одной системе: свойство безопасности, где контрпример — конечный путь, и свойство живости, где он бесконечен. Предъявите оба контрпримера как трассы.

⑥ *Ловушка записи*

Покажите, как легко обмануться в записи свойства. Сравните $\Box(p \rightarrow \Diamond q)$, $\Box \Diamond q$ на одной системе. Объясните на контрпримере, где ваша формула соврала о вашем намерении.

Итог: работающий STL-чекер с журналом «система $\times$ свойство $\rightarrow$ выполняется или контрпример (конечный путь или бесконечное лассо)», где видно, почему безопасность обнаруживается конечным перебором, а живость — нет.

XXXIV

Интуиционистская логика

Существуют ли иррациональные числа a, b , для которых a^b рационально? Классическое доказательство занимает две строки. Рассмотрим число $\sqrt{2}^{\sqrt{2}}$. Если оно рационально, нужная пара найдена: $a = b = \sqrt{2}$. Если иррационально, положим

$$a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}.$$

Тогда $a^b = 2$. В обоих случаях ответ утвердительный: такие пары существуют.

Но, закрыв доказательство, нельзя назвать ни одной такой пары. Рассуждение разобрало два случая и не сообщило, какой из них имеет место: существование гарантировано, а объект не предъявлен. Тот же приём — «либо p , либо не p , и в обоих случаях всё сходится» — опирается на закон исключённого третьего из главы 1. Загадка не декоративная: долгое время никто не знал, рационально ли само $\sqrt{2}^{\sqrt{2}}$. Лишь теорема Гельфонда–Шнайдера показала, что оно трансцендентно.²⁴¹

Для Брауэра такое рассуждение — не доказательство. Математика для него — конструктивная деятельность ума: утверждение истинно, когда предъявлено его доказательство, а доказательство существования обязано построить объект. «Существовать» в математике значит «быть построенным». За этим стоит один из самых острых споров в истории математики — спор об основаниях: когда в начале XX века в теории множеств обнаружили парадоксы, пришлось решать, на чём вообще стоит математика.

Классическая логика из главы 1 покоится на двух допущениях: законе исключённого третьего ($p \vee \neg p$) и снятии двойного отрицания. Снятие двойного отрицания — правило, по которому из $\neg\neg p$ следует p . Оба кажутся естественными, но ни один не выводится из более простых принципов. Это выбор: логической необходимости здесь нет. *Интуиционизм* — это отказ от обоих допущений, и отказ меняет само понятие истины. Классика понимает истину как соответствие реальности, интуиционизм — как предъявленное доказательство.

В главе 33 семантика уже менялась: истина стала относительной — истинно в одном мире, ложно в другом. Здесь сдвиг глубже: меняется то, что для формулы значит быть истинной. Семантика Крипке вернётся и сюда, но в другом чтении: миры станут *состояниями знания*, а достижимость — расширением знания. Отрицание

²⁴¹А. О. Гельфонд. «О седьмой проблеме Гильберта». Доклады АН СССР, 1934; T. Schneider. «Transzendenzuntersuchungen periodischer Funktionen». 1934.

смотрит вперёд: $\neg p$ истинно сейчас, если никакое будущее состояние знания не подтвердит p .

Философский спор обернулся инженерным открытием. Соответствие Карри–Ховарда (глава 28) показало: интуиционистское доказательство — это программа, а формула — тип. Доказательство импликации — функция, доказательство конъюнкции — пара, доказательство дизъюнкции — размеченный выбор: ровно те конструкции, из которых строятся программы. Снятие двойного отрицания, неприемлемое для интуициониста, соответствует оператору `call/cc` — управлению продолжениями, нелокальному выходу, исключениям.

Цена такого чтения — потеря классических доказательств существования, которые не предъявляют объекта. Награда — извлечение алгоритмов: если доказано $A \vee B$, то доказана одна из сторон. Если доказано существование, предъявлен свидетель. Классическая логика говорит «что-то существует», интуиционистская — «вот оно». На этой основе работают доказательные ассистенты: `Coq` и `Lean` проверяют доказательства как типизированные программы, и ошибка в рассуждении становится ошибкой типизации, которую машина отвергает.

Всё начинается с вопроса, который кажется наивным: что значит для утверждения быть истинным? Какую логику получит тот, кто откажется от закона исключённого третьего, — и что он вместе с ней потеряет?

ИСТОРИЯ Спор об основаниях

Интуиционизм был самым радикальным из ответов на кризис оснований начала XX века. Брауэр защитил в 1907 году диссертацию, название которой обещало опровержение аристотелевского закона исключённого третьего. В 1908 году он напечатал статью «О ненадёжности логических принципов»²⁴², где закон исключённого третьего объявлен надёжным только в конечном: перебор конечной совокупности решает вопрос всегда, за её пределами гарантий нет. К тому времени Брауэр уже прославился классическими результатами в топологии: теорема о неподвижной точке и инвариантность размерности (1911) принадлежат ему. Собственную программу он называл интуиционизмом: математика — деятельность мышления, и рассуждения о бесконечных совокупностях должны перестраиваться под построения.

Формализация заняла два десятилетия и прошла через трёх математиков. В 1925 году двадцатидвухлетний Колмогоров построил первую аксиоматизацию и перевод классических теорем на конструктивный язык²⁴³. Гливенко в 1929 доказал точную связь с классикой, о которой пойдёт речь в конце семантического раздела. В 1930 году Гейтинг опубликовал правила интуиционистского исчисления²⁴⁴,

²⁴²L. E. J. Brouwer. «De onbetrouwbaarheid der logische principes». 1908.

²⁴³А. Н. Колмогоров. «О принципе tertium non datur». Математический сборник, 1925; A. Kolmogorov. «Zur Deutung der intuitionistischen Logik». Mathematische Annalen, 1932.

²⁴⁴A. Heyting. «Die formalen Regeln der intuitionistischen Logik». 1930.

которыми пользуются до сих пор, а в 1932 году Колмогоров предложил читать доказательства как решения задач — по именам этих трёх математиков названа конструктивная интерпретация связок.

Полемика с Гильбертом вышла за пределы теории: в 1928 году Гильберт добился исключения Брауэра из редколлегии *Mathematische Annalen*, где оба состояли.

ОБЗОР ГЛАВЫ

Разберём семантику интуиционизма: ВНК-интерпретация (от англ. Brouwer–Heyting–Kolmogorov, далее — БХК), алгебры Хейтинга, семантика Крипке, двойное отрицание и теорема Гливенко. Затем рассмотрим промежуточные логики: пополнения интуиционизма, не доходящие до классики. Факультативный раздел покажет линейную логику, трактующую гипотезы как расходуемые ресурсы. Коснёмся и компактности — классического существования без построения: она обещает модель, не предъявляя её (глава 3). Для логики второго порядка компактность уже неверна (глава 29). Отдельный факультативный раздел посвящён доказательным ассистентам (Coq, Lean), проверяющим доказательства как типы термов.

После этой главы вы сможете:

1. объяснять, почему классическое доказательство существования не предъявляет объекта;
2. различать классическую и интуиционистскую истину;
3. применять ВНК-интерпретацию и семантику Крипке;
4. объяснять теорему Гливенко и роль двойного отрицания;
5. связывать интуиционизм с соответствием Карри–Ховарда и доказательными ассистентами.

§ 34.1 Семантика интуиционизма

Интуиционистская логика отказывается от классической семантики, в которой значений ровно два. У неё — несколько эквивалентных семантик:

- конструктивная (БХК-интерпретация).
- алгебраическая (алгебры Хейтинга).
- реляционная (семантика Крипке).

Все они сходятся в одном: снятие двойного отрицания и закон исключённого третьего — не тавтологии.

34.1.1 БХК-интерпретация

Интуиционистская истина конструктивна: истинность утверждения — наличие предъявленного доказательства. БХК-интерпретация фиксирует, что считается доказательством каждой связки.

ОПРЕДЕЛЕНИЕ 34.1 БХК-интерпретация

Доказательством формулы является:

- $A \wedge B$ — пара: доказательство A и доказательство B .
- $A \vee B$ — доказательство A или доказательство B , с пометкой, какое именно.
- $A \rightarrow B$ — способ превратить любое доказательство A в доказательство B .
- $\neg A$ — способ из доказательства A получить противоречие.
- $\forall x.A(x)$ — способ для любого x построить доказательство $A(x)$.
- $\exists x.A(x)$ — конкретный x вместе с доказательством $A(x)$.

Отрицание сводится к импликации: $\neg A \equiv A \rightarrow \perp$.

Пример Конструкция доказательства импликации

Покажем по БХК, что формула $A \rightarrow (B \rightarrow A)$ доказуема: предъявим способ превращать доказательства.

Пусть a — доказательство A , которое нам дали. Нужно построить доказательство $B \rightarrow A$: способ, который любое доказательство B превращает в доказательство A . Такой способ есть: константная функция, которая доказательство B игнорирует и возвращает a . Обозначим её $\lambda b.a$.

Итак, доказательство $A \rightarrow (B \rightarrow A)$ — это функция. Она принимает доказательство A и возвращает функцию $\lambda b.a$. Полное доказательство записывается как $\lambda a.\lambda b.a$. Первый аргумент — доказательство A , второй — доказательство B , результат — доказательство A . С этой функцией мы встретимся ещё раз в примере о Lean ниже.

Из БХК-интерпретации следует: дизъюнкция и существование конструктивны. $p \vee \neg p$ истинно, только если предъявлено доказательство p или доказательство $\neg p$. Такого доказательства нет, пока p не решено.

Пример «Завтра будет дождь»: утверждение без доказательства

Пусть p — «завтра будет дождь». Сегодня нет доказательства p : прогноз вопроса не решает. Нет и доказательства $\neg p$: способа превратить доказательство дождя в противоречие не предъявлено. По БХК это значит: доказательства нет ни у p , ни у $\neg p$, ни у $p \vee \neg p$, потому что доказательство дизъюнкции предъявляет одну из сторон.

Завтра вопрос решится. Если идёт дождь, наблюдение — доказательство p . Если дождя нет, запись сухого дня даёт способ превратить любое утверждение о

дожде в противоречие с фактами — это доказательство $\neg p$. Истинность здесь привязана ко времени: утверждение, не истинное сегодня, станет истинным завтра.

В булевой логике та же формула $p \vee \neg p$ — тавтология: её таблица истинности даёт 1 при любом значении p . Разница в том, что считать значением: классика — положение дел, БХК — предъявленное доказательство.

Снятие двойного отрицания $\neg\neg A \rightarrow A$ требует из «нельзя опровергнуть A » извлечь «построение A » — БХК этого не гарантирует.

Пример Почему не проходит снятие двойного отрицания

Попробуем по БХК построить доказательство формулы $\neg\neg A \rightarrow A$. Доказательство посылки — функция f , которая любое доказательство $\neg A$, то есть $A \rightarrow \perp$, превращает в доказательство $\perp$. Из такой функции нужно извлечь доказательство A .

Первая попытка: подать f какое-нибудь доказательство $\neg A$, получить $\perp$ и вывести из него A по правилу взрыва (ex falso). Но чтобы подать f аргумент, нужно иметь доказательство $\neg A$ — опровержение A , которого у нас нет.

Вторая попытка: разобрать случаи. Если A доказуемо, всё готово. Если нет, хочется объявить это доказательством $\neg A$, но отсутствие доказательства A не есть конструкция из A в $\perp$. И сам разбор случаев опирается на исключённое третье, которое мы как раз и не принимаем.

Третья попытка: посмотреть, не выдаёт ли f что-нибудь полезное сам по себе. Не выдаёт: f — потребитель опровержений, он возвращает $\perp$ только в ответ на аргумент типа $A \rightarrow \perp$. Аргумент такого типа мы предъявить не можем.

Конструкция застревает: f производит $\perp$ только при наличии аргумента, а аргумента у нас нет. Поэтому $\neg\neg A \rightarrow A$ не выводится из БХК-требований. Алгебры Хейтинга и семантика Крипке ниже покажут, что она недоказуема.

Пример Одно направление двойного отрицания проходит

Формула $p \rightarrow \neg\neg p$ интуиционистски выводима. По БХК: доказательство p — это a . Нужно построить доказательство формулы $\neg\neg p$. Это $(p \rightarrow \perp) \rightarrow \perp$. Берём любое доказательство $f : p \rightarrow \perp$ и применяем его к a : получаем $\perp$. Полное доказательство — функция $\lambda a. \lambda f. f(a)$. Двойное отрицание добавляет слой отрицания, и конструкция сохраняется. Не проходит лишь обратный переход — снятие двойного отрицания.

Выводимость $\vdash$ здесь — интуиционистское исчисление: правила натуральной дедукции главы 2 без правила косвенного доказательства.

ТЕОРЕМА 34.1 Свойство дизъюнкции

Если $\vdash A \vee B$ в интуиционистской логике, то $\vdash A$ или $\vdash B$.

Набросок доказательства

Докажем разбором последнего правила нормального вывода. Нормализуем вывод $A \vee B$. В нормальном выводе заключение правила введения не бывает большой посылкой удаления: такая пара — редекс, и нормализация её устраняет. Будем подниматься от последнего правила по цепочке больших посылок. Если последнее правило — введение, то по виду заключения $A \vee B$ это введение $\vee$, и его посылка — вывод A или вывод B . Если последнее правило — удаление, подъём по цепочке обрывается. Оборваться он может только на правиле из $\perp$: открытых допущений у вывода теоремы нет, а большие посылки-заключения введений запрещены нормальной формой. Тогда $\perp$ выводима, и по правилу взрыва выводима любая формула, в том числе обе стороны дизъюнкции.

Классическая логика этим свойством не обладает: $\vdash p \vee \neg p$, но ни $\vdash p$, ни $\vdash \neg p$ не выводимы. БХК-чтение объясняет свойство до всякой нормализации: доказательство $A \vee B$ по определению несёт пометку стороны.

Пример Свойство дизъюнкции в действии

Формула $p \vee \neg p$ показывает, что это свойство запрещает. Классически $p \vee \neg p$ доказуема, хотя ни один дизъюнкт не доказуем: формула p с переменной p теоремой быть не может, и $\neg p$ — тоже. В интуиционистской логике такая ситуация невозможна: если бы $p \vee \neg p$ была теоремой, свойство дизъюнкции дало бы доказательство p или доказательство $\neg p$. А ни того, ни другого не существует. Объяснение недоказуемости закона исключённого третьего получается прямо из БХК, без всякой семантики.

Положительная сторона: доказуемая дизъюнкция несёт в себе готовый выбор. Формула $(p \rightarrow p) \vee q$ доказуема, потому что доказуем левый дизъюнкт. Доказательство $p \rightarrow p$ — тождественная функция. Это функция $\lambda a.a$, возвращающая аргумент без изменений. Дизъюнкция получается правилом введения $\vee$ слева: выбран левый дизъюнкт, пометка не нужна.

ТЕОРЕМА 34.2 Свойство существования

Если $\vdash \exists x.A(x)$ в интуиционистской логике, то $\vdash A(t)$ для некоторого терма t .

Набросок доказательства

Воспользуемся нормализацией вывода.

Нормальный вывод $\exists x.A(x)$ заканчивается правилом введения $\exists$, которое предъявляет *свидетеля* — терм t с $A(t)$.

Именно поэтому из интуиционистского доказательства существования извлекается алгоритм, а из классического — нет. Классическое доказательство может рассуждать по случаям, не предъявив ни одного свидетеля.

Пример «В ящике лежит клад»: существование без свидетеля

Пусть p — «в ящике лежит клад». Формально это $\exists x.A(x)$: x — предмет из ящика, $A(x)$ — « x — клад». Классическое обоснование: ящик заперт, ключ у казначея, казначей утверждает, что клад в ящике. Существование обосновано, но ни один предмет не предъявлен.

По БХК такое обоснование — не доказательство p : доказательство обязано назвать предмет x и показать, что это клад. Доказательство появится, когда ящик откроют и предмет достанут. Свойство существования делает это обязательным: доказуемое $\exists x.A(x)$ всегда несёт свидетеля t с доказательством $A(t)$.

В булевой логике $\exists x.A(x)$ истинно по положению дел: клад в ящике либо есть, либо его нет, и свидетель на истинность не влияет. В интуиционистской логике истинно то, что предъявлено, — ровно как с парой иррациональных a, b из начала главы.

Проверка БХК-интерпретация

1. Что по БХК является доказательством $A \rightarrow B$, и почему отрицание сводится к импликации?
2. Почему из «нельзя опровергнуть A » не следует построение A ?
3. Почему свойство дизъюнкции получается прямо из определения доказательства $A \vee B$?

34.1.2 Алгебры Хейтинга

У классической логики есть простая семантика: два истинностных значения, таблицы истинности. Что считать «истинным значением» интуиционистской формулы? Одной бинарной шкалы мало: нужно, чтобы закон исключённого третьего не был тавтологией и чтобы из $\neg\neg p$ не следовало p .

Простейшая небулева алгебра Хейтинга — трёхэлементная цепочка $\{0, \frac{1}{2}, 1\}$. Она не совпадает с трёхзначной логикой Лукасевича²⁴⁵: там импликация иная. В логике Лукасевича снятие двойного отрицания — тавтология ($\neg\neg p \rightarrow p$ истинно при всех трёх значениях), а в цепочке Хейтинга оно принимает $\frac{1}{2}$ при $p = \frac{1}{2}$. Двух элементов для контрпримера не хватает: алгебра $\{0, 1\}$ — булева, и в ней выполняются оба

²⁴⁵J. Łukasiewicz. «O logice trójwartościowej». 1920.

классических принципа (исключённое третье и снятие двойного отрицания). Любая трёхэлементная решётка — цепочка, а трёхэлементная цепочка — единственная трёхэлементная алгебра Хейтинга. Значения читаются так: 0 — ложь, 1 — истина, $\frac{1}{2}$ — «неизвестно, ещё не построено».

Импликация в этой шкале:

- $a \rightarrow b = 1$, если $a \leq b$.
- $a \rightarrow b = b$ иначе.

При $a = \frac{1}{2}, b = 0$ импликация даёт 0. Двойное отрицание считается по шагам. Сначала отрицание $\frac{1}{2}$:

$$\neg\left(\frac{1}{2}\right) = \left(\frac{1}{2} \rightarrow 0\right) = 0.$$

Затем двойное отрицание:

$$\begin{aligned}\neg\neg\left(\frac{1}{2}\right) &= \neg 0 \\ &= (0 \rightarrow 0) = 1.\end{aligned}$$

При $p = \frac{1}{2}$ формула $\neg\neg p \rightarrow p$ принимает значение

$$\begin{aligned}\neg\neg\left(\frac{1}{2}\right) \rightarrow \frac{1}{2} &= 1 \rightarrow \frac{1}{2} \\ &= \frac{1}{2} \neq 1.\end{aligned}$$

Снятие двойного отрицания — не тавтология.

Проверим: при $p = \frac{1}{2}$ формула $p \vee \neg p$ принимает значение

$$\begin{aligned}\frac{1}{2} \vee \neg\frac{1}{2} &= \frac{1}{2} \vee 0 \\ &= \frac{1}{2}.\end{aligned}$$

Дизъюнкция не всегда истинна.

Все значения сведены в таблицу:

p	$\neg p$	$p \vee \neg p$	$\neg\neg p$	$\neg\neg p \rightarrow p$
0	1	1	0	1
$\frac{1}{2}$	0	$\frac{1}{2}$	1	$\frac{1}{2}$
1	0	1	1	1

При $p = \frac{1}{2}$ обе классические тавтологии принимают значение $\frac{1}{2}$.

Пример «Завтра будет дождь» на трёхэлементной цепочке

Возьмём p — «завтра будет дождь». Сегодня вопрос не решён, поэтому $p = \frac{1}{2}$. Посчитаем связки на цепочке $\{0, \frac{1}{2}, 1\}$:

Формула	Выкладка	Значение	Прочтение
p	—	$\frac{1}{2}$	«завтра будет дождь» — не решено
$\neg p$	$\frac{1}{2} \rightarrow 0 = 0$	0	«завтра не будет дождя» — ложно
$p \vee \neg p$	$\frac{1}{2} \vee 0 = \frac{1}{2}$	$\frac{1}{2}$	«дождь будет или не будет» — не доказано
$\neg\neg p$	$\neg 0 = 0 \rightarrow 0 = 1$	1	«неверно, что дождя не будет» — истинно
$\neg\neg p \rightarrow p$	$1 \rightarrow \frac{1}{2} = \frac{1}{2}$	$\frac{1}{2}$	снятие двойного отрицания — не проходит

Та же таблица в булевой логике:

Формула	Булево значение	Значение на цепочке при $p = \frac{1}{2}$
$p \vee \neg p$	1 при любом p	$\frac{1}{2}$
$\neg\neg p \rightarrow p$	1 при любом p	$\frac{1}{2}$
$p \rightarrow p$	1 при любом p	1

Прочтения в первой таблице описывают состояние знания сегодня: 0 означает «опровергнуто на текущем знании», а не «установлено, что не так». Строка $\neg p = 0$ при $p = \frac{1}{2}$ не утверждает, что дождя не будет: опровергнуть дождь сегодня нельзя, поэтому значение $\neg p$ равно 0, пока p не решено. Когда завтра p примет 0 или 1, значение $\neg p$ пересчитается: $\neg 1 = 0$, $\neg 0 = 1$.

В булевой таблице $p \vee \neg p$ — константа 1: тавтология. Формула $\neg\neg p \rightarrow p$ — тоже константа 1: тавтология. На цепочке те же формулы принимают $\frac{1}{2}$: тавтологиями они не являются. Формула $p \rightarrow p$ остаётся истинной и на цепочке: интуиционизм отменяет два классических принципа, исключённое третье и снятие двойного отрицания, а не всю классическую логику. При $p = \frac{1}{2}$ формула $\neg\neg p$ получает значение 1: «неверно, что завтра не будет дождя» истинно, а сам дождь не предъявлен. Двойное отрицание снимает опровержение, но не строит доказательства.

ОПРЕДЕЛЕНИЕ 34.2 Алгебра Хейтинга

Алгебра Хейтинга — дистрибутивная решётка с наименьшим элементом $\perp$ и наибольшим элементом $\top$, в которой для любых a, b существует наибольший элемент c с условием $a \wedge c \leq b$.

Здесь:

- **Дистрибутивная решётка** — множество с операциями $\wedge$ и $\vee$, согласованными с порядком: операции дают грани пары, наибольшую нижнюю и наименьшую верхнюю.
- **Дистрибутивность** — тождество $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.
- **Относительное псевдодополнение** a до b — элемент c из определяющего условия. Обозначается $a \rightarrow b$.

Отрицание в такой решётке определяется как $\neg a \equiv a \rightarrow \perp$, то есть «насколько a далеко от дна».

Классическая булева алгебра — частный случай алгебры Хейтинга, где $\neg\neg a = a$ для всех a .

Что значит «формула истинна в алгебре Хейтинга»? Оценка назначает каждой переменной элемент алгебры. Значения связок вычисляются операциями алгебры: конъюнкция — $\wedge$, дизъюнкция — $\vee$, импликация — относительное псевдодополнение, отрицание — $a \rightarrow \perp$. Формула истинна в алгебре, если её значение — $\top$ при всех оценках.

ТЕОРЕМА 34.3 Полнота для интуиционизма

Формула интуиционистски выводима тогда и только тогда, когда она истинна во всех алгебрах Хейтинга.

Набросок доказательства

Докажем эквивалентность в обе стороны.

Формулы рассматриваются с точностью до интуиционистской эквивалентности. Классы эквивалентности образуют алгебру Хейтинга — *алгебру Линденбаума–Тарского*, порядок задаётся выводимостью импликаций между представителями. В этой алгебре $[\varphi] = \top$ тогда и только тогда, когда φ выводима. Если φ не выводима, то $[\varphi] \neq \top$, и φ ложна в этой алгебре: невыводимость всегда опровергается конкретной алгеброй. Это интуиционистский аналог леммы Линденбаума из главы 2.

Обратно: истинность выводимых формул во всех алгебрах Хейтинга проверяется индукцией по выводу.

Степень $[\varphi] = \top$ стоит разобрать: $\top$ — класс заведомо истинной формулы, например $p \rightarrow p$. Равенство $[\varphi] = [p \rightarrow p]$ означает выводимость импликаций в обе стороны. Из $(p \rightarrow p) \rightarrow \varphi$ по правилу *modus ponens* выводится сама φ . Поэтому $\top$ — ровно классы теорем, и невыводимая формула при оценке своими классами получает значение, отличное от $\top$.

Алгебры Хейтинга дают интуиционизму полную семантику — как булевы алгебры дают её классической логике.

Пример Алгебры Хейтинга в топологии

Открытые множества топологического пространства образуют алгебру Хейтинга. Импликация $A \rightarrow B$ — объединение всех открытых U с $U \cap A \subseteq B$. Это наибольшее открытое множество, «почти содержащееся» в B . Отрицание $\neg A$ — внутренность дополнения. Так интуиционистская логика оказывается логикой *открытых* свойств: свойство, истинное в точке, истинно и в некоторой окрестности.

Проверка Алгебры Хейтинга

1. Почему двух элементов недостаточно для контрпримера к закону исключённого третьего?
2. Как устроена импликация на трёхэлементной цепочке, и почему $\frac{1}{2} \rightarrow 0 = 0$?
3. Почему $\neg\neg(\frac{1}{2}) = 1$? Что это даёт для формулы $\neg\neg p \rightarrow p$ при $p = \frac{1}{2}$?

34.1.3 Семантика Крипке

Интуиционистскую истину задаёт и семантика Крипке (глава 33), но с монотонностью: знание не отменяется.

ОПРЕДЕЛЕНИЕ 34.3 Интуиционистская модель Крипке

Модель — фрейм (W, R) с предпорядком R : рефлексивным и транзитивным отношением. Антисимметричность не требуется: взаимно достижимые миры описывают одно и то же знание. Миры — состояния знания, а $R(w, w')$ означает: в w' знают не меньше, чем в w .

Оценка монотонна: если $w \in V(p)$ и $R(w, w')$, то $w' \in V(p)$ — став истинным, атом остаётся истинным.

Принуждение $w \Vdash A$ определяется индукцией:

- $w \Vdash p$ тогда и только тогда, когда $p \in V(w)$.
- $w \Vdash A \wedge B$ тогда и только тогда, когда $w \Vdash A$ и $w \Vdash B$.
- $w \Vdash A \vee B$ тогда и только тогда, когда $w \Vdash A$ или $w \Vdash B$.
- $w \Vdash A \rightarrow B$ тогда и только тогда, когда для всех w' с $R(w, w')$ из $w' \Vdash A$ следует $w' \Vdash B$.
- $w \Vdash \neg A$ тогда и только тогда, когда ни для какого w' с $R(w, w')$ не верно $w' \Vdash A$.

Замечание Две структуры с одним именем

Название «модель Крипке» глава 33 уже заняла: там миры связывает произвольное бинарное отношение, а принуждение читает модальности. Здесь та же пара слов означает другую структуру: отношение — предпорядок, оценка моно-

тонна, а $\rightarrow$ и $\neg$ смотрят вперёд по нему. Это перегрузка имени, а не совпадение структур: общая у них только картина миров и достижимости.

Примечание Пропозициональный случай

Определение дано для пропозициональных формул. Для кванторов модель пополняется предметными областями, монотонно растущими по мирам, — здесь это не требуется: глава работает с высказываниями.

Импликация и отрицание смотрят вперёд: $A \rightarrow B$ истинно сейчас, если в любом будущем состоянии знания из A следует B . Поэтому $\neg A$ означает: никакое расширение знания не подтвердит A .

Закон исключённого третьего не выполняется: мир может не знать ни p , ни $\neg p$, если в будущем p может стать известным.

Пример Модель Крипке: неопровержимое, но не доказанное

Возьмём два мира $u < v$. Оценка: $V(p) = \{v\}$ — атом p истинен только в верхнем мире. В мире u про p пока ничего не известно. В v его уже доказали. Оценка монотонна: расширив знание до v , мы p не теряем. Модель из двух миров — на рис. 126.

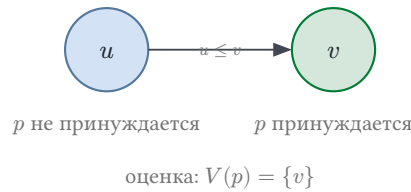


Рис. 126. Два мира $u < v$.

Посчитаем принуждение шаг за шагом, начиная с u .

1. Атом p : $p \notin V(u)$. Значит, $u \Vdash p$ не выполняется.
2. Отрицание $\neg p$: для принуждения нужно, чтобы никакой мир $w' \geq u$ не принуждал p . Но $v \geq u$ принуждает p . Поэтому $u \Vdash \neg p$ не выполняется.
3. Двойное отрицание $\neg\neg p$: для принуждения нужно, чтобы никакой мир $w' \geq u$ не принуждал $\neg p$. Для u это только что показано. Для v : если бы выполнялось $v \Vdash \neg p$, ни один мир над v не принуждал бы p . Но сам v принуждает p , значит, $v \Vdash \neg p$ не выполняется. Ни один мир над u не принуждает $\neg p$, поэтому $u \Vdash \neg\neg p$ выполняется.

Итак, в u принуждается $\neg\neg p$, но не p . Импликация $\neg\neg p \rightarrow p$ в модели не выполняется: посылка истинна в u , заключение — нет. Заодно видно, почему не принуждается $p \vee \neg p$: в u нет ни p , ни $\neg p$.

Мир	p	$\neg p$	$\neg\neg p$	$p \vee \neg p$
u	нет	нет	да	нет

Мир	p	$\neg p$	$\neg\neg p$	$p \vee \neg p$
v	да	нет	да	да

Замечание «Неверно, что завтра не будет дождя»

Пример с двумя мирами читается в бытовых терминах: p — «завтра будет дождь», мир u — сегодняшний вечер, мир v — завтрашнее утро, когда дождь идёт. В u принуждается $\neg\neg p$: «неверно, что завтра не будет дождя», потому что мир с дождём совместим с сегодняшним знанием. А p в u не принуждается: дождь ещё не пошёл. Отсюда и невыполнимость $\neg\neg p \rightarrow p$: из «дождь не исключён» не следует «дождь будет». В булевой логике $\neg\neg p$ совпадает с p : снятие двойного отрицания — тавтология.

Проверка Семантика Крипке

1. Почему в модели из примера в мире u не принуждается ни p , ни $\neg p$?
2. Почему $\neg\neg p \rightarrow p$ не выполняется в u , хотя посылка там принуждается?
3. Зачем в определении импликации и отрицания нужен квантор по всем будущим мирам?

Пример «Если завтра будет дождь, я возьму зонт»: импликация в Крипке

Пусть p — «завтра будет дождь», q — «я возьму зонт». Сегодня, в мире w_0 , не известно ни p , ни q : дождя не обещают, зонт в сумку не положен. Модель добавляет над w_0 два несравнимых мира, w_1 и $w_{1'}$. В мире w_1 дождь подтверждён и зонт взят: $V(p) = V(q) = \{w_1\}$. В мире $w_{1'}$ дождь подтверждён, а зонт не взят: $V(p) = \{w_{1'}\}$, $V(q) = \emptyset$. Мир $w_{1'}$ описывает нарушенную договорённость. Устройство модели — на рис. 127.

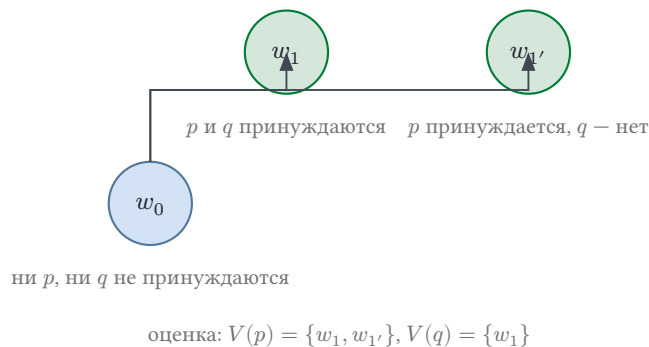


Рис. 127. Корень w_0 и два несравнимых мира над ним.

Мир	p	q	$p \rightarrow q$	$p \vee \neg p$
w_0	нет	нет	нет	нет
w_1	да	да	да	да
$w_{1'}$	да	нет	нет	да

Импликация $p \rightarrow q$ в w_0 требует: во всяком мире $w' \geq w_0$ из p следует q . В w_1 это выполняется, в $w_{1'}$ — нет: там дождь идёт, а зонта нет. Поэтому $w_0 \Vdash p \rightarrow q$ не выполняется: модель допускает расширение, в котором договорённость нарушена. Если убрать $w_{1'}$ из модели, импликация станет истинной в w_0 : во всяком расширении с дождём зонт взят. Истинность импликации — истинность обязательства: она не требует, чтобы сегодня были известны p или q . Дизъюнкция $p \vee \neg p$ в w_0 не принуждается: не принуждается ни p , ни $\neg p$. В булевой таблице $p \rightarrow q$ ложна ровно на строке $p = 1, q = 0$ — строке мира $w_{1'}$. Семантика Крипке добавляет к этой строке условие: она должна отсутствовать во всех расширениях знания, а не только в текущем мире.

34.1.4 Двойное отрицание и Гливенко

В трёх семантиках снятие двойного отрицания не проходит: БХК не предъявляет конструкцию, на цепочке $\neg\neg p \rightarrow p$ принимает $\frac{1}{2}$ при $p = \frac{1}{2}$, в модели Крипке корень принуждает $\neg\neg p$, но не p . За этим сравнением стоит многолетний спор об основаниях, и вторую его сторону занял Гильберт. Отказ от закона исключённого третьего обесценил бы классический анализ с его нефинитными доказательствами существования. Гильберт предложил другую стратегию: сохранить классическую математику целиком, формализовав её в исчислениях и обеспечив надёжность финитной теорией доказательств. Непротиворечивость исчисления доказывается комбинаторикой конечных объектов, и тогда всё выводимое в нём легитимно. Закон исключённого третьего при таком взгляде — соглашение игры с символами: пока непротиворечивость держится, игра безопасна. Разногласие видно на формуле из начала главы: рассуждение о степенях $\sqrt{2}^{\sqrt{2}}$ для Гильберта — законный способ доказывать существование, для Брауэра — незавершённый эскиз.

Исход решил Гёдель: теоремы о неполноте (глава 26) показали, что финитного доказательства непротиворечивости арифметики не существует. Теория доказательств пережила программу, и её центральный вопрос — точное соотношение классической и интуиционистской выводимости — остался. Для логики высказываний ответ даёт теорема Гливенко.

ТЕОРЕМА 34.4 Теорема Гливенко

Формула φ выводима в классической логике высказываний тогда и только тогда, когда в интуиционистской выводимо $\neg\neg\varphi$.²⁴⁶

Набросок доказательства

Докажем отрицательным переводом: сопоставим каждой формуле её двойное отрицание и проверим, что классический вывод переносится вместе с ним.

²⁴⁶В. И. Гливенко. «Sur quelques points de la logique de M. Brouwer». 1929.

Два опорных факта выводятся в интуиционистском исчислении. Из φ выводимо $\neg\neg\varphi$, поэтому обёртывание сохраняет все интуиционистские аксиомы. Обёртывание сохраняет и *modus ponens*: из $\neg\neg(A \rightarrow B)$ и $\neg\neg A$ выводится $\neg\neg B$. Остаётся проверить классические аксиомы, которых у интуиционистского исчисления нет. Их обёртывания $\neg\neg(p \vee \neg p)$ и $\neg\neg(\neg\neg p \rightarrow p)$ интуиционистски выводимы, хотя сами аксиомы — нет. По индукции по классическому выводу каждая классическая теорема φ получает интуиционистское доказательство $\neg\neg\varphi$.

Обратная импликация проста: интуиционистский вывод $\neg\neg\varphi$ остаётся классическим, а классика снимает двойное отрицание.

У перевода есть граница: для логики предикатов обёртывание целиком не переносится, и там работает перевод Гёделя—Генцена, меняющий образы $\forall$ и $\exists$. Его предвосхитил Колмогоров в 1925 году. В приложении к арифметике перевод даёт относительную непротиворечивость: классическая арифметика непротиворечива, если непротиворечива интуиционистская (Гёдель, 1933, Генцен, 1936).

Замечание Что это значит

Классическое доказательство φ — это интуиционистское доказательство $\neg\neg\varphi$: «нельзя опровергнуть φ ». Классическая логика — интуиционистская, в которой доказательство ослаблено до опровержения опровержения. Утверждение считается доказанным, когда опровергнуто его отрицание, то есть построен вывод $\neg\varphi \rightarrow \perp$. На уровне алгебр Хейтинга: классическая логика — алгебры Хейтинга, где $\neg\neg a = a$ всегда, то есть булевы алгебры.

§ 34.2 * Промежуточные логики

Теорема Гливенко ставит классику вплотную к интуиционизму: классика получается из интуиционистской логики добавлением снятия двойного отрицания. Между этими крайностями есть собственная территория: пополнения интуиционистского исчисления новыми схемами, не доходящие до классики.

ОПРЕДЕЛЕНИЕ 34.4 Промежуточная логика

Логика L называется *промежуточной*, если каждая теорема интуиционистской логики является теоремой L , каждая теорема L является теоремой классической логики и существует классическая тавтология, не являющаяся теоремой L .

Инструменты здесь те же, что и у самого интуиционизма: семантика Крипке и алгебры Хейтинга. Каждая новая схема аксиом отсекает часть моделей, и разные наборы схем дают разные логики. Промежуточных логик континуум.

Ближайший к интуиционизму шаг — слабое исключённое третье, схема $\neg p \vee \neg\neg p$. Она интуиционистски не выводима, а её пополнение называют логикой Янкова. Эта логика полна относительно слитых фреймов Крипке, в которых любые два мира имеют общий мир выше обоих. Схема $(p \rightarrow q) \vee (q \rightarrow p)$ задаёт логику Даммета: она полна относительно линейно упорядоченных миров и выражает сравнимость любых двух состояний знания.

Свойство дизъюнкции при пополнениях не обязано выживать, и контрпример даёт сама логика Янкова.

Пример Логика Янкова теряет свойство дизъюнкции

В логике Янкова формула $\neg p \vee \neg\neg p$ — теорема, но ни один из дизъюнктов теоремой не является. Для $\neg\neg p$ возьмём цепочку $r < v$, где p ложна всюду: тогда $r \Vdash \neg p$, и двойное отрицание в корне опровергнуто. Для $\neg p$ возьмём ту же цепочку с p , истинной только в v : вершина подтверждает p , поэтому $r \Vdash \neg p$ не выполняется. Обе цепочки слиты, а на слитых фреймах схема $\neg p \vee \neg\neg p$ валидна. Если бы в корне опровергались обе стороны, мир с $\neg p$ и мир с p имели бы общее продолжение, которое подтверждает p и опровергает её одновременно. Значит, обе модели удовлетворяют всем теоремам логики Янкова и опровергают по одному дизъюнкту: свойство дизъюнкции рушится первым же пополнением.

Есть и мост к главе 33: перевод Гёделя читает каждую интуиционистскую формулу внутри модальной системы S4, и промежуточные логики продолжаются до расширений S4.

§ 34.3 * Линейная логика

Промежуточные логики усиливают интуиционизм, сохраняя все его правила. Линейная логика Жирара²⁴⁷ устроена иначе: она ослабляет структурные правила, и каждая гипотеза становится ресурсом, расходуемым ровно один раз. Конъюнкция расщепляется на две: мультипликативная (A и B параллельно) и аддитивная (выбор между ними, за потребителем).

Чтобы вернуть классическое рассуждение, вводятся экспоненциальные модальности $!$ и $?$. Модальность $!A$ означает: формула доступна неограниченное число раз. Модальность $?A$ — двойственная: формула доступна любое число раз, включая ноль. В классическом вложении она соответствует отрицанию $!A$: как $!$ восстанавливает право сокращать и ослаблять гипотезы, так $?$ — то же право для их отрицаний.

²⁴⁷J.-Y. Girard. «Linear Logic». Theoretical Computer Science, 1987.

Пример Ресурс можно израсходовать только один раз

Формула $A \rightarrow A \otimes A$ в линейной логике не выводится: заключение требует два экземпляра A , а посылка поставяет ровно один. *Мультипликативная конъюнкция* $\otimes$ соединяет два разных ресурса, и удвоить гипотезу нечем: структурное правило сокращения отсутствует. Разрешение на повторное использование объявляется явно: из $!A$ вывод $A \otimes A$ проходит, потому что $!A$ доступна неограниченное число раз. Проходит и аддитивный вариант $A \rightarrow A \& A$: *аддитивная конъюнкция* — это выбор, и один экземпляр A обслуживает его без размножения.

В общем случае снятие двойного отрицания в линейной логике не выводится. Классика возвращается вложением: перевод заменяет классическую импликацию $A \rightarrow B$ на $!A \rightarrow B$ в линейной логике, а отрицание оборачивает модальностью. Модальности $!$ и $?$ восстанавливают право сокращения и ослабления гипотез. Точные правила — в статье Жирара. Здесь важно следствие: классическая выводимость сохраняется переводом, и обратно. Линейная логика моделирует ресурсы: владение памятью (Rust), сетевые взаимодействия (типы сессий для параллельных процессов).

Классическая логика при этом осталась на месте: она непротиворечива и по-прежнему применима, просто перестала быть единственным вариантом. Каждая альтернатива соответствует своему классу вычислений: интуиционистская — функциям и типам (глава 28), линейная — ресурсам, которые можно израсходовать только один раз. Выбор логики определяет, что в ней можно выразить и какой ценой.

§ 34.4 Компактность и существование без построения

Компактность (глава 3) — классическое существование без построения: если каждое конечное подмножество теории имеет модель, то модель имеет и вся теория. Теорема обещает модель, не предъявляя её, — ровно то существование, которое интуиционист не признаёт доказательством.

Теорема Гливенко оставляет такое доказательство лишь как доказательство двойного отрицания.

Для логики второго порядка компактность неверна (глава 29): выражается утверждение, которое никакое конечное подмножество теории не охватывает. Конкретный пример: утверждение «домен конечен» и теория из утверждений «элементов не меньше n » для всех n . Каждое конечное подмножество такой теории имеет конечную модель: достаточно домена, покрывающего максимальное из требований подмножества. А вся теория модели не имеет: её модель была бы конечной и бесконечной одновременно.

У вопроса «что считать доказательством» есть инженерная сторона: доказательство должно допускать машинную проверку.

§ 34.5 * Доказательные ассистенты (Coq и Lean)

Соответствие Карри–Ховарда (глава 28) превращает доказательство в программу. *Доказательные ассистенты* (*proof assistants*) — системы, которые проверяют доказательства машинно, эксплуатируя этот изоморфизм. Математик пишет доказательство как терм в зависимо-типизированном исчислении, машина проверяет его тип.

ОПРЕДЕЛЕНИЕ 34.5 Доказательный ассистент

Доказательный ассистент — интерактивная система, в которой:

- утверждение — это тип;
- доказательство — терм этого типа;
- проверка доказательства — проверка типов (type checking).

Машина не предлагает доказательства сама, но проверяет каждый шаг: ошибка в рассуждении — ошибка типизации.

- Coq (1989, INRIA) — один из первых зрелых ассистентов. На нём формализована теорема о четырёх красках Жоржем Гонтье²⁴⁸.
- Lean (2013, Microsoft Research) — современный ассистент с мощной автоматизацией. На нём ведётся проект mathlib — формализация значительной части современной математики.
- Agda (Ульф Норелл, 2007) — ассистент на основе интуиционистской теории типов Мартина-Лёфа.

Пример Как выглядит доказательство в Lean

Утверждение «если n чётно, то n^2 чётно» — тип функции. В Lean оно записывается так:

```
example (n : Nat) : Even n -> Even (n ^ 2) := by
  intro h
  rcases h with (k, hk)
  use 2 * k ^ 2
  rw [hk]
  ring
```

Здесь `Even n` — стандартное определение чётности: существует k с $n = k + k$. Доказательство — терм, который тактики строят шаг за шагом: `intro` принимает доказательство посылки, `rcases` разбирает его на свидетель k и равенство hk , `use` предъявляет свидетель существования, `rw` подставляет равенство, `ring` проверяет арифметическое тождество. Проверка типов машиной гарантирует: терм корректен — доказательство действительно. Фрагмент проверен в Lean 4 (v4.33.0) с библиотекой Mathlib.

²⁴⁸G. Gonthier. «Formal Proof—The Four-Color Theorem». Notices of the AMS, 2008.

Чисто логическое утверждение доказывается теми же тактиками, только без арифметики. Формула $A \rightarrow (B \rightarrow A)$ — тип функции. Она принимает доказательство A и возвращает доказательство $B \rightarrow A$.

```
example (A B : Prop) : A -> B -> A := by
  intro a -- принимаем доказательство A
  intro b -- принимаем доказательство B
  exact a -- возвращаем доказательство A
```

Терм, собранный тактиками, — `fun a b => a`. Первый аргумент — доказательство A , второй — доказательство B , результат — доказательство A . Это ровно константная функция из БХК-интерпретации: доказательство B игнорируется, ответ всегда — a . Этот фрагмент проверен в Lean 4 (v4.33.0) без библиотек.

Автоматизация (тактики) помогает строить термы: `omega` решает линейную арифметику, `simp` — упрощение, `ring` — полиномиальные тождества.

Ассистенты — наследники мечты Лейбница (глава 1): рассуждение, сводимое к вычислению. Разница в том, что Лейбниц представлял это философски, а Coq и Lean делают это инженерно. Ошибки случаются и здесь (в самих системах), но проверка каждого шага — механическая, как в арифметике.

§ 34.6 Итоги главы

Классическая логика — выбор: закон исключённого третьего и снятие двойного отрицания не навязаны извне, и отказ от них открывает другие логики. Интуиционизм связывает истину с предъявленным доказательством: БХК-интерпретация объясняет, что значит построить доказательство каждой связки — конъюнкция требует обеих частей, дизъюнкция — конкретной стороны, импликация — способа превращения доказательства посылки в доказательство заключения. Главное следствие — классическое доказательство существования не предъявляет объекта, и интуиционист его не признаёт.

Алгебры Хейтинга и семантика Крипке дают этому точную семантику — от трёх-элементной цепочки до миров знания. Снятие двойного отрицания перестаёт быть тавтологией, и теорема Гливенко описывает связь точнее: классическое доказательство — это интуиционистское доказательство двойного отрицания.

Соответствие Карри–Ховарда превращает эту позицию в инженерную: доказательство — программа, утверждение — тип, и доказательные ассистенты (Coq, Lean) проверяют рассуждения как типы термов (глава 28). Между интуиционистской и классической логиками лежат промежуточные: их континуум, и уже слабое исключённое третье, первый шаг от интуиционизма, разрушает свойство дизъюнкции. В стороне остаётся отказ от другого типа правил: линейная логика ослабляет структурные правила и трактует гипотезы как ресурсы, расходующиеся ровно один раз.

Полнота логики первого порядка даёт компактность — источник нестандартных моделей (главы 3 и 29). Выбор логики — выбор цены: каждое допущение открывает одни доказательства и закрывает другие.

Проверка Проверьте себя

1. Почему интуиционист не принимает закон исключённого третьего, и что в БХК-интерпретации требуется от доказательства $A \vee B$?
2. Почему снятие двойного отрицания $\neg\neg A \rightarrow A$ недоказуемо, и на каком шаге застревает конструкция?
3. В модели Крипке из примера: почему $\neg\neg p$ принуждается в нижнем мире, а p — нет?
4. Как теорема Гливленко связывает классическую доказуемость с интуиционистской?
5. Что проверяет доказательный ассистент, когда «проверяет доказательство», и что такое терм в смысле Карри–Ховарда?

Что дальше?

Интуиционизм связывает истину с доказательством. В главе 35 мы сделаем последний шаг от классической логики — к градуальной принадлежности. Градуальная принадлежность размоет и саму границу «истинно/ложно».

§ 34.7 Упражнения

Упражнение о компактности опирается на раздел выше и на теорему главы 3.

Интуиционизм и БХК.

1. Почему закон исключённого третьего неприемлем для интуициониста?
2. По БХК-интерпретации: что является доказательством $A \rightarrow B$? А доказательством $\exists x.A(x)$?
3. Что соответствует конъюнкции и дизъюнкции при соответствии Карри–Ховарда?
4. Чем линейная логика отличается от классической?
5. Проверьте на трёхэлементной алгебре Хейтинга: $p \vee \neg p$ не тавтология, а $\neg\neg(p \vee \neg p)$ — тавтология.
6. * Постройте модель Крипке с корнем w_0 и двумя несравнимыми мирами w_1 , w_2 над ним, в которой импликация $p \rightarrow q$ принуждается в w_0 , хотя сегодня не известно ни p , ни q . Укажите оценку и объясните, почему обязательство выполняется. Подсказка: сделайте w_1 миром, где и p , и q . В w_2 пусть p не принуждается.

7. * Сформулируйте теорему Гливенко и объясните, почему классическое доказательство — это интуиционистское доказательство двойного отрицания.
8. * Объясните, почему компактность неверна для логики второго порядка. Что именно выражается, но не охватывается конечными подмножествами?
9. Формула $\neg p \vee \neg\neg p$ — классическая тавтология. Постройте модель Крипке с корнем w_0 и двумя несравнимыми мирами w_1, w_2 , где p принуждается в w_1 и не принуждается в w_2 , и покажите, что в корне не принуждается ни $\neg p$, ни $\neg\neg p$. Как называется пополнение интуиционистской логики этой схемой и что происходит в нём со свойством дизъюнкции?
10. * Пусть A — атомарная формула (базовый тип, отличный от $\perp$). Запишите снятие двойного отрицания $\neg\neg A \rightarrow A$ как тип, используя $\neg A = A \rightarrow \perp$. Докажите, что в чистом типизированном λ -исчислении (только переменные, абстракция и аппликация) не существует замкнутого терма такого типа. Какой оператор языка программирования «обитает» этот тип и тем самым восстанавливает классическую логику?
11. Как доказательный ассистент связан с соответствием Карри–Ховарда? Что проверяет машина, когда «проверяет доказательство»?

ПРОЕКТ Извлечение свидетеля из доказательства существования

Постройте интерпретатор-экстрактор по соответствию Карри–Ховарда: доказательство — терм, формула — тип, и из доказательства существования механически извлекается свидетель. Исходная точка — классическое доказательство существования иррациональных a, b с рациональным a^b : оно занимает две строки, но не называет ни одной пары. Рассуждение разбирает случаи « $\sqrt{2}^{\sqrt{2}}$ рационально или нет» и опирается на закон исключённого третьего: существование гарантировано, а объект не предъявлен. Интуиционист такое доказательство не признаёт: для него «существует» значит «вот оно».

① Исчисление

Соберите типы-формулы и термы-доказательства: импликация, произведение, размеченная сумма, пустой тип, отрицание $\neg A = A \rightarrow \perp$. Проверка типов и нормализация. Продумайте, как формула $\exists x.A(x)$ кодируется парой «свидетель, доказательство».

② Экстрактор

Научитесь извлекать из терма типа $A \vee B$ выбранную сторону и её доказательство. Из терма типа $\exists x.A(x)$ извлекайте свидетеля и доказательство $A(x)$. Объясните, почему извлечение сводится к вычислению нормальной формы и чтению её головы.

③ Необитаемость

Проверьте полным перебором нормальных форм, что замкнутого терма типа $\neg\neg A \rightarrow A$ в чистом исчислении нет, и докажите его отсутствие струк-

турно (бесконечный спуск), а не конечной выборкой размеров. Покажите, что оператор call/cc даёт терм, населяющий этот тип, и терм для $A \vee \neg A$.

④ *Классическая загадка*

Проверьте, что классическое рассуждение о $\sqrt{2}^{\sqrt{2}}$ не типизируется: разбор случаев требует обитателя $\text{RatS} \vee \neg \text{RatS}$ для нерешённого атома, а ветви дают разные типы. Предъявите конструктивную пару иррациональных a, b с рациональным a^b и проверьте программой, что a^b действительно рационально.

⑤ *Запуск*

Запустите извлечённых свидетелей на конкретных входах и сведите в таблицу: формула, есть ли обитатель, терм-свидетель, программа, которая его вычисляет.

Итог: работающий интерпретатор-экстрактор, где из доказательства существования механически получается программа, предъявляющая объект, и строгое обоснование того, что снятие двойного отрицания необитаемо без call/cc.

XXXV

Нечёткие множества

Классическая теория множеств опирается на закон исключённого третьего. Для любого элемента x и любого множества A верно ровно одно из двух: либо x принадлежит A , либо не принадлежит. Характеристическая функция фиксирует эту дихотомию: каждому элементу — 0 или 1. На этом фундаменте стоит вся дискретная математика курса — и фундамент работает.

Однако некоторые понятия сопротивляются бинарной классификации. Естественное определение множества «высоких людей» — через порог: высоким считается человек ростом от 180 см. Тогда при росте 180 см человек высок, а при 179.9 см — нет. Один миллиметр решает принадлежность к множеству.

Это не вопрос точности измерений — это свойство самого понятия. Парадокс кучи, известный со времён Эвбулида Милетского, указывает на тот же структурный факт: для некоторого класса понятий любая бинарная граница произвольна. «Молодой возраст», «высокая температура», «большое число» — какую границу ни проведи, найдутся элементы, которые интуиция относит к одной категории, а формальное определение — к другой.

В главе 31 точность уже принесена в жертву гарантии: вместо значений переменных брались их знаки, а свойства программ приближались элементами решёток. Абстракция теряла информацию, но взамен давала ответ, верный для всех запусков сразу. Следующий шаг — нечёткость: там, где абстрактная интерпретация жертвовала точностью ради вычислимости, нечёткость отказывается от ответа «да или нет» ради выразительности. Обе конструкции начинаются с одного признания: резкая граница часто есть свойство описания, а не самого понятия.

Поводов отказаться от порога в практике достаточно. Кондиционер решает, жарко в комнате или нет, — но жара нарастает плавно, и правило «если жарко, то охлаждение сильное» требует, чтобы степень жары была числом. Будь правило бинарным, регулятор метался бы на границе: чуть жарче порога — полная мощность, чуть холоднее — ноль. Автопилот с порогом 40 км/ч на скорости 40.1 жмёт тормоз, на 39.9 — газ, и машину швыряет на каждом переходе границы. Для рекомендательной системы и сегментации изображений граница так же размыта: степень соответствия фильму вкусу, переход между небом и зданием на снимке. Для таких понятий бинарная принадлежность неверна.

Нечёткое множество (fuzzy set) — понятие другой аксиоматики, а не компромисс с неточностью. Вместо характеристической функции, отвечающей «да» или «нет»,

вводится *функция принадлежности*: она приписывает каждому элементу степень от нуля до единицы. Тридцатилетний человек — «молодой» со степенью 0.6 и «средний» со степенью 0.3, и оба утверждения верны, но в разной мере.

Классическое множество остаётся частным случаем: когда функция принадлежности принимает только значения 0 и 1, бинарная принадлежность возвращается.

Отказ от закона исключённого третьего здесь — сознательный выбор, расширяющий пространство выразимых понятий. Цена выбора — часть привычных выводов: чем больше понятий выражает формализм, тем меньше общезначимых схем он признаёт. Аппарат — операции над степенями и срезы, связывающие нечёткость с классическими множествами.

Числами от 0 до 1 курс уже пользовался дважды: булева алгебра (только 0 и 1, глава 10) и вероятность (шанс события, глава 20). Нечёткие множества — третий способ: степень соответствия понятию. Курс начался с логических связок и двузначной истины, а заканчивается там, где «да или нет» уступает место степени. Как заменить бинарную границу степенью, не потеряв математическую строгость?

ИСТОРИЯ Термин, который был ругательством

Статья Заде «Fuzzy Sets»²⁴⁹ положила начало целой области. Слово «fuzzy» — «нечёткий», «расплывчатый» — звучало для точных наук как вызов, и в 1960-е и 1970-е годы «fuzzy thinking» было почти ругательством. Признание пришло от инженеров: промышленные контроллеры на нечёткой логике — от стиральных машин до метро в Сендае — работали, и термин перестал быть насмешкой. Этот шаг Заде был не первым: уже трёхзначная логика Яна Лукасевича²⁵⁰ допускала третье значение высказывания, промежуточное между истиной и ложью.

ОБЗОР ГЛАВЫ

Начнём с функции принадлежности — главного инструмента главы — и характеристик нечёткого множества: носителя, ядра, высоты и α -срезов. Теорема декомпозиции покажет, что по срезам множество восстанавливается целиком. Треугольные нечёткие числа перенесут ту же идею на числовую прямую. Затем введём операции пересечения, объединения и дополнения и увидим, что с ними исчезает закон исключённого третьего. Выбор конкретной операции окажется выбором t-нормы, а у каждой t-нормы будет собственная семантика «и». От множеств нечёткость перейдёт к логике: лингвистические переменные, правила вида «если жарко, то охлаждение сильное» и дефаззификация, превращающая нечёткий вывод в число, которым управляет регулятор. В конце посмотрим на применения — кластериза-

²⁴⁹L. A. Zadeh. «Fuzzy Sets». Information and Control, 1965.

²⁵⁰J. Łukasiewicz. «O logice trójwartościowej». 1920.

цию, принятие решений, обработку естественного языка — и на место нечёткости среди трёх способов работать с числами от 0 до 1.

После этой главы вы сможете:

1. строить функции принадлежности и характеристики нечётких множеств;
2. пользоваться α -срезами и теоремой декомпозиции;
3. применять операции с t -нормами и объяснять, почему исчезает закон исключённого третьего;
4. строить нечёткие правила и выполнять дефаззификацию;
5. объяснять связь нечётких множеств с классическими множествами и с вероятностью.

§ 35.1 Функция принадлежности

ОПРЕДЕЛЕНИЕ 35.1 Нечёткое множество

Нечётким множеством (fuzzy set) A на универсуме U называется функция

$$\mu_A : U \rightarrow [0, 1],$$

сопоставляющая каждому элементу $x \in U$ степень принадлежности $\mu_A(x)$.

Значение $\mu_A(x) = 0$ означает «точно не принадлежит», значение $\mu_A(x) = 1$ — «точно принадлежит», а промежуточные значения отвечают частичной принадлежности.

ЛОВУШКА Нечёткость $\neq$ вероятность

Степень принадлежности 0.7 легко прочесть как вероятность — это ошибка. Нечёткость говорит о мере соответствия понятию и к частоте события отношения не имеет. Различие двух родов неопределённости — в ремарке ниже.



Неопределённость бывает двух родов. Вероятность отвечает на вопрос «как часто?» и предполагает повторение: бросок монеты, шум в канале, доля дефектных деталей. Нечёткость отвечает на вопрос «в какой мере?» и повторения не требует: высокий рост, горячая вода, большой город — понятия, к которым объект относится по близости. Обе формализуют неполноту знания, но разную: неполноту статистическую и неполноту лингвистическую.

Классическое множество — частный случай: его характеристическая функция принимает только значения 0 и 1. Нечёткое множество *обобщает* классическое: его функция принадлежности (та же характеристическая функция) принимает значения во всём отрезке $[0, 1]$.

Пример Нечёткое множество «высокий рост»

$U = [140, 210]$ (рост в сантиметрах). Функция принадлежности могла бы выглядеть так:

$$\mu_{\text{high}}(x) = \begin{cases} 0 & \text{при } x \leq 160 \\ \frac{x-160}{30} & \text{при } 160 < x < 190 \\ 1 & \text{при } x \geq 190 \end{cases}$$

- Человек ростом 175 см: $\mu_{\text{high}}(175) = \frac{175-160}{30} = 0.5$ — «наполовину высокий».
- Человек ростом 185 см: $\mu_{\text{high}}(185) = \frac{185-160}{30} \approx 0.83$ — «почти высокий».

Форма функции принадлежности — инженерное решение, которое часто берётся из опроса экспертов.

Спросим сто банкиров, с какого балла кредит можно называть «хорошим». Хорошим сочли 750 все сто, 700 — половина, 650 — никто. Доли согласия прочерчивают кривую: по точкам $(750, 1)$, $(700, 0.5)$, $(650, 0)$ строится функция принадлежности слова «хороший кредит».

Соседние кривые перекрываются, и перекрытие — честное отражение того, что эксперты не договорились о границе. Три самые ходовые конфигурации — треугольная, трапецидальная и гауссова. Все три — на рис. 128.

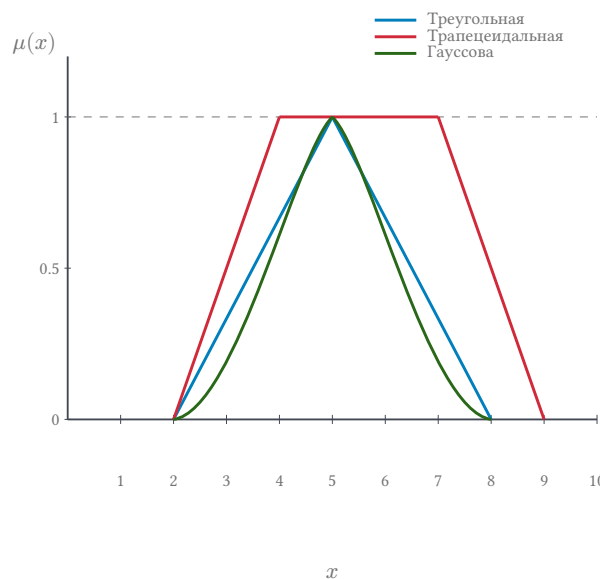


Рис. 128. Функции принадлежности: треугольная $(2, 5, 8)$, трапецидальная $(2, 4, 7, 9)$ и гауссова с центром 5.

Функция принадлежности задаёт множество целиком, но работать с ним удобнее через отдельные характеристики: где множество начинается, где держится на полной уверенности и насколько высоко поднимается. Носитель, ядро и высота отвечают на эти три вопроса по отдельности, а α -срез переводит нечёткое множество на язык классических — по одному множеству на каждый уровень α .

ОПРЕДЕЛЕНИЕ 35.2 Носитель

Носителем (support) нечёткого множества A называется классическое множество $\{x \in U \mid \mu_A(x) > 0\}$.

ОПРЕДЕЛЕНИЕ 35.3 Ядро

Ядром (core) называется $\{x \in U \mid \mu_A(x) = 1\}$.

ОПРЕДЕЛЕНИЕ 35.4 Высота

Высотой (height) называется $\max_{x \in U} \mu_A(x)$.

ОПРЕДЕЛЕНИЕ 35.5 α -срез

α -**срезом** называется классическое множество

$$\{x \in U \mid \mu_A(x) \geq \alpha\}.$$

α -срезы связывают нечёткие множества с классическими. При $\alpha = 0$ получаем весь универсум U , а при $\alpha = 1$ — ядро.

Пример Характеристики множества «молодой возраст»

Возьмём нечёткое множество «молодой возраст» на универсуме $U = [0, 100]$ (возраст в годах). Функция принадлежности:

$$\mu_{\text{young}}(x) = \begin{cases} 1 & \text{при } x \leq 25 \\ \frac{40-x}{15} & \text{при } 25 < x < 40 \\ 0 & \text{при } x \geq 40 \end{cases}$$

Носитель — возрасты с положительной степенью: $[0, 40)$. Ядро — возрасты со степенью 1: $[0, 25]$. Высота — наибольшая степень: 1.

α -срез собирается из условия $\mu(x) \geq \alpha$. Для $\alpha > 0$ он имеет вид

$$A_\alpha = [0, 40 - 15\alpha].$$

- При $\alpha = 0.5$: $[0, 32.5]$.
- При $\alpha = 0.25$: $[0, 36.25]$.
- При $\alpha = 0$: весь универсум $[0, 100]$.

Для функции принадлежности «высокого роста» из примера выше носитель — росты с положительной степенью, ядро — росты со степенью 1, а α -срез — все росты со степенью не ниже α . Любое нечёткое множество можно восстановить по семейству своих α -срезов (теорема декомпозиции).

ЛОВУШКА α -срез при $\alpha = 0 \neq$ ядро

Крайние срезы путают чаще всего. При $\alpha = 0$ срез A_0 — весь универсум U : условие $\mu_A(x) \geq 0$ выполнено для каждого элемента, в том числе с нулевой степенью. При $\alpha = 1$ срез A_1 — ядро: точки со степенью $\mu_A(x) = 1$. Носитель требует строгого неравенства $\mu_A(x) > 0$ и потому уже универсума. С A_0 он совпадает лишь когда ни одна степень не обращается в ноль.

ТЕОРЕМА 35.1 Декомпозиция по α -срезам

Любое нечёткое множество A на U однозначно восстанавливается по своим α -срезам:

$$\mu_A(x) = \sup \{ \alpha \in [0, 1] \mid x \in A_\alpha \}$$

для каждого $x \in U$.

Набросок доказательства

Зафиксируем x . Условие $x \in A_\alpha$ по определению среза означает неравенство

$$\mu_A(x) \geq \alpha.$$

Множество уровней α , при которых x принадлежит срезу, — это в точности интервал

$$[0, \mu_A(x)].$$

Супремум этого интервала равен его правому концу, то есть $\mu_A(x)$. Значит, степень принадлежности каждого элемента восстанавливается по срезам однозначно — а вместе с ней и всё множество.

Пример Восстановление по срезам

Вернёмся к множеству «молодой возраст» и возьмём возраст $x = 30$. Степень принадлежности: $\mu_{\text{young}}(30) = \frac{40-30}{15} = \frac{2}{3} \approx 0.67$. Возраст 30 попадает в срез A_α ровно для тех α , которые не превосходят $\frac{2}{3}$. Множество таких уровней — интервал $[0, \frac{2}{3}]$. Супремум этих уровней равен $\frac{2}{3}$, то есть $\mu_{\text{young}}(30)$. Теорема декомпозиции восстановила степень по срезам.

Замечание

Чайки рода *Larus* живут кольцом вокруг Северного полюса. Каждая популяция даёт плодовитое потомство с соседней. Крайние — сомкнувшие кольцо — уже не скрещиваются. Принадлежность к одному виду убывает с расстоянием: это склон.

Биолог в поле не может сказать: «вот здесь кончается один вид и начинается другой». Как никто не может сказать: «вот здесь кончается высокий человек и начинается невысокий». Чёткая граница — конструкция ума. Эволюция рисовала переходы, а не пороги.

Проверка Функция принадлежности

1. Может ли ядро нечёткого множества быть пустым, и что это говорит о его высоте?
2. Почему любая бинарная граница для понятия «высокий рост» произвольна, и как функция принадлежности устраняет эту произвольность?

§ 35.2 Нечёткие числа

Нечёткое множество на числовой прямой, моделирующее понятие «около x_0 », удобно задавать треугольной формой. **Треугольные нечёткие числа (TFN)** широко используются в принятии решений и нечёткой арифметике.

ОПРЕДЕЛЕНИЕ 35.6 Треугольное нечёткое число

Треугольным нечётким числом (TFN) называется нечёткое множество с функцией принадлежности, задаваемой тройкой (l, n, r) , где $l < n < r$:

$$\mu_{(l,n,r)}(x) = \begin{cases} 0 & \text{при } x \leq l \text{ или } x \geq r \\ \frac{x-l}{n-l} & \text{при } l < x \leq n \\ \frac{r-x}{r-n} & \text{при } n < x < r \end{cases}$$

Параметры тройки задают форму функции:

1. l — левая граница замыкания носителя.
2. n — мода и ядро множества, единственная точка со значением $\mu = 1$.
3. r — правая граница замыкания.

α -срез TFN — это замкнутый интервал:

$$A_\alpha = [l + \alpha(n - l), r - \alpha(r - n)].$$

- При $\alpha = 0$: $[l, r]$ — замыкание носителя. Сам носитель — открытый интервал (l, r) , ведь там $\mu > 0$.
- При $\alpha = 1$: $[n, n]$ — ядро. Ядро — одна точка.
- При $\alpha = 0.5$ получаем интервал половинной уверенности.

Пример Арифметика треугольных нечётких чисел

$A = (1, 2, 3)$ — «около 2». $B = (4, 5, 6)$ — «около 5».

Сложение (поточечное через α -срезы):

$$A + B = (1 + 4, 2 + 5, 3 + 6) = (5, 7, 9).$$

Результат — «около 7».

Вычитание:

$$A - B = (1 - 6, 2 - 5, 3 - 4) = (-5, -3, -1).$$

Умножение (приближённое):

$$A \cdot B \approx (1 \cdot 4, 2 \cdot 5, 3 \cdot 6) = (4, 10, 18).$$

Результат — «около 10».

Арифметика TFN обобщает интервальную арифметику. Вместо одного интервала мы имеем семейство вложенных интервалов — по одному на каждый уровень уверенности α .

§ 35.3 Операции над нечёткими множествами

Заде определил операции над нечёткими множествами через поточечные минимумы и максимумы.

ОПРЕДЕЛЕНИЕ 35.7 Пересечение

Пересечением нечётких множеств A и B на U называется нечёткое множество с функцией принадлежности $\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$.

ОПРЕДЕЛЕНИЕ 35.8 Объединение

Объединением нечётких множеств A и B на U называется нечёткое множество с функцией принадлежности $\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$.

ОПРЕДЕЛЕНИЕ 35.9 Дополнение

Дополнением нечёткого множества A на U называется нечёткое множество с функцией принадлежности $\mu_{\neg A}(x) = 1 - \mu_A(x)$.

Пересечение принадлежит обоим множествам настолько, насколько принадлежит менее принадлежащему из них, объединение — настолько, насколько принадлежит более принадлежащему, а дополнение — настолько, насколько не хватает до полной принадлежности.

Пример Дополнение множества «высокий рост»

Возьмём функцию принадлежности «высокого роста» из первого примера главы. Дополнение считается по формуле $\mu_{\neg A}(x) = 1 - \mu_A(x)$.

- При росте 150 см: $\mu_A(150) = 0$, значит $\mu_{\neg A}(150) = 1$ — «точно не высокий».
- При росте 175 см: $\mu_A(175) = 0.5$, значит $\mu_{\neg A}(175) = 0.5$ — «высокий» и «не высокий» в равной мере.
- При росте 200 см: $\mu_A(200) = 1$, значит $\mu_{\neg A}(200) = 0$.

Рис. 129 показывает действие трёх операций на примере двух треугольных функций принадлежности. Объединение берёт максимум, пересечение — минимум, дополнение — зеркальное отражение $1 - \mu$. Функции принадлежности μ_A и μ_B нарисованы тонкими линиями, результат каждой операции — жирной.

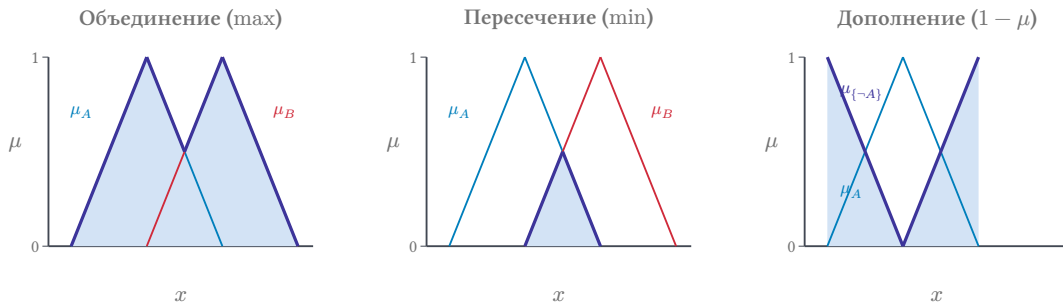


Рис. 129. Операции над нечёткими множествами: объединение, пересечение и дополнение.

Законы де Моргана — привычные факты о булевых операциях — переносятся и на нечёткий случай, но требуют отдельной проверки: операции изменились, и законы могли не сохраниться.

ТЕОРЕМА 35.2 Законы де Моргана для нечётких множеств

Для любых нечётких множеств A, B на U и любого элемента $x \in U$:

$$\mu_{\neg(A \cup B)}(x) = \mu_{\neg A \cap \neg B}(x),$$

$$\mu_{\neg(A \cap B)}(x) = \mu_{\neg A \cup \neg B}(x).$$

Доказательство

Докажем первый закон разбором случаев по соотношению $a = \mu_A(x)$ и $b = \mu_B(x)$. Второй закон доказывается аналогично. По определениям дополнения и объединения

$$\mu_{\neg(A \cup B)}(x) = 1 - \max(\mu_A(x), \mu_B(x)),$$

$$\mu_{\neg A \cap \neg B}(x) = \min(1 - \mu_A(x), 1 - \mu_B(x)).$$

Остаётся проверить, что для любых чисел $a, b \in [0, 1]$ верно равенство

$$1 - \max(a, b) = \min(1 - a, 1 - b).$$

Случай 1. Пусть $a \geq b$. Левая часть равна $1 - a$, а правая — $\min(1 - a, 1 - b) = 1 - a$, поскольку $1 - a \leq 1 - b$.

Случай 2. При $b > a$ рассуждение симметрично. В обоих случаях значения совпадают при каждом x , поэтому множества равны.

В отличие от классических множеств, для нечётких множеств закон исключённого третьего *не* выполняется. Классическая принадлежность — выключатель: свет либо горит, либо нет. Нечёткая — диммер: яркость плавно растёт от нуля до максимума, и промежуточных положений бесконечно много. Закон исключённого третьего — утверждение про выключатель, и диммер его нарушает: на половине яркости лампа одновременно и горит, и не горит.

$$\mu_{A \cup \neg A}(x) = \max(\mu_A(x), 1 - \mu_A(x)).$$

Если $\mu_A(x) = 0.6$, то

$$\mu_{A \cup \neg A}(x) = \max(0.6, 0.4) = 0.6,$$

а не 1. Аналогично, закон противоречия: $\mu_{A \cap \neg A}(x) = \min(0.6, 0.4) = 0.4$ — не ноль.

Именно это свойство позволяет нечёткой логике моделировать размытые понятия. Человек ростом 175 см одновременно «высокий» (0.5) и «не высокий» (0.5) — и в этом нет противоречия: про такого человека нельзя сказать ни «высокий», ни «не высокий» с уверенностью.

Замечание Что исчезает вместе с законом исключённого третьего?

Классическая теория множеств покоится на одном основании. Для любого элемента x и любого множества A либо $x \in A$, либо $x \notin A$, и третьего не дано. Это не эмпирический факт — это аксиоматический выбор. Из этого выбора вырастает всё здание бинарной логики: доказательство от противного, принцип двойственности, булева алгебра, разрешающие алгоритмы, которые всегда дают определённый ответ.

Нечёткие множества снимают это основание. Когда $\mu_{A \cup \neg A}(x) < 1$, закон исключённого третьего не выполняется. Когда $\mu_{A \cap \neg A}(x) > 0$, закон противоречия — тоже. Моделируя градуальные понятия — «высокий», «молодой», «комфортный» — мы теряем и доказательство от противного: оно требует, чтобы противоречия были невозможны, а при градуальной принадлежности они возникают.

Это структурное следствие определения, а не недостаток или преимущество. Расширяя пространство выразимых понятий, мы *сужаем* пространство допустимых логических ходов. Выбор между выразительностью и строгостью вывода остаётся за тем, кто строит модель.

Стандартные операции Заде ($\min/\max/1 - x$) — лишь один из возможных выборов. Математически это частный случай более общего семейства: *треугольных норм* (t-норм для пересечения) и *треугольных конорм* (s-норм для объединения).

От пересечения двух степеней принадлежности разумно требовать немногого. Для классических множеств таблица истинности AND диктует эти требования: элемент принадлежит пересечению тогда и только тогда, когда принадлежит обоим множествам. В нечётком случае пересечение — это функция двух степеней, и требования к ней диктует интуиция о самой операции.

Порядок факторов не должен влиять на результат. «Высокий и средний» — то же самое, что «средний и высокий», поэтому пересечение обязано быть коммутативным.

Расстановка скобок тоже не должна влиять: принадлежность пересечению трёх множеств не зависит от того, какое пересечение выполнено первым.

Рост одной из степеней не может уменьшить пересечение: если элемент стал увереннее принадлежать одному из множеств, его принадлежность пересечению не упадёт.

Полная принадлежность ничего не меняет: пересечение со степенью 1 возвращает исходную степень. Нулевая принадлежность обнуляет пересечение: элемент, точно не принадлежащий одному множеству, не принадлежит и пересечению.

Эти четыре требования — коммутативность, ассоциативность, монотонность и граничные условия — задают целый класс операций.

ОПРЕДЕЛЕНИЕ 35.10 t-норма

t-нормой называется функция $I : [0, 1]^2 \rightarrow [0, 1]$, удовлетворяющая аксиомам:

1. Коммутативность: $I(a, b) = I(b, a)$
2. Ассоциативность: $I(a, I(b, c)) = I(I(a, b), c)$
3. Монотонность: если $a \leq a', b \leq b'$, то $I(a, b) \leq I(a', b')$
4. Граничные условия: $I(a, 1) = a, I(a, 0) = 0$

ОПРЕДЕЛЕНИЕ 35.11 s-норма

s-нормой (конормой) называется функция $U : [0, 1]^2 \rightarrow [0, 1]$ с теми же аксиомами, кроме граничных условий: $U(a, 1) = 1, U(a, 0) = a$.

t-норма моделирует пересечение (AND), s-норма — объединение (OR). Дополнение всегда стандартное: $\mu_{\neg A}(x) = 1 - \mu_A(x)$.

Три основные пары (t-норма, s-норма) собраны в таблице 13.

Название	t-норма $I(a, b)$	s-норма $U(a, b)$	Применение
Заде (стандартная)	$\min(a, b)$	$\max(a, b)$	Универсальная, самая распространённая
Вероятностная	$a \cdot b$	$a + b - a \cdot b$	Статистические приложения
Лукасевича	$\max(0, a + b - 1)$	$\min(1, a + b)$	Многозначная логика (Łukasiewicz, 1920)

Таблица 13. Три основные пары t-норм и s-норм.

Для стандартной t-нормы ($\min$) пересечение $A \cap \neg A$ даёт ≤ 0.5 , как мы видели. Для вероятностной нормы $a \cdot (1 - a) \leq 0.25$ — ещё меньше. Для нормы Лукасевича $\max(0, a + (1 - a) - 1) = 0$ — закон противоречия восстановлен. Но идемпотентность потеряна: $I(a, a) = \max(0, 2a - 1)$, и при $a < 1$ результат меньше a .

Выбор t-нормы — содержание операции пересечения в конкретной задаче: от него зависит поведение нечёткой системы. Различие между тремя парами из таблицы 13 — в идемпотентности: в том, что происходит, когда один и тот же вход учитывается дважды.

Норма $\min$ идемпотентна: $I(a, a) = a$. Повторное пересечение ничего не меняет. Два датчика, измеряющих одну и ту же величину, в пересечении дают то же, что и один. Избыточность безопасна: дублирующие свидетельства не наказываются.

Произведение $a \cdot b$ идемпотентным не является: $I(a, a) = a^2$. При $a < 1$ результат меньше a . Согласие двух неуверенных мнений даёт меньшее значение, чем каждое по отдельности. Два эксперта с оценкой риска 0.7 каждый в совокупности дают 0.49 — с каждым подтверждением система становится осторожнее.

Норма Лукасевича нильпотентна: как только $a + b \leq 1$, пересечение обнуляется. Две половинные принадлежности по 0.5 в сумме дают 1 — и конъюнкция уже равна нулю. Это самый строгий из трёх AND: слабая суммарная уверенность ($a + b \leq 1$) обнуляет пересечение.

Для двух половинных принадлежностей по 0.5 три нормы дадут 0.5, 0.25 и 0 — выбор нормы меняет результат. Инженер решает, какое из этих чисел описывает его задачу.

Пример Пересечение нечётких множеств

A = «высокий рост» (как выше). B = «средний рост» с функцией

$$\mu_B(x) = \begin{cases} 1 & \text{при } 160 \leq x \leq 170 \\ \frac{180-x}{10} & \text{при } 170 < x < 180 \\ 0 & \text{иначе} \end{cases}$$

Для человека ростом 175 см: $\mu_A(175) = 0.5$. $\mu_B(175) = \frac{180-175}{10} = 0.5$. Пересечение: $\mu_{A \cap B}(175) = \min(0.5, 0.5) = 0.5$ — «одновременно высокий и средний» с половинной уверенностью. Объединение: $\mu_{A \cup B}(175) = \max(0.5, 0.5) = 0.5$.

Здесь $0.5 + 0.5 = 1$ — степени «высокий» и «средний» сошлись в сумме на единицу. Для других μ_A, μ_B и других x это не так.

Проверка Операции над нечёткими множествами

1. Законы де Моргана выполняются и для нечётких операций, но их пришлось проверять заново. Почему нельзя было просто перенести их с классических множеств?
2. Почему идемпотентность не входит в аксиомы t-нормы, хотя классическое пересечение идемпотентно?

Примечание Нечёткие отношения

Нечёткое отношение — это нечёткое множество на декартовом произведении $X \times Y$. Степень принадлежности пары (x, y) показывает, насколько x связано с y . На нечётких отношениях строятся нечёткий вывод и нечёткие контроллеры.

Пример Нечёткое отношение «х близко к у»

$X = Y = \{1, 2, 3, 4\}$ — четыре города на одной трассе. Степень близости: $\mu_R(x, y) = \max\left(0, 1 - \frac{|x - y|}{3}\right)$.

- Пара $(1, 1)$: $\mu_R = 1$ — город близок к себе.
- Пара $(1, 2)$: $\mu_R = 1 - \frac{1}{3} = \frac{2}{3}$.
- Пара $(1, 3)$: $\mu_R = 1 - \frac{2}{3} = \frac{1}{3}$.
- Пара $(1, 4)$: $\mu_R = 1 - 1 = 0$ — крайние города не близки.

Отношение симметрично: $\mu_R(x, y) = \mu_R(y, x)$, порядок аргументов не важен. Транзитивность для степеней требует, чтобы прямая связь не была слабее цепочки: $\mu_R(x, z) \geq \min(\mu_R(x, y), \mu_R(y, z))$ для любых x, y, z . Здесь требование нарушено: цепочка через город 2 даёт $\min(\mu_R(1, 2), \mu_R(2, 3)) = \frac{2}{3}$, а прямая связь $\mu_R(1, 3) = \frac{1}{3}$ слабее. При степенях 0 и 1 неравенство вырождается в классическую транзитивность (глава 5): если x связано с y и y с z , то x связано с z .

§ 35.4 Нечёткая логика

Нечёткая логика переносит идею градуальной принадлежности на логические утверждения. Вместо двух значений (истина/ложь) — бесконечная шкала $[0, 1]$.

Логические связки обобщаются:

- $\mu_{P \wedge Q} = \min(\mu_P, \mu_Q)$

- $\mu_{P \vee Q} = \max(\mu_P, \mu_Q)$
- $\mu_{\neg P} = 1 - \mu_P$
- $\mu_{P \rightarrow Q} = \min(1, 1 - \mu_P + \mu_Q)$ (импликация Лукасевича, есть и другие варианты)

Примечание Почему импликаций несколько

Импликация не выбирается отдельно от конъюнкции. Импликация J называется остатком t -нормы I , если $J(a, b)$ — наибольшее c с условием $I(c, a) \leq b$. Для нормы $\min$ остаток — импликация Гёделя: значение 1 при $a \leq b$ и значение b в противном случае. Цепь $[0, 1]$ с операциями $\min$, $\max$ и этой импликацией — алгебра Хейтинга (глава 34), только на непрерывной шкале вместо трёхэлементной цепочки. Импликация Лукасевича из списка выше — остаток нормы $\max(0, a + b - 1)$, и у каждой пары свой набор сохраняющихся классических законов.

ОПРЕДЕЛЕНИЕ 35.12 Лингвистическая переменная

Лингвистической переменной называется переменная, значениями которой являются нечёткие множества на некоторой числовой шкале.

Например, «температура» может принимать значения «холодно», «прохладно», «комфортно», «тепло», «жарко» — каждое из которых есть нечёткое множество на шкале градусов.

Пример Нечёткий контроллер кондиционера

Правила (Мамдани, 1974):

1. Если температура «жарко», то охлаждение «сильное».
2. Если температура «тепло», то охлаждение «слабое».
3. Если температура «комфортно», то охлаждение «выключено».

Зададим функции принадлежности термов «жарко» и «тепло» на шкале температуры:

$$\mu_{\text{жарко}}(x) = \begin{cases} 0 & \text{при } x \leq 25 \\ \frac{x-25}{5} & \text{при } 25 < x < 30 \\ 1 & \text{при } x \geq 30 \end{cases}$$

$$\mu_{\text{тепло}}(x) = \begin{cases} 0 & \text{при } x \leq 18 \\ \frac{x-18}{6} & \text{при } 18 < x \leq 24 \\ \frac{34-x}{10} & \text{при } 24 < x < 34 \\ 0 & \text{при } x \geq 34 \end{cases}$$

При температуре 27°C : $\mu_{\text{жарко}}(27) = \frac{27-25}{5} = 0.4$. $\mu_{\text{тепло}}(27) = \frac{34-27}{10} = 0.7$. Оба правила срабатывают пропорционально степени истинности условия. Результат

— взвешенная комбинация «сильного» и «слабого» охлаждения. Математически это сводится к операциям $\min$ и $\max$ над функциями принадлежности.

По такому принципу работали контроллеры стиральных машин и метро Сендай (Япония, 1987) — первая крупная промышленная система на нечёткой логике.

Тот же контроллер на чёткой логике — лестница из if-else: меньше 26 градусов — охлаждение выключено, меньше 28 — слабое, иначе — сильное. Правка одного порога тянет переписывание веток, а в нечёткой версии достаточно сдвинуть одну функцию принадлежности.

Правила остаются человекочитаемыми: эксперт читает «если жарко, то сильно», не расшифровывая таблицу порогов. Для программиста это главный практический аргумент нечёткости: сопровождаемость.

Агрегированная функция принадлежности, полученная на выходе правил, — это нечёткое множество. Но управляющий сигнал (например, мощность охлаждения) должен быть конкретным числом. Переход от нечёткого множества к числу называется *дефаззификацией*.

Метод Мамдани (max-min):

1. Правило i имеет firing strength f_i . Firing strength — степень истинности антецедента.
2. Выходная функция правила i «срезается» сверху на уровне f_i .

$$\mu_i^{\text{out}}(y) = \min(\mu_{\text{consequent}_i}(y), f_i)$$

3. Результаты всех правил объединяются через $\max$: $\mu^{\text{out}}(y) = \max_i \mu_i^{\text{out}}(y)$

Метод Ларсена (max-product, Larsen, 1980):

1. Отличается только способом комбинирования: вместо $\min$ используется произведение
2. $\mu_i^{\text{out}}(y) = \mu_{\text{consequent}_i}(y) \cdot f_i$
3. Результаты правил также объединяются через $\max$

При одном и том же наборе правил Mamdani²⁵¹ даёт «срезанную» функцию (плато), Larsen — «масштабированную» (пики той же формы, но меньшей высоты). Что предпочесть — решает задача.

После агрегации дефаззификация превращает нечёткое множество в одно число. Регулятору нужен один сигнал: заслонке, насосу, реле подаётся точка на оси.

Методов несколько, потому что кривую можно сжать в точку разными способами — и каждый способ по-своему отвечает на вопрос, что в функции главное. Центр тяжести усредняет по всей кривой: результат плавный, малые изменения входов не дёргают выход. Mean of Max берёт самое уверенное значение и не размазывает

²⁵¹E. H. Mamdani. «Application of Fuzzy Algorithms for Control of Simple Dynamic Plant». Proceedings of the IEEE, 1974.

решение по хвостам. Bisector делит площадь пополам: он близок к центру тяжести, но длинный хвост или выброс сдвигает его меньше.

Это дизайн-решение, и три метода дают три характера результата — усреднённый, решительный, устойчивый к выбросам.

Основные методы:

1. **Centroid** (центр тяжести): $y^* = \frac{\int y \cdot \mu(y) dy}{\int \mu(y) dy}$ — самый распространённый, даёт усреднённый результат
2. **Mean of Max** (МОМ): среднее точек, в которых $\mu(y)$ максимальна — «бери самое уверенное»
3. **Bisector**: вертикаль, делящая площадь под кривой пополам

Пример Дефаззификация трапециевидной функции

Дана трапеция с вершинами $(2, 0)$, $(2.5, 1)$, $(3, 1)$, $(4.5, 0)$.

1. Centroid: ≈ 3.06 — центр тяжести смещён вправо из-за длинного спада
2. MOM: $\frac{2.5+3}{2} = 2.75$ — середина плато
3. Bisector: 3.0 — ближе к центру, но не равен centroid

Разные методы дают разные результаты — инженер выбирает метод под задачу.

§ 35.5 Применения и связь с остальной книгой

Нечёткие множества — рабочий инструмент в областях, где нет чётких границ.

Кластеризация: алгоритм fuzzy c-means (Bezdek, 1981) — каждый объект принадлежит каждому кластеру со своей степенью, вместо жёсткого приписывания к одному кластеру. Алгоритм минимизирует

$$J_m = \sum_{i=1}^N \sum_{j=1}^C u_{ij}^m \|x_i - c_j\|^2,$$

где u_{ij} — степень принадлежности объекта i кластеру j , $m > 1$ — fuzzifier (обычно $m = 2$), c_j — центр кластера j .

При $m \rightarrow 1$ алгоритм стремится к жёсткому k-means. При $m \rightarrow \infty$ все объекты равномерно распределяются по кластерам.

Качество разбиения измеряется Fuzzy Partition Coefficient: $FPC = \left(\frac{1}{N}\right) \sum_i \sum_j u_{ij}^2$ — чем ближе к 1, тем «чётче» разбиение. Это даёт более информативную картину для объектов на границах кластеров.

Принятие решений: многокритериальный выбор с нечёткими весами критериев — обобщение метода анализа иерархий (АНП) на случай, когда предпочтения не точны.

Обработка естественного языка: квантификаторы «несколько», «много», «почти все» задаются как нечёткие множества на шкале количества или доли.

Примечание Нечёткость и вероятность — не одно и то же

Вероятность (глава 20) измеряет *шанс* события, которое ещё не произошло. После броска монеты неопределённость исчезает — выпал орёл или решка.

Нечёткость измеряет *степень соответствия* понятию для уже известного факта. Рост человека уже измерен — 175 см. Неопределённость — в *классификации*: измерив рост, мы всё ещё не знаем, насколько он «высокий».

Формально вероятность подчиняется закону исключённого третьего и закону противоречия: $P(A \cup \neg A) = 1$, $P(A \cap \neg A) = 0$. Степень принадлежности им не подчиняется: $\mu_{A \cup \neg A}(x) = \max(\mu_A(x), 1 - \mu_A(x))$. При промежуточных $\mu_A(x)$ эта величина меньше единицы. Пересечение $\mu_{A \cap \neg A}(x) = \min(\mu_A(x), 1 - \mu_A(x))$ — больше нуля. Аксиоматика разная — применения разные.

Нечёткие множества обобщают классическую теорию множеств (глава 4) и булеву алгебру (глава 10), заменяя бинарную принадлежность градуальной. Это последняя тема нашего курса — переход к областям, где границы перестают быть чёткими.

§ 35.6 * Интерполяция и схема Сугено

Мы разобрали схему Мамдани и способы дефаззификации. Теперь добавим то, чего в базовой схеме нет: взгляд на вывод как на интерполяцию, сравнение с классическим регулятором и схему Сугено.

Нечёткий вывод — это интерполяция, только узлы заданы словами. Правила «если жарко, то сильно» и есть модель, собранная из опыта эксперта.

Классический регулятор требует уравнения процесса: теплопереноса, динамики поезда, риска заёмщика. Нечёткому достаточно слов: там, где механизм неясен или слишком сложен, правила собираются из опыта. Физическую модель часто можно не строить — в этом практическая ценность подхода.

В схеме Мамдани заключения правил — нечёткие множества. Это интуитивно: эксперт говорит «если жарко, то охлаждение сильное», и «сильное» — размытое понятие.

Схема Сугено (Sugeno, 1985) устроена иначе: заключение правила — функция от входов. Если температура x , то мощность охлаждения — линейная функция $y = ax + b$. Выход системы — взвешенная сумма этих функций, где веса — firing strength правил.

Пример Схема Сугено на числах

Вернёмся к кондиционеру из раздела о нечёткой логике. Правила Сугено: если температура «жарко», то мощность $y = 2x - 40$. Если температура «тепло», то мощность $y = x - 20$. При температуре 27° : $\mu_{\text{жарко}}(27) = 0.4$. $\mu_{\text{тепло}}(27) = 0.7$ — обе степени посчитаны в примере с кондиционером. Заключение первого правила: $y_1 = 2 \cdot 27 - 40 = 14$. Заключение второго: $y_2 = 27 - 20 = 7$. Выход — взвешенная сумма:

$$y = \frac{0.4 \cdot 14 + 0.7 \cdot 7}{0.4 + 0.7} = \frac{10.5}{1.1} \approx 9.5.$$

Ответ 9.5 ближе к 7, чем к 14: правило «тепло» сработало сильнее и тянет результат к себе.

Сугено быстрее считает и лучше поддаётся автоматической настройке параметров. Мамдани прозрачнее для человека. Выбор между ними — вопрос требований: скорость против прозрачности.

Какой бы метод дефаззификации ни завершал конвейер, на выходе — конкретное число. Такой конвейер работает в ABS, автофокусе камер и климат-контроле: входные данные размыты, а решение должно быть точным.

§ 35.7 Итоги главы

Парадокс кучи показал, что для многих понятий бинарная граница произвольна, и ответ — градуальная принадлежность. Функция $\mu_A : U \rightarrow [0, 1]$ измеряет степень соответствия понятию, а операции над функциями переносят на нечёткие множества алгебру классических. Нечёткость отвечает на вопрос «в какой мере?», а не «как часто?» — этим она отличается от вероятности. Её функция принадлежности принимает значения во всём отрезке $[0, 1]$, тогда как классическое множество — частный случай с характеристической функцией в $\{0, 1\}$.

α -срезы и теорема декомпозиции связывают новую конструкцию со старой: нечёткое множество восстанавливается по семейству обычных. Выбор t-нормы оказывается выбором семантики «и» — пересечение по $\min$ отличается от пересечения по произведению. Так понятия без резкой границы получают точный язык: носитель, ядро, высота и треугольные нечёткие числа задают форму заранее.

Дальше язык заработал: лингвистические переменные и правила «если жарко, то охлаждение сильное» складываются в конвейер Мамдани, который превращает нечёткий вывод в число для регулятора — так работали контроллеры стиральных машин и метро Сендай. Инженерная ценность — в сопровождаемости: правила читает эксперт, а правка одной функции принадлежности заменяет переписывание порогов.

Нечёткие множества — третий способ работать с числами от 0 до 1 рядом с булевой алгеброй (глава 10) и вероятностью (глава 20): переключатели, шансы и степени принадлежности. Классическое множество (глава 4) возвращается, как только степеням принадлежности запрещены промежуточные значения, и курс замыкает круг: от закона исключённого третьего до его осознанного нарушения.

Проверка Проверьте себя

1. Чем нечёткость отличается от вероятности?
2. Что такое α -срез и зачем он нужен? Как восстановить нечёткое множество по семейству его α -срезов?
3. Почему для нечётких множеств не выполняется закон исключённого третьего? Приведите численный пример.
4. Что такое t-норма и чем пересечение по $\min$ отличается от пересечения по произведению?
5. Как работает дефаззификация? Почему регулятору нельзя подать самую агрегированную функцию принадлежности?
6. Что такое лингвистическая переменная и как она участвует в работе нечёткого контроллера?

Конец курса

Введение книги поставило вопрос: что значит описывать точно и где проходит граница точного описания. Курс отвечал по частям. Первой стала задача языка: записывать дискретные структуры — от логических связей до множеств, отношений, функций и графов. Затем добавился счёт: комбинаторика для числа вариантов, теория чисел для арифметики остатков, вероятность для случайных событий. Дальше описание превратилось в вычисление: автоматы и грамматики распознали языки, а машины Тьюринга определили, что вообще вычислимо. Обнаружились и границы: проблема остановки не решается никаким алгоритмом, а закон исключённого третьего в интуиционистской логике перестал быть тавтологией.

Последняя глава отвечает на собственный вопрос: как заменить бинарную границу степенью, не потеряв математическую строгость. Ответ дают конкретные конструкции: функция принадлежности $\mu_A : U \rightarrow [0, 1]$ задаёт степени, а t-нормы аксиоматизируют операции над ними. α -срезы и теорема декомпозиции восстанавливают множество целиком и держат связь с классической теорией.

Применения указывают продолжение: контроллеры на нечёткой логике работают от стиральных машин до метро в Сендае. Градуальные степени тем временем вошли в кластеризацию, принятие решений и обработку естественного языка. Словарь программиста, инженера и теоретика, обещанный во введении, собран — от таблиц истинности до t-норм. Этим словарём описывают и дискретные структуры, и понятия без резких границ.

§ 35.8 Упражнения

Функция принадлежности и нечёткие числа.

1. Определите функцию принадлежности нечёткого множества «средний возраст» на универсуме $U = [0, 120]$ (возраст в годах). Пусть ядро — интервал $[35, 50]$. Носитель — интервал $[20, 65]$. Используйте кусочно-линейную функцию. Вычислите $\mu(30)$, $\mu(42)$, $\mu(60)$.
2. Даны два нечётких множества A, B на универсуме $U = \{x_1, x_2, x_3, x_4\}$:

$$A = \left\{ \frac{0.3}{x_1}, \frac{0.8}{x_2}, \frac{1.0}{x_3}, \frac{0.1}{x_4} \right\},$$

$$B = \left\{ \frac{0.9}{x_1}, \frac{0.4}{x_2}, \frac{0.6}{x_3}, \frac{0.0}{x_4} \right\}.$$

Вычислите $A \cap B$, $A \cup B$, $\neg A$.

3. Для нечёткого множества «высокий рост» из текста главы найдите носитель, ядро и высоту.
4. Для треугольного нечёткого числа $(1, 4, 7)$ вычислите α -срезы. Для уровней $\alpha = 0, 0.5, 1$.

Операции и t-нормы.

5. Выполните арифметические операции над треугольными нечёткими числами: $A = (2, 5, 8)$. $B = (1, 3, 5)$. Найдите $A + B$, $A - B$.
6. Сравните три t-нормы (Заде, вероятностная, Лукасевича) на одном примере: $a = 0.6$, $b = 0.7$. Вычислите $I(a, b)$ для каждой и объясните разницу.
7. Проверьте, выполняется ли дистрибутивность для стандартных операций Заде: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. Докажите или приведите контрпример.
8. * Докажите законы де Моргана для нечётких множеств: $\neg(A \cup B) = \neg A \cap \neg B$ и двойственный. Почему их пришлось доказывать заново, хотя для классических множеств они выполняются?

Нечёткая логика и нечёткий вывод.

9. Постройте нечёткий контроллер для простой системы: если температура «низкая», то обогрев «сильный». Если «комфортная», то обогрев «слабый». Задайте

функции принадлежности для «низкая», «комфортная», «сильный» и «слабый». Для входной температуры 22° вычислите firing strength каждого правила.

10. * Докажите теорему декомпозиции. Любое нечёткое множество A восстанавливается по семейству своих α -срезов. А именно: $\mu_A(x) = \sup \{\alpha \in [0, 1] \mid x \in A_\alpha\}$.
11. * Для выходной функции принадлежности, заданной трапецией с вершинами $(0, 0)$, $(2, 1)$, $(4, 1)$, $(5, 0)$, вычислите centroid (центр тяжести) и Mean of Max. Объясните, почему результаты различаются.
12. ** Покажите, что для стандартных операций Заде закон исключённого третьего не выполняется, а закон противоречия — тоже. Приведите численный пример. Затем покажите, что для t -нормы Лукасевича закон противоречия выполняется. Но нарушается идемпотентность: $I(a, a) < a$ при $a < 1$. Какое практическое следствие имеет выбор между этими t -нормами в нечётком контроллере?
13. * Разберите «доказательство» того, что степень принадлежности — это вероятность: «если $\mu_A(x) = 0.7$, то при многократных наблюдениях элемент x принадлежит множеству A в 70% случаев, значит нечёткое множество — это вероятностное распределение». Где именно ошибка, и что теряется, если прочитать степень принадлежности как вероятность?

ПРОЕКТ Сравнение порогового и нечёткого регуляторов

Постройте два регулятора термостата — чёткую лестницу и нечёткий регулятор Мамдани — и сравните их на шумном дрейфующем входе. Исходная точка — дребезг бинарного правила «если жарко, то полная мощность» на границе: чуть теплее порога — полная мощность, чуть холоднее — ноль, и каждый переход границы даёт скачок управляющего сигнала. Нечёткий регулятор заменяет переключатель градуальной функцией принадлежности: правило срабатывает пропорционально степени истинности условия, и выход меняется плавно.

① Нечёткие множества и t -нормы

Реализуйте функции принадлежности (треугольная, трапецеидальная) и операции Заде: объединение ($\max$), пересечение ($\min$), дополнение ($1 - \mu$). Носитель, ядро, высота и α -срезы, а затем восстановите множество по срезам (теорема декомпозиции). Затем сравните три пары (t -норма, s -норма), задающие конъюнкцию и дизъюнкцию: Заде ($\min, \max$), вероятностная ($a \cdot b, a + b - a \cdot b$) и Лукасевича ($\max(0, a + b - 1), \min(1, a + b)$). Проверьте аксиомы, законы де Моргана и невыполнение закона исключённого третьего.

② Два регулятора

Соберите чёткую лестницу из if-else и нечёткий регулятор Мамдани с t -нормой и дефаззификатором как параметрами. Задайте базу из 15 правил «если ошибка X и производная Y , то мощность Z ».

③ Методы дефаззификации

Дефаззификация превращает нечёткое множество в одно число, и разные методы делают это по-разному. Реализуйте центр тяжести, среднее из максимумов и Bisector и сравните их на одной функции принадлежности. На какой форме все три метода дают одно число, а на какой — расходятся? (Подумайте о симметрии формы.) Объясните, почему: центр тяжести — центр площади, среднее из максимумов — середина одного связного плато максимумов, Bisector делит площадь пополам.

④ *Дребезг и параметры*

Подайте дрейфующий шумный вход, многократно пересекающий порог, и измерьте амплитуду осцилляций выхода, число переключений и суммарное управляющее усилие каждого регулятора. Затем прогоните три t -нормы и три дефаззификатора у границы: управляющие сигналы около уставки, степень срабатывания правил и метрики дребезга для девяти пар.

⑤ *Парадокс кучи*

Проведите цепочку «1 зерно, ..., 10000» через чёткий порог и нечёткую «кучу». Покажите, что у порога есть ровно одно зерно, где куча «возникает», а у нечёткой принадлежности подъём плавный, без скачка.

⑥ *Контроллер Сугено*

Мамдани использует нечёткое правило с нечётким следствием и дефаззификацией площади. Сугено заменяет следствие функцией от входов: правило — «если x есть A , то $y = ax + b$ ». Реализуйте Сугено-контроллер: фаззификация, веса правил, выход как взвешенное среднее следствий. Сравните с Мамдани на том же примере: где выход различается и почему.

Итог: измеренное сравнение двух регуляторов на одном входе: метрики дребезга, таблица девяти пар (t -норма, дефаззификатор) и разбор парадокса кучи как цены бинарной границы.

XXXVI

Теория категорий

Элемент декартова произведения множеств восстанавливается по проекциям: из $\pi_1(x) = \pi_1(y)$ и $\pi_2(x) = \pi_2(y)$ следует $x = y$. Состояние прямого произведения автоматов восстанавливается по компонентам: из равенства обеих компонент следует равенство состояний. Утверждения говорят о разном — о множествах и о машинах, а доказательства совпадают дословно: это одно рассуждение о парах и проекциях, и язык глав определит его как равенство композиций.

За первым совпадением тянутся другие. Конъюнкция $p \wedge q$ выпускает два правила вывода: из $p \wedge q$ следует p и следует q . Обратное тоже верно: любое высказывание, из которого следуют и p , и q , влечёт и $p \wedge q$. Пересечение подмножеств лежит в каждом из них, и любое подмножество, лежащее в обоих, лежит и в пересечении. Наибольший общий делитель двух чисел делится на любой их общий делитель. Материал и здесь разный — формулы, множества, числа — а устройство одно: объект с парой стрелок к двум данным объектам, через который пропускается любая другая такая пара.

У одной конструкции набралось пять имён, и каждая глава доказывала её заново: у множеств свои элементы, у формул свои таблицы истинности, у чисел свои делители, у автоматов свои переходы. Общего языка для повторяющегося рассуждения не было. Этот язык вырос из задачи с другого конца математики. В 1940-е топологи различали изоморфизмы естественные и зависящие от выбора базиса: разница была очевидна каждому, точного определения у неё не было. Чтобы его дать, пришлось определить, что такое естественность, — и определение потребовало нового типа математического объекта, в котором сами построения и их соответствия стали предметом теории.

Язык оказался шире задачи. Тот же словарь сегодня описывает типы и функции в языках программирования: контейнеры с операцией `map`, полиморфные функции, цепочки вычислений с отказами. Как один язык делает повторяемость конструкций видимой до всяких вычислений — и почему этим же языком оказались типы и функции в программировании?

ИСТОРИЯ Эйленберг, Мак Лейн и естественность

Сэмюэл Эйленберг и Сондерс Мак Лейн познакомились в 1940 году в Мичигане: тополог и алгебраист, оказавшиеся над одной задачей — гомологиями, алгебраическими инвариантами, сопоставляющими пространству с операцией

умножения группу. Статья 1945 года «General Theory of Natural Equivalences»²⁵² начиналась с наблюдения, что слово «естественный» в математике использовали постоянно, не определяя. Изоморфизм конечномерного векторного пространства V с его двойственным (пространством всех линейных функций на V) строится по выбору базиса, а с дважды двойственным — без всякого базиса: второй изоморфизм «естественный», первый — нет. Разница ощущалась всеми, но не была формализована.

Формализация потребовала ввести сразу три понятия: категорию, функтор и естественное преобразование. Оба слова позаимствованы у философии: «категория» — у Аристотеля и Канта, «функтор» — у Рудольфа Карнапа, из его книги «Логический синтаксис языка». Мак Лейн признавался, что брал эти слова «с удовольствием украв их у философов»²⁵³.

Дальше язык зажил собственной жизнью. Александр Гротендик в 1960-е перестроил на нём алгебраическую геометрию целиком. Уильям Ловер в 1970-е вместе с Майлом Тирни построил элементарные топосы — категории, внутри которых можно вести математику — и связал их с интуиционистской логикой. С 2010-х идёт прикладная волна: категории нашли применение в базах данных, теории сетей, вероятностях и машинном обучении, а сама дисциплина получила имя прикладной теории категорий.

ОБЗОР ГЛАВЫ

Глава строит язык, в котором объект определяется не элементами, а стрелками. Сначала — знакомые миры: множества с функциями, моноиды, порядки и графы, на которых выводится общее устройство, фиксируемое затем определением категории. Затем два способа сравнивать категории между собой: функторы, переводящие объекты вместе со стрелками, и естественные преобразования, сравнивающие функторы. На этой тройке держится всё остальное: универсальные свойства и пределы, сводящие произведение, сумму и наибольший общий делитель к одной диаграмме, лемма Йонеды, сопряжённые функторы и монады — вплоть до категорной семантики типов. Факультативные разделы уводят в стороны, где язык стрелок работает в полный рост: стринг-диаграммы, свёртки списков, метрики, нечёткость и топосы с внутренней логикой.

После этой главы вы сможете:

1. проверять, что множество с композицией образует категорию, и узнавать категории в моноидах, порядках и графах;

²⁵²S. Eilenberg, S. Mac Lane. «General Theory of Natural Equivalences». Transactions of the American Mathematical Society, 1945.

²⁵³S. Mac Lane. «Categories for the Working Mathematician». 2-е изд., Springer, 1998.

2. отличать изоморфизм, мономорфизм и эпиморфизм от их теоретико-множественных двойников и пользоваться двойственностью;
3. строить функторы и естественные преобразования и проверять их законы;
4. формулировать универсальные свойства и узнавать произведения, копроизведения и терминальные объекты в разных категориях;
5. объяснять лемму Йонеды и сопряжённые функторы на примерах, включая связки Галуа из анализа программ;
6. связывать монады с композицией эффективных вычислений и читать категорную семантику простых типов.

§ 36.1 Категория и её стрелки

Композиция — самая устойчивая операция во всей математике. Если функция f переводит A в B , а g — B в C , за f можно выполнить g . Если a делит b , а b делит c , то a делит c . Если один путь в графе заканчивается там, где начинается другой, их можно пройти подряд. Во всех трёх случаях цепочка шагов не зависит от расстановки скобок — и моноид (глава 12) был ровно об этом: множество, на котором композиция определена всегда и имеет нейтральный элемент.

Миры различаются тем, какие пары шагов допускаются к композиции. В моноиде перемножаются любые два элемента, функции стыкуются, только когда выход одной служит входом другой, пути сцепляются концом в начало. Общее правило уже видно: у каждого шага есть вход и выход, и складываются шаги при совпадении стыка.

Начнём с мира, где всё уже доказано, — с функций между множествами. Объекты здесь — множества, стрелки — функции с началом и концом, и к композиции допускаются только пары с совпавшим стыком. Композиция — это последовательное применение g после f , и оба её свойства нам уже знакомы (глава 6): тройные группировки равны, а тождественные функции ничего не меняют. Занесём этот мир в опись: объекты, стрелки с началом и концом, композиция при совпадении стыка, ассоциативность, тождественные стрелки.

Пример Моноид: один объект, стрелки-элементы

Возьмём моноид $(M, \star, e)$ (глава 12). Объект будет один — точка $\tilde{M}$, без внутреннего устройства. Стрелками объявим сами элементы M : элемент $a \in M$ получает вторую роль — роль стрелки $a : \tilde{M} \rightarrow \tilde{M}$. Композиция определена операцией моноида: стрелка b после стрелки a — это элемент $b \star a$, который сам служит стрелкой, $b \circ a = b \star a$. Тожественная стрелка — нейтральный элемент в роли стрелки, $\text{id}_{\tilde{M}} = e$: равенства $e \star a = a \star e = a$ превращаются в закон тождества. Ассоциативность композиции — это в точности ассоциативность операции. Таблица Кэли становится таблицей композиции стрелок без единой правки — все пять пунктов описи на месте.

Петли различаются именами, а не геометрией: рисунок разводит их вокруг точки лишь затем, чтобы глаз мог пересчитать. Две петли с разными именами — разные стрелки, как две разные функции между одной парой множеств. Композиция здесь — операция, заданная таблицей Кэли: в частности, $e \circ a = a \circ e = a$. Единственный объект делает стык совпадающим всегда, поэтому компонуются любые две стрелки. Композиция при этом не порождает новых стрелок: результат операции — снова элемент M , и стрелка этого элемента уже существует. Интуиция «пройти по стрелке» верна для путей следующего примера, где стрелки — настоящие дороги. Петли моноида — элементы в роли стрелок, и маршрутного смысла у них нет.

Один объект и его петли-стрелки — на рис. 130.

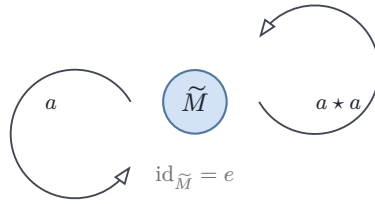


Рис. 130. Один объект, стрелки — элементы, композиция — операция.

Пример Порядок: стрелка вместо неравенства

Возьмём частично упорядоченное множество $(P, \leq)$ (глава 8). Объекты — элементы P , и стрелку $a \rightarrow b$ проведём ровно в том случае, когда $a \leq b$. Композиция — транзитивность: из $a \leq b$ и $b \leq c$ следует $a \leq c$, тождественная стрелка — рефлексивность $a \leq a$. Между любыми двумя объектами получается не более одной стрелки, и вся информация о мире заключена в том, какие стрелки существуют.

Как выглядят стрелки этого мира, показывает цепочка на рис. 131. Пунктирная стрелка $0 \rightarrow 2$ — не дополнительная: в тонкой категории она единственная и равна композиции двух сплошных. Тождественные петли $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 2$ на рисунках таких категорий не изображаются.

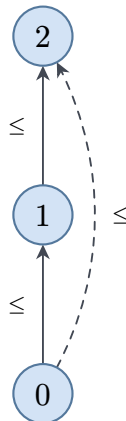


Рис. 131. Цепочка $0 \leq 1 \leq 2$: каждая стрелка означает отношение, пунктирная — транзитивность.

Описи трёх миров совпали во всех пунктах: материал разный, скелет один. Выпишем этот скелет отдельным определением и дадим ему имя.

ОПРЕДЕЛЕНИЕ 36.1 Категория

Категория $\mathcal{C}$ состоит из:

1. совокупности **объектов** $A, B, C, \dots$;
2. совокупности **стрелок** (морфизмов) $f : A \rightarrow B$, где объект A называется *началом*, а B — *концом* стрелки f ;
3. операции **композиции**, сопоставляющей паре стрелок $f : A \rightarrow B$ и $g : B \rightarrow C$ стрелку $g \circ f : A \rightarrow C$,

при условии, что для любых компонуемых стрелок выполнены два закона:

- **ассоциативность**: $h \circ (g \circ f) = (h \circ g) \circ f$;
- **тождество**: у каждого объекта A есть стрелка $\text{id}_A : A \rightarrow A$ с $\text{id}_B \circ f = f = f \circ \text{id}_A$ для каждой $f : A \rightarrow B$.

Композиция читается справа налево: $g \circ f$ означает «сначала f , затем g » — так же, как в записи $g(f(x))$ функция f применяется первой. Тожественная стрелка единственна: для двух кандидатов e, e' на роль id_A выполнено $e = e \circ e' = e'$ — тот же аргумент, что доказывает единственность нейтрального элемента моноида (глава 12). В этой конструкции объекты — только бирки. У них нет «внутренности», которая вскрывалась бы вопросом «из чего состоит»: весь материал категории разложен по стрелкам. Такое решение самое непривычное в определении и самое рабочее на практике: рассуждению, опирающемуся на одну композицию, незачем знать, что внутри объектов, и оно автоматически верно во всех категориях сразу.

Теперь имена встают на места. Мир множеств и функций — это категория Set . Моноид — категория с одним объектом, все стрелки которой — петли. Мир порядка — *тонкая* категория: между любыми двумя объектами не более одной стрелки, и вся информация заключена в наличии стрелок. Решётка (глава 8) — тонкая категория, в которой инфимум и супремум окажутся произведением и копроизведением. Об этом расскажет секция 36.5.

Словарь моноида и категории с одним объектом симметричен. Пример выше переводил моноид в категорию. Теперь проделаем обратный путь — от произвольной категории с единственным объектом к моноиду.

УТВЕРЖДЕНИЕ 36.1 Категория с одним объектом — моноид

Пусть $\mathcal{C}$ — категория с единственным объектом A . Тогда совокупность всех стрелок $\mathcal{C}$ с композицией в роли операции и стрелкой id_A в роли нейтрального элемента — моноид.

Доказательство

Докажем проверкой аксиом моноида (глава 12). Операция тотальна: начало и конец любой стрелки совпадают с A , поэтому любые две стрелки композиемы, и композиция снова даёт стрелку той же совокупности. Ассоциативность — аксиома категории. Нейтральность id_A — аксиома тождества: $\text{id}_A \circ f = f = f \circ \text{id}_A$ для каждой стрелки f .

Оба перевода отменяют друг друга. Из $\tilde{M}$ доказательство возвращает тот же моноид: стрелки категории — элементы M , композиция — операция $\star$. Из моноида стрелок пример выше строит категорию с тем же единственным объектом и теми же стрелками — исходную категорию. Единственный объект — бирка, а бирка в единственном экземпляре не добавляет информации: вся структура живёт в петлях.

Словарь доходит и до кода. Типокласс — контракт на категорию с одним объектом. Носитель здесь — тип, операция и нейтральный элемент — его методы, а законы — в точности аксиомы категории.

```
// Типокласс моноида --- контракт на категорию с одним объектом.
trait Monoid {
  fn e() -> Self; // нейтральный элемент =
  тождественная стрелка
  fn star(self, other: Self) -> Self; // операция = композиция стрелок
}
// a.star(b).star(c) == a.star(b.star(c)) --- ассоциативность
// a.star(Self::e()) == a && Self::e().star(a) == a --- тождество

impl<A> Monoid for Vec<A> {
  fn e() -> Self { vec![] }
  fn star(mut self, other: Self) -> Self { self.extend(other); self }
}
```

`impl` для `Vec` — конкретный экземпляр: свободный моноид из 12, списки — петли, конкатенация — композиция, пустой список — тождественная стрелка. Законы типокласса компилятор не проверяет — они живут контрактами рядом с кодом. Для конечного моноида проверка законов — перебор по таблице Кэли. Ровно этим занят проект в конце главы.

Словарь доходит и до отображений: гомоморфизм моноидов из 12 — функтор между категориями с одним объектом, и обратно. Единственный объект переходит в единственный сам собой, а сохранение операции и нейтрального элемента — в точности два закона функтора (секция 36.3). Занесём словарь в таблицу.

Моноид $(M, \star, e)$	Категория $\tilde{M}$
носитель M	совокупность всех стрелок $\tilde{M}$
элемент $a \in M$	стрелка $a : \tilde{M} \rightarrow \tilde{M}$
операция $b \star a$	композиция $b \circ a$
нейтральный элемент e	тождественная стрелка $\text{id}_{\tilde{M}}$
ассоциативность	аксиома ассоциативности

Моноид $(M, \star, e)$	Категория $\tilde{M}$
нейтральность e	аксиома тождества
обратимый элемент	изоморфизм
гомоморфизм моноидов	функтор

Пример Граф порождает свободную категорию

Ориентированный граф задаёт объекты и образующие стрелки, но не композицию. Добавим её принудительно: стрелками объявим все *пути* — конечные последовательности рёбер, в которых конец каждого ребра совпадает с началом следующего. Композиция — приписывание одного пути к концу другого, тождество — пустой путь в каждой вершине. Ассоциативность здесь — соглашение о скобках при конкатенации последовательностей, уже встречавшееся для списков в главе 12.

Полученная категория называется *свободной*: в ней нет никаких соотношений между путями, кроме склейки концов — каждый новый путь обязан появиться, и лишних не возникает.

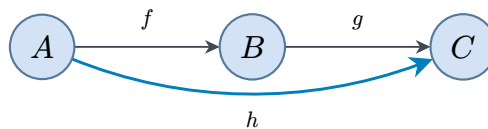


Рис. 132. Свободная категория графа: путь $g \circ f$ становится самостоятельной стрелкой.

Теперь прочтём рёбра графа как шаги, которые можно проходить один за другим. Получится категория, где вдобавок к рёбрам появились составные пути: из A в C ведёт новый морфизм $g \circ f$ — «пройти f , затем g ». Регулярные выражения (глава 23) перечисляют такие пути, а эpsilon-переходы недетерминированного конечного автомата (НКА) — формально добавленные стрелки, не меняющие распознаваемый язык.

Стрелки каждого примера можно собрать в данные — и это второй столп языка после самого определения категории.

ОПРЕДЕЛЕНИЕ 36.2 **hom-множество**

Множество всех стрелок категории $\mathcal{C}$ с началом A и концом B называется **hom-множеством** пары и обозначается $\mathcal{C}(A, B)$.

Читается запись так: «стрелки категории $\mathcal{C}$ из A в B ». Имя — от «homomorphism»: гомоморфизмы (глава 12) были стрелками между моноидами, и имя перешло на произвольные стрелки. Три мира дают три образца чтения. В $\mathbf{Set}$ hom-множество — знакомое множество всех функций: $\mathbf{Set}(\{1, 2\}, \{a, b, c\})$ насчитывает девять функций, потому что каждый из двух элементов области выбирает один из трёх образов. В тонкой категории порядка hom-множество работает индикатором неравенства:

$\mathcal{C}(a, b)$ одноэлементно при $a \leq b$ и пусто иначе. В категории моноида M единственное hom-множество — это сам M : таблица Кэли и есть таблица композиции стрелок.

Через hom-множества категория перезаписывается целиком — и в этой записи видна её алгебраическая суть. Композиция становится семейством умножений, по одному на каждый промежуточный объект:

$$\circ_{A,B,C} : \mathcal{C}(A, B) \times \mathcal{C}(B, C) \rightarrow \mathcal{C}(A, C).$$

Тождество — семейством отметок единицы $\star \mapsto \text{id}_A \in \mathcal{C}(A, A)$. hom-множества — рабочая валюта всей главы: функторы действуют на них, лемма Йонеды пересчитывает через них естественные преобразования, а сопряжение задаётся биекцией между двумя hom-множествами.

Примечание Как читать диаграммы

Рисунок 132 — это не иллюстрация, а утверждение. Узлы — объекты, стрелки — морфизмы, и диаграмма *коммутирует*: любые два пути по стрелкам из одного узла в другой дают равные композиции.

Проверка коммутативности треугольника на рис. 133 — это вычисление одного равенства $g \circ f = h$, и вся сила диаграмм в том, что сложные цепочки равенств сворачиваются в один взгляд на картинку.

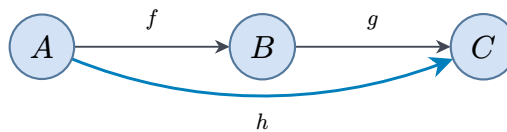


Рис. 133. Коммутативный треугольник: две дороги из A в C дают одну стрелку.

Проверка Категория и её примеры

1. Проверьте, что $(\mathbb{Z}, +, 0)$ задаёт категорию с одним объектом. Какая стрелка будет тождественной, и что значит закон ассоциативности в терминах сложения?
2. Почему для тонкой категории из частичного порядка ассоциативность композиции — это транзитивность?
3. Выпишите все стрелки свободной категории графа из двух рёбер $A \rightarrow B \rightarrow C$ вместе с тождественными. Сколько стрелок ведёт из A в C ?
4. Что неверно в «категории», где тождественная стрелка объекта A действует как $f \circ \text{id}_A = f$, но $\text{id}_B \circ f = g$ для некоторой $g \neq f$?

§ 36.2 Стрелки вместо элементов

В теории множеств базовый вопрос к объекту — «какие у него элементы». В категории базовый вопрос — «какие стрелки в него ведут и из него выходят». Разница работающая: половина свойств, привычно формулируемых через элементы, переформулируется языком стрелок, и переформулировка действует там, где элементов нет вовсе.

Начнём с привычного. Для функций знакомы два свойства: инъективность — разные элементы переходят в разные, сюръективность — каждый элемент накрыт. Оба определения требуют заглянуть *внутри* множеств. Следующие определения выглядят иначе и не требуют ничего, кроме композиции.

ОПРЕДЕЛЕНИЕ 36.3 Изоморфизм

Стрелка $f : A \rightarrow B$ называется **изоморфизмом**, если существует $g : B \rightarrow A$ с $g \circ f = \text{id}_A$ и $f \circ g = \text{id}_B$. Стрелка g называется *обратной* к f .

Обратная стрелка отменяет действие f : туда-обратно — тождество в обе стороны. В категории Set изоморфизмы — это в точности биекции, и это первый сигнал, что язык стрелок не слабее привычного. В категории моноида $(M, \star, e)$ изоморфизмы — обратимые элементы: равенства $g \circ f = \text{id}$ и $f \circ g = \text{id}$ дословно превращаются в $g \star f = e$ и $f \star g = e$ — определение обратного элемента из 12.

ОПРЕДЕЛЕНИЕ 36.4 Мономорфизм

Стрелка $f : A \rightarrow B$ называется **мономорфизмом**, если для любых параллельных $g_1, g_2 : X \rightarrow A$ (с общим началом X и общим концом A) из $f \circ g_1 = f \circ g_2$ следует $g_1 = g_2$.

В Set любая инъекция — мономорфизм: разные входы дают разные образы, и по образу вход восстанавливается.

ОПРЕДЕЛЕНИЕ 36.5 Эпиморфизм

Стрелка $f : A \rightarrow B$ называется **эпиморфизмом**, если для любых параллельных $g_1, g_2 : B \rightarrow Y$ из $g_1 \circ f = g_2 \circ f$ следует $g_1 = g_2$.

В Set любая сюръекция — эпиморфизм: значения g_1 и g_2 , равные на всех прообразах, равны и на всём B .

Оба определения читаются как сокращение общего множителя: если произведение с множителем f одинаково, то и второй сомножитель тот же. Инъективность и сюръективность требуют заглядывать *внутри* множеств, а моно- и эпиморфизм проверяются по одной композиции — и потому имеют смысл в любой категории.

УТВЕРЖДЕНИЕ 36.2 В Set стрелочные определения совпадают с элементными

В категории Set мономорфизмы — в точности инъективные функции, эпиморфизмы — в точности сюръективные, изоморфизмы — биекции.

Доказательство

Докажем все три соответствия по очереди.

Мономорфизм инъективен. Пусть $f : A \rightarrow B$ — мономорфизм, и $f(x_1) = f(x_2)$ для элементов $x_1, x_2 \in A$. Рассмотрим функции $g_1, g_2 : 1 \rightarrow A$ из одноэлементного множества $1 = \{*\}$, переводящие $*$ в x_1 и x_2 соответственно. Тогда $f \circ g_1 = f \circ g_2$ как функции из 1 в B , и свойство сокращения даёт $g_1 = g_2$, то есть $x_1 = x_2$.

Инъективность даёт мономорфизм. Пусть f инъективна и $f \circ g_1 = f \circ g_2$ для функций $g_1, g_2 : X \rightarrow A$. Тогда для каждого $x \in X$ выполнено $f(g_1(x)) = f(g_2(x))$, откуда инъективность даёт $g_1(x) = g_2(x)$, а значит, $g_1 = g_2$.

Эпиморфизм сюръективен. Если элемент $y \in B$ не покрыт f , возьмём две функции $g_1, g_2 : B \rightarrow \{0, 1\}$, отличающиеся только на y . После f они совпадают, вопреки сокращению справа.

Сюръективность даёт эпиморфизм. Пусть f сюръективна и $g_1 \circ f = g_2 \circ f$. Каждый $y \in B$ имеет вид $f(x)$, и равенство на x даёт $g_1(y) = g_2(y)$, то есть $g_1 = g_2$.

Изоморфизм — биекция. Пусть f — изоморфизм с обратной g . Инъективность: из $f(x_1) = f(x_2)$ следует $x_1 = g(f(x_1)) = g(f(x_2)) = x_2$. Сюръективность: каждый $y \in B$ имеет вид $f(g(y))$, значит накрыт. Обратно, у биекции обратная функция существует, и обе композиции тождественны.

Тонкая категория из порядка устроена иначе, и это отрезвляющий пример.

Пример Мономорфизмы порядка

В тонкой категории частичного порядка между любыми двумя объектами есть не более одной стрелки, поэтому любые два параллельных g_1, g_2 совпадают автоматически. Свойство сокращения выполняется для всякой стрелки: каждая стрелка — и мономорфизм, и эпиморфизм, независимо от того, как выглядят элементы. Инъективность и сюръективность к стрелкам такой категории неприменимы в принципе: у стрелки нет элементов, а стрелочные свойства выполняются для всех стрелок сразу. При этом изоморфизмом стрелка быть не обязана: в цепочке $0 \leq 1 \leq 2$ стрелка $0 \rightarrow 1$ — и мономорфизм, и эпиморфизм, но обратной к ней нет, потому что стрелки $1 \rightarrow 0$ не существует. Мономорфизм вместе с эпиморфизмом обратной стрелки не гарантируют.

Порядок и множества дают две крайности: в Set стрелочные определения совпадают с элементными, в тонкой категории элементы исчезают полностью. Между ними живут все остальные категории, и стрелочный язык — единственный, работающий сразу везде.

Мы договорились описывать объекты только через их стрелки: объект считается известным ровно настолько, насколько известны его стрелки, и все дальнейшие конструкции этой границы не переступают.

ОПРЕДЕЛЕНИЕ 36.6 Противоположная категория

Категория с теми же объектами и стрелками, что у $\mathcal{C}$, но с обращёнными началами и концами называется **противоположной** и обозначается $\mathcal{C}^{\text{op}}$: стрелка $f : A \rightarrow B$ читается в ней как $f : B \rightarrow A$, а композиция обращает порядок, $g \circ^{\text{op}} f = f \circ g$.

Для тонкой категории порядка обращение стрелок — это переход к обратному порядку: $\mathcal{C}^{\text{op}}$ цепочки $0 \leq 1 \leq 2$ — цепочка $2 \leq 1 \leq 0$.

Утверждение, доказанное для произвольной категории $\mathcal{C}$, верно и для $\mathcal{C}^{\text{op}}$. Подставив $\mathcal{C}^{\text{op}}$ вместо $\mathcal{C}$ и развернув определение противоположной категории, получаем новое верное утверждение: все стрелки перевёрнуты, начала и концы поменялись местами, а композиции читаются в обратном порядке. Вот образец такого хода: в $\mathcal{C}^{\text{op}}$ условие эпиморфизма $g_1 \circ^{\text{op}} f = g_2 \circ^{\text{op}} f$ по определению читается как $f \circ g_1 = f \circ g_2$ — а это условие мономорфизма в $\mathcal{C}$. Это **принцип двойственности**: у каждого понятия и теоремы есть двойник. Мономорфизм двойственен эпиморфизму, терминальный объект — начальному, произведение — копроизведению, предел — копределу. Половина теории категорий — умение доказывать одну теорему и получать вторую бесплатно.

Проверка Стрелки и двойственность

1. Почему биекция — это в точности изоморфизм в Set ? Проверьте обе композиции.
2. Возьмём множество $\{a, b\}$ с отношением «каждый элемент меньше или равен каждому». Будет ли стрелка $a \rightarrow b$ изоморфизмом в тонкой категории этого отношения?
3. Убедитесь сначала, что композиция двух мономорфизмов — мономорфизм. Затем сформулируйте двойственное утверждение и получите его двойственностью, без нового доказательства.
4. Что такое категория $\mathcal{C}^{\text{op}}$ для тонкой категории порядка $(\mathbb{Z}, \leq)$? Какому порядку она соответствует?

§ 36.3 Функторы и перенос структуры

Между моноидами (глава 12) разумными отображениями были не любые функции, а гомоморфизмы: они сохраняли операцию. Между автоматами (глава 23) — не любые пары функций, а те, что согласованы с переходами. Общий рецепт уже виден: отображение структур заслуживает имени, когда оно уносит за собой не только элементы, но и устройство. Категория собрана из объектов, стрелок и композиции, поэтому честное отображение категорий переводит все три части.

ОПРЕДЕЛЕНИЕ 36.7 Функтор

Функтор $F : \mathcal{C} \rightarrow \mathcal{D}$ сопоставляет каждому объекту A категории $\mathcal{C}$ объект $F(A)$ категории $\mathcal{D}$, а каждой стрелке $f : A \rightarrow B$ — стрелку $F(f) : F(A) \rightarrow F(B)$ так, что

$$F(g \circ f) = F(g) \circ F(f), \quad F(\text{id}_A) = \text{id}_{F(A)}.$$

Контравариантным называют функтор $\mathcal{C}^{\text{op}} \rightarrow \mathcal{D}$: на объектах он ведёт себя обычно, но каждая стрелка $f : A \rightarrow B$ переходит в стрелку $F(A) \leftarrow F(B)$ противоположного направления. Обычный функтор $\mathcal{C} \rightarrow \mathcal{D}$ в противовес называют *ковариантным*. Знакомый пример контравариантности — прообраз подмножества, он ждёт ниже.

Законы функтора говорят: сначала компоновать, потом переводить — то же самое, что сначала переводить, потом компоновать. Диаграммы сохраняются: коммутующий квадрат переходит в коммутующий квадрат.

Примечание Зачем закон тождества

Отображение, переводящее каждую функцию в постоянную, почти проходит первый закон: постоянные склеиваются так же, как исходные. Но $F(\text{id}_A)$ оказывается постоянной функцией — не $\text{id}_{F(A)}$. Закон тождества отсекает такие вырожденные кандидаты и же делает $F(f^{-1})$ обратной стрелкой к $F(f)$ в теореме о сохранении изоморфизмов.

Программисты живут в категории типов: объекты — типы, стрелки — функции, композиция и тождества — обычные. В ней функтор — это контейнер с операцией `map`: список с `map`, опциональное значение с `map`. Обе аксиомы читаются как «применять композицию к отображённому — то же, что отображать композицию».

Этот взгляд стоит зафиксировать кодом, потому что дальше он станет главным героем.

```
// Функтор "список": map уважает композицию и тождество.
fn map<T, U>(xs: &[T], f: impl Fn(&T) -> U) -> Vec<U> {
    xs.iter().map(f).collect()
}
```



```
// map(xs, |x| g(f(x))) == map(map(xs, f), g)  --- композиция
// map(xs, |x| x)          == xs              --- тождество
```

Первый классический функтор — булеан, конструкция, знакомая по каждому разговору о множествах.

Пример Булеан в обе вариантности

Ковариантный булеан переводит множество A в $\mathcal{P}(A)$, а функцию $f : A \rightarrow B$ — в отображение образа $\text{img}_f : \mathcal{P}(A) \rightarrow \mathcal{P}(B)$. На подмножество $S \subseteq A$ оно действует правилом $S \mapsto f(S) = \{f(x) \mid x \in S\}$. Тождественная функция переходит в тождественную, образ композиции — композиция образов, так что это функтор $\text{Set} \rightarrow \text{Set}$.

Контравариантная версия переводит $f : A \rightarrow B$ в прообраз $f^{-1} : \mathcal{P}(B) \rightarrow \mathcal{P}(A)$: подмножество $T \subseteq B$ возвращается в $f^{-1}(T) = \{x \in A \mid f(x) \in T\}$. На конкретных множествах: для $A = \{1, 2\}$, $B = \{a, b, c\}$ и $f(1) = a$, $f(2) = b$ образ $\{1\} = \{a\}$, прообраз $\{a, c\} = \{1\}$, прообраз $\{c\} = \emptyset$. Здесь направление стрелок меняется: чем больше информации о B , тем больше известно и об A .

Обе вариантности собраны на рис. 134.

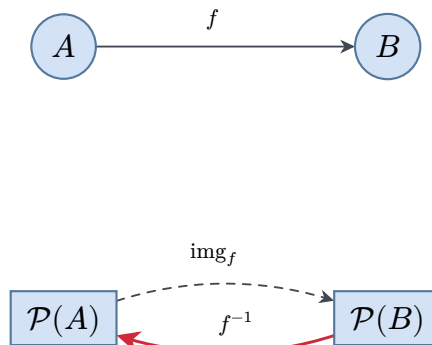


Рис. 134. Функция f поднимается на подмножества дважды: образ идёт вперёд, прообраз — назад.

УТВЕРЖДЕНИЕ 36.3 Прообраз контравариантен

Отображение $f \mapsto f^{-1}$ определяет контравариантный функтор на Set : $\text{id}^{-1} = \text{id}$ и $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$.

Доказательство

Докажем второе равенство включением в обе стороны. Первое получается той же проверкой.

Включение $\subseteq$. Пусть $x \in (g \circ f)^{-1}(T)$ для подмножества $T \subseteq C$. По определению прообраза $g(f(x)) \in T$, то есть $f(x) \in g^{-1}(T)$, а значит $x \in f^{-1}(g^{-1}(T))$.

Включение $\supseteq$. Пусть $x \in f^{-1}(g^{-1}(T))$. Тогда $f(x) \in g^{-1}(T)$, то есть $g(f(x)) \in T$, что и означает $x \in (g \circ f)^{-1}(T)$.

Прообраз — рабочая конструкция теории вероятностей: события выражаются через прообразы случайных величин (глава 20), и контравариантность объясняет, почему распределение вероятностей тянет события «в обратную сторону». Тот же функтор незаметно работал в алгебре логики и в анализе программ: предусловие команды — прообраз множества постсостояний.

Следующая пара функторов — мост между двумя мирами, и она ещё вернётся в секции 36.8.

Пример Свободный и забывающий

Забывающий функтор $U : \mathbf{Mon} \rightarrow \mathbf{Set}$ переводит моноид в его носитель, выбрасывая операцию: стрелки-гомоморфизмы переходят в обычные функции. *Свободный* функтор $L : \mathbf{Set} \rightarrow \mathbf{Mon}$ переводит множество A в моноид строк A^* с конкатенацией (глава 12), и его называют *свободным*.

Функция $f : A \rightarrow B$ продолжается в гомоморфизм $L(f) : A^* \rightarrow B^*$ применением f к каждому элементу списка: пустой список остаётся пустым, конкатенация сохраняется. Для $f : \{0, 1\} \rightarrow \{a, b\}$ с $f(0) = a$, $f(1) = b$ гомоморфизм $L(f)$ переводит список «011» в «abb», а «01» и «10» — в «ab» и «ba» по отдельности, поэтому конкатенация не разрушается. Программист узнаёт здесь тар над списками: свободный функтор — контейнер, который ничего не помнит кроме элементов и потому переносит любые функции.

Примечание Осторожно со словом «функтор»

В главе 16 функтором назывался конструктор терма: `parent(tom, mary)` — функтор `parent` с двумя аргументами. Здесь функтор — отображение между категориями. Две структуры носят одно имя по случайности истории: карнаповское слово прижилось и в Прологе, и в теории категорий. Родственна и другая коллизия: «категоричность» аксиом из теории моделей (глава 22) — про единственность модели, а не про категории этой главы.

Третий пример — самый важный для этой главы, хотя выглядит скромнее всех.

Пример Функтор стрелок из объекта

Зафиксируем объект A категории $\mathcal{C}$. Сопоставление $B \mapsto \mathcal{C}(A, B)$ переводит объект в множество стрелок, ведущих из A в него. Стрелка $h : B \rightarrow C$ переводится в операцию посткомпозиции $\mathcal{C}(A, B) \rightarrow \mathcal{C}(A, C)$, действующую правилом $f \mapsto h \circ f$. Ассоциативность композиции — в точности функториальность: $(h_2 \circ h_1) \circ f = h_2 \circ (h_1 \circ f)$.

Получен функтор $\mathcal{C}(A, -) : \mathcal{C} \rightarrow \text{Set}$, *представимый* объектом A . Его контравариантный родственник $\mathcal{C}(-, B)$ строится предкомпозицией: стрелка $h : X \rightarrow Y$ действует правилом $u \mapsto u \circ h$, отображением $\mathcal{C}(Y, B) \rightarrow \mathcal{C}(X, B)$.

Вернёмся к изоморфизмам: функтор не может испортить то, что сформулировано на языке стрелок.

УТВЕРЖДЕНИЕ 36.4 Функтор сохраняет изоморфизмы

Если $f : A \rightarrow B$ — изоморфизм в $\mathcal{C}$ и $F : \mathcal{C} \rightarrow \mathcal{D}$ — функтор, то $F(f) : F(A) \rightarrow F(B)$ — изоморфизм в $\mathcal{D}$, причём $(F(f))^{-1} = F(f^{-1})$.

Доказательство

Пусть $g : B \rightarrow A$ — обратная стрелка: $g \circ f = \text{id}_A$ и $f \circ g = \text{id}_B$. Применим F к обоим равенствам и воспользуемся законами функтора:

$$F(g) \circ F(f) = F(g \circ f) = F(\text{id}_A) = \text{id}_{F(A)},$$

и аналогично $F(f) \circ F(g) = \text{id}_{F(B)}$. Стрелка $F(g)$ — искомый обратный к $F(f)$.

Следствие для практики: если две структуры изоморфны, то любые их функториальные инварианты совпадают. Инварианты, построенные из композиций, — количества, диаграммы, устройства — переносятся без пересчёта. Именно так устроены доказательства вида «графы не изоморфны: у одного есть цикл длины 5, у другого нет» — здесь только закон указан явно.

Проверка Функторы

1. Проверьте аксиомы функтора для ковариантного булеана: что нужно проверить для id и для композиции?
2. Забывающий функтор $\text{Grp} \rightarrow \text{Set}$ переводит группу (моноид, где у каждого элемента есть обратный) в её носитель. Что он делает со стрелками и почему гомоморфизм переходит в функцию?
3. Почему `Option` с операцией `map` — функтор на программистской категории типов? Проверьте оба закона на примерах `None`.
4. Функтор $\mathcal{C}(A, -)$ переводит объект A в множество. Что он делает с изоморфизмом $h : B \rightarrow C$? Будет ли отображение множеств $\mathcal{C}(A, B) \rightarrow \mathcal{C}(A, C)$ биекцией?

§ 36.4 Естественные преобразования

Два функтора $F, G : \mathcal{C} \rightarrow \mathcal{D}$ едут из одной категории в другую. Сравнить их — значит для каждого объекта A дать стрелку $\alpha_A : F(A) \rightarrow G(A)$, и сделать это *согласованно*. Согласованность и есть та самая естественность, ради которой Эйленберг и Мак Лейн построили весь аппарат.

Порядок появления понятий здесь обратен порядку их изложения. Категории ввели не ради самих категорий: определение понадобилось, чтобы строго сказать, что такое функтор, а функтор — чтобы определить естественное преобразование. Верхний этаж башни оказался главным, и нижние держат его до сих пор.

ОПРЕДЕЛЕНИЕ 36.8 Естественное преобразование

Пусть $F, G : \mathcal{C} \rightarrow \mathcal{D}$ — функторы. Семейство стрелок $\alpha_A : F(A) \rightarrow G(A)$, по одной на каждый объект A категории $\mathcal{C}$, называется **естественным преобразованием** $\alpha : F \Rightarrow G$, если для каждой стрелки $f : A \rightarrow B$ коммутирует квадрат

$$G(f) \circ \alpha_A = \alpha_B \circ F(f).$$

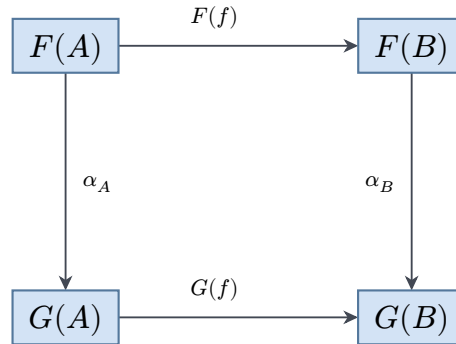


Рис. 135. Квадрат естественности: два пути из $F(A)$ в $G(B)$ по сторонам квадрата дают одно и то же.

Естественность читается как честность семейства: компоненты α_A согласованы со всеми стрелками категории, поэтому «не зависят от случайного выбора».

Разница видна на паре простых семейств. Вложение $x \mapsto \{x\}$ множества A в булеан $\mathcal{P}(A)$ — естественное преобразование из тождественного функтора (оставляющего каждое множество на месте) в булеан: оно определено одним правилом, без выбора чего бы то ни было. А семейство «выбрать по одному представителю в каждом непустом множестве» даёт стрелки из одноэлементного 1 в каждое A , но каждому своему способу: правило зависит от выбора. Для $f : \{0, 1\} \rightarrow \{a, b\}$ с $f(0) = a, f(1) = b$ выбор 0 в $\{0, 1\}$ и b в $\{a, b\}$ разрушает квадрат: путь через f даёт a , прямой путь — b . Различие «одно правило против выбора» математики 1930-х ощущали пальцами, не имея языка, — пока язык не появился.

Программисты используют естественные преобразования ежедневно, называя их полиморфными функциями.

Пример Безопасный первый элемент

Зафиксируем два функтора на категории типов: список `List` и опциональное значение `Option`. Семейство `head` переводит список любого типа A в `Option`, возвращая первый элемент или отсутствие: пустой список переходит в `None`, непустой — в `Some(x)`.

Естественность — это равенство $\text{map } g \text{ (head } xs) = \text{head (map } g \text{ xs)}$ для любой функции g : достать первый элемент и преобразовать его — то же, что преобразовать все и достать первый. Здесь F — список, G — `Option`, компонента α_A — `head`: левая часть равенства — путь через `Option(A)`, правая — путь через список B , и это в точности два пути квадрата 135. Для `None` и для `Some(x)` равенство проверяется в одну строку каждое.

Семейство `head` не зависит от типа элементов и от конкретной функции — только от формы контейнеров. Параметричность — имя этого явления: дженерик без ограничений на тип-параметр не может подсматривать в элементы, и квадраты естественности выдаются ему автоматически (ограничение вида `T: SomeTrait` уже подсматривает — здесь его нет). Свободные теоремы Вадлера: каждая полиморфная функция удовлетворяет теореме, которую не нужно доказывать — она следует из одного типа²⁵⁴.

Функторы одной пары категорий сами складываются в категорию.

ОПРЕДЕЛЕНИЕ 36.9 Категория функторов

Категория, объекты которой — функторы $\mathcal{C} \rightarrow \mathcal{D}$, а стрелки — естественные преобразования между ними, называется **категорией функторов** и обозначается $[\mathcal{C}, \mathcal{D}]$, если композиция задана покомпонентно, $(\beta \circ \alpha)_A = \beta_A \circ \alpha_A$, а тождество тождественно покомпонентно.

Простейший пример: если $\mathcal{C}$ — дискретная категория из двух объектов (только тождественные стрелки), то функтор в `Set` — это пара множеств, а естественное преобразование — пара функций. Категория функторов поднимает на этаж выше: стрелки $\mathcal{C}$ продолжают жить среди стрелок нового мира, и ничто не мешает подниматься дальше.

Естественный изоморфизм — естественное преобразование, у которого каждая компонента α_A — изоморфизм. Ниже $\text{id}_{\mathcal{C}}$ обозначает тождественный функтор, оставляющий категорию на месте.

²⁵⁴P. Wadler. «Theorems for free!». Conference on Functional Programming Languages and Computer Architecture, 1989.

ОПРЕДЕЛЕНИЕ 36.10 Эквивалентность категорий

Функтор $F : \mathcal{C} \rightarrow \mathcal{D}$ задаёт **эквивалентность категорий**, если найдётся $G : \mathcal{D} \rightarrow \mathcal{C}$ с естественными изоморфизмами $\text{id}_{\mathcal{C}} \cong G \circ F$ и $F \circ G \cong \text{id}_{\mathcal{D}}$.

Знак $\cong$ читается «между ними существует естественный изоморфизм» — это не равенство функторов.

Изоморфизм категорий требует буквально равных композиций, $G \circ F = \text{id}_{\mathcal{C}}$ и $F \circ G = \text{id}_{\mathcal{D}}$. Эквивалентность слабее: обратный функтор обязан возвращать каждый объект лишь с точностью до изоморфизма. Двойственность Стоуна (глава 10) — эквивалентность категории стоуновых пространств и категории, противоположной категории булевых алгебр: объекты не совпадают буквально, но вся комбинаторика переезжает целиком. В теории моделей близкое явление называют категоричностью (глава 22): теория имеет единственную модель с точностью до изоморфизма. Эквивалентность категорий переносит тот же идеал на целые миры: две эквивалентные категории неразличимы категорными средствами.

Проверка Естественные преобразования

1. Выпишите квадрат естественности для head и проверьте его явно на списках $[]$ и $[1, 2]$ с функцией $g(x) = x + 1$.
2. Семейство $\alpha_A(x) = \{x\}$, вкладывающее множество A в булеан $\mathcal{P}(A)$ одноэлементными подмножествами, естественно ли относительно функтора $\mathcal{P}$?
3. Почему композиция естественных преобразований покомпонентно снова естественна? Достаточно одной строки для каждой компоненты.
4. Чем эквивалентность категорий отличается от изоморфизма категорий? Какое из двух требований слабее и почему этого достаточно для переноса теории?

§ 36.5 Универсальные свойства

Вернёмся к совпадениям из введения — теперь на языке стрелок. У произведения множеств, прямого произведения автоматов и конъюнкции обобщий вид такой, как на рис. 136: объект P с парой стрелок $p_1 : P \rightarrow A$ и $p_2 : P \rightarrow B$, через который пропускается любая другая пара. «Пропускается» — значит для любого X с парой $f : X \rightarrow A$, $g : X \rightarrow B$ найдётся ровно одна стрелка $\langle f, g \rangle : X \rightarrow P$ с $p_1 \circ \langle f, g \rangle = f$ и $p_2 \circ \langle f, g \rangle = g$. Свойство говорит сразу обо всех объектах категории — любой X обязан согласовать свою пару с P — и потому называется *универсальным*: оно про положение среди стрелок, а не про внутреннее устройство P .

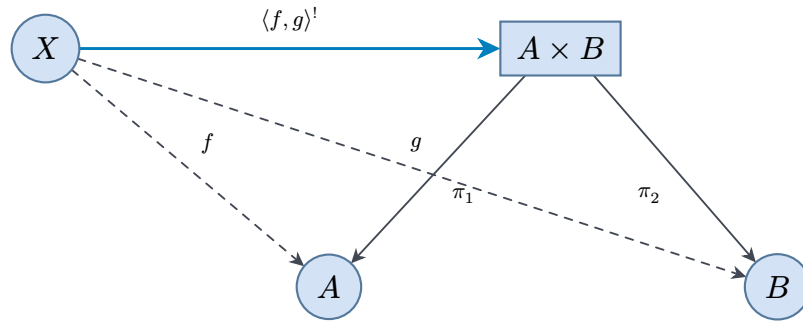


Рис. 136. Универсальное свойство произведения: любая пара стрелок в A и B раскладывается через проекции ровно одним способом.

Самая экономная версия той же идеи — объект, определяемый одним универсальным свойством без пар.

ОПРЕДЕЛЕНИЕ 36.11 Терминальный объект

Объект 1 называется **терминальным**, если из каждого объекта категории ведёт ровно одна стрелка в 1 .

В терминальный объект вся категория «сваливается» единственным способом: в Set это одноэлементное множество $1 = \{*\}$, и функция в него ровно одна — она забывает про элемент всё. В тонкой категории порядка терминальный объект — наибольший элемент: в него ведёт стрелка-неравенство от каждого.

ОПРЕДЕЛЕНИЕ 36.12 Начальный объект

Объект 0 называется **начальным**, если из 0 в каждый объект категории ведёт ровно одна стрелка.

Начальный объект — его зеркальное отражение: в Set это пустое множество $\emptyset$, из него в каждое множество ведёт единственная (пустая) функция, а в порядке начальный объект — наименьший элемент. В категории с одним объектом, порождённой тривиальным моноидом, этот объект одновременно начальный и терминальный. Уже у $(\mathbb{Z}, +, 0)$ стрелок из объекта в себя бесконечно много, так что ни начального, ни терминального объекта нет: универсальные объекты существуют не во всякой категории.

ОПРЕДЕЛЕНИЕ 36.13 Произведение

Произведением пары объектов A и B называется объект $A \times B$ с парой стрелок $\pi_1 : A \times B \rightarrow A$ и $\pi_2 : A \times B \rightarrow B$, через который пропускается любая пара стрелок в A и B : для $f : X \rightarrow A$ и $g : X \rightarrow B$ существует ровно одна стрелка $\langle f, g \rangle : X \rightarrow A \times B$ с $\pi_1 \circ \langle f, g \rangle = f$ и $\pi_2 \circ \langle f, g \rangle = g$.

В Set это декартово произведение с обычными проекциями, и слово «ровно одна» здесь — теорема: пара функций задаёт функцию $h(x) = (f(x), g(x))$ в $A \times B$, и разные пары дают разные функции, потому что проекции восстанавливают h обратно.

ОПРЕДЕЛЕНИЕ 36.14 Копроизведение

Двойственная к произведению конструкция называется **копроизведением** пары объектов A и B и устроена так: объект $A + B$ со стрелками $\iota_1 : A \rightarrow A + B$ и $\iota_2 : B \rightarrow A + B$ пропускает через себя любую пару стрелок, выходящих из A и из B . А именно, для $f : A \rightarrow X$ и $g : B \rightarrow X$ существует ровно одна стрелка $[f, g] : A + B \rightarrow X$ с $[f, g] \circ \iota_1 = f$ и $[f, g] \circ \iota_2 = g$.

В Set копроизведение — дизъюнктное объединение с включениями ι_1, ι_2 : чтобы задать функцию на нём, достаточно задать её на каждой из частей по отдельности.

Пример Пять глав одной диаграммы

- Set: произведение — декартово произведение с проекциями (глава 4), копроизведение — дизъюнктное объединение с включениями.
- Логика высказываний (глава 1): возьмём категорию из двух истинностных значений со стрелкой $0 \rightarrow 1$ — произведение объектов p и q — их минимум, значение $p \wedge q$, копроизведение — максимум $p \vee q$, а тавтологии вроде $p \rightarrow (q \vee p)$ читаются как стрелки.
- Тонкая категория порядка (глава 8): произведение — точная нижняя грань $a \wedge b$, копроизведение — точная верхняя $a \vee b$; универсальность — определение грани через импликации между неравенствами.
- Делимость целых чисел: произведение — наибольший общий делитель, копроизведение — наименьшее общее кратное; универсальность НОД — «любой общий делитель делит его».
- Автоматы (глава 23): в категории автоматов со стрелками-функциями, сохраняющими переходы и множества принимающих состояний, прямое произведение автоматов — произведение; проекции — компоненты пары состояний.

Категория	Произведение	Копроизведение	Терминальный
Set	пары $A \times B$	дизъюнктное объединение	одноэлементное
логика	$p \wedge q$	$p \vee q$	$\top$ («истина»)
порядок	$a \wedge b$ — инфимум	$a \vee b$ — супремум	наибольший
делимость	НОД(a, b)	НОК(a, b)	0 (на всё делится)
автоматы	прямое произведение	дизъюнктное объединение состояний	одно принимающее состояние

Разберём одну строку таблицы руками, чтобы универсальность перестала быть заклинанием. В делимости стрелка $a \rightarrow b$ означает « a делит b », и произведение пары $(12, 18)$ — объект с делящими их обоими стрелками, через который пропускается любой общий делитель. НОД 6 подходит: $6 \rightarrow 12$ и $6 \rightarrow 18$ есть, а для $X = 2$ со стрелками $2 \rightarrow 12$, $2 \rightarrow 18$ стрелка $2 \rightarrow 6$ существует и единственна (тонкая категория). Любой общий делитель $d \neq 6$ проваливается одинаково: d делит 6, а 6 не делит

меньшее d , поэтому стрелки $6 \rightarrow d$ нет — достаточно взять $X = 6$ с его стрелками в 12 и 18. «Наибольший» оказался не про сравнение чисел, а про универсальное положение среди делителей.

Универсальные объекты определяются не однозначно, а «с точностью до изоморфизма» — и вот первое полноценное доказательство этого принципа.

ТЕОРЕМА 36.5 Единственность универсальных объектов

Любые два произведения пары A, B изоморфны, причём существует ровно один изоморфизм, согласованный с проекциями ($p_1 \circ h = p'_1$ и $p_2 \circ h = p'_2$). То же верно для копроизведений, терминальных и начальных объектов.

Доказательство

Докажем для произведения. Для остальных конструкций рассуждение одинаково, а для копроизведения — двойственно.

Пусть P с проекциями p_1, p_2 и P' с проекциями p'_1, p'_2 — два произведения A и B . Применим универсальное свойство P к паре $p'_1 : P' \rightarrow A, p'_2 : P' \rightarrow B$: существует единственная стрелка $h : P' \rightarrow P$ с $p_1 \circ h = p'_1$ и $p_2 \circ h = p'_2$. Симметрично, существует единственная $h' : P \rightarrow P'$ с $p'_1 \circ h' = p_1$ и $p'_2 \circ h' = p_2$.

Композиция $h \circ h' : P \rightarrow P$ удовлетворяет $p_1 \circ h \circ h' = p_1$ и $p_2 \circ h \circ h' = p_2$. Этим же свойством обладает тождественная стрелка id_P , а универсальность P даёт *единственность*: $h \circ h' = \text{id}_P$. Аналогично $h' \circ h = \text{id}_{P'}$. Стрелка h — искомый изоморфизм, и он единственный по построению.

Доказательство единственности написано один раз, а применимо ко всем пяти строкам таблицы: множества, формулы, порядки, числа и автоматы покрыты одной теоремой.

§ 36.6 Пределы и копределы

Произведение решало задачу о паре объектов, между которыми нет стрелок. Обобщение берёт произвольную диаграмму: набор объектов и стрелок между ними, например, три объекта A, B, C со стрелками $A \rightarrow C$ и $B \rightarrow C$.

ОПРЕДЕЛЕНИЕ 36.15 Конус над диаграммой

Объект N с семейством стрелок-ножек $n_A : N \rightarrow A$, по одной в каждый объект диаграммы D , называется **конусом** над D , если семейство согласовано со стрелками диаграммы: для каждой её стрелки $u : A \rightarrow C$ выполнено $n_C = u \circ n_A$.

Конус над диаграммой из пары объектов A и B без стрелок — это просто объект с парой стрелок в A и B . Над диаграммой $A \rightarrow C \leftarrow B$ конус обязан ещё согласовывать ножки: композиции $N \rightarrow A \rightarrow C$ и $N \rightarrow B \rightarrow C$ обе равны ножке $N \rightarrow C$.

ОПРЕДЕЛЕНИЕ 36.16 Предел

Конус $\lim D$ с ножками l_A называется **пределом** диаграммы D , если из любого другого конуса N с ножками n_A в него ведёт ровно одна стрелка $m : N \rightarrow \lim D$ с $n_A = l_A \circ m$ для каждого объекта A диаграммы. Предел — терминальный объект среди конусов.

Двойственное понятие — **копредел**: коконус, объект N со стрелками $A_i \rightarrow N$, согласованными со стрелками диаграммы в обратную сторону, через который единственным образом пропускается любой другой коконус. Оба героя — конус и терминальный конус — показаны на рис. 137.

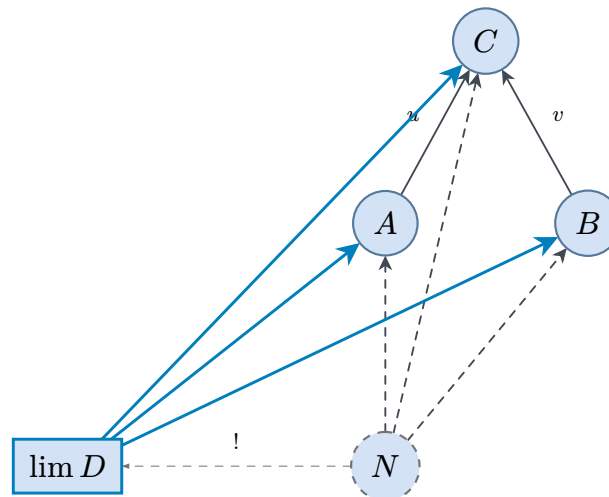


Рис. 137. Конус N и терминальный конус $\lim D$: через $\lim D$ пропускается любой конус, как через произведение — любая пара.

Пределы узнаются в конструкциях всей книги:

1. произведение — предел диаграммы из двух объектов без стрелок
2. пересечение подмножеств $A_i \subseteq U$ — предел диаграммы вложений $A_i \rightarrow U$ в общее надмножество
3. наименьшая общая верхняя грань цепочки — копредел цепочки
4. p -адические числа — предел цепочки отображений взять-остаток $\mathbb{Z}/p^{n+1} \rightarrow \mathbb{Z}/p^n$ (обратная система): предел склеивает совместимые по всем модулям вычеты, материал за рамками курса, но схема та же

Единой теоремой о пределах закрываются сразу все эти построения: правые сопряжённые функторы сохраняют пределы, — докажем это в секции 36.8.

Проверка Универсальные свойства

1. Проверьте, что в тонкой категории порядка произведение пары — точная нижняя грань: выпишите универсальное свойство на языке неравенств.
2. Почему копроизведение в $\mathbf{Set}$ не может быть обычным объединением $A \cup B$? Какую роль играют включения ι_1, ι_2 ?
3. Найдите терминальный объект тонкой категории делимости на множестве $\{2, 3, 6, 12\}$. Существует ли в ней начальный объект?

Проверка Пределы

1. Конус над диаграммой из одного объекта A без стрелок — это просто объект со стрелкой в A . Что такое предел такой диаграммы?
2. Нарисуйте диаграмму $A \rightarrow C \leftarrow B$ и конус над ней с вершиной N . Какому равенству стрелок соответствует согласование ножек?
3. Почему терминальный объект — это предел пустой диаграммы (диаграммы без объектов и стрелок)?

§ 36.7 Лемма Йонеды

Функторы вида $\mathcal{C}(A, -)$ из примера о представимых функторах устроены скромно: они ничего не знают об объектах, кроме стрелок из A . Лемма Йонеды утверждает, что и знать больше не нужно: любой функтор $\mathcal{C} \rightarrow \mathbf{Set}$ можно понять через такие «стрелочные» функторы, и словарь перевода предельно прост.

ТЕОРЕМА 36.6 Лемма Йонеды

Пусть $\mathcal{C}$ — категория, A — её объект, $F : \mathcal{C} \rightarrow \mathbf{Set}$ — функтор. Естественные преобразования $\alpha : \mathcal{C}(A, -) \rightarrow F$ находятся во взаимно однозначном соответствии с элементами множества $F(A)$, и соответствие естественно по A и по F .

Доказательство

Докажем, что отображение, переводящее преобразование α в элемент $\alpha_A(\mathrm{id}_A)$, — биекция, построив обратное в обе стороны.

От семейства к элементу. Каждое естественное преобразование α даёт элемент $\alpha_A(\mathrm{id}_A) \in F(A)$: компонента α_A — функция из множества стрелок $\mathcal{C}(A, A)$ в $F(A)$, применённая к тождественной.

От элемента к семейству. Пусть $x \in F(A)$. Для каждого объекта B определим компоненту $\alpha_B^x : \mathcal{C}(A, B) \rightarrow F(B)$ правилом $f \mapsto F(f)(x)$: стрелка $f : A \rightarrow B$

переводится функтором F в функцию $F(f) : F(A) \rightarrow F(B)$, действующую на элементе x .

Взаимная обратность. Семейство α^x естественно: для $h : B \rightarrow C$ выполнено $F(h) \circ \alpha_B^x(f) = F(h)(F(f)(x)) = F(h \circ f)(x) = \alpha_C^x(h \circ f)$, что и есть коммутативность квадрата естественности. В другую сторону, естественность α на стрелке $f : A \rightarrow B$ даёт равенство $F(f) \circ \alpha_A = \alpha_B \circ \mathcal{C}(A, f)$, применённое к id_A . Подставив элемент, получаем $F(f)(\alpha_A(\text{id}_A)) = \alpha_B(f \circ \text{id}_A) = \alpha_B(f)$, то есть $\alpha_B^{\alpha_A(\text{id}_A)} = \alpha_B$. Семейства совпадают покомпонентно. Естественность соответствия по A — согласованность с предкомпозицией стрелок $A' \rightarrow A$, по F — с естественными преобразованиями $F \Rightarrow F'$. Обе проверяются теми же покомпонентными вычислениями.

Читается лемма так: *функтор узнаётся по значениям на стрелках из одного объекта*. Вся информация о F , доступная из A , упакована в одно множество $F(A)$. Согласование со всеми стрелками не сужает выбор: естественных семейств ровно столько, сколько элементов. Лемма допускает и машинную проверку: в крейте `code/categories` конечная категория хранится таблицей композиции, и пример `yoneda_check` прогоняет соответствие леммы перебором — каждое естественное преобразование $\mathcal{C}(A, -) \rightarrow F$ переводится в элемент $F(A)$ и обратно — на моноиде $\mathbb{Z}_4$, цепи $0 < 1 < 2$ и свободной категории графа $A \rightarrow B \rightarrow C$.

Следствие при $F = \mathcal{C}(B, -)$ переводит смысл в форму: естественные преобразования $\mathcal{C}(A, -) \rightarrow \mathcal{C}(B, -)$ — это в точности стрелки $B \rightarrow A$ исходной категории.

УТВЕРЖДЕНИЕ 36.7 Вложение Йонеды

Соответствие, переводящее объект B в функтор $\mathcal{C}(B, -)$, продолжается до вполне верного функтора $y : \mathcal{C}^{\text{op}} \rightarrow [\mathcal{C}, \text{Set}]$. Вполне верный означает, что отображение на каждом множестве параллельных стрелок взаимно однозначно: разные стрелки дают разные естественные преобразования, и каждое преобразование возникает из стрелки.

Набросок доказательства

Стрелка $g : B \rightarrow A$ — это стрелка $A \rightarrow B$ в $\mathcal{C}^{\text{op}}$, и она порождает естественное преобразование $y(g)_C : \mathcal{C}(A, C) \rightarrow \mathcal{C}(B, C)$ предкомпозицией $f \mapsto f \circ g$. Лемма Йонеды при $F = \mathcal{C}(B, -)$ превращает соответствие $g \mapsto y(g)$ в биекцию со множеством $\mathcal{C}(B, A)$: обратное направление восстанавливает g как образ тождества id_A . Биективность на каждом множестве параллельных стрелок — и есть полная верность.

Функтор из моноида-категории $\tilde{M}$ в Set — это множество S , на котором каждый элемент $m \in M$ действует функцией $S \rightarrow S$: законы функтора — в точности законы действия моноида из 12.

Практический смысл лучше всего виден в программировании. Объект, о котором известно только, какие стрелки-наблюдения в него ведут, известен полностью: любое свойство-функтор F вычисляется через стрелочный функтор. Продолжения в функциональных языках — та же идея в коде: продолжение — функция, которой вычисление передаёт промежуточный результат, чтобы та решила, что делать дальше. Тип $\forall B. (A \rightarrow B) \rightarrow F(B)$ взаимно однозначен с $F(A)$, и это не совпадение, а лемма Йонеды, прочитанная в категории типов.

Замечание Почему это работает

Заметим, что вся сила леммы держится на тождественной стрелке id_A : элемент $x \in F(A)$ и семейство функций «из стрелок» определяют друг друга через $x = \alpha_A(\text{id}_A)$. Универсальные свойства из предыдущей секции — та же механика, увиденная сверху: функтор, сопоставляющий X пары стрелок $X \rightarrow A$ и $X \rightarrow B$, представим объектом $A \times B$, а единственность из 36.5 — тень единственности представляющего элемента.

Проверка Лемма Йонеды

1. Пусть $\mathcal{C}$ — тонкая категория порядка. Что такое $\mathcal{C}(A, B)$ как множество, и что означает лемма Йонеды для предпорядка?
2. Для моноида-категории $\tilde{M}$ множество $\mathcal{C}(\tilde{M}, \tilde{M})$ — это сам M . Во что превращается лемма Йонеды для функтора из $\tilde{M}$ в Set — множества с действием моноида?
3. Восстановите по элементу $x \in F(A)$ значение компоненты α_B на стрелке $f : A \rightarrow B$. Почему без функториальности F построение бы сломалось?
4. Объясните без формул, почему тип-продолжение $(A \rightarrow B) \rightarrow F(B)$ для всех B «знает» столько же, сколько $F(A)$.

§ 36.8 Сопряжённые функторы

Свободный моноид над множеством A — список — устроен расточительно: он хранит все последовательности. Забывающий функтор идёт обратно, и пара не возвращается к исходному множеству: $U \circ L$ раздувает A до множества всех списков, а $L \circ U$ забывает операцию моноида. Пара функторов связана, однако, чем-то бóльшим, чем композиция: гомоморфизмы из свободного моноида A^* в моноид M — это в точности функции из A в носитель M . Гомоморфизм собирается из значений на буквах, и собирается единственным образом. Такое соответствие двух множеств стрелок и есть сопряжение — центральное понятие теории категорий, к которому в конце концов сводятся и пределы, и монады, и почти все универсальные конструкции.

ОПРЕДЕЛЕНИЕ 36.17 Сопряжённые функторы

Функторы $F : \mathcal{C} \rightarrow \mathcal{D}$ и $G : \mathcal{D} \rightarrow \mathcal{C}$ образуют **сопряжение** $F \dashv G$, если для любых объектов $X \in \mathcal{C}$, $Y \in \mathcal{D}$ задана биекция

$$\varphi_{X,Y} : \mathcal{D}(F(X), Y) \rightarrow \mathcal{C}(X, G(Y)),$$

естественная по X и по Y . Функтор F называют *левым сопряжённым*, G — *правым*.



Рис. 138. Сопряжение — биекция между множествами стрелок $F(X) \rightarrow Y$ и $X \rightarrow G(Y)$, согласованная со всеми остальными стрелками.

Само соответствие показано на рис. 138: слева стрелки $F(X) \rightarrow Y$, справа $X \rightarrow G(Y)$, между ними — биекция. Слово «естественная» означает согласованность со стрелками: для $x : X' \rightarrow X$, $y : Y \rightarrow Y'$ и $h : F(X) \rightarrow Y$ образ перестраивается по закону

$$G(y) \circ \varphi_{X,Y}(h) \circ x = \varphi_{X',Y'}(y \circ h \circ F(x)),$$

— по образцу квадрата естественности из 36.4. На практике это значит: соответствие двух множеств стрелок устраивается «одним правилом», без выбора.

На первом же примере $L \dashv U$ единица и коединица различимы руками: $\eta_A(a) = [a]$ — вставка элемента в одноэлементный список, а ε_M — свёртка списка элементов M операцией моноида (для $(\mathbb{Z}, +, 0)$: $[1, 2, 3] \mapsto 6$). Единица $\eta_X = \varphi_{X, F(X)}(\text{id}_{F(X)})$ идёт $X \rightarrow GF(X)$, а коединица $\varepsilon_Y = \varphi_{G(Y), Y}^{-1}(\text{id}_{G(Y)})$ идёт $FG(Y) \rightarrow Y$.

Первый пример уже разобран руками.

Пример Свободный сопряжён к забывающему

Для $L \dashv U$ между множествами и моноидами биекция переводит гомоморфизм $h : A^* \rightarrow M$ в функцию $x \mapsto h([x])$ на одноэлементных списках.

Обратное направление продолжает функцию $f : A \rightarrow U(M)$ до гомоморфизма: пустой список — в нейтральный элемент, конкатенация — в операцию моноида. Сопряжение говорит: *продолжение существует и единственно*, — и это в точности универсальное свойство свободной структуры, с которым мы уже встречались в 36.3.

Сопряжение между функторами на предпорядках — конструкция, которую курс уже использовал молча.

Пример Связки Галуа

Пусть $\mathcal{C}$ и $\mathcal{D}$ — тонкие категории предпорядков, а функторы между ними — монотонные отображения. Биекция множеств стрелок в тонкой категории — это эквивалентность «стрелка есть» на «стрелка есть», и сопряжение $F \dashv G$ принимает вид

$$F(x) \leq y \Leftrightarrow x \leq G(y).$$

Это определение связки Галуа (глава 31): пара α, γ — имена той главы, абстракция и конкретизация, к естественным преобразованиям отношения не имеющие — с условием $\alpha(X) \preceq d \Leftrightarrow X \subseteq \gamma(d)$. Весь статический анализ программ построен на этом сопряжении: приближение α и конкретизация γ — левый и правый сопряжённые между решётками состояний, и корректность переноса свойств — следствие сопряжённости.

Третье сопряжение — самое близкое программисту.

Пример Каррирование

В категории множеств (и в категории типов) биекция

$$\text{Set}(A \times B, C) \cong \text{Set}(A, C^B)$$

переводит функцию двух аргументов в функцию, возвращающую функцию: для битов $f(x, y) = x \oplus y$ переходит в $x \mapsto (y \mapsto x \oplus y)$. Объект C^B называется экспоненциалом: он изображает функции из B в C как единый объект.

Для сопряжения нужны естественности по A и по C . Естественность по B — отдельный факт, делающий $(-)^B$ функтором по B . Пара функторов $- \times B$ и $(-)^B$ образует сопряжение: взятие произведения слева сопряжено к экспоненцированию. Декартово замкнутая категория — категория с конечными произведениями и всеми экспоненциалами. Категория типов функционального языка декартово замкнута, и каррирование (глава 27) — проявление этого сопряжения. Здесь и собирается категорная семантика простых типов: тип — объект, функция — стрелка, произведение типов — $\times$, а каррирование — биекция сопряжения.

УТВЕРЖДЕНИЕ 36.8 Правый сопряжённый сохраняет пределы

Пусть $F \dashv G$ и в $\mathcal{D}$ существует предел $\lim D$ диаграммы D . Тогда $G(\lim D)$ — предел диаграммы GD в $\mathcal{C}$.

Набросок доказательства

Конус над GD с вершиной X — это семейство стрелок $X \rightarrow G(D_i)$, биективно соответствующее семейству $F(X) \rightarrow D_i$, то есть конусу над D с вершиной $F(X)$. Универсальный конус $\lim D$ пропускает его через себя единственным образом, и биекция сопряжения переводит это разложение обратно в разложение через $G(\lim D)$. Единственность сохраняется биекцией.

Про сопряжения говорят, что они «возникают везде», и это справедливо: почти каждая универсальная конструкция этой главы получается из некоторого сопряжения. Произведения — правый сопряжённый к диагональному функтору $\Delta : \mathcal{C} \rightarrow \mathcal{C} \times \mathcal{C}$, копирующему $X \mapsto (X, X)$: биекция $\mathcal{C}(X, A) \times \mathcal{C}(X, B) \cong \mathcal{C}(X, A \times B)$ — знакомое «стрелка в произведение — это пара стрелок». Свободные структуры — левые сопряжённые к забывающим. Даже связка Галуа из анализа программ — сопряжение между решётками, прочитанное в тонких категориях.

Проверка Сопряжённые функторы

1. Проверьте естественность биекции $L \dashv U$: что происходит с гомоморфизмом $A^* \rightarrow M$ при предкомпозиции функцией $k : A' \rightarrow A$?
2. Выпишите единицу сопряжения $- \times B \dashv (-)^B$: в какую стрелку переходит тождественная функция на $A \times B$?
3. Почему связка Галуа из 31 обязана сохранять точные грани одной из своих сторон? Какая именно сторона и почему?
4. Предположим, $F \dashv G$ и $F' \dashv G$. Покажите, используя биекцию, что $F \cong F'$ естественным образом.

§ 36.9 Монады и композиция эффектов

Программистская функция из A в B редко бывает просто функцией. Она может отказать (`Option`), вернуть несколько ответов (`Vec`), обратиться к вводу-выводу, изменить состояние, ждать. У всех этих историй общая скелетная схема: тип результата оборачивается, и композиция таких функций требует склеивать обёртки. Монада — категорная формализация этой склейки, до того удачная, что стала конструкцией языков программирования²⁵⁵.

Начнём с отказа, самого простого эффекта. Парсер числа в строке — функция `&str -> Option<i32>`, а валидатор — `i32 -> Option<Email>`. Как компоновать парсер с валидатором, чтобы отказ любого звена останавливал цепочку, видно из кода.

²⁵⁵B. Milewski. «Category Theory for Programmers». 2019.


```
// Композиция Клейсли: отказ любого звена отказывает всю цепочку.
fn and_then<T, U, V>(
    parse: impl Fn(T) -> Option<U>,
    validate: impl Fn(U) -> Option<V>,
) -> impl Fn(T) -> Option<V> {
    move |x| parse(x).and_then(|y| validate(y))
}
// and_then(p, v)(x) = None, если p(x) = None --- короткое замыкание
```

За этим кодом стоит категория. Объекты — типы, стрелки из A в B — функции $A \rightarrow \text{Option}(B)$. Композиция стрелок — склейка из кода выше, тождество из A в A — функция $x \mapsto \text{Some}(x)$. Ассоциативность и тождественность напрямую проверяются — и не случайно: перед нами *категория Клейсли* монады `Option`, построенная по общему рецепту, который появится сейчас.

ОПРЕДЕЛЕНИЕ 36.18 Монада

Монада на категории $\mathcal{C}$ — это функтор $T : \mathcal{C} \rightarrow \mathcal{C}$ вместе с естественными преобразованиями (Id здесь — тождественный функтор, не тождественная стрелка id_A)

- единицей $\eta : \text{Id} \rightarrow T$ (в каждой компоненте $\eta_A : A \rightarrow T(A)$) и
- умножением $\mu : T \circ T \rightarrow T$ (в каждой компоненте $\mu_A : T(T(A)) \rightarrow T(A)$),

удовлетворяющими законам (покомпонентно, в каждом объекте A):

$$\mu_A \circ T(\mu_A) = \mu_A \circ \mu_{T(A)}, \quad \mu_A \circ \eta_{T(A)} = \text{id}_{T(A)} = \mu_A \circ T(\eta_A).$$

Здесь $T(\mu_A)$ — применение функтора T к стрелке μ_A : функтор действует на преобразование покомпонентно. Первый закон говорит, что сплющивание тройной обёртки не зависит от порядка двух сплющиваний, второй — что единичные обёртки сплющиваются бесследно.

Для монады `Option` единица оборачивает значение в `Some`, а умножение `join` сплющивает `Option<Option<T>>` в `Option<T>`: внешний `None` отказывает всё, внешний `Some(x)` оставляет внутреннее. Для списка единица — одноэлементный список, умножение — конкатенация списка списков. Законы монады гарантируют, что склейки не зависят от расстановки скобок и что лишние обёртки ничего не меняют.

Пример Категория Клейсли

Категория Клейсли $\mathcal{C}_T$ монады T имеет те же объекты, что $\mathcal{C}$, а стрелки из A в B — стрелки $A \rightarrow T(B)$ исходной категории. Для стрелок Клейсли $f : A \rightarrow T(B)$ и $g : B \rightarrow T(C)$ композиция определена через единицу и умножение. Композиции справа идут в исходной категории $\mathcal{C}$, а сплющивание происходит в цели $\mathcal{C}$, как `join` в `and_then`:

$$g \circ_T f = \mu_C \circ T(g) \circ f, \quad \text{id}_A = \eta_A.$$

Для `Option` это в точности `and_then` из кода, для списка — `flat_map`. Законы монады — необходимое и достаточное условие того, что эта композиция ассоциативна и тождественна.

Происхождение монад в природе почти всегда одно: сопряжение, как в следующем примере. Это ещё одна причина, по которой сопряжения названы центральным понятием теории.

Пример Монада списка из сопряжения

Композиция $T = U \circ L$ свободного и забывающего функторов из 36.3 переводит множество A в множество всех списков над A . Единица $\eta : \text{Id} \rightarrow T$ — вставка элемента в одноэлементный список, умножение $\mu : TT \rightarrow T$ — конкатенация списка списков.

Обе структуры естественны, и законы монады выполняются: конкатенация списка списков списков не зависит от порядка сплющивания, а одноэлементные списки нейтральны. Сопряжение $L \dashv U$ породило монаду, и механизм общий: для любого $F \dashv G$ монада $T = G \circ F$ получает η — единицу сопряжения, а $\mu = G \varepsilon F$ — коединицу, пропущенную через G . Для списков это одноэлементный список и конкатенация, а схема свёртки — на рис. 139.

$$T = G \circ F$$



Рис. 139. Сопряжение $F \dashv G$ сворачивается в монаду $T = G \circ F$ на исходной категории.

Замечание Монада — моноид в категории эндофункторов

Знаменитую фразу «монада — это просто моноид в категории эндофункторов» стоит разобрать без мемов. Эндофункторы $\mathcal{C} \rightarrow \mathcal{C}$ образуют категорию — частный случай категории функторов из 36.4: объекты — эндофункторы, стрелки — естественные преобразования, композиция служит «умножением». Моноид из 12 — это объект с ассоциативной операцией и нейтральным элементом. В категории функторов ту же роль играют μ и η . Законы монады — моноидные законы, прочитанные построчно: μ ассоциативно, η нейтрально. Шутка остаётся шуткой, но за ней — точное утверждение: монады продолжают линию моноидов в новый мир, и моноид из одной главы стал строительным блоком другой.

Практическая отдача монад — дисциплина эффектов. Функция с типом $A \rightarrow \text{Option}(B)$ честно объявляет возможность отказа, $A \rightarrow \text{Vec}(B)$ — множественность, $A \rightarrow \text{State}(s, B)$ — доступ к состоянию ($\text{State}(s, B)$ — сокращение для $s \rightarrow (B, s)$: входное состояние и результат с новым состоянием). Тип — контракт, и категория

Клейсли — система композиции контрактов, в которой эффекты не теряются и не дублируются. Оператор `?` в Rust, `and_then` в идиоматичном коде, `>=>` в Haskell — синтаксис поверх одной и той же диаграммы.

Проверка Монады

1. Проверьте закон тождества для монады `Option`: `join(Some(x)) = x` и `join` после `unit` — тождественная функция.
2. Что означает ассоциативность умножения для списка списков списков? Приведите пример из трёх уровней вложенности.
3. Выпишите компоненты η и μ монады, порождённой сопряжением из 36.8 для свободной группы (моноида, где у каждого порождающего есть формальная обратная буква) вместо свободного моноида.
4. Почему для монады `Vec` композиция Клейсли — это `flat_map`, а не `map`? Что сломалось бы при замене?

§ 36.10 * Стринг-диаграммы

Стрелочная запись держит диаграмму в одном измерении: композиция идёт вдоль направления стрелок. Для функторов двух аргументов, *бифункторов* $F(A, B)$, удобнее второе измерение. Запишем объекты вертикальными *проводами*, а стрелки — узлами на них: операция $m : A \times B \rightarrow C$ рисуется узлом с двумя входящими проводами и одним исходящим, как на рис. 140.

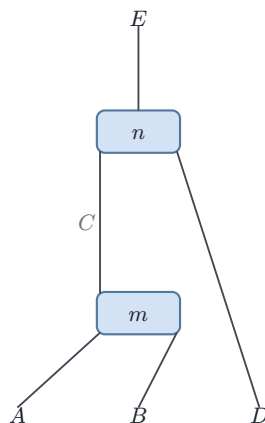


Рис. 140. Стринг-диаграмма: узлы — операции, провода — объекты, композиция идёт снизу вверх.

Композиция морфизмов — вертикальное сцепление узлов, а операции на разных проводах соединяются параллельно — горизонтально: $f : A \rightarrow B$ рядом с $g : C \rightarrow D$ дают $f \times g : A \times C \rightarrow B \times D$. Категория с операцией $\times$ на парах объектов и согласованными с ней стрелками (моноидальная категория — точное определение оставим за кадром: в ней $\times$ ассоциативно лишь с точностью до изоморфизма $(A \times B) \times C \cong A \times (B \times C)$, называемого ассоциатором) — аналог моноида, поднятого на

уровень категорий, и стринг-диаграммы — её родная нотация. Равенство диаграмм с точностью до непрерывной деформации проводов — тень законов категории, и в моноидальном мире перестановки проводов сами становятся объектами теории.

Для программиста провода — потоки данных, а узлы — операции. Поточковые графы обработки и dataflow-пайплайны — стринг-диаграммы, встреченные в индустрии. Теория категорий даёт им алгебру: когда два пайплайна эквивалентны — вопрос о деформации диаграммы, а не о пошаговой симуляции.

§ 36.11 * Алгебры и коалгебры

Свёртка списка с нулём и шагом (глава 12) до сих пор оставалась просто идиомой кода. Категорный взгляд превращает её в теорему о начальном объекте.

Фиксируем функтор F на категории. F -алгеброй называется объект X со стрелкой $F(X) \rightarrow X$: алгебра — способ сворачивать « F -образные конструкции над X » в элемент X . Для списка подходит $F(X) = 1 + A \times X$: конструкция — либо пустой маркер, либо пара (голова, хвост). Алгебра — пара операций: что вернуть для пустого и как обработать голову с хвостом, то есть в точности аргументы `fold`.

Стрелки этой категории — гомоморфизмы F -алгебр: функции h между носителями, сохраняющие операцию свёртки, $h \circ \alpha = \beta \circ F(h)$.

ОПРЕДЕЛЕНИЕ 36.19 Начальная алгебра

F -алгебра $\alpha : F(\mu F) \rightarrow \mu F$ называется **начальной**, если из неё в любую алгебру $\beta : F(X) \rightarrow X$ ведёт единственный гомоморфизм.

Обозначение μF — стандартное имя носителя начальной алгебры, «наименьшая неподвижная точка» функтора F (это μ — не монадное умножение выше, а лишь буква). Единственный гомоморфизм из μF в X называют **катаморфизмом**: свёртка структуры по шагу β . Для списка это `fold`, для дерева — свёртка по вершинам: в обоих случаях свёртка — единственный гомоморфизм из начальной алгебры.

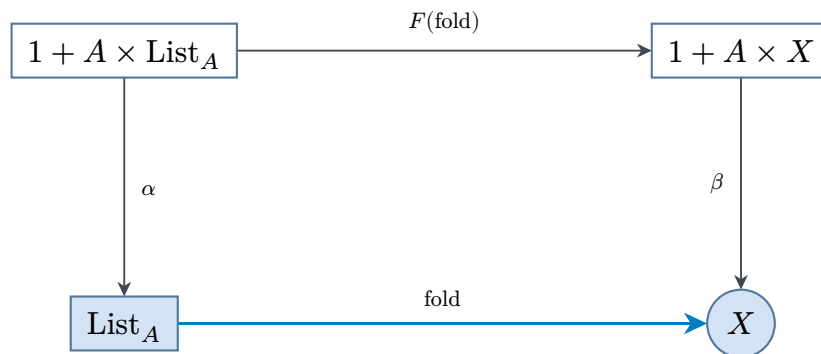


Рис. 141. Список — начальная алгебра $1 + A \times (-)$: свёртка в любую структуру существует и единственна.

Начальная алгебра функтора $1 + A \times (-)$ — список A^* : его алгебра — конструкторы nil и cons , а катаморфизм — свёртка по этим конструкторам, как на рис. 141. Начальность честна: каждый элемент списка собирается из конструкторов за конечное число шагов, поэтому значение гомоморфизма на нём вычисляется шаг за шагом, и задаётся оно значениями на nil и cons однозначно. Единственность из универсального свойства объясняет, почему две свёртки с одинаковыми нулём и шагом совпадают: раньше требовалась индукция по списку, теперь достаточно универсального свойства. Рекурсивные типы в функциональных языках — неподвижные точки функторов, и компилятор с их универсальным свойством разбирается за нас.

Двойственная конструкция смотрит в другую сторону: F -коалгебра — объект X со стрелкой $X \rightarrow F(X)$, способ *разворачивать* элемент в структуру. Коалгебра функтора $2 \times (-)^A$ — автомат (2 — двухэлементное множество ответов «принимает/не принимает»): элемент-состояние разворачивается в пару «принимаящее ли это состояние» и «функция перехода по букве»²⁵⁶. *Финальная* коалгебра собирает все возможные поведения, а уникальное отображение в неё — минимизация: два состояния, неразличимые наблюдениями, склеиваются (глава 23). Алгебры задают, как строить данные из частей, а коалгебры — как наблюдать системы в работе. Индукция и коиндукция — двойственные взгляды на одно и то же определение, а ближе всего программистам двойственность видна в теории автоматов.

§ 36.12 * Метрики, вероятности и нечёткость

Определение категории требует, чтобы стрелки каждого hom -множества составляли множество — обычную структуру без внутреннего устройства. Потребуем большего: пусть $\mathcal{C}(A, B)$ живёт в другой категории, например в категории чисел. Такие категории называются *обогащёнными*, и главный пример — метрика.

Зафиксируем категорию чисел $[0, \infty]$: один «морфизм» $a \rightarrow b$ при $a \geq b$, композиция — транзитивность неравенства (направление выбрано так, чтобы композиция обогащённой категории превратилась в неравенство треугольника, а не в обратное к нему). Поверх композиции живёт моноидальная операция — сложение с нейтральным нулём: она складывает «расстояния», и именно она стоит в правиле композиции обогащённой категории. Обогащённая категория над ней задаётся функцией $d(A, B) \in [0, \infty]$ на парах объектов, где композиция и тождество читаются как неравенства: $d(A, A) = 0$ и $d(A, C) \leq d(A, B) + d(B, C)$. Второе — неравенство треугольника: категорное определение выдало аксиомы метрики почти бесплатно. Взгляд Ловер на метрические пространства как на обогащённые категории²⁵⁷ перекидывает мост между анализом и алгеброй: пополнение пространства оказывается копределом, а сжимающие отображения — функторами. Аксиомы при этом дают

²⁵⁶J. J. M. M. Rutten. «Universal coalgebra: a theory of systems». Theoretical Computer Science, 2000.

²⁵⁷F. W. Lawvere. «Metric spaces, generalized logic, and closed categories». Rendiconti del Seminario Matematico e Fisico di Milano, 1973.

квазиметрику: симметрия и делимость из обогащения не следуют и требуют отдельных условий.

Нечёткие множества (глава 35) встают в ту же рамку. Заменяем $[0, \infty]$ на отрезок $[0, 1]$ с t -нормой — ассоциативной операцией с нейтральным элементом 1, нечётким аналогом сложения (точное определение — в главе 35): «композиция» степеней принадлежности a, b — это $t(a, b)$. Обогащённая категория над $[0, 1]$ — структура с *порогами*: степень $d(A, B)$ того, что « A переходит в B », удовлетворяет треугольному неравенству $d(A, C) \leq t(d(A, B), d(B, C))$. Функция принадлежности $\mu : U \rightarrow [0, 1]$ (глава 35) — та же конструкция, прочитанная обогащённо: стрелка в категорию значений, а выбор t -нормы — выбор моноидальной структуры на значениях. Вероятности — третий сюжет той же рамки: в марковских категориях морфизмы несут распределения, а условные вероятности становятся композициями.

В одной секции встретились три сюжета из разных концов математики, и механизм у них общий: Ном-объекты несут внутреннюю структуру, а категорные аксиомы согласуют эту структуру с композицией. Это и есть категорный стиль работы: не изобретать язык для каждой области заново, а находить уже готовый.

§ 36.13 * Топосы и внутренняя логика

В категории множеств логика живёт внутри. Подмножество $S \subseteq A$ — это то же самое, что характеристическая функция $A \rightarrow \{0, 1\}$ (глава 4), и логические операции над подмножествами — поточечные операции над нулями и единицами. Пересечение — минимум значений, объединение — максимум, дополнение — переворот. Категории, в которых логика живёт так же без всяких элементов, определяются двумя ингредиентами. Декартова замкнутость из примера о каррировании даёт произведения и экспоненциалы — «функции как объекты». А логику даёт *классификатор подобъектов*: объект Ω вместе с естественной биекцией между подобъектами любого объекта A (подмножествами с точностью до переименования) и стрелками $A \rightarrow \Omega$. В $\mathbf{Set}$ это в точности соответствие «подмножество $\leftrightarrow$ характеристическая функция»: $\Omega = \{0, 1\}$. Категория с конечными произведениями, экспоненциалами и классификатором подобъектов называется *топосом*.

Классификатор превращает логику в геометрию. Стрелка $A \rightarrow \Omega$ — это «высказывание об A с доказательствами»: в $\mathbf{Set}$ оно принимает значение 1 ровно на элементах подмножества. Операции логики становятся стрелками над Ω : унарные — из Ω , как отрицание $\Omega \rightarrow \Omega$, бинарные — из $\Omega \times \Omega$, как конъюнкция и импликация. Они задаются не таблицей, а устройством категории, и таблица получается как следствие.

Дальше происходит неожиданное: алгебра операций на Ω — это алгебра Гейтинга из интуиционистской главы курса (глава 34). Импликация Гейтинга удовлетворяет $p \wedge (p \rightarrow q) \rightarrow q$, но закон исключённого третьего $p \vee (\neg p)$ не обязан выполняться: в общем топосе у Ω больше двух «степеней истинности», и отрицание возвращает не всюду ложь. Интуиционистская логика оказалась точной логикой внутреннего

устройства топосов, а классическая — её частным случаем: классика возвращается там, где Ω булева, как в $\mathbf{Set}$ или в топосах пучков над дискретным пространством.

Примеры топосов показывают масштаб конструкции. Категория множеств — простейший топос, его внутренняя логика классична. Категория пучков над топологическим пространством — топос, чья логика чувствует геометрию пространства: истинность может меняться от точки к точке, и «существует» означает «существует в окрестности». Двойственность Стоуна из 10 в этой рамке — теорема о представлении булевых алгебр: каждая булева алгебра собирается из открытых множеств своего стоунового пространства.

Для программиста след топоса — теория типов с зависимыми типами и доказательные ассистенты. Предикат-семейство типов, высказывание-тип с доказательствами-элементами: логика и типы населяют одну структуру, и соответствие Карри–Говарда из 27, продолженное Ламбеком до категорий, — первый пример того, как логика и типы садятся в одну структуру²⁵⁸.

§ 36.14 Итоги главы

Глава началась с совпадений: произведение множеств, прямое произведение автоматов, конъюнкция, НОД — разные главы, один рисунок с парой проекций и универсальным свойством. Ответ оказался языком: категория — объекты, стрелки и композиция — в котором «быть произведением» — свойство положения среди стрелок, и оно проверяется один раз, а работает в любой категории. Язык отработал на всём пройденном материале. Моноиды и порядки оказались категориями в двух крайних режимах, связка Галуа из анализа программ — сопряжением, свёртка списка — катаморфизмом начальной алгебры, а каррирование и нечёткость встали в общий ряд конструкций.

Три уровня языка сложились в пирамиду. Категории описывают отдельные миры, функторы переносят между мирами, сохраняя композицию, а естественные преобразования сравнивают переносы. На этой тройке держатся универсальные свойства и пределы, лемма Йонеды с её «объект — сеть отношений», сопряжения и монады. На каждом уровне стояли рабочие примеры: булеан и прообраз, свободный список и забывающий функтор, `Option` и `Vec` в роли монад.

Для программиста итог конкретнее. Типы и функции образуют категорию, полиморфные конструкции — функторы и естественные преобразования, алгебраические типы — начальные алгебры, эффекты — монады, а тип с эффектом — стрелка Клейсли. Категорный язык не заменяет практику кода, но даёт ей словарь, в котором `map`, `and_then`, `flat_map` и `fold` стоят точные универсальные свойства.

²⁵⁸J. C. Baez, M. Stay. «Physics, Topology, Logic and Computation: A Rosetta Stone». New Structures for Physics, 2011.

Проверка Проверьте себя

1. Что общего у моноида и частично упорядоченного множества как категорий и чем они различаются?
2. Сформулируйте универсальное свойство произведения и объясните, почему НОД ему удовлетворяет в категории делимости.
3. Что утверждает лемма Йонеды и как она объясняет фразу «объект определяется своими отношениями»?
4. Почему связка Галуа из абстрактной интерпретации — частный случай сопряжения функторов?
5. Как монада делает композицию функций с эффектами ассоциативной? Что в ней играет роль нейтрального элемента?
6. Что общего между свёрткой списка и минимизацией автомата?

Что дальше?

Категорный аппарат не заканчивается на этой главе — он только начат. Моноидальные категории и стринг-диаграммы ведут в квантовые вычисления и теорию сетей, высшие категории — в топологию и гомотопическую теорию типов, пучки и топосы — в геометрию и логику. Прикладная волна применяет тот же язык к базам данных, вероятностям и машинному обучению. Словарь у этого языка один на всех, а глубина владения им растёт с запасом примеров.

§ 36.15 Упражнения*Категории и примеры.*

1. Проверьте, что $(\mathbb{Z}_n, +, 0)$ задаёт категорию с одним объектом. Выпишите таблицу композиции стрелок для $n = 4$ и найдите изоморфизмы.
2. Постройте свободную категорию графа с рёбрами $A \rightarrow B$, $B \rightarrow C$, $A \rightarrow C$. Сколько стрелок из A в C ? Какие пары стрелок компонуемы?
3. Докажите, что в тонкой категории порядка каждая стрелка — мономорфизм и эпиморфизм одновременно.
4. * Найдите категорию, в которой стрелка является мономорфизмом и эпиморфизмом, но не изоморфизмом. Укажите объекты, стрелки и проверьте все три свойства.

Функторы и естественность.

5. Проверьте, что взятие носителя переводит категорию **Mon** в **Set**: что происходит с тождественным гомоморфизмом и композицией?

6. Для функции $f : \{1, 2\} \rightarrow \{a, b\}$, $f(1) = a$, $f(2) = a$ вычислите образ множества $\{1, 2\}$ и прообраз множества $\{b\}$. Проверьте на этом примере контравариантность прообраза для композиции с $g : \{a, b\} \rightarrow \{x, y\}$, $g(a) = x$, $g(b) = y$.
7. Докажите, что семейство $\alpha_A(x) = \{x\}$ из A в $\mathcal{P}(A)$ естественно: выпишите квадрат и проверьте обе функции на элементах.
8. * Функтор $\mathcal{C}(A, -)$ переводит объект в множество стрелок. Докажите, что он сохраняет произведения: $\mathcal{C}(A, B \times C) \cong \mathcal{C}(A, B) \times \mathcal{C}(A, C)$.

Универсальные свойства и Йонеда.

9. Найдите терминальные и начальные объекты в категории Set^{op} . Сравните с Set .
10. Проверьте универсальное свойство НОД на паре $(12, 18)$: любой общий делитель делит НОД, и как из этого следует единственность.
11. Постройте соответствие леммы Йонеда для функтора $\mathcal{P}$ булеана на одноэлементном множестве $A = \{1\}$: перечислите естественные преобразования $\mathcal{C}(A, -) \rightarrow \mathcal{P}$ и укажите их образы в $\mathcal{P}(A)$.
12. * Докажите, что $\mathcal{P}(A) \cong \text{Set}(A, 2)$: булеан — представимый функтор. Какой объект его представляет?

Сопряжения и монады.

13. Выпишите биекцию сопряжения $L \dashv U$ явно для двухэлементного моноида $(\{e, s\}, \star)$ с таблицей $s \star s = e$ и множества $A = \{a\}$: сколько гомоморфизмов $A^* \rightarrow \{e, s\}$ и сколько функций $A \rightarrow \{e, s\}$?
14. Проверьте, что (α, γ) (глава 31) со знаковой областью задаёт связку Галуа: $\alpha(X) \preceq d \Leftrightarrow X \subseteq \gamma(d)$ для $X = \{-2, 3\}$, $d = +$.
15. Для монады `Option` проверьте закон тождества на формах `join(Some(Some(x))) = Some(x)` и `join(Some(None)) = None`, а ассоциативность умножения — на тройной вложенности `Option<Option<Option<i32>>>>`.
16. * Пусть T — монада списка. Что такое категория Клейсли для T ? Выпишите композицию стрелок $A \rightarrow T(B)$ и $B \rightarrow T(C)$ через `flat_map` и проверьте ассоциативность на примере списков длины 2.

ПРОЕКТ Конечные категории под микроскопом

Реализуйте библиотеку конечных категорий и проверьте на ней главные теоремы — от аксиом до леммы Йонеда — конечным перебором. Категория представляется списком объектов и таблицей композиции стрелок, поэтому каждое утверждение главы превращается в проверяемый эксперимент.

① Категории из графов, порядков и моноидов

Структуры: категория из моноида (объект один, стрелки — элементы), тонкая категория из отношения порядка и свободная категория ориенти-

рованного графа (стрелки — пути, композиция — конкатенация путей). Проверка аксиом: тотальность композиции, ассоциативность и тождества — циклом по таблице. Убедитесь, что нетранзитивное отношение честно проваливает проверку.

② **Функторы и естественность перебором**

Функтор — пара отображений (объекты, стрелки) с проверкой законов. Постройте функторы между своими категориями: забывающий, коллапс свободной категории в цепочку, удвоение моноида. Естественные преобразования перечисляются декартовым произведением Ном-множеств с фильтром по коммутативности квадратов.

③ **Лемма Йонеды**

Для объекта A постройте Ном-функтор $\mathcal{C}(A, -)$ как функтор в конечные множества. Перечислите все естественные преобразования $\mathcal{C}(A, -) \rightarrow F$ для нескольких F и сравните с элементами $F(A)$: биекция из леммы должна проявиться численно, включая соответствие $\alpha \mapsto \alpha_A(\text{id}_A)$.

④ **Клейсли в коде**

Реализуйте композицию Клейсли для `Option` и `Vec` и проверьте ассоциативность и тождества на наборах примеров, включая отказы на середине цепочки. Сравните число успешных прогонов `Vec`-цепочки с предсказанием списочной монады.

Итог: библиотека с проверкой аксиом, таблицей перечисленных естественных преобразований и численным подтверждением биекции Йонеды на трёх категориях: моноид $\mathbb{Z}_4$, цепочка $0 < 1 < 2$, свободная категория графа из двух рёбер.

Заключение

Мы начали с различия между непрерывным и дискретным. Всё, что вычисление обрабатывает, распадается на отдельные элементы: биты, состояния, шаги. Между соседними элементами дискретного нет ничего промежуточного, и эта отделённость позволяет говорить о нём точно: про каждый элемент можно сказать, входит он в рассматриваемый объект или нет.

Первую часть пути мы учились строить такое описание. Множества, отношения, функции, графы — это способы называть дискретные объекты и связи между ними, ничего не пропуская и не добавляя лишнего. Определение — договор о том, что слово значит, и пока договора нет, рассуждать не о чем.

Следующим шагом стало доказательство. Оказалось, что доказательство можно устроить как последовательность шагов, где каждый опирается только на определения и на предыдущие. Такое рассуждение не требует доверия к говорящему: его можно проверить механически.

Потом язык описания стал инструментом построения. Мы строили числа из множеств: от натуральных, заданных как ординалы фон Неймана, до действительных, собранных сечениями Дедекинда. Логика, записанная как действия с нулями и единицами, ложится в основу схем. Коды позволяют исправить ошибку в переданном слове, не требуя повторной передачи. Криптография защищает сообщение от чужого чтения.

Затем мы научились считать. Комбинаторика считает огромные множества вариантов, не перебирая их: число перестановок шестидесяти предметов превышает число атомов в наблюдаемой Вселенной и всё же уместается в одну строку. Вероятность приписывает исходам веса там, где единственного исхода нет.

Конечные автоматы описывают простейшую из вычислительных моделей. Машины Тьюринга и лямбда-исчисление — два способа сказать, что значит «вычислимо», и они равносильны (тезис Чёрча–Тьюринга). Теория типов добавила к вычислению дисциплину: она решает, каким выражениям придать смысл, прежде чем их вычислять.

Наконец мы дошли до границ вычисления. Одни задачи не решаются никаким алгоритмом. Другие решаются, но так медленно, что дождаться ответа нельзя. Эти границы следуют из устройства самих задач. Рядом с этой границей работает и инженерия: статический анализ программ отвечает приближённо там, где точный ответ невозможен.

Ограничено не только вычисление — ограничено и само точное описание. Не всё сводится к однозначному описанию: есть случайность, нечёткость, приближение.

С ними тоже можно работать — измерять случайность, допускать степени между «да» и «нет», оценивать погрешность заранее.

В самом конце пути язык структур собрался в одну рамку: теория категорий описывает множества, группы, автоматы и типы одними и теми же понятиями — объектом, стрелкой, диаграммой. Это язык, на котором разделы математики разговаривают между собой, и открытая дверь в её дальнейшие области.

Сквозная тема всех этих шагов — точность: описать точно, рассуждать точно, считать точно и знать, где точность недостижима. Определения всех понятий собраны в словаре терминов в конце книги. Навык точного мышления приобретается только собственной работой.

Этот навык открывает математическую литературу: статьи, монографии и документацию можно читать без переводчика. Главы заканчивались указанием, куда можно двинуться дальше. Выбор направления остаётся за читателем.

Указания к упражнениям

Глава 1. Логика и доказательства

- **Задача 3** (штрих Шеффера). Выразите сначала отрицание через штрих Шеффера, затем подставьте его в закон де Моргана. Штрих Шеффера — самостоятельная связка: для выражения дизъюнкции хватит одного типа вложений.
- **Задача 9** (контрпример к делимости). Проверьте, что влечёт одновременная делимость на 6 и на 10, — и сравните с произведением 60. Ищите значение, которое делится на оба числа, но не на их произведение.
- **Задача 11** (тройки Хоара). Вычислите слабое условие: подставьте $x + 1$ вместо x в постусловие и сравните с данным предусловием. Так проверяются обе тройки одновременно: одна станет корректной, другая — нет.
- **Задача 14** (ошибка в индукции про лошадей). Ошибку ищите так: проверьте индукционный переход на минимальных k , начиная с базы, и посмотрите, сломается ли он. В правильном доказательстве шаг обязан работать для самого первого перехода, где ещё нет пересечения групп.
- **Задача 15** (сильная индукция и разложение на простые). Для составного k разложите его $k = ab$; оба множителя строго меньше k . К ним применимо индукционное предположение — в отличие от делителей, никак не связанных с $k - 1$.
- **Задача 16** (ошибка в индукции про Фибоначчи). Проверьте сам индукционный переход: что даёт сумма двух нечётных чисел $F_k + F_{k+1}$? Свойство наследуется от *пары* предыдущих — посмотрите, что происходит с этой парой по модулю 2.

Глава 2. Дедукция

- **Задача 12** (закон Пирса). Закон Пирса классический: доказывайте косвенно, предположив $\neg P$, и найдите, на каком шаге нужен классический приём. Из $\neg P$ и гипотезы $(P \rightarrow Q) \rightarrow P$ попробуйте вывести противоречие.
- **Задача 13** (импликация и дизъюнкция). Прямой стратегии нет: из $A \rightarrow B$ не следует ни $\neg A$, ни B . Попробуйте косвенно: предположите отрицание цели и выведите противоречие.

- **Задача 14** (мнимое интуиционистское снятие двойного отрицания). Ищите место, где правило вывода применяется неявно: снятие двойного отрицания — классическое правило. В интуиционистском выводе ему неоткуда взяться, кроме косвенного доказательства.

Глава 3. Логика предикатов

- **Задача 5** (многосортная запись). Введите сорта и предикат, связывающий их, и ограничьте кванторы по сортам. Сравните с односортной записью, где кванторы ограничиваются предикатами-ограничителями.
- **Задача 7** (компактность и следствие конечности). Доказывайте от противного: предположите, что φ следует из Γ , но не из конечного подмножества. Посмотрите, что скажет компактность о множестве $\Gamma \cup \{\neg\varphi\}$.

Глава 4. Множества

- **Задача 6** (разность и пересечение). Докажите двойным включением, расписав условие $x \in A \setminus (B \cap C)$ через принадлежность и отрицания. Ключ — логический закон де Моргана, применённый к условию «в A и вне пересечения».
- **Задача 9** (дистрибутивность декартова произведения). Распишите членство пары (x, y) в $A \times (B \cup C)$ как два условия и примените дистрибутивность логической конъюнкции. Для пересечения та же цепочка с конъюнкцией вместо дизъюнкции.
- **Задача 13** (типы Option и Result). Option и Result — размеченные (дизъюнктные) объединения: формально произведение метки на тип, слитое дизъюнктно. Конструкторы играют роль меток, разделяющих варианты.

Глава 5. Отношения

- **Задача 3** (антисимметричность и асимметричность). Сравните определения на диагонали: что делает каждое из свойств при $a = b$? Различие именно там, где a и b совпадают.
- **Задача 6** (транзитивные замыкания). R^+ — пути длины не менее 1, R^* добавляет пути длины 0. Нарисуйте граф с циклом и сравните: какие пары добавляет R^* сверх R^+ ровно из-за нулевых путей?
- **Задача 9** (отношения эквивалентности и числа Белла). Отношение эквивалентности на A соответствует разбиению A : классы и есть блоки. Осталось понять, сколько бывает разбиений — свяжите с числами Белла.

- **Задача 10** (смежные классы по рациональным). Проверьте три аксиомы через замкнутость $\mathbb{Q}$: вычитание, взятие противоположного и композицию разностей. Класс элемента — сдвиг; оцените его мощность и число классов, опираясь на счётность $\mathbb{Q}$.
- **Задача 12** (теорема Рамсея). Зафиксируйте одного человека и посмотрите на его связи с остальными пятью: примените принцип Дирихле. Одна из групп (знакомых/незнакомых с ним) содержит не менее трёх человек — исследуйте связи внутри неё.

Глава 6. Функции

- **Задача 6** (число инъекций и сюръекций). Инъекция — выбор нескольких различных значений из области; посчитайте число таких выборов (убывающий факториал). Для сюръекций оцените мощность образа: куда отображаются элементы меньшего множества?
- **Задача 7** (биекция $\mathbb{N} \rightarrow \mathbb{Z}$). Перечислите целые числа зигзагом и задайте f отдельно на нечётных и на чётных n . Разбейте на три ветки и проверьте, что их образы не пересекаются и покрывают всё.
- **Задача 10** (обратная к композиции). Разворачивать цепочку нужно в обратном порядке: что композиция делала первым, то снимается последним. Проверьте на отдельном элементе, применяя шаги в обратном порядке.
- **Задача 12** (прообраз и пересечение). Прообраз действует поточечно: $x \in f^{-1}(C \cap D)$ распишите как два условия об одном объекте. Для образа проверьте, чем он отличается: он может «склеить» разные точки.
- **Задача 14** (симметричность Θ). Сравните определения $\mathcal{O}$ и Ω : одно неравенство переходит в другое обращением константы. Примените это наблюдение к обоим границам Θ .

Глава 7. Кардиналы

- **Задача 3** (счётность $\mathbb{Z}$). Постройте биекцию с $\mathbb{N}$ зигзагом и задайте f отдельно на нечётных и на чётных n . Проверьте, что образы веток не пересекаются и покрывают всё $\mathbb{Z}$.
- **Задача 8** (мощность континуума). Сопоставьте подмножеству его характеристическую функцию, а затем свяжите $\{0, 1\}^\infty$ с $[0, 1]$ двоичным разложением. Неоднозначность разложения — только у двоично-рациональных чисел; посмотрите, как от неё избавиться.

- **Задача 10** (квадрат континуума). Сведите $\mathbb{R}$ к $(0, 1)$ и постройте инъекции в обе стороны: одна — вложение, другая — чередование. Далее примените Кантора–Бернштейна–Шрёдера.
- **Задача 13** (невычислимые языки). Языков столько же, сколько функций $\mathbb{N} \rightarrow \{0, 1\}$, а программ — счётно много. Диагональный аргумент примените к любому перечислению таких функций.
- **Задача 14** (ошибка в «счётности» интервала). Проверьте, покрывает ли предложенное перечисление бесконечные двоичные дроби. Сопоставьте с диагональным аргументом о несчётности интервала.

Глава 8. Отношения порядка

- **Задача 4** (включение как частичный порядок). Проверьте три аксиомы включения; инфимум и супремум пары — пересечение и объединение. Нелинейность покажите на несравнимой паре.
- **Задача 9** (число частичных порядков). Для $n = 3$ переберите варианты по числу сравнимых пар: антицепь, одна пара, цепь. Сверьте с известной последовательностью 1, 3, 19 для $n = 1, 2, 3$.
- **Задача 13** (законы поглощения). Докажите два неравенства и примените антисимметричность. Покажите, что a — верхняя грань множества $\{a, a \wedge b\}$, и что она же — его точная верхняя грань.
- **Задача 14** (теорема Кнастера–Тарского). В конечной решётке запустите итерацию снизу и докажите неубывание по монотонности. Стабилизация в конечной решётке даёт неподвижную точку; наименьшесть — сравнением с произвольной неподвижной.
- **Задача 16** (словарный порядок без минимума). Подберите множество, в котором каждая строка имеет меньшую соседнюю в лексикографическом порядке. Смотрите на первые различные символы.
- **Задача 19** (единственная топологическая сортировка). Рассмотрите два случая: все пары сравнимы — позиция фиксирована; есть несравнимая пара — её можно упорядочить двумя способами.
- **Задача 20** (ошибка про максимальный элемент). Проверьте, где доказательство опирается на существование наибольшего элемента. В бесконечном ЧУМ цепь может быть неограниченной сверху — найдите такой пример.

Глава 9. Графы

- **Задача 2** (число вершин нечётной степени чётно). Посмотрите на сумму всех степеней по модулю 2. Каждое слагаемое чётной степени даёт ноль, а нечётной — единицу.
- **Задача 4** (формула Кэли для $n = 4$). Прямое перечисление удобно через коды Прюфера: они биективны деревьям. Сравните число таких последовательностей с формулой Кэли.
- **Задача 5** (ошибка: дерево и путь через все вершины). Проверьте индукционный шаг: всегда ли удалённый лист можно дописать в конец получившегося пути? Посмотрите на малое дерево, где сосед листа — не конец пути.
- **Задача 7** (единственность остова при различных весах). Обменный аргумент: в двух разных остовах возьмите лёгчайшее ребро одного, отсутствующее в другом, и переставьте. Разные веса гарантируют, что полученный остов оказался бы легче — противоречие с минимальностью.
- **Задача 11** (граф Петерсена и условие Дирака). Условие Дирака лишь достаточное — его невыполнение ничего не говорит. Что должно остаться в дополнении гамильтонова цикла, если он есть? Посчитайте совершенные паросочетания.
- **Задача 12** (Дейкстра и отрицательные веса). Корректность Дейкстры опирается на неотрицательность: извлечённая из кучи вершина уже не улучшится. Попробуйте добавить одно ребро отрицательного веса и проследите, где это сломает аргумент.
- **Задача 16** (непланарность K_5 и $K_{3,3}$). Начните со стандартного неравенства $E \leq 3V - 6$ для K_5 . Для двудольного $K_{3,3}$ усильте его: каждая грань ограничена минимум четырьмя рёбрами.
- **Задача 19** (паросочетание как поток). Постройте сеть, в которой поток величины k отвечает паросочетанию из k рёбер. Целочисленность Форда–Фалкерсона даёт поток из нулей и единиц.
- **Задача 20** (ошибка в индукции про двудольность). Проверьте, когда вершину можно вернуть в две доли после удаления. Критерий двудольности — отсутствие нечётных циклов: попробуйте граф, где индукция «докажет» слишком много.

Глава 10. Булева алгебра

- **Задача 3** (линейные булевы функции). Линейная функция — полином Жегалкина без мономов второй степени. Посчитайте, сколько коэффициентов остаётся свободными.
- **Задача 6** (критерий Поста для $\{\text{IMPLY}, 0\}$). Проверьте каждый из пяти предположений Поста: что нарушает импликация, а что — константа 0? Достаточно, чтобы ни один класс не вмещал всё множество.

- **Задача 9** (единственность СДНФ). Минтерм истинен ровно на одном наборе — СДНФ собрана из минтермов единичных строк. Покажите, что любой минтерм, попавший в одну СДНФ, обязан быть и в другой.
- **Задача 11** (существенные импликанты функции большинства). Найдите для каждого терма набор, где истинна только он, — тогда он не поглощается остальными. Каждый из трёх термов покрывает свою «личную» единицу.
- **Задача 14** (единственность полинома Жегалкина). Сравните число полиномов Жегалкина с числом булевых функций. Над $GF(2)$ сложение $\oplus$ не коллапсирует: $x \oplus x = 0$.
- **Задача 16** (ROBDD и порядок переменных). ROBDD каноничен при фиксированном порядке, но размер зависит от числа сливаемых узлов. Попробуйте два порядка на одной функции и сравните размеры.
- **Задача 17** (XOR линейна). Сумма по модулю 2 — полином первой степени. Разберите классы Поста отдельно: какие сохраняет, какие нарушает $\oplus$?

Глава 11. Схемы

- **Задача 6** (схема на одних NAND). Выразите каждую операцию через NAND и сложите размеры. Отрицание и конъюнкция на NAND — самый короткий путь.
- **Задача 9** (ripple-carry и carry-lookahead). Сравните, как перенос распространяется в каждой схеме: цепочкой или деревом. Глубина и размер меняются в противоположных направлениях.
- **Задача 11** (вторая половина кода Грея). Вторая половина — обращение первой. Посмотрите, зачем нужно обращение, на стыке половин. Сравните последнюю строку первой половины с первой строкой второй.
- **Задача 13** (мощностной аргумент Шеннона). Сравните число схем размера не более s с числом функций от n переменных. Посчитайте, сколько схем и сколько функций, и найдите, где аргумент становится решающим.
- **Задача 14** (неконструктивность мощностного аргумента). Аргумент лишь сравнивает мощности: доказывает существование, не предъявляя функцию. Подумайте, что даёт таблица истинности такой функции.

Глава 12. Алгебраические структуры

- **Задача 3** (подсчёт операций с нейтральным элементом). Строки и столбец с индексом e предопределены тождеством; остаются свободные клетки. Посчитайте число свободных клеток и значений для них, а затем проверьте ассоциативность на примере.

- **Задача 6** (подгруппы порядка 4 в $\mathbb{Z}_{16}$). В циклической группе на каждый порядок, делящий порядок группы, — ровно одна подгруппа. Найдите элемент порядка 4: он порождает подгруппу нужного размера.
- **Задача 8** (малая теорема Ферма из Лагранжа). Группа $\mathbb{Z}_p^*$ ненулевых вычетов имеет порядок $p - 1$. Порядок элемента делит порядок группы — примените это к a^{p-1} .
- **Задача 11** (поле $\text{GF}(4)$). Проверьте неприводимость $x^2 + x + 1$ над $\mathbb{Z}_2$ и работайте по модулю неё. Найдите примитивный элемент: его степени должны пробегать все ненулевые элементы.
- **Задача 13** (поле $\text{GF}(8)$). Работайте по модулю $x^3 + x + 1$: выражайте каждую следующую степень через предыдущие. Смотрите, когда степени вернутся к 1.
- **Задача 17** (орбита и стабилизатор). Свяжите орбиту со смежными классами по стабилизатору: $g \cdot x = g' \cdot x$ равносильно $g'^{-1}g \in \text{Stab}(x)$. Примените к действиям поворота на вершинах и на раскрасках квадрата.

Глава 13. Коды

- **Задача 3** (нет кода длины 5 с 4 словами и расстоянием 4). Переносом сделайте одно слово нулём (расстояния сохраняются) и оцените веса остальных. Посмотрите на минимальное расстояние между словами большого веса.
- **Задача 6** (код Хэмминга $(7, 4)$ совершенный). Посчитайте объём шара радиуса 1 в $\{0, 1\}^7$ и сравните с числом слов. Признак совершенности однократно исправляющего кода: произведение числа слов и объёма шара равно 2^n .
- **Задача 9** (гауссов биномиальный коэффициент). Считайте упорядоченные базисы: сколько способов выбрать первый и второй вектор? Поделите на число упорядоченных базисов в одном подпространстве.
- **Задача 11** (префиксность кода Хаффмана). В дереве Хаффмана символы хранятся только в листьях. Свяжите префиксность с тем, что лист одного слова не может лежать на пути к другому.
- **Задача 13** (неконструктивность теоремы Шеннона). Доказательство выбирает код случайно и показывает, что средняя вероятность ошибки мала. Раз среднее мало — есть код не хуже среднего, но доказательство не указывает, какой именно.
- **Задача 16** (совершенство кода-повторения). Код-повторение исправляет $t = \lfloor (n - 1)/2 \rfloor$ ошибок; посмотрите на сумму размеров шаров. При нечётном n посимметрируйте биномиальную сумму. При чётном n проверьте слово с ровно $n/2$ единицами.

- **Задача 17** (расстояние кода Рида—Соломона). Многочлен степени не выше $k - 1$ имеет не более $k - 1$ корней. Сравните число совпадений двух кодовых слов — это даст минимальное расстояние и границу Синглтона.

Глава 14 SAT

- **Задача 5** (выполнимость и общезначимость). Свяжите двойственность: одна общезначима тогда и только тогда, когда другая невыполнима. Проверьте на парах противоположных формул.
- **Задача 8** (2-SAT за полиномиальное время, 3-SAT NP-полна). В 2-SAT дизъюнкт даёт две импликации и задачу сводится к графу импликаций. Третий литерал ломает эту редукцию — посмотрите, почему попарная замена терпит неудачу.
- **Задача 11** (гамильтонов цикл как SAT). Введите переменные «вершина занимает позицию» и выразите ограничения задачи. Сравните наивное перечисление перестановок с полиномиальным числом переменных и дизъюнктов.
- **Задача 13** (анализ конфликта CDCL). Постройте граф импликаций и посмотрите, какие решения вносят вклад в конфликт. Конфликтный дизъюнкт собирается из «наиболее свежих» решений.

Глава 15. SMT

- **Задача 15** (subset sum NP-полна). Заполните таблицу истинности, разбив разряды на переменные и служебные числа. Для переменной заведите числа с единицей в переменном разряде, для дизъюнкта — служебные числа. Целевая сумма должна обеспечить выбор ровно одного литерала на переменную и выполнение каждого дизъюнкта.
- **Задача 16** (жадный алгоритм не решает SAT). Жадный выбор значения по текущим дизъюнктам не гарантирует выполнимости остатка. Постройте контр-пример: на каждом шаге выбор кажется разумным, но остаётся невыполненный дизъюнкт, хотя формула выполнима.

Глава 15 SMT

- **Задача 5** (разностные ограничения и граф). Приведите все ограничения к виду $u - v \leq c$ и постройте граф. Смотрите на суммарный вес циклов: система невыполнима тогда и только тогда, когда есть цикл отрицательного веса.
- **Задача 15** (EUF и теорема Чёрча). Равенства в $\mathcal{T}_{\text{EUF}}$ решаются слиянием классов эквивалентности. Кванторы разрушают алгоритм — по теореме Чёрча проблема разрешения неразрешима.

- **Задача 16** (bit-blasting сложения). Введите биты и сформулируйте сложение с переносом через $\oplus$ и $\wedge$. Выпишите КНФ для каждого равенства и найдите выполняющее присваивание.

Глава 16. Логическое программирование

- **Задача 15** (отрицание как провал). $\text{not}(P)$ как провал — это «поиск P не дал ни одного решения», а не « P ложно». Сравните $\text{not}(\text{not}(\text{flies}(X)))$ с $\text{flies}(X)$: чем отличаются результаты?

Глава 17. Матроиды

- **Задача 9** (ранг графического матроида). Независимые множества — ациклические наборы рёбер, базы — остовные леса. Ранг подмножества — размер максимального ациклического поднабора: посмотрите, где в графе образуется цикл.

Глава 18. Комбинаторика

- **Задача 5** (числа 1 и 2 не стоят рядом). Посчитайте перестановки, где 1 и 2 стоят рядом, и вычтите их по принципу дополнения. Склейте 1 и 2 в один объект в двух порядках.
- **Задача 9** (тождество Вандермонда). Посчитайте дважды число способов выбрать r человек из двух групп. Один способ — сумма по числу выбранных из первой группы.
- **Задача 13** (уравнение с ограничением $x_1 \leq 8$). Начните со всех неотрицательных решений и вычтите запрещённые, заменой переменной снимая ограничение. Сверьте «звёзды и перегородки» для запрещённого случая.
- **Задача 16** (неоднородная рекуррента). Решите характеристическое уравнение и отделите частное решение для неоднородной части. Если правая часть совпадает с корнем, ищите частное решение с резонансной поправкой.
- **Задача 18** (расстановка скобок). Зафиксируйте последнее внешнее умножение — оно делит произведение на две независимые части. Получите рекурренту и решите её производящей функцией.
- **Задача 21** (раскраски граней куба). Примените лемму Бёрнсайда: усредните число неподвижных раскрасок по всем вращениям. Вращение с s циклами на гранях оставляет неподвижными 3^s раскрасок.
- **Задача 22** (рамсеевское число $R(3, 4)$). Примените верхнюю оценку через $R(s - 1, t) + R(s, t - 1)$. Сверьте, можно ли её усилить, когда оба слагаемых чётны.

- **Задача 23** (коэффициент через свёртку). Коэффициент — число решений с ограничениями на каждую переменную. Сначала без верхних ограничений, затем включения-исключения по выходящим за них.

Глава 19. Теория чисел и криптография

- **Задача 3** (алгоритм Евклида на числах Фибоначчи). На паре (F_{n+1}, F_n) посмотрите, какие частные и остатки получаются при делении. Пара уменьшается до $(1, 0)$ за число шагов, равное... — наихудший случай (теорема Ламе).
- **Задача 6** (критерий обратимости по модулю). Из $ax \equiv 1 \pmod{m}$ выведите, что общий делитель a и m делит единицу. Обратно — расширенный алгоритм Евклида даёт коэффициенты Безу.
- **Задача 9** (маллобильность RSA). Вспомните мультипликативность возведения в степень: $(m_1 m_2)^e \equiv m_1^e m_2^e$. Подумайте, как злоумышленник, видя c , подменяет сообщение без закрытого ключа.
- **Задача 12** (атака Полига–Хеллмана). Порядок группы раскладывается на множители; атака сводит задачу к подгруппам этих порядков. Посмотрите, что проецирует возведение в степень $p - 1/q$.
- **Задача 13** (безопасное простое). Трудность дискретного логарифма зависит от разложения $p - 1$: Полиг–Хеллман разбирает его по малым делителям. Что меняется при $p = 2q + 1$ с простым q ?
- **Задача 16** (квантовая угроза). Факторизация и дискретный логарифм сводятся к нахождению периода функции. Квантовое преобразование Фурье находит период за полиномиальное время. Почему у AES нет такой уязвимости?
- **Задача 17** (341 — псевдопростое). Сведите $2^{340} \pmod{341}$ к двум малым модулям через китайскую теорему об остатках. Посмотрите, почему тест Ферма ошибочно соглашается.

Глава 20. Вероятность

- **Задача 4** (связность случайного графа). Считайте связные через дополнение: разберите несвязные по числу компонент. Всего графов — степень двойки от числа возможных рёбер.
- **Задача 8** (первый орёл по очереди). Выпишите, в какой бросок может впервые выпасть орёл у каждого игрока. Вероятности образуют геометрические прогрессии.
- **Задача 10** (задача Монти Холла). Разберите случаи по расположению автомобиля: когда переключение проигрывает? Учтите, что ведущий открывает дверь не случайно — его выбор несёт информацию.

- **Задача 14** (ожидаемое число пустых ящиков). Введите индикатор пустоты ящика и посчитайте его матожидание. Линейность матожидания работает и для зависимых индикаторов.
- **Задача 18** (Чебышёв и Чернов). Оцените хвост через стандартное отклонение (Чебышёв) и через экспоненциальное неравенство (Чернов). Сравните, какая оценка точнее и почему.
- **Задача 20** (задача о шляпах для пяти человек). Выберите k счастливых, остальные должны все ошибиться — это беспорядок. Соберите вероятность из выбора и беспорядка.
- **Задача 21** (вероятностный метод для $R(k, k)$). В случайном графе оцените вероятность, что фиксированные k вершин образуют клику. Сложите по всем k -подмножествам (неравенство Буля) и найдите n , где эта вероятность меньше 1.
- **Задача 22** (ожидание до первой шестёрки). Обозначьте E искомое ожидание и составьте уравнение, связывающее E с самим собой после первого броска.

Глава 21. Производящие функции

- **Задача 4** (ОПФ для суммы двух кубиков). Один бросок кости описывается многочленом; сумма двух бросков — его квадрат. Коэффициент при нужной степени — число способов набрать соответствующую сумму.
- **Задача 8** (ОПФ частичных сумм Фибоначчи). Частичные суммы накапливаются умножением ОПФ на $\frac{1}{1-x}$. Умножьте ОПФ чисел Фибоначчи на этот множитель.
- **Задача 9** (числа Каталана из функционального уравнения). Решите $C(x) = 1 + xC(x)^2$ как квадратное уравнение. Выбор знака определяется условием $C(0) = 1$.
- **Задача 12** (формула Кэли). Корневое дерево — корень и набор корневых поддеревьев: выразите ЭПФ. Обращение Лагранжа даёт число корневых деревьев; некорневое получается выбором корня.
- **Задача 14** (сумма биномиальных коэффициентов). $(1+x)^n$ — ОПФ биномиальных коэффициентов. Подстановка $x = 1$ суммирует их.

Глава 22. Конструкции чисел

- **Задача 3** (категоричность аксиоматики Пеано). Во втором порядке индукция — одна аксиома для всех подмножеств; в первом — схема только для определимых. Сравните, что даёт это различие для изоморфизма моделей.
- **Задача 7** (сокращение на ноль). Сокращение означает умножение на обратное к $a - b$. Посмотрите, что происходит, когда $a = b$.

- **Задача 11** (парадокс Банаха–Тарского и аксиома выбора). В доказательстве используется выбор представителя из каждой орбиты действия группы вращений. Подумайте, почему без аксиомы выбора парадокс невозможен.
- **Задача 12** (множество Витали). Введите на $[0, 1)$ отношение через сдвиг на рациональное и выберите по одному представителю из каждого класса. Счётные сдвиги множества нигде не пересекаются — сведите это к противоречию с мерой.
- **Задача 14** (независимость континуум-гипотезы). Совместимость с ZFC доказывается построением модели: конструктивный универсум (Гёдель) и форсинг (Коэн). Подумайте, сколько «новых» подмножеств натурального ряда появляется в $M[G]$.
- **Задача 15** (умножение сечений Дедекинда). Докажите $\alpha \cdot \alpha = 2$ двумя включениями: сначала прямое, затем обратное с подбором рационального между. В обратную сторону используйте плотность квадратов рациональных чисел.

Глава 23. Конечные автоматы и регулярные языки

- **Задача 3** (пятый с конца символ). Состояние должно помнить пять последних прочитанных символов. Сколько таких окон и какие из них принимающие?
- **Задача 7** (нет трёх подряд одинаковых). Ограничение касается длины серий одинаковых символов. Серии нулей и единиц чередуются: запишите язык как чередование.
- **Задача 14** (индекс отношения $\simeq_L$). Если ДКА имеет меньше состояний, чем классов $\simeq_L$, два различных слова попадают в одно состояние. Покажите, что это даёт одинаковый ответ на любое продолжение.
- **Задача 17** (первые половины слов). По ДКА для L постройте НКА с состояниями-парами (p, q) . Первая компонента читает первую половину, вторая — вторую в обратную сторону.
- **Задача 18** (делимость двоичного числа на 7). Пусть состояние — остаток от деления. Как он меняется при дописывании бита?
- **Задача 19** (ошибка в «доказательстве» регулярности). Накачка даёт необходимое, но не достаточное условие. Покажите, что накачиваться должны ВСЕ слова при ЛЮБОМ допустимом разбиении.
- **Задача 20** (перемешивание регулярных языков). Строка из SHUFFLE читается двумя вызываемыми друг друга головками. Постройте НКА с состояниями-парами и ϵ -переходами.

Глава 24. Контекстно-свободные языки

- **Задача 4** (зеркальные слова). Каждое рекурсивное правило добавляет одинаковые буквы по краям, а терминальное — кладёт разделитель в центр. Однозначность: выбор правила вынужден первым символом невыведенной части.
- **Задача 7** (два типа скобок). Первая скобка слова определяет первое правило; парная находится счётчиком глубины. Покажите, что эта позиция единственна, — и сравните с неоднозначной грамматикой, порождающей тот же язык.
- **Задача 10** (условие $i = j$ или $j = k$). Язык распадается в объединение двух КС-языков по тому, какое равенство следим. Автомат угадывает стратегию ε -переходом: первая сравнивает a с b , вторая — b с c .
- **Задача 13** (не контекстно-свободен, но накачивается). Постройте язык, который является объединением неконтекстно-свободного «ядра» и контекстно-свободного «моря». Ядро должно содержать лишь слова накачки, а море — поглощать остальные.
- **Задача 18** (КЗ-грамматика для $a^n b^n c^n$). Монотонная грамматика порождает $a^n (BC)^n$, а перестановки $CB \rightarrow BC$ собирают блоки в $a^n B^n C^n$. Терминальные правила превращают блоки в буквы слева направо.
- **Задача 19** (неоднозначность $S \rightarrow SS \text{ mid } a$). Слово aaa имеет два разных дерева разбора; но язык a^+ допускает однозначную грамматику. Сравните: неоднозначность — свойство грамматики или языка?
- **Задача 20** (замкнутость КС-языков). Для объединения, конкатенации и звезды Клини добавьте новый стартовый нетерминал и нужные правила. Переименуйте нетерминалы G_1 и G_2 , чтобы они не пересекались.

Глава 25. Машины Тьюринга

- **Задача 4** (машины Тьюринга счётны). Каждая машина задаётся конечной строкой (состояния, алфавит, таблица переходов). Сопоставьте с числом языков.
- **Задача 7** (язык $a^n b^n c^n$). Одной головкой хранить три счётчика нельзя — отмечайте пройденные буквы. За проход проверяйте по одной букве каждого типа и сверяйте порядок.
- **Задача 8** (копирование строки). Поставьте разделитель в конец входа и переносите символы по одному, пометая исходные. Когда все перенесены, восстановите исходные буквы по маркерам.
- **Задача 12** (тезис Чёрча–Тьюринга). Тезис связывает формальную вычислимость с интуитивным понятием алгоритма. Подумайте, почему его нельзя доказать как теорему, и что говорит в его пользу.

Глава 26. Разрешимость и неразрешимость

- **Задача 6** (язык $L(M) = \Sigma^*$). Сведите проблему останова: по паре (M, w) постройте машину M' , которая на любом входе x запускает M на w и принимает x , если M остановилась. Тогда $L(M') = \Sigma^*$ ровно тогда, когда M останавливается на w .
- **Задача 7** (эквивалентность двух машин). Сведите задачу пустоты: фиксируйте машину $M_\emptyset$ с пустым языком и сопоставьте каждой машине M пару $(M, M_\emptyset)$. Тогда $L(M) = \emptyset \leftrightarrow L(M) = L(M_\emptyset)$, то есть код машины лежит в языке эквивалентности ровно тогда, когда её язык пуст. Если бы эквивалентность была разрешима, разрешима была бы и задача пустоты.
- **Задача 10** (теорема Райса). Рассмотрите случай, когда пустой язык не обладает свойством P . Противоположный случай сводится к нему через дополнение, ведь разрешимость сохраняется при дополнении. Выберите машину A , язык которой обладает свойством, и по паре (M, w) постройте машину M' : она запускает M на w и затем ведёт себя как A . Если M останавливается на w , то $L(M') = L(A)$, и свойство выполнено. Если M закичивается, то $L(M') = \emptyset$, и свойство не выполнено.
- **Задача 14** (занятый бобр). Пусть T_f — машина с c состояниями, вычисляющая функцию f . Для каждого n постройте машину N_n , которая записывает n , выполняет T_f и печатает $f(n) + 1$ единиц. Для всех достаточно больших n число состояний N_n не превосходит n , поэтому $\Sigma(n) \geq f(n) + 1 > f(n)$. Если бы Σ была вычислимой, мы могли бы подставить $f = \Sigma$. Тогда получилось бы $\Sigma(n) > \Sigma(n)$ — противоречие.
- **Задача 15** (невычислимость Σ). Предположите, что машина B вычисляет $\Sigma(n)$. Ключевая оценка: машина, моделирующая M на w и дописывающая по единице за шаг, имеет $k = |\langle M \rangle| + |w| + c$ состояний. Она останавливается не позднее чем за $\Sigma(k)$ шагов. Тогда моделирование M на w в течение $\Sigma(k)$ шагов решает проблему останова — противоречие.

Глава 27. Лямбда-исчисление

- **Задача 9** (теорема Чёрча–Россера). Конфлюэнтность β -редукции: если терм редуцируется двумя способами, то и их результаты сводятся к общему. Подумайте о последствиях для независимости от стратегии редукции.
- **Задача 14** (возведение в степень над числами Чёрча). Возведение в степень — это применение m к самому себе n раз. Проверьте на паре маленьких чисел Чёрча, сколько раз применяется функция.
- **Задача 16** (предшественник числа Чёрча). Шаг сдвига переводит пару (a, b) в $(b, \text{succ } b)$. Запустите его n раз из пары $(0, 0)$ и посмотрите на первый элемент.

- **Задача 19** (абстракция скобок). Переводите терм изнутри наружу: $[x]x = I$, $[x]y = Ky$ при $y \neq x$, $[x](MN) = S([x]M)([x]N)$. Проверьте результат на простом терме обратного порядка.
- **Задача 20** (ошибка: не каждый терм имеет НФ). Утверждение «редукция уменьшает терм» неверно: β -шаг копирует аргумент. Приведите терм, где β -редукция не завершается нормальной формой.
- **Задача 21** (ошибка: редукция не коммутативна). Редукция направлена: $M \rightarrow N$ не означает $N \rightarrow M$. Сравните с тем, что означала бы коммутативность для нормальных форм.

Глава 28. Теория типов

- **Задача 4** (деревья вывода). Постройте дерево вывода для каждого терма. Сравните их форму и то, чем отличается итоговый тип.
- **Задача 6** (самоприменение xx в $\lambda \rightarrow$). Попробуйте вывести тип для xx : каким должен быть тип x (app)? Сведите к уравнению на типы и покажите его неразрешимость в $\lambda \rightarrow$.
- **Задача 9** (комбинатор S). Постройте тип по терму: зафиксируйте типы x, y, z и выведите тип тела. Посмотрите, какая тавтология ей соответствует.
- **Задача 11** (сильная нормализация и обходные пути). Обходной путь — введение импликации, сразу за которым идёт её удаление. В термах ему отвечает β -редекс; посмотрите, как его устраняет нормализация.
- **Задача 12** (зависимые типы). Тип $\text{Vec } n$ зависит от значения терма n ; в $\lambda \rightarrow$ терм не входит в тип. Что нужно добавить, чтобы тип зависел от терма?

Глава 29. За пределами натуральных чисел

- **Задача 3** (почему схема индукции не исключает нестандартные элементы). В PA первого порядка индукция — схема по формулам языка, а не квантор по всем подмножествам. Свойство « x стандартно» выразимо ли формулой первого порядка?
- **Задача 8** (ассоциативность ординального сложения). Проводите трансфинитную индукцию по третьему аргументу. Предельный шаг — ключевой: работайте с супремумом.
- **Задача 12** (нестандартные модели и сюрреальные числа). Нестандартные модели — непреднамеренное расширение; сюрреальные числа — преднамеренная конструкция по дням. Сравните, какое из них получается из 0 конечным числом шагов.

- **Задача 13** (нестандартная модель действительных чисел). Расширьте язык константой c и добавьте аксиомы $c > n$ для всех натуральных n . Примените компактность и посмотрите на свойства c и $1/c$.
- **Задача 14** (ошибка: не все ординалы счётны). Предельный шаг трансфинитной индукции: объединение счётно, только если объединение берётся по счётному множеству. Что происходит при $\lambda = \omega_1$?

Глава 30. Сложность вычислений

- **Задача 4** (замкнутость NP относительно объединения и пересечения). Постройте верификатор: где нужен дополнительный бит угадывания, а где — пара сертификатов?
- **Задача 8** (coNP-полнота TAUTOLOGY). Свяжите задачу с SAT через отрицание: что означает $\varphi \notin \text{TAUTOLOGY}$? Сводите дополнение SAT к TAUTOLOGY отображением.
- **Задача 11** (теорема Мэхэни о разреженных NP-полных языках). Разреженность означает малое число строк каждой длины. Ведите фронт частично означенных формул и прореживайте его, пока не останется полином.
- **Задача 12** (PSPACE-полнота TQBF). Принадлежность: рекурсивная оценка кванторов с переиспользованием памяти. Трудность: выразите достижимость конфигураций квантифицированной формулой, используя трюк Сэвича.
- **Задача 13** (усиление вероятности в BPP). Запустите машину несколько раз независимо и возьмите большинство. Оцените вероятность ошибки большинства через неравенство Чернова.
- **Задача 14** (ошибка: NP не подмножество P этим аргументом). Обращение верификатора не порождает алгоритма поиска сертификата. Подумайте, сколько кандидатов пришлось бы перебрать.
- **Задача 15** (ошибка: NP не подмножество coNP). Верификатор дополнения не равен отрицанию верификатора языка. Что должен проверять coNP-верификатор для $x \in L$ и для $x \notin L$?

Глава 31. Абстрактная интерпретация

- **Задача 4** (сложение знаков: $+ \boxplus - = \top$). Сложение знаков — точнейшая абстракция множества конкретных сумм. Покажите, что множество $\{u + v \mid u > 0, v < 0\}$ совпадает со всем $\mathbb{Z}$.
- **Задача 8** (домен, различающий «строго положительное» и «неотрицательное»). Добавьте элемент пнэг с подходящей конкретизацией и постройте таблицу сложения. Сравните результат на программе с условием $x \geq 0$ со знаковым доменом.

- **Задача 11** (домен чётности). Работайте в решётке $\{\perp, \text{even}, \text{odd}, \top\}$ в арифметике по модулю 2. Проверьте корректность каждого переноса включением конкретизаций.
- **Задача 12** (неподвижная точка цикла и widening). Наивная итерация счётчика с неизвестной границей не стабилизируется. Попробуйте widening, а затем narrowing по условию цикла.
- **Задача 13** (связка Галуа для знаков). Проверьте условие связки: $\alpha(x) \preceq d \leftrightarrow x \in \gamma(d)$. Абстракция — знак числа, конкретизация — все числа данного знака.

Глава 32. Формальная верификация

- **Задача 11** (слабейшее предусловие для модуля). Воспользуйтесь определением wr для условного оператора: конъюнкция импликаций с постусловием каждой ветки. Подстановка в присваивании — ключ: замените переменную в постусловии.

Глава 33. Модальная и темпоральная логика

- **Задача 5** (куб $TB4$). Каждая ось — своё свойство отношения достижимости: T — рефлексивность, B — симметричность, 4 — транзитивность. Составьте все 2^3 комбинации и назовите известные системы.
- **Задача 11** (сложность модальной выполнимости). Свидетель для SAT — оценка, а для модальной выполнимости — модель Крипке с потенциально экспоненциальным числом миров. Вложенные модальности — чередующиеся кванторы; свяжите с QBF.
- **Задача 6** (аксиома B и симметричность). Аксиома B : $p \rightarrow \Box \Diamond p$. Проверьте её на фрейме из двух миров с отношением в одну сторону. Оцените p только в одном мире — какая аксиома требует симметричности?
- **Задача 9** (CTL против LTL: $A \Box E \Diamond p$). $E \Diamond p$ — «из текущего состояния есть путь к p »; $\Box \Diamond p$ — «на каждом пути p бесконечно часто». Постройте модель, где первая истинна, а вторая — нет: петля и путь к p из одного состояния.
- **Задача 12** (живость и безопасность как неподвижные точки). Живость «когда-нибудь p » — наименьшая неподвижная точка итерации снизу; безопасность «всегда p » — наибольшая, сверху. Различие E и A — в пре-образе: один требует хотя бы одного преемника, другой — всех.
- **Задача 13** (model checking LTL через автоматы Бюхи). Постройте автомат Бюхи для отрицания свойства и синхронное произведение с моделью. Проверьте пустоту языка произведения: есть ли принимающее состояние на цикле?
- **Задача 14** (табличный метод для K). Примените правило $\Box \varphi$ и постройте дерево, раскрывая импликации. Формула общезначима в K, если все ветви замкнуты.

Глава 34. Интуиционизм

- **Задача 6** (модель Крипке для $p \rightarrow q$). Для опровержения импликации нужен мир, где принуждается p , но не q . Подумайте, какие другие миры должны существовать над таким миром.
- **Задача 7** (теорема Гливенко). Утверждение касается перевода классической выводимости в двойное отрицание. Проверьте на примере $p \vee \neg p$: каков «перевод» этой формулы?
- **Задача 8** (компактность второго порядка). Во втором порядке конечность выражима формулой: существует инъективная, но не сюръективная функция. Соедините это с бесконечным множеством попарно различных констант.
- **Задача 10** (необитаемость типа $((A \rightarrow \perp) \rightarrow \perp) \rightarrow A$). Попробуйте построить терм этого типа в чистом STLC. Откуда взять значение A ? Сопоставьте с call/cc и классической логикой.

Глава 35. Нечёткие множества

- **Задача 8** (законы де Моргана для нечётких множеств). Равенство поточечно — сведите оба закона к тождествам для степеней принадлежности. Проверьте разбором случаев $a \geq b$ и $b > a$.
- **Задача 10** (теорема декомпозиции). Зафиксируйте x и посмотрите, при каких уровнях α он попадает в срез. Восстановите степень принадлежности из супремума этих уровней.
- **Задача 11** (центроид и Mean of Max). Вычислите площадь и первый момент фигуры под функцией принадлежности. Сравните центроид с серединой плато максимума: почему они различаются?
- **Задача 12** (исключённое третье и t-нормы). Проверьте оба закона на $\mu_A(x) = 0.5$ для t-нормы Заде. Попробуйте t-норму Лукасевича — что она даёт?
- **Задача 13** (степень $\neq$ вероятность). Степень принадлежности не имеет частотной интерпретации. Сравните аксиомы: вероятности счётно-аддитивны, а μ — любая функция.

Глава 36. Категории

- **Задача 4** (мономорфизм и эпиморфизм, но не изоморфизм). Начните с частичного порядка: в тонкой категории каждая стрелка мономорфизм и эпиморфизм. Но изоморфизм требует обратной стрелки — найдите порядок, где стрелка есть, а обратной нет.

- **Задача 8** (функтор $\mathcal{C}(A, -)$ и произведения). Напомните себе универсальное свойство произведения и примените его в $\mathcal{Set}$. Показать изоморфизм — построить биекцию в обе стороны между множествами стрелок.
- **Задача 12** (булеан — представимый функтор). Представимый функтор — это $\mathcal{Set}(A, -)$. Какой объект A надо взять, чтобы $\mathcal{Set}(A, B)$ отождествился с $\mathcal{P}(B)$?
- **Задача 16** (категория Клейсли для монады списка). Композиция стрелок $A \rightarrow T(B)$ и $B \rightarrow T(C)$ в категории Клейсли — это $x \mapsto \text{flat_map}(g)(f(x))$: список $f(x)$ отображается поэлементно и склеивается из образов его элементов. Проверьте ассоциативность на списках: сгруппируйте по-разному и сравните результаты.

Словарь терминов

В этом словаре собраны основные понятия курса: по одному краткому определению на термин. Рядом с каждым термином — ссылка на раздел, где он вводится и обсуждается подробно. Записи выстроены по дуге курса — от языка описания дискретных структур к вычислению и его границам.

§ 36.16 Язык дискретных структур

- **Высказывание** — повествовательное утверждение с определённым истинностным значением (**T** — истина, **F** — ложь) (раздел 1.1).
- **Логические связи** — отрицание $\neg p$, конъюнкция $p \wedge q$, дизъюнкция $p \vee q$, импликация $p \rightarrow q$ и эквиваленция $p \leftrightarrow q$ (раздел 1.1).
- **Правильно построенная формула** (*well-formed formula*) — формула, собранная из атомов по грамматике связок (раздел 1.1).
- **Интерпретация** — приписывание каждой пропозициональной переменной истинностного значения (раздел 1.1).
- **Таблица истинности** — перечисление всех интерпретаций атомов и вычисление значения формулы на каждой (раздел 1.1).
- **Тавтология** — формула, истинная при всех интерпретациях (раздел 1.1).
- **Противоречие** — формула, ложная при всех интерпретациях, она же невыполнимая (раздел 1.1).
- **Выполнимая формула** (*satisfiable formula*) — формула, истинная хотя бы при одной интерпретации (раздел 1.1).
- **КНФ** — конъюнктивная нормальная форма: конъюнкция дизъюнктов (раздел 1.1).
- **Предикат** — утверждение с переменной $P(x)$, становящееся высказыванием при подстановке значения или связывании квантором (раздел 3.1).
- **Предметная область** (*domain of discourse*) — множество значений, которые может принимать переменная предиката (раздел 3.1).
- **Математическая индукция** — метод доказательства: база $P(0)$ и шаг $\forall k(P(k) \rightarrow P(k+1))$ влекут $\forall n P(n)$ (раздел 1.2).
- **Логическая эквивалентность** — $\varphi \equiv \psi$, если $\varphi \leftrightarrow \psi$ — тавтология. Эквивалентные формулы взаимозаменяемы в любом контексте (раздел 1.1).
- **Логическое следствие** — $\varphi_1, \dots, \varphi_n \models \psi$, если каждая интерпретация, делающая посылки истинными, делает истинным и заключение (раздел 1.1).
- **ДНФ** — дизъюнктивная нормальная форма: дизъюнкция конъюнкций литералов (раздел 1.1).
- **Законы де Моргана** — $\neg(p \wedge q) \equiv \neg p \vee \neg q$ и $\neg(p \vee q) \equiv \neg p \wedge \neg q$ (раздел 1.1).

- **Квантор** — $\forall$ и $\exists$ превращают предикат в высказывание. $\forall x \in A$ и $\exists x \in A$ — ограниченные (раздел 3.2).
- **Сколемизация** — замена квантора существования функцией от охватывающих его кванторов всеобщности: $\forall x \exists y P(x, y)$ превращается в $\forall x P(x, f(x))$ (раздел 3.5).
- **Предварённая нормальная форма (ПНФ)** — вид $Q_1 x_1 \dots Q_k x_k \psi$, в котором все кванторы вынесены в начало, а ψ их не содержит (раздел 3.5).
- **Многосортная логика** (*many-sorted logic*) — логика первого порядка с сортами: переменные типизированы, $\forall x : S. \varphi$ — сокращение для $\forall x (x \in S \rightarrow \varphi)$. Лежит в основе SMT-солверов (раздел 3.8).
- **Сигнатура** — набор символов языка с указанной арностью: константы, функциональные и предикатные символы (раздел 3.10).
- **Общезначимость** — формула $\models A$ истинна в каждой интерпретации, аналог тавтологии для логики первого порядка (раздел 3.10).
- **Структурная индукция** — доказательство свойства для всех рекурсивно определённых объектов: база для атомов и шаг для каждого конструктора (раздел 1.2).
- **Тройка Хоара** — $\{P\} S \{Q\}$: если предусловие P выполнено перед S , то постусловие Q выполнено после (глава 32, первое знакомство — раздел 1.3).
- **Ассерт** — логическая формула, размещённая в коде и обязанная быть истинной в этой точке выполнения. Не путать с «утверждением» как высказыванием (глава 32).
- **Инвариант цикла** — формула I , которую сохраняет тело цикла и которая вместе с условием выхода даёт постусловие. Позволяет доказывать корректность циклов индукцией (глава 32).
- **Слабейшее предусловие** — $\text{wp}(S, Q)$: самая широкая формула P , для которой верна тройка $\{P\} S \{Q\}$. Для присваивания $\text{wp}(x := E, Q) = Q[x := E]$ (глава 32).
- **Условие верификации** (*verification condition*) — логическая формула, истинность которой гарантирует корректность программы. Проверяется SMT-солвером на общезначимость (глава 32).
- **Вариант цикла** (*loop variant*) — функция от состояния программы со значениями во вполне упорядоченном множестве, строго убывающая на каждой итерации. Доказывает завершение цикла (глава 32).
- **Частичная и тотальная корректность** — частичная: если оператор завершается, то постусловие верно. Тотальная: оператор завершается и постусловие верно (глава 32).
- **Система вывода** — набор аксиом и правил вывода (глава 2).
- **Аксиома** — формула, принимаемая системой без доказательства (глава 2).
- **Правило вывода** — разрешённый переход от посылок заданной формы к заключению заданной формы (глава 2).
- **Гипотеза** — формула, принимаемая в выводе без доказательства, но не входящая в систему (глава 2).

- **Логический вывод** (*derivation*) — конечная последовательность формул: аксиомы, гипотезы и результаты правил. Запись $\Gamma \vdash \varphi$ означает, что такой вывод существует (глава 2).
- **Натуральная дедукция** — система вывода без аксиом, с правилами ввода и удаления для каждой связки. Ближе к рассуждению математика, чем системы Гильберта (глава 2).
- **Нотация Фитча** — запись натурального вывода: колонка пронумерованных строк, поддоказательства отступом, снятие гипотез (глава 2).
- **Поддоказательство** — блок вывода с временной гипотезой. По завершении гипотеза снимается и снаружи недоступна (глава 2).
- **Косвенное доказательство** — правило IP: из вывода противоречия из $\neg A$ следует A . Классическое, отсутствует в интуиционизме (глава 2).
- **Секвенция** — $\Gamma \vdash \Delta$: из всех формул списка Γ следует хотя бы одна формула списка Δ (раздел 2.10).
- **Литерал и клауза** — литерал: атом или его отрицание. Клауза: дизъюнкция литералов (раздел 2.11).
- **Введение и удаление кванторов** — правила натуральной дедукции для $\forall$ и $\exists$ с побочными условиями на свежесть переменной и свободу подстановки (раздел 2.13).
- **Корректность и полнота** — $\vdash \subseteq \models$ (всё выводимое истинно) и $\models \subseteq \vdash$ (всё истинное выводимо). Вместе отождествляют синтаксис и семантику (глава 2).
- **Множество** — неупорядоченный набор различных объектов. Принадлежность $a \in A$ (раздел 4.1).
- **Булеан** — множество всех подмножеств $\mathcal{P}(A) = \{X \mid X \subseteq A\}$. $|\mathcal{P}(A)| = 2^{|A|}$ (раздел 4.3).
- **Декартово произведение** — $A \times B = \{(a, b) \mid a \in A, b \in B\}$. Фундамент для отношений и функций (раздел 4.7).
- **Пустое множество** — $\emptyset$: множество без элементов. Не путать с синглетоном $\{\emptyset\}$ (раздел 4.1).
- **Экстенциональность** — множества равны, когда состоят из одних и тех же элементов: $A = B$, если $\forall x(x \in A \leftrightarrow x \in B)$ (раздел 4.2).
- **Подмножество** — $A \subseteq B$, если каждый элемент A является элементом B . $A \subset B$ — собственное (раздел 4.2).
- **Разбиение** — набор непустых попарно непересекающихся подмножеств, объединение которых равно всему множеству (раздел 4.8).
- **Парадокс Рассела** — противоречие из допущения о множестве $R = \{x \mid x \notin x\}$. Показал несостоятельность принципа свёртки (раздел 4.9).
- **Аксиома выделения** — для любого множества A и свойства $P(x)$ существует $\{x \in A \mid P(x)\}$. Исключает парадокс Рассела (раздел 4.9).
- **Бинарное отношение** — подмножество $R \subseteq A \times A$. aRb означает $(a, b) \in R$ (раздел 5.1).
- **Замыкание отношения** — минимальное расширение отношения заданным свойством: рефлексивное $R^=$, транзитивное R^+ (раздел 5.1).

- **Отношение эквивалентности** — рефлексивное, симметричное и транзитивное отношение. Разбивает множество на классы (раздел 5.2).
- **Рефлексивность** — aRa для всех a (раздел 5.1).
- **Симметричность** — из aRb следует bRa (раздел 5.1).
- **Транзитивность** — из aRb и bRc следует aRc (раздел 5.1).
- **Антисимметричность** — из aRb и bRa следует $a = b$ (раздел 5.1).
- **Класс эквивалентности** — $[a] = \{b \in A \mid a \sim b\}$. Классы либо совпадают, либо не пересекаются (раздел 5.2).
- **Функция** — бинарное отношение, в котором каждому $a \in A$ соответствует ровно один $b \in B$ (раздел 6.1).
- **Область определения** — множество допустимых входов A функции $f : A \rightarrow B$, обозначается $\text{Dom}(f)$ (раздел 6.1).
- **Кодомен** — множество разрешённых выходов B функции $f : A \rightarrow B$. Не путать с областью значений — образом $f(A)$, фактически достигаемым функцией (раздел 6.1).
- **Инъекция** — функция, при которой $f(x) = f(y) \rightarrow x = y$. «разные входы — разные выходы» (раздел 6.2).
- **Сюръекция** — функция, при которой $\forall b \in B \exists a \in A : f(a) = b$. «каждый элемент задействован» (раздел 6.3).
- **Биекция** — функция, являющаяся инъекцией и сюръекцией одновременно. Взаимно-однозначное соответствие (раздел 6.4).
- **Композиция функций** — $(g \circ f)(x) = g(f(x))$. Композиция инъекций — инъекция, сюръекций — сюръекция (раздел 6.6).
- **Образ функции** — $f(A) = \{f(a) \mid a \in A\}$: множество фактически достигаемых значений (раздел 6.1).
- **Прообраз** — $f^{-1}(Y) = \{a \in A \mid f(a) \in Y\}$. Определён для любого Y (раздел 6.1).
- **Обратная функция** — $f^{-1} : B \rightarrow A$ существует тогда и только тогда, когда f — биекция (раздел 6.1).
- **Частичная функция** — $f : A \mapsto B$, определённая лишь на части A . Модель вычисления, которое может не завершиться (раздел 6.1).
- **Характеристическая функция** — $\chi_X : U \rightarrow \{0, 1\}$: равна 1 на элементах X и 0 вне X (раздел 6.9).
- **О-большое** — $f(n) \in \mathcal{O}(g(n))$, если $f(n) \leq C \cdot g(n)$ при всех $n \geq n_0$. f растёт не быстрее g (раздел 6.8).
- **Равномощность** — $|A| = |B|$, если существует биекция $f : A \rightarrow B$. Обобщает «одинаковое количество» на бесконечные множества (раздел 7.1).
- **Конечное и бесконечное множество** — конечное равномощно отрезку $\{0, 1, \dots, n-1\}$ для некоторого n , бесконечное не является конечным (раздел 7.1).
- **Счётное множество** — множество, конечное или равномощное $\mathbb{N}$. Элементы можно занумеровать (раздел 7.3).
- **Диагональный аргумент Кантора** — построение элемента, отличающегося от каждого в предполагаемом перечислении. Доказывает несчётность $\mathbb{R}$ (раздел 7.3).

- **Теорема Кантора** — $|A| < |\mathcal{P}(A)|$ для любого A . Булеан всегда строго больше исходного множества (раздел 7.4).
- **Континуум** — мощность $\mathbb{R}$: $|\mathbb{R}| = 2^{\aleph_0}$. Множество всех языков имеет мощность континуума (раздел 7.6).
- **Несчётное множество** — бесконечное множество, не равномощное $\mathbb{N}$. Его элементы нельзя занумеровать (раздел 7.3).
- **Сравнение мощностей** — $|A| \leq |B|$, если существует инъекция $A \rightarrow B$. $|A| < |B|$, если биекции нет (раздел 7.5).
- **Теорема Кантора–Бернштейна–Шрёдера** — если $|A| \leq |B|$ и $|B| \leq |A|$, то $|A| = |B|$ (раздел 7.5).
- **Кардинальное число** — класс равномощных множеств. $|\mathbb{N}| = \aleph_0$, $|\mathbb{R}| = 2^{\aleph_0}$ (раздел 7.6).
- **Кардинальная арифметика** — $\kappa + \lambda = \kappa \cdot \lambda = \max(\kappa, \lambda)$ для бесконечных кардиналов. $2^\kappa > \kappa$ (раздел 7.6).
- **Алеф- и бет-числа** — алефы перечисляют все бесконечные кардиналы по возрастанию, беты получаются итерациями булеана: $\beth_{\alpha+1} = 2^{\beth_\alpha}$ (раздел 7.6).
- **Диаграмма Хассе** — граф покрытия частичного порядка. Транзитивные рёбра опускаются (раздел 8.2).
- **Линейный порядок** — частичный порядок, в котором любые два элемента сравнимы (раздел 8.1).
- **Супремум и инфимум** — наименьшая верхняя грань ($a \vee b$) и наибольшая нижняя грань ($a \wedge b$) (раздел 8.5).
- **Решётка** — ЧУМ, в котором каждая пара элементов имеет супремум и инфимум (раздел 8.6).
- **Подрешётка** — подмножество решётки, замкнутое относительно $\vee$ и $\wedge$ (раздел 8.6).
- **Дистрибутивная решётка** — решётка с тождеством $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$. По теореме Биркгофа дистрибутивна тогда и только тогда, когда не содержит подрешёток M_3 и N_5 (раздел 8.6).
- **Модулярная решётка** — решётка, в которой из $a \preceq c$ следует $a \vee (b \wedge c) = (a \vee b) \wedge c$. Характеризуется отсутствием подрешётки N_5 (раздел 8.6).
- **Характеризация** — необходимое и достаточное условие (утверждение вида «тогда и только тогда») (раздел 8.6).
- **Топологическая сортировка** — линейное продолжение частичного порядка. Планирование задач с зависимостями (раздел 8.7).
- **Частичный порядок** (*partial order*) — рефлексивное, антисимметричное и транзитивное отношение. Пара $(A, \preceq)$ называется ЧУМ (*poset*) (раздел 8.1).
- **Строгий порядок** — иррефлексивное транзитивное отношение $\prec$, нестрогий получается добавлением равенства (раздел 8.1).
- **Цепь и антицепь** — цепь: подмножество попарно сравнимых элементов ЧУМ. Антицепь: подмножество попарно несравнимых (раздел 8.1).

- **Отношение покрытия** — a покрывает b , если $b < a$ и между ними нет промежуточных элементов. Только рёбра покрытия рисуются в диаграмме Хассе (раздел 8.1).
- **Экстремальные элементы** — минимальный и максимальный: не имеют строго меньшего/большого. Наименьший и наибольший — единственны и сравнимы со всеми (раздел 8.4).
- **Лексикографический порядок** — порядок на $A \times B$: сравниваются первые компоненты, при равенстве — вторые (раздел 8.8).
- **Неподвижная точка** — элемент x с $f(x) = x$. Монотонная функция на полной решётке имеет наименьшую и наибольшую неподвижные точки (раздел 8.10).
- **Граф** — $G = (V, E)$: множество вершин V и множество рёбер E (раздел 9.1).
- **Неориентированный и ориентированный граф** — рёбра неориентированного — неупорядоченные пары $\{u, v\}$, дуги ориентированного — упорядоченные пары (u, v) (раздел 9.1).
- **Простой граф и мультиграф** — простой не содержит петель и кратных рёбер, мультиграф разрешает и то и другое (раздел 9.1).
- **Взвешенный граф** — пара (G, w) , где весовая функция $w : E \rightarrow \mathbb{R}$ сопоставляет каждому ребру число (раздел 9.1).
- **Изоморфизм графов** — биекция вершин, сохраняющая рёбра и направления дуг: изоморфные графы отличаются только именами вершин (раздел 9.1).
- **Дерево** — связный ациклический граф. n вершин, $n - 1$ рёбер (раздел 9.6).
- **Эйлеров цикл** — цикл, проходящий каждое ребро ровно один раз (раздел 9.8).
- **Гамильтонов цикл** — цикл, проходящий каждую вершину ровно один раз. NP-полная задача (раздел 9.10).
- **Планарный граф** — граф, который можно нарисовать на плоскости без пересечений рёбер (раздел 9.17).
- **Раскраска графа** — приписывание цветов вершинам так, чтобы смежные вершины имели разные цвета. $\chi(G)$ — хроматическое число (раздел 9.18).
- **Степень вершины** — $\deg(v)$: число инцидентных рёбер. Сумма степеней всех вершин равна $2|E|$ (раздел 9.1).
- **Путь и цикл** — путь — маршрут без повторяющихся вершин. Цикл — замкнутый путь (раздел 9.3).
- **Связность** — граф связан, если между любыми двумя вершинами есть путь. Компонента связности — максимальный связный подграф (раздел 9.3).
- **Двудольный граф** — вершины разбиты на две доли, каждое ребро соединяет разные доли. Критерий — отсутствие циклов нечётной длины (раздел 9.13).
- **Формула Эйлера** — для связного планарного графа $V - E + F = 2$, где F — число граней (раздел 9.17).

§ 36.17 Алгебра и инженерия

- **Бинарная операция** — функция $\star : A \times A \rightarrow A$, сопоставляющая паре элементов элемент того же множества (раздел 12.1).

- **Ассоциативность** — $(a \star b) \star c = a \star (b \star c)$ (раздел 12.1).
- **Коммутативность** — $a \star b = b \star a$ (раздел 12.1).
- **Нейтральный элемент** — $e \star a = a \star e = a$ (раздел 12.1).
- **Обратный элемент** — $a \star a^{-1} = e$ (раздел 12.1).
- **Полугруппа и моноид** — множество с ассоциативной операцией, моноид дополнительно снабжён нейтральным элементом (раздел 12.2).
- **Группа** — множество G с ассоциативной бинарной операцией $\cdot$, в которой есть нейтральный элемент e и обратный a^{-1} для каждого $a \in G$ (раздел 12.3).
- **Абелева группа** — группа, операция которой коммутативна: $a \cdot b = b \cdot a$ для всех $a, b \in G$ (раздел 12.3).
- **Циклическая группа** — группа с образующим g , чьи степени g^k пробегает все элементы группы (раздел 12.3).
- **Подгруппа** — непустое подмножество $H \subseteq G$, замкнутое относительно операции и взятия обратного (раздел 12.4).
- **Смежный класс** — множество $aH = \{a \cdot h \mid h \in H\}$. Классы попарно не пересекаются и покрывают группу, на чём стоит теорема Лагранжа (раздел 12.4).
- **Нормальная подгруппа** — подгруппа H с условием $gHg^{-1} = H$ для всех $g \in G$, равносильным совпадению левых и правых смежных классов (раздел 12.9).
- **Теорема Лагранжа** — порядок подгруппы конечной группы делит порядок группы: $|H|$ делит $|G|$ (раздел 12.4).
- **Орбита** — множество $G \cdot x = \{g \cdot x \mid g \in G\}$ точек, достижимых из x действием группы (раздел 12.5).
- **Стабилизатор** — подгруппа $\text{Stab}(x) = \{g \in G \mid g \cdot x = x\}$ элементов, оставляющих точку x на месте (раздел 12.5).
- **Кольцо** — множество R с операциями $+$ и $\cdot$: по сложению абелева группа, по умножению полугруппа, связанные дистрибутивностью (раздел 12.6).
- **Поле** — кольцо, в котором ненулевые элементы образуют абелеву группу по умножению (раздел 12.7).
- **Конечное поле** — поле с конечным числом элементов $|K| = p^m$, обозначается $\text{GF}(p^m)$ (раздел 12.8).
- **Неприводимый многочлен** — многочлен над $\mathbb{Z}_p$, который нельзя представить произведением многочленов меньших степеней (раздел 12.8).
- **Примитивный элемент** — образующий мультипликативной группы поля: его степени пробегает все ненулевые элементы $\text{GF}(q)$ (раздел 12.8).

- **Гомоморфизм** — отображение $\varphi : A \rightarrow B$, сохраняющее операцию: $\varphi(x \star y) = \varphi(x) \cdot \varphi(y)$ (раздел 12.9).
- **Изоморфизм** — взаимно однозначное соответствие, сохраняющее структуру: в алгебре — биективный гомоморфизм, в категории — стрелка с обратной. Изоморфные объекты одинаковы с точностью до переименования (раздел 12.9, категорный взгляд — раздел 36.2).
- **Ядро гомоморфизма** — множество $\ker(\varphi) = \{x \in G \mid \varphi(x) = e\}$ элементов, переходящих в нейтральный элемент (раздел 12.9).
- **Образ гомоморфизма** — множество значений гомоморфизма φ : все $\varphi(x)$ для $x \in G$ (раздел 12.9).
- **Факторгруппа** — группа $\frac{G}{H}$ смежных классов нормальной подгруппы H . Умножение классов через представителей корректно именно благодаря нормальности (раздел 12.9).
- **Векторное пространство** — множество векторов с операциями сложения и умножения на скаляры поля (раздел 12.10).
- **Булева функция** — функция $f : \{0, 1\}^n \rightarrow \{0, 1\}$: связки, вентили и схемы вычисляют именно такие функции (раздел 10.1).
- **Булева алгебра** — алгебраическая структура $(B, \vee, \wedge, \neg, 0, 1)$ с операциями, удовлетворяющими аксиомам коммутативности, ассоциативности, дистрибутивности и дополнения (раздел 10.1).
- **СДНФ/СКНФ** — совершенная дизъюнктивная/конъюнктивная нормальная форма: каноническое представление булевой функции через минтермы/макстермы (раздел 10.5).
- **Функциональная полнота** — система связей полна, если через неё выражается любая булева функция (раздел 10.1).
- **Критерий Поста** — система связей полна тогда и только тогда, когда не содержится целиком ни в одном из пяти предполных классов (раздел 10.1).
- **Замыкание** — наименьшее множество булевых функций $[F]$, содержащее F и проекции и замкнутое относительно композиции (раздел 10.2).
- **Клон** — множество булевых функций с проекциями x_i , замкнутое относительно композиции. Предполные классы Поста — максимальные собственные клоны (раздел 10.2).
- **Карта Карно** — визуальный метод минимизации булевых выражений (до 4 переменных) (раздел 10.6).
- **BDD** — Binary Decision Diagram: каноническое представление булевой функции в виде ориентированного ациклического графа (раздел 10.8).

- **Дизъюнкт** — дизъюнкция литералов, элементарная часть КНФ. Резолюция выводит из дизъюнктов новые дизъюнкты (раздел 10.3).
- **Минтерм и макстерм** — конъюнкция/дизъюнкция литералов, в которую каждая переменная входит ровно один раз. Истинна/ложна ровно на одной строке таблицы истинности (раздел 10.3).
- **Полином Жегалкина (АНФ)** — представление булевой функции XOR-суммой мономов над $GF(2)$. Единственно для каждой функции (раздел 10.7).
- **Простая импликанта** — конъюнкция литералов, имплицитующая функцию и не расширяемая без потери импликации (раздел 10.6).
- **Разложение Шеннона** — $f = (\overline{x_i} \wedge f[x_i := 0]) \vee (x_i \wedge f[x_i := 1])$: функция определяется своим поведением при $x_i = 0$ и $x_i = 1$. Основа BDD (раздел 10.8).
- **ROBDD** — упорядоченная и редуцированная BDD: каноническое представление, единственное при фиксированном порядке переменных (раздел 10.8).
- **Логический вентиль** — транзисторная схема, вычисляющая элементарную булеву функцию (AND, OR, NOT, NAND, NOR, XOR) (раздел 11.2).
- **Комбинационная схема** — ациклическое соединение вентилях. Выход определяется только текущими входами, без памяти (раздел 11.2).
- **Сумматор** — схема для сложения битовых строк: полусумматор (без входящего переноса) и полный сумматор (с переносом) (раздел 11.3).
- **Размер схемы** — число вентилях, мера аппаратных затрат (раздел 11.2).
- **Глубина и задержка схемы** — длиннейший путь от входа к выходу: глубина определяет задержку распространения сигнала и тактовую частоту (раздел 11.2).
- **Позиционная система счисления** — представление числа $d_k \dots d_0$ как $\sum d_i b^i$. При $b = 2, 8, 16$ — двоичная, восьмеричная, шестнадцатеричная (раздел 11.1).
- **Код Грея** — упорядочение всех 2^n двоичных строк длины n , в котором соседние строки отличаются ровно в одном бите (раздел 11.4).
- **Мультиплексор и декодер** — селектор: выдаёт значение одного из 2^k входов по k адресным битам. Декодер активирует ровно один из 2^k выходов (раздел 11.3).
- **Семейство схем** — последовательность схем $C_1, C_2, \dots$, где C_n имеет n входов. Равномерное, если существует алгоритм, строящий C_n по n (раздел 11.2).
- **Схемная сложность** — минимальный размер схемы, вычисляющей булеву функцию. Глубинная — минимальная глубина такой схемы (глава 11).
- **Код** — отображение сообщений в кодовые слова над алфавитом для защиты или сжатия данных (раздел 13.1).
- **Расстояние Хэмминга** — количество позиций, в которых два кодовых слова различаются (раздел 13.1).

- **Параметры кода** — тройка $(n, |C|, d)$: длина слов, число кодовых слов и кодовое расстояние (раздел 13.1).
- **Код Хэмминга** — линейный код с расстоянием 3, исправляющий одну ошибку. Параметры $(2^r - 1, 2^r - r - 1, 3)$ (раздел 13.2).
- **Линейный код** — код, кодовые слова которого образуют линейное подпространство над конечным полем (раздел 13.4).
- **Вес слова** — число ненулевых позиций $\text{wt}(x)$. Кодовое расстояние линейного кода равно минимальному весу ненулевого кодового слова (раздел 13.4).
- **Скорость кода** — доля информационных битов $R = \frac{k}{n}$, дополнение до единицы — избыточность (раздел 13.4).
- **Граница Синглтона** — $d \leq n - k + 1$. Коды, достигающие границы, называются MDS (раздел 13.11).
- **Префиксный код** — код, в котором ни одно слово не является началом другого. Такие коды декодируются однозначно и без задержки (раздел 13.7).
- **Код Хаффмана** — префиксный код с минимальной средней длиной для заданного распределения частот (раздел 13.7).
- **Кодовое расстояние** — минимальное расстояние Хэмминга между парами различных кодовых слов. Определяет исправляющую способность (раздел 13.1).
- **Порождающая и проверочная матрицы** — G кодирует сообщение (uG) . H проверяет слово: $Hx^T = 0$ для корректного (раздел 13.4).
- **Граница Хэмминга** — $|C| \cdot \sum_{i=0}^t \binom{n}{i} \leq 2^n$: верхняя оценка числа кодовых слов кода, исправляющего t ошибок (раздел 13.11).
- **Циклический код** — линейный код, замкнутый относительно циклического сдвига. Задаётся порождающим многочленом (раздел 13.9).
- **Код Рида–Соломона** — кодирует сообщение-многочлен его значениями в точках конечного поля. MDS-код, достигает границы Синглтона (раздел 13.10).
- **Пропускная способность канала** — $C = 1 - H(p)$ для двоичного симметричного канала: граница скорости надёжной передачи (раздел 13.6).
- **Модель** — набор значений переменных, при котором формула истинна. Формула выполняема, если модель существует (раздел 14.1).
- **SAT** — задача булевой выполнимости: существует ли набор значений переменных, при котором КНФ-формула истинна (раздел 14.1).
- **Теорема Кука–Левина** — SAT NP-полна: любая задача из NP полиномиально сводится к SAT (раздел 14.8).
- **Резолюция** — правило вывода: из $(C \vee x)$ и $(D \vee \bar{x})$ следует $(C \vee D)$. Опровержение — вывод пустого дизъюнкта (раздел 2.11).

- **CDCL** — Conflict-Driven Clause Learning: алгоритм для SAT с обучением на конфликтах и нехронологическим возвратом (раздел 14.4).
- **2-SAT** — полиномиальный случай SAT: выполнимость сводится к проверке компонент сильной связности в графе импликаций (раздел 14.1).
- **k -SAT** — SAT, в котором каждый дизъюнкт содержит ровно k литералов. 2-SAT разрешима за полиномиальное время, 3-SAT NP-полна (раздел 14.1).
- **Граф импликаций** — ориентированный граф на литералах: дизъюнкт $(l_1 \vee l_2)$ даёт рёбра $\overline{l_1} \rightarrow l_2$ и $\overline{l_2} \rightarrow l_1$ (раздел 14.1).
- **Преобразование Цейтина** — перевод схемы в КНФ с линейным ростом размера: по вспомогательной переменной на вентиль (раздел 14.1).
- **DPLL** — поиск с возвратом с распространением единичного дизъюнкта и устранением чистых литералов. Предшественник CDCL (раздел 14.4).
- **Ограничение мощности** — кодирование ограничений «не более k из n » схемами вроде sequential counter за $O(nk)$ дизъюнктов (раздел 14.4).
- **Теория первого порядка** — пара $\mathcal{T} = (\Sigma, \mathcal{M})$ из сигнатуры и класса допустимых интерпретаций (глава 15).
- **Разрешимость теории** — множество общезначимых предложений теории разрешимо: существует завершающаяся процедура, отвечающая «да» или «нет» на вопрос общезначимости (глава 15).
- **SMT** — Satisfiability Modulo Theories: выполнимость кванторно-свободных формул по модулю теории. SAT-решатель на булевом скелете плюс theory-солвер (глава 15).
- **Theory-солвер** ($\mathcal{T}$ -солвер) — процедура решения (*decision procedure*) конъюнкций литералов в конкретной теории: арифметике, равенстве, массивах (глава 15).
- **DPLL(T)** — архитектура SMT-солвера: CDCL на булевом скелете, theory-солвер как оракул совместности, конфликты теории — выученные дизъюнкты (глава 15).
- **Congruence closure** — алгоритм разрешимости равенств с неинтерпретируемыми функциями: слияние классов термов по равенству и конгруэнтности (глава 15).
- **Разностная логика** (*difference logic*, DL) — фрагмент линейной арифметики из литералов $x - y \leq c$. ограничения читаются и над целыми, и над вещественными числами (DL — фрагмент одновременно LIA и LRA). Ограничению $u - v \leq c$ отвечает ребро $v \rightarrow u$. конъюнкция выполнима тогда и только тогда, когда в графе ограничений нет отрицательного цикла (глава 15).
- **Битовый вектор** (*bit-vector*) — целое фиксированной разрядности, арифметика по модулю 2^n как в процессоре. Решается bit-blasting'ом — разворачиванием битов в булеву схему (глава 15).

- **Линейная арифметика** — теории линейных неравенств: LRA над вещественными и LIA над целыми числами. Линейный терм — сумма переменных с константными коэффициентами, умножение переменных запрещено. Кванторно-свободная выполнимость LRA полиномиальна, LIA NP-полна (глава 15).
- **Арифметика Пеано** ($\mathcal{T}_{PA}$) — теория натуральных чисел с сигнатурой $\{0, S, +, \times, =\}$ и аксиомами, включающими умножение (глава 15).
- **Арифметика Пресбургера** — теория натуральных чисел с $0, S, +$ и $=$: арифметика Пеано без умножения. Разрешима (в отличие от арифметики Пеано), но любая процедура решения требует в худшем случае двойной экспоненты (глава 15).
- **Теория массивов** ($\mathcal{T}_{AX}$) — теория памяти с функциями $\text{read}(a, i)$ и $\text{write}(a, i, v)$. Аксиомы: read-over-write ($\text{read}(\text{write}(a, i, v), i) = v$, запись не трогает другие индексы) и экстенциональность (массивы, равные по каждому индексу, равны). Кванторно-свободный фрагмент NP-полон, полная теория неразрешима (глава 15).
- **Nelson-Oppen** (метод Нельсона–Оппена) — комбинация theory-солверов разных теорий: пурификация, отдельное решение, обмен выведенными равенствами и неравенствами об общих переменных до неподвижной точки. Условия применимости: непересекающиеся сигнатуры (теории делят только равенство) и стабильная бесконечность (глава 15).
- **Логическое программирование** — парадигма, в которой программа — множество определённых клауз, а выполнение — поиск доказательства (глава 16).
- **Терм** — переменная, константа или структура $f(t_1, \dots, t_n)$. Данные логической программы (глава 16).
- **Грунтовый терм** — терм без переменных (раздел 16.1).
- **Подстановка переменных** — конечное отображение переменных в термы. Применяется одновременно (глава 16).
- **Унификация** — решение уравнения между термами. Наиболее общий унификатор (МГУ) единствен с точностью до переименования (глава 16).
- **Occurs-check** — запрет связывать переменную с термом, который её содержит (глава 16).
- **Атом** — атомарная формула $p(t_1, \dots, t_n)$ из предикатного символа и термов. Из атомов собираются определённые клаузы и цели (раздел 16.4).
- **Определённая клауза** — факт или правило вида $A \leftarrow B_1, \dots, B_n$. Программа — конечное множество таких клауз (глава 16).
- **SLD-резолюция** — процедура доказательства: выбрать левую цель, унифицировать с головой переименованной клаузы, подставить тело (глава 16).
- **Бэктрекинг** — возврат к следующему отложенному выбору при неудаче ветви. Поиск в глубину по SLD-дереву (глава 16).

- **Отрицание как провал** — $\neg(P)$ истинно, если поиск P не дал решений. Опирается на замкнутый мир, не является логическим отрицанием (глава 16).
- **Задача выбора** — из элементов с весами собрать допустимое подмножество наибольшего суммарного веса (глава 17).
- **Жадный алгоритм** — упорядочивает элементы по убыванию веса и добавляет очередной, только если допустимость сохраняется. На матроидах оптимален, вне их может ошибаться (глава 17).
- **Матроид** — система независимости со свойством замены. Жадный алгоритм оптимален на матроидах (глава 17).
- **Система независимости** — семейство множеств, замкнутое относительно подмножеств (наследственность) (глава 17).
- **Свойство замены** — при $|A| < |B|$ для допустимых A и B множество A пополняется элементом из $B \setminus A$ с сохранением допустимости. Именно оно отделяет матроиды от систем независимости (глава 17).
- **База и ранг матроида** — максимальное независимое множество и его размер. Все базы матроида имеют одинаковый размер (глава 17).
- **Теорема Радо–Эдмондса** — жадный алгоритм находит независимое множество максимального веса на матроиде. Вне класса матроидов может ошибаться (глава 17).

§ 36.18 Счёт, случайность и бесконечность

- **Перестановки и сочетания** — $P(n, k) = \frac{n!}{(n-k)!}$ (порядок важен). $C(n, k) = \binom{n}{k}$ (порядок не важен) (раздел 18.3).
- **Размещение** (k -permutation) — выбор k из n объектов с учётом порядка, $P(n, k) = \frac{n!}{(n-k)!}$ (раздел 18.3).
- **Сочетания с повторениями** — выбор k элементов из n типов, где повторения разрешены, а порядок не важен, $\binom{n+k-1}{k}$. Формула «звёзд и перегородок» (раздел 18.3).
- **Перестановки с повторениями** — упорядочение n объектов с неразличимыми экземплярами, мультиномиальный коэффициент $\frac{n!}{n_1! \dots n_k!}$. Считает анаграммы слова (раздел 18.3).
- **Бином Ньютона** — $(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$. Тожество Паскаля, свёртка Вандермонда (раздел 18.4).
- **Включение-исключение** (*inclusion-exclusion*) — формула для размера объединения пересекающихся множеств: $|\cup A_i| = \sum |A_i| - \sum |A_i \cap A_j| + \dots$ (раздел 18.5).
- **Линейное рекуррентное соотношение** — уравнение вида $a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$ с постоянными коэффициентами, задающее следующий член через k предыдущих. Решается методом характеристического уравнения (глава 18).

- **Числа Каталана** — $C_n = \frac{1}{n+1} \binom{2n}{n}$. Считают бинарные деревья, скобочные последовательности, триангуляции (раздел 18.7).
- **Принцип Дирихле** (*pigeonhole principle*) — если n объектов размещены по m ящикам и $n > m$, то хотя бы в одном ящике не менее двух объектов (раздел 18.1).
- **Лемма Бёрнсайда** — количество орбит $= \frac{1}{|G|} \sum_{g \in G} |X^g|$. Подсчёт с точностью до симметрии (раздел 18.11).
- **Правило дополнения** — $|A| = |U| - |\bar{A}|$: применяется, когда посчитать «не- A » проще, чем A (раздел 18.1).
- **Правило суммы и произведения** — $|A \cup B| = |A| + |B|$ для непересекающихся множеств. $|A \times B| = |A| \cdot |B|$ для независимых шагов (раздел 18.1).
- **Числа Стирлинга** — $S(n, k)$ — число разбиений n -множества на k блоков (2-го рода). $c(n, k)$ — число перестановок с k циклами (1-го рода) (раздел 18.7).
- **Числа Белла** — $B_n = \sum_{k=0}^n S(n, k)$: общее число разбиений n -элементного множества (раздел 18.7).
- **Двенадцатеричный путь** — классификация распределений n шаров по k ящикам по трём осям: различимость шаров, ящиков и ограничение на заполнение (раздел 18.10).
- **Числа Рамсея** — $R(s, t)$: минимальное n , при котором любой граф на n вершинах содержит клику размера s или независимое множество размера t (раздел 18.13).
- **НОД** (наибольший общий делитель) — наибольшее целое число, делящее a и b , обозначается $\gcd(a, b)$. Взаимная простота — случай $\gcd(a, b) = 1$ (раздел 19.2).
- **Алгоритм Евклида** — нахождение НОД повторным делением с остатком. Расширенный вариант даёт коэффициенты Безу (раздел 19.2).
- **Коэффициенты Безу** — целые x и y с $ax + by = 1$ для взаимно простых a и m , которые находит расширенный алгоритм Евклида. Тогда x — обратный элемент к a по модулю m (раздел 19.2).
- **Малая теорема Ферма** — $a^{p-1} \equiv 1 \pmod{p}$ для простого p и a , не делящегося на p (раздел 19.4).
- **Теорема Эйлера** — $a^{\varphi(m)} \equiv 1 \pmod{m}$ для взаимно простых a, m . Функция Эйлера (раздел 19.4).
- **Псевдопростое число** — составное n , взаимно простое с a , удовлетворяющее $a^{n-1} \equiv 1 \pmod{n}$. Обманывает тест Ферма по основанию a . Числа Кармайкла (наименьшее 561) псевдопросты по всем взаимно простым основаниям (раздел 19.5).
- **Китайская теорема об остатках** — система сравнений $x \equiv a_i \pmod{m_i}$ при попарно взаимно простых m_i имеет решение, единственное по модулю произведения (раздел 19.6).
- **RSA** — шифрование на основе трудности разложения: открытый ключ (n, e) , закрытый d с $ed \equiv 1 \pmod{\varphi(n)}$ (раздел 19.7).
- **Образующий** — такой элемент g группы $\mathbb{Z}_p^*$, что его степени $g^0, g^1, \dots, g^{p-2}$ пробегают все ненулевые остатки (раздел 19.8).
- **Дискретный логарифм** — по $g^a \pmod{p}$ найти a . Предположительно труден, лежит в основе обмена Диффи–Хеллмана (раздел 19.8).

- **Принцип Керкгоффса** — секретность должна жить в ключе, а алгоритм может быть открытым. Безопасность проверяется атакой (раздел 19.1).
- **Сравнение по модулю** — $a \equiv b \pmod{m}$, если m делит $a - b$. Классы остатков образуют кольцо (раздел 19.3).
- **Обратный элемент по модулю** — $a \cdot a^{-1} \equiv 1 \pmod{m}$. Существует тогда и только тогда, когда $\gcd(a, m) = 1$ (раздел 19.3).
- **Односторонняя функция** — функцию легко вычислить, но трудно обратить (факторизация, дискретный логарифм). Фундамент RSA (раздел 19.7).
- **Протокол Диффи–Хеллмана** — установление общего секрета $g^{ab} \pmod{p}$ по открытому каналу (раздел 19.8).
- **Атака «человек посередине»** — злоумышленник вклинивается между участниками обмена ключами и ведёт с каждым из них отдельный протокол, выдавая себя за другого (раздел 19.8).
- **Эллиптическая кривая** — $E : y^2 = x^3 + ax + b$ над $\text{GF}(p)$. Точки образуют абелеву группу (раздел 19.9).
- **Вероятностное пространство** — пара (Ω, P) , где Ω — конечное или счётное множество исходов, P — вероятностная мера (раздел 20.1).
- **Событие** — любое подмножество $A \subseteq \Omega$ множества исходов. Событие происходит, когда наступивший исход принадлежит A (раздел 20.1).
- **Условная вероятность** — $P(A | B) = \frac{P(A \cap B)}{P(B)}$. Пересчёт вероятности при получении новой информации (раздел 20.2).
- **Формула Байеса** — $P(H | E) = \frac{P(E|H)P(H)}{P(E)}$. Переход от априорной вероятности к апостериорной (раздел 20.3).
- **Независимость событий** — $P(A \cap B) = P(A) \cdot P(B)$. Знание об одном событии не меняет оценку другого (раздел 20.2).
- **Матожидание** (*expected value*) — $E[X] = \sum x \cdot P(X = x)$. Взвешенное среднее возможных значений случайной величины (раздел 20.4).
- **Линейность матожидания** — $E[aX + bY] = a E[X] + b E[Y]$ всегда, без условий о независимости (раздел 20.5).
- **Индикатор события** — случайная величина 1_A , равная 1, если событие произошло, и 0 иначе. Мост от событий к матожиданию: $E[1_A] = P(A)$ (раздел 20.5).
- **Случайная величина** — функция $X : \Omega \rightarrow \mathbb{R}$ на вероятностном пространстве. Числовой исход эксперимента (раздел 20.4).
- **Распределение** — набор вероятностей $P(X = x)$ по всем значениям случайной величины. Полностью описывает её вероятностное поведение (раздел 20.4).
- **Дисперсия** — $\text{Var}[X] = E[(X - E[X])^2]$: мера разброса значений относительно среднего (раздел 20.4).
- **Стандартное отклонение** — $\sigma[X] = \sqrt{\text{Var}[X]}$: разброс в тех же единицах, что и сама величина (раздел 20.4).
- **Бернулли, биномиальное и геометрическое распределения** — три распределения одного базового опыта: исход испытания с вероятностью успеха p , число успехов за n испытаний, номер первого успеха. Биномиальное: $P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$. Геометрическое: $P(X = k) = (1 - p)^{k-1} p$ (раздел 20.6).

- **Неравенства Маркова и Чебышёва** — $P(X \geq a) \leq \frac{E[X]}{a}$ и $P(|X - \mu| \geq a) \leq \frac{\text{Var}[X]}{a^2}$: оценки хвостов через среднее и дисперсию (раздел 20.7).
- **Стандартное нормальное распределение** — «колокол» с плотностью $\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$: вероятность промежутка равна площади под кривой. $\Phi(z) = P(Z \leq z)$ — накопленная площадь слева (раздел 20.9).
- **Центральная предельная теорема (ЦПТ)** — стандартизованная сумма n независимых одинаково распределённых величин сходится к стандартному нормальному распределению: предельная форма одна для всех исходных распределений (раздел 20.9).
- **Вероятностный метод** — если вероятность, что случайный объект обладает свойством, положительна, то такой объект существует. Неконструктивное доказательство существования (раздел 20.12).
- **ОПФ** — обычная производящая функция $A(x) = \sum a_n x^n$. Упаковка последовательности в формальный степенной ряд (раздел 21.1).
- **ЭПФ** — экспоненциальная производящая функция $E(x) = \sum a_n \frac{x^n}{n!}$. Для помеченных объектов и факториального роста (раздел 21.2).
- **Свёртка последовательностей** — произведению ОПФ соответствует свёртка: $A(x)B(x) = \sum c_n x^n$, где $c_n = \sum a_i b_{n-i}$ (раздел 21.1).
- **Решение рекурренций через ОПФ** — рекуррента $\rightarrow$ уравнение на $A(x) \rightarrow$ простейшие дроби $\rightarrow$ коэффициент $[x^n]A(x)$ (раздел 21.1).
- **Сдвиг последовательности** — умножение ОПФ на x вставляет ноль в начало последовательности. Деление на x после удаления константы сдвигает влево (раздел 21.1).
- **Метод простейших дробей** — разложение рациональной ОПФ $\frac{P(x)}{Q(x)}$ на слагаемые $\frac{c}{(1-\rho x)^m}$ для извлечения коэффициента $[x^n]$ (раздел 21.1).
- **Обобщённый бином Ньютона** — $(1+x)^k$ при отрицательном или нецелом k — бесконечный ряд с коэффициентами $\binom{k}{n}$ (раздел 21.1).
- **Вероятностная производящая функция** — $G_X(s) = \sum p_k s^k$: кодирует распределение дискретной случайной величины (раздел 21.3).
- **Композиция числа** — представление n в виде упорядоченной суммы положительных слагаемых. Их ровно 2^{n-1} (раздел 21.3).
- **Комбинаторный вид** — правило, сопоставляющее каждому конечному множеству-носителю множество структур, устойчивое к переносу вдоль биекций. Число F -структур извлекается из ЭПФ: $|F[n]| = n![x^n]F(x)$ (глава 21).
- **Сумма, произведение, композиция и набор видов** — операции исчисления видов, переводящиеся в операции над ЭПФ: $F + G$, $F \cdot G$, $F(G(x))$, $e^{F(x)}$ (глава 21).
- **Аксиомы Пеано** — аксиоматика натуральных чисел: ноль, следование, индукция. В первом порядке имеет нестандартные модели (раздел 22.1).
- **Индуктивное множество** — содержит $\emptyset$ и замкнуто относительно следования $x \mapsto x \cup \{x\}$. Натуральные числа — пересечение всех индуктивных множеств (раздел 22.3).
- **Ординалы фон Неймана** — построение натуральных чисел из пустого множества: $0 = \emptyset$, $n + 1 = n \cup \{n\}$. ω — множество всех натуральных чисел (раздел 22.3).

- **Сечения Дедекинда** — построение $\mathbb{R}$ как множества подмножеств $\mathbb{Q}$, замкнутых вниз и не имеющих наибольшего элемента (раздел 22.6).
- **Аксиома выбора** — из любого семейства непустых множеств можно выбрать по элементу. Эквивалентна лемме Цорна и теореме Цермело (раздел 22.7).
- **Континуум-гипотеза** — между $\aleph_0$ и $2^{\aleph_0}$ нет промежуточных кардиналов. Независимость от ZFC доказали Гёдель (1940) и Коэн (1963) (раздел 22.11).
- **Недостижимый кардинал** — несчётный кардинал κ , для которого из $\lambda < \kappa$ следует $2^\lambda < \kappa$ (сильный предел) и который не является объединением менее чем κ множеств меньшей мощности (регулярность). Существование нельзя доказать в ZFC (глава 7).
- **Кумулятивная иерархия** — построение всех множеств по уровням: $V_0 = \emptyset$, $V_{\alpha+1} = \mathcal{P}(V_\alpha)$, на предельном ординале — объединение предыдущих (глава 22).
- **Аксиома большого кардинала** — постулат, утверждающий существование кардинала, которое ZFC (если она непротиворечива) не может доказать. Первый и простейший пример — недостижимый кардинал (глава 22).
- **Алгебра Дедекинда** — тройка (A, a, f) с выделенным элементом, инъективным следованием и принципом индукции. Любые две алгебры Дедекинда изоморфны (раздел 22.2).
- **Целые числа** — построение $\mathbb{Z} = \frac{\mathbb{N} \times \mathbb{N}}{\sim}$ с $(a, b) \sim (c, d)$ при $a + d = b + c$ (раздел 22.4).
- **Рациональные числа** — построение $\mathbb{Q} = \frac{\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})}{\sim}$ с $(a, b) \sim (c, d)$ при $ad = bc$ (раздел 22.5).
- **Парадокс Банаха–Тарского** — шар в $\mathbb{R}^3$ разбивается на конечное число частей, из которых движениями собираются два шара того же радиуса (раздел 22.9).
- **Множество Витали** — подмножество $[0, 1)$, которому нельзя приписать длину. Построено с помощью аксиомы выбора (раздел 22.10).

§ 36.19 Вычислимость и её границы

- **ДКА** — детерминированный конечный автомат $(Q, \Sigma, \delta, q_0, F)$. В каждом состоянии по каждому символу ровно один переход (раздел 23.2).
- **НКА** — недетерминированный конечный автомат. $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ допускает несколько переходов из одного состояния (раздел 23.3).
- **Регулярный язык** — язык, распознаваемый конечным автоматом. Эквивалентно описывается регулярным выражением (раздел 23.2).
- **Расширенная функция переходов** — $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$, доопределяющая δ с одиночных символов на целые слова (раздел 23.2).
- **Лемма о накачке** (*pumping lemma*) — инструмент доказательства нерегулярности: всякая достаточно длинная цепочка регулярного языка содержит накачиваемую середину (раздел 23.8).
- **Теорема Майхилла–Нерода** — регулярность языка эквивалентна конечности числа классов эквивалентности отношения неразличимости (раздел 23.9).

- **Минимизация ДКА** — построение автомата с минимальным числом состояний через объединение неразличимых состояний (раздел 23.10).
- **Алфавит** — конечное множество Σ символов, неделимых значков, из которых собираются слова (раздел 23.1).
- **Слово** — конечная последовательность символов алфавита, длина слова $|w|$ равна числу его символов (раздел 23.1).
- **Конкатенация** — приписывание слова v к концу слова u с обозначением uv , пустое слово ϵ служит единицей (раздел 23.1).
- **Формальный язык** — произвольное подмножество Σ^* . Слово — конечная последовательность символов алфавита (раздел 23.1).
- **Регулярное выражение** — алгебраическое описание языка операциями объединения, конкатенации и звезды Клини (раздел 23.5).
- **Теорема Клини** — регулярные выражения и конечные автоматы описывают один и тот же класс языков (раздел 23.5).
- **Конструкция подмножеств** — из НКА с n состояниями строится эквивалентный ДКА с не более чем 2^n состояниями. Недетерминизм не добавляет силы (раздел 23.4).
- **ϵ -переход** — спонтанный переход НКА, меняющий состояние без чтения входного символа (раздел 23.3).
- **Контекстно-свободная грамматика** — правила вида $A \rightarrow \alpha$. Порождает вложенные структуры языков программирования (раздел 24.1).
- **Автомат с магазинной памятью** (*pushdown automaton*) — конечный автомат со стеком. Распознаёт в точности КС-языки (раздел 24.2).
- **Иерархия Хомского** — классификация формальных языков: регулярные $\subset$ контекстно-свободные $\subset$ контекстно-зависимые $\subset$ рекурсивно перечислимые (раздел 24.8).
- **Лемма о накачке для КС-языков** — инструмент доказательства неконтекстно-свободности: достаточно длинная цепочка содержит две накачиваемые середины (раздел 24.3).
- **Дерево разбора** — представление вывода в КС-грамматике: корень S , внутренние вершины — нетерминалы, листья — терминалы (раздел 24.3).
- **Неоднозначная грамматика** — грамматика, в которой некоторое слово имеет более одного дерева разбора (раздел 24.3).
- **Эквивалентность КСГ и МП-автомата** — язык порождается КС-грамматикой тогда и только тогда, когда распознаётся автоматом с магазинной памятью (раздел 24.2).
- **Нормальная форма Хомского** — приведённая КС-грамматика с правилами $A \rightarrow BC$ и $A \rightarrow a$, к которой приводится любая КС-грамматика (раздел 24.4).
- **Контекстно-зависимая грамматика** — правила $\alpha A \beta \rightarrow \alpha \gamma \beta$: замена нетерминала разрешена только в контексте (раздел 24.7).
- **Машина Тьюринга** — модель вычислений с бесконечной лентой, читающей/пишущей головкой и конечным управлением $(Q, \Gamma, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ (раздел 25.1).

- **Проблема останковки** — неразрешимая задача определения, остановится ли машина Тьюринга на данном входе (раздел 26.2).
- **Частичная вычислимость** — функция f частично вычислима, если существует МТ, останавливающаяся с $f(w)$ на ленте для всех w из области определения (раздел 25.5).
- **m-сводимость** (*many-one reduction*) — $A \leq_m B$ если существует вычислимая f , такая что $x \in A \leftrightarrow f(x) \in B$ (раздел 26.3).
- **Разрешимость** — язык разрешим, если существует МТ, которая всегда останавливается и принимает в точности его строки (раздел 26.1).
- **Теорема Райса** — любое нетривиальное экстенциональное свойство распознаваемых языков алгоритмически неразрешимо (раздел 26.4).
- **Первая теорема Гёделя о неполноте** — непротиворечивая рекурсивно аксиоматизируемая теория, содержащая арифметику, имеет предложение, которое теория не может ни доказать, ни опровергнуть (глава 26).
- **Вторая теорема Гёделя о неполноте** — такая теория не доказывает собственную непротиворечивость (глава 26).
- **Функция занятого бобра** — $\Sigma(n)$: максимум единиц, который записывает на пустой ленте останавливающаяся МТ с n состояниями. Рост любой вычислимой функции рано или поздно обгоняет (глава 26).
- **Диофантово уравнение** — уравнение $p(x_1, \dots, x_n) = 0$ с многочленом p целых коэффициентов, которое решают в целых числах (раздел 26.10).
- **Диофантово множество** — $A \subseteq \mathbb{N}^m$, задаваемое параметрами $y_1, \dots, y_k$, зануляющими многочлен с целыми коэффициентами. По теореме MRDP это в точности перечислимые множества (раздел 26.10).
- **Символьное исполнение** — анализ, при котором переменные получают символьные выражения, ветвление разветвляет состояния по условиям пути, а решатель отбрасывает невыполнимые пути (раздел 31.8).
- **Машина Тьюринга с оракулом** — МТ с лентой-оракулом, отвечающей на вопросы принадлежности языку. Формализует относительную вычислимость (раздел 26.5).
- **Арифметическая иерархия** — уровни Σ_n и Π_n : классы языков по числу чередований кванторов $\exists$ и $\forall$ поверх разрешимого предиката (раздел 26.6).
- **Тезис Чёрча–Тьюринга** — класс интуитивно вычислимых функций совпадает с классом функций, вычислимых на МТ (раздел 25.5).
- **Конфигурация** — мгновенное описание МТ: состояние, содержащее ленты и позиция головки (раздел 25.1).
- **Распознаваемый язык** — язык, все строки которого МТ принимает. На остальных входах может не останавливаться (раздел 25.1).
- **Универсальная машина Тьюринга** — МТ, симулирующая любую другую по её описанию. Программа и данные неразличимы (раздел 25.6).
- **Квайн** — программа, печатающая собственный текст: неподвижная точка трансформации, вклеивающей описание программы в её же вывод (раздел 25.6).

- **λ -терм** — выражение из переменных, аппликаций (MN) и абстракций $(\lambda x.M)$. Множество всех термов — Λ (раздел 27.1).
- **β -редукция** — правило вычисления: $(\lambda x.M)N$ редуцируется в $M[x := N]$. Основной шаг вычисления в λ -исчислении (раздел 27.2).
- **Нормальная форма** — терм без β -редексов. Если существует — единственна по теореме Чёрча–Россера (раздел 27.2).
- **Числа Чёрча** — кодирование натуральных чисел как $n = \lambda f x. f^n(x)$. Сложение, умножение — через композицию итераций (раздел 27.3).
- **Y-комбинатор** — терм, строящий неподвижную точку для любого F . Позволяет выразить рекурсию без рекурсивных определений (раздел 27.4).
- **Свободные и связанные переменные** — абстракция $\lambda x.M$ связывает свободные вхождения x в M (раздел 27.1).
- **Комбинатор** — λ -терм без свободных переменных (раздел 27.1).
- **α -конверсия** — переименование связанной переменной с условиями, предотвращающими захват (раздел 27.1).
- **α -эквивалентность** — совпадение термов с точностью до переименования связанных переменных (раздел 27.1).
- **Подстановка терма** — $M[x := N]$: замена свободных вхождений x в M на N с переименованием связанных при необходимости (раздел 27.1).
- **Теорема Чёрча–Россера** — β -редукция конфлюэнтна: у терма не более одной нормальной формы (раздел 27.2).
- **λ -определимость** — класс λ -определимых функций совпадает с классом вычислимых на МТ (раздел 27.5).
- **Простые типы** — базовые типы (Nat , Bool) и функциональные $(\sigma \rightarrow \tau)$. Приписываются термам в $\lambda \rightarrow$ (раздел 28.1).
- **Subject Reduction** — тип терма не меняется при β -редукции. Вычисление сохраняет типовую корректность (раздел 28.4).
- **Strong Normalization** — каждый типизируемый терм в $\lambda \rightarrow$ сильно нормализуем. Бесконечных циклов нет (раздел 28.4).
- **Соответствие Карри–Ховарда** — изоморфизм: типы = формулы, термы = доказательства, редукция = нормализация доказательств (раздел 28.5).
- **λ -куб Барендрегта** — восемь типовых систем, различающихся разрешёнными видами зависимостей: $\lambda \rightarrow$, $\lambda 2$, λP , $\lambda \omega$, λC (раздел 28.6).
- **Зависимый функциональный тип** — $\Pi x : A. B(x)$: функции, результат которых на каждом $a : A$ лежит в своём типе $B(a)$ (раздел 28.6).
- **Зависимое произведение** — $\Sigma x : A. B(x)$: пары (a, b) , где тип второго компонента $B(a)$ подобран по первому (раздел 28.6).
- **System F ($\lambda 2$)** — типовая система с квантификацией по типам $\forall \alpha. \sigma$, формализация полиморфизма (раздел 28.6).
- **Правила типизации** — правила var , app , abs : соответствуют аксиоме, $\rightarrow$ -удалению и $\rightarrow$ -введению натурального вывода (раздел 28.2).
- **Нетипизируемость** — терм $\Omega = (\lambda x. xx)(\lambda x. xx)$ не имеет типа: самоприменение требует $\sigma = \sigma \rightarrow \tau$ (раздел 28.3).

- **Семантика Крипке** — модель $M = (W, R, V)$: миры, отношение достижимости и оценка атомов. $\Box p$ истинно в мире, где p истинно во всех достижимых мирах (глава 33).
- **Автомат Бюхи** — автомат над бесконечными словами: слово принимается, если принимающее состояние встречается бесконечно часто. Основа model checking LTL (глава 33).
- **Интуиционистская логика** — логика без закона исключённого третьего: доказательство — конструкция. Таблица истинности здесь ни при чём. Соответствие Карри–Ховарда: доказательства — программы (глава 34).
- **Линейная логика** — логика ресурсов: гипотеза расходуется ровно один раз, конъюнкция расщепляется на мультипликативную и аддитивную. Моделирует владение памятью (Rust) и сетевые сессии (глава 34).
- **Пруф-ассистенты** — инструменты формальной верификации (Coq, Lean, Agda) на основе соответствия Карри–Ховарда (раздел 28.6).
- **Теорема компактности** — если каждое конечное подмножество множества предложений имеет модель, то и всё множество имеет модель (раздел 29.1).
- **Нестандартная модель арифметики** — модель PA первого порядка, содержащая элементы, большие всех натуральных чисел (раздел 29.1).
- **Трансфинитная индукция** — обобщение математической индукции: база, переход и предельный шаг для ординалов (раздел 29.2).
- **Ординал** — транзитивное множество, вполне упорядоченное отношением принадлежности, например ω или $\omega + 1$ (раздел 29.2).
- **Наследственная запись** — запись числа по основанию b , в которой показатели степеней сами записаны в наследственной записи (раздел 29.2).
- **Сюрреальные числа** — числа, построенные рекурсивно как пары множеств $\{L \mid R\}$. Содержат $\mathbb{R}$, ординалы и бесконечно малые в одном поле (раздел 29.3).
- **Ординальная арифметика** — сложение и умножение ординалов некоммутативны: $1 + \omega = \omega$, но $\omega + 1 > \omega$ (раздел 29.2).
- **Предельный ординал** — ординал без предшественника. Требуется предельного шага в трансфинитной индукции (раздел 29.2).
- **Нестандартный анализ** — расширение $\mathbb{R}$ до поля гиперреальных чисел с бесконечно малыми и бесконечно большими элементами (раздел 29.4).
- **Класс P** — задачи, разрешимые детерминированной машиной Тьюринга за полиномиальное время (раздел 30.1).
- **Класс NP** — задачи, проверка решения которых лежит в P . Формально: разрешимые недетерминированной МТ за полиномиальное время (раздел 30.2).
- **Полиномиальная сводимость** — $A \leq_p B$ если существует полиномиально вычислимая f , такая что $x \in A \leftrightarrow f(x) \in B$ (раздел 30.3).
- **NP-полнота** — задача NP-полна, если она в NP и к ней полиномиально сводится любая задача из NP (раздел 30.3).
- **P против NP** — открытая проблема: верно ли, что $P = NP$? Одна из семи задач тысячелетия института Клэя (раздел 30.2).

- **Сертификат** — полиномиально проверяемое свидетельство принадлежности входа языку. Определение NP через угадывание сертификата (раздел 30.2).
- **NP-трудная задача** — задача, к которой полиномиально сводится любая задача из NP. NP-полная — NP-трудная и лежащая в NP (раздел 30.3).
- **Класс PSPACE** — языки, разрешимые МТ с полиномиальной памятью. $\text{NPSPACE} = \text{PSPACE}$ (теорема Сэвича) (раздел 30.7).
- **FPT** — параметризованная задача, разрешимая за $f(k) \cdot n^{O(1)}$: экспонента загнана в параметр (раздел 30.5).
- **Ядро** — алгоритм, переводящий за полиномиальное время экземпляр (x, k) в эквивалентный (x', k') размера, ограниченного функцией от параметра k (раздел 30.6).
- **Класс coNP** — языки, дополнения которых лежат в NP (раздел 30.7).
- **BPP** — языки, разрешаемые вероятностной МТ за полиномиальное время с вероятностью ошибки не более $\frac{1}{3}$ (раздел 30.7).
- **Полиномиальная иерархия** — классы Σ_i^P, Π_i^P над NP, порождаемые чередующимися кванторами (раздел 30.8).
- **Абстрактная интерпретация** — статический анализ программы через аппроксимацию множеств состояний элементами решётки (раздел 31.1).
- **Домен знаков** — решётка из $\perp, -, 0, +$ и $\top$: каждое целое значение абстрагируется его знаком (раздел 31.1).
- **Информационный порядок** — $a \preceq b$, если конкретный смысл a содержится в конкретном смысле b , движение вверх означает потерю точности (раздел 31.1).
- **Абстрактный домен** — решётка $(D, \preceq, \perp, \top, \sqcup, \sqcap)$ абстрактных состояний программы (раздел 31.1).
- **Конкретизация** — $\gamma : D \rightarrow \mathcal{P}(\mathcal{C})$. Абстрактное значение к множеству конкретных состояний (раздел 31.1).
- **Абстракция** — $\alpha : \mathcal{P}(\mathcal{C}) \rightarrow D$. Множество состояний к наиболее точному абстрактному значению (раздел 31.1).
- **Переносная функция** — абстрактная семантика оператора: $\llbracket s \rrbracket^\# : D \rightarrow D$ (раздел 31.2).
- **Widening** — оператор расширения, заставляющий итерацию неподвижной точки сойтись на бесконечных доменах (раздел 31.5).
- **Корректность анализа** — анализ не пропускает реальных ошибок. Плата — возможные ложные срабатывания (раздел 31.6).
- **Интервальный домен** — абстрактный домен из интервалов $[l, h]$. Доказывает свойства вроде выхода за границы массива (раздел 31.3).
- **Домен констант и домен вычетов** — абстракции переменной конкретным числом c или классом вычетов $c \bmod m$, отслеживающие равенства и делимость (раздел 31.3).
- **Наименьшая неподвижная точка** — результат анализа цикла. Достигается итерацией от $\perp$ по монотонной функции (раздел 31.4).
- **Narrowing** — оператор сужения, возвращающий точность, потерянную widening (раздел 31.5).

- **Модальная логика** — логика необходимости $\Box$ и возможности $\Diamond$ на фреймах миров с отношением достижимости (глава 33).
- **Фрейм** — пара (W, R) : множество миров и отношение достижимости. Аксиомы модальной логики соответствуют свойствам R (глава 33).
- **Модель Крипке** — тройка (S, R, L) : состояния, переходы, разметка атомарных утверждений. Семантика темпоральных логик (глава 33).
- **LTL** — линейная темпоральная логика: формулы о путях с операторами $\bigcirc$ (next), U (until), $\Box$ (always), $\Diamond$ (eventually) (глава 33).
- **CTL** — логика ветвящегося времени: кванторы пути A/E перед темпоральными операторами. Выражает свойства деревьев вычислений (глава 33).
- **Неподвижные точки CTL** — $E(\varphi U \psi)$ задаётся наименьшей, а $E\Box\varphi$ наибольшей неподвижной точкой своего уравнения (глава 33).
- **Model checking** — автоматическая проверка темпорального свойства на конечной модели: разметка состояний для CTL, автоматы Бюхи для LTL (глава 33).
- **Исчисление первого порядка** — система вывода для логики предикатов: аксиомы, правила и кванторные правила, механически проверяемые (глава 3). Выводимость обозначается $\vdash$.
- **Полнота** — свойство системы вывода: всё общезначимое выводимо ($\models \varphi$ влечёт $\vdash \varphi$). Для логики первого порядка — теорема Гёделя о полноте (глава 3).
- **Компактность** — свойство логики первого порядка: множество формул выполнимо тогда и только тогда, когда выполнимо каждое его конечное подмножество. Следствие полноты, источник нестандартных моделей (главы 3, 29).
- **БХК-интерпретация** (от англ. BHK, Brouwer–Heyting–Kolmogorov) — конструктивное чтение связок: доказательство — это построение (пара, функция, свидетель) (глава 34).
- **Алгебра Хейтинга** — дистрибутивная решётка с псевдодополнением $a \rightarrow b$. Полная семантика интуиционистской логики (глава 34).
- **Интуиционистская модель Крипке** — фрейм с предпорядком R : в мире w' известно не меньше, чем в w , если $R(w, w')$ (глава 34).
- **Теорема Гливенко** — классическая выводимость φ равносильна интуиционистской выводимости $\neg\neg\varphi$ (глава 34).
- **Промежуточная логика** — логика между интуиционистской и классической: все теоремы интуиционизма в ней выводимы, её теоремы верны классически, а некоторая классическая тавтология в ней не выводима (глава 34).
- **Нечёткое множество** (*fuzzy set*) — обобщение классического множества: элемент принадлежит с некоторой степенью $\mu_A(x) \in [0, 1]$ (раздел 35.1).
- **Функция принадлежности** — $\mu_A : X \rightarrow [0, 1]$. Для каждого x число $\mu_A(x)$ — степень принадлежности к нечёткому множеству (раздел 35.1).
- **α -срез** — $A_\alpha = \{x \in X \mid \mu_A(x) \geq \alpha\}$. Мост между нечёткими и обычными множествами (раздел 35.1).
- **Нечёткое отношение** — нечёткое множество на декартовом произведении $X \times Y$. Градуальные связи между элементами (раздел 35.3).

- **Нечёткий вывод** — схема: фаззификация $\rightarrow$ применение правил $\rightarrow$ агрегация $\rightarrow$ дефаззификация (раздел 35.6).
- **Дефаззификация** — переход от нечёткого результата к конкретному числу. Завершающий шаг нечёткого вывода (раздел 35.6).
- **Носитель, ядро и высота** — характеристики нечёткого множества: элементы с $\mu_A(x) > 0$, с $\mu_A(x) = 1$ и максимум μ_A (раздел 35.1).
- **Теорема декомпозиции** — нечёткое множество однозначно восстанавливается по α -срезам (раздел 35.1).
- **Треугольное нечёткое число** — нечёткое множество с треугольной функцией принадлежности (l, n, r) (раздел 35.2).
- **t -норма и s -норма** — аксиоматические обобщения пересечения и объединения нечётких множеств (раздел 35.3).
- **Лингвистическая переменная** — переменная, значениями которой служат нечёткие множества на числовой шкале (раздел 35.4).
- **Категория** — объекты, стрелки между ними и ассоциативная композиция с тождественными стрелками: моноид — категория из одного объекта (раздел 36.1).
- **hom-множество** — $\mathcal{C}(A, B)$: множество всех стрелок из A в B (раздел 36.1).
- **Мономорфизм и эпиморфизм** — стрелки, отменяемые слева и справа в композиции: стрелочные двойники инъекции и сюръекции (раздел 36.2).
- **Противоположная категория** — $\mathcal{C}^{\text{op}}$ с обращёнными началами и концами стрелок, источник двойственных утверждений (раздел 36.2).
- **Функтор** — отображение между категориями, переводящее объекты и стрелки с сохранением композиции и тождеств (раздел 36.3).
- **Естественное преобразование** — семейство стрелок $\alpha_A : F(A) \rightarrow G(A)$ между функторами, согласованное с каждой стрелкой категории (раздел 36.4).
- **Эквивалентность категорий** — функтор с псевдообратным: обе композиции естественно изоморфны тождественным функторам (раздел 36.4).
- **Терминальный и начальный объект** — объекты, в которые и из которых из каждого объекта ведёт ровно одна стрелка (раздел 36.5).
- **Произведение и копроизведение** — объект с парой проекций или вложений, через который единственным образом проходит любая такая пара стрелок: универсальное свойство произведения множеств (раздел 36.5).
- **Предел** — терминальный объект среди конусов над диаграммой, обобщение произведения на произвольную схему объектов и стрелок (раздел 36.6).
- **Монада** — эндофунктор $T : \mathcal{C} \rightarrow \mathcal{C}$ с единицей η и умножением μ , дисциплинирующий эффекты вроде отказа или множественных ответов (раздел 36.9).