λ-исчисление

От синтаксиса термов до монад в Haskell

2026

УДК 510.6
ББК 22.12

Елюков А. С.

λ-исчисление: от синтаксиса термов до монад в Haskell.

Небольшая книга о лямбда-исчислении: синтаксис термов, бета-редукция, кодирование Чёрча, Y-комбинатор, типизированные исчисления и построенные на них концепты Haskell — классы типов, функторы, аппликативы, моноиды, монады и трансформеры монад.

Для студентов и разработчиков, знакомых с основами математической логики и функционального программирования.

Оглавление

Глава 1

Базовые понятия

Синтаксис термов

Как в математике невероятное количество направлений — анализ, ТФКП и т. д. — держится на нескольких аксиомах, которые принимаются на веру, так и в λ-исчислении всего три конструкции являются основой. Весь язык строится из переменной, абстракции и аппликации; всё остальное — числа, логика, структуры данных и рекурсия — появляется уже поверх них.

$$\Lambda \ ::= \ x \ \mid \ (\lambda x.\, M) \ \mid \ (M\, N) \tag{1.1}$$

где x — переменная; $(\lambda x.\, M)$ — *абстракция*: анонимная функция с параметром x и телом M; $(M\, N)$ — *аппликация*: применение функции M к аргументу N. При этом N может быть любым термом: переменной x, другой аппликацией $(P\, Q)$ или целой абстракцией $(\lambda y.\, y)$. Например: $M\, x$, $M\, (P\, Q)$, $M\, (\lambda y.\, y)$.

Несколько важных замечаний: аппликация лево-ассоциативна: $M\, N\, P \equiv ((M\, N)\, P)$; тело абстракции простирается максимально вправо: $\lambda x.\, M\, N \equiv \lambda x.\, (M\, N)$; цепочка абстракций сокращается: $\lambda x\, y\, z.\, M \equiv \lambda x.\, \lambda y.\, \lambda z.\, M$.

Терм — это дерево: листья — переменные, внутренние узлы — абстракции и аппликации. Полезно приучить себя видеть терм как дерево, а не как строку.

Связанные и свободные переменные

В этой главе рассмотрим достаточно тривиальные вещи, но они будут очень полезны в последующем материале. Введём такое понятие, как свободные переменные, которые обозначим FV. Вот несколько примеров применения этого понятия ко всем известным нам термам.

$$\mathrm{FV}(x) = \{x\}, \qquad \mathrm{FV}(\lambda x.\, M) = \mathrm{FV}(M) \setminus \{x\}, \qquad \mathrm{FV}(M\, N) = \mathrm{FV}(M) \cup \mathrm{FV}(N) \tag{1.2}$$

У одиночной переменной x свободна сама x. В абстракции $\lambda x.\, M$ переменная x связывается ближайшей лямбдой, поэтому из свободных переменных тела M её надо убрать. В применении $M\, N$ свободные переменные берутся из обеих частей: всё свободное в M плюс всё свободное в N. Терм без свободных переменных называется

замкнутым, или комбинатором; при этом одна и та же буква может встретиться и свободно, и связанно, например, в $x\,(\lambda x.\,x)$.

Напомню, что терм представляет собой дерево, а переменные — листья. Так вот, в листья можно подставлять, в свою очередь, поддеревья, то есть термы. Такая операция называется подстановкой. Вот несколько примеров таких подстановок:

$$x[x{:=}N] = N, \qquad y[x{:=}N] = y \;\; (y \neq x) \tag{1.3}$$

$$(M_1\,M_2)[x{:=}N] = M_1[x{:=}N]\;M_2[x{:=}N] \tag{1.4}$$

$$(\lambda y.\,M)[x{:=}N] = \lambda y.\,M[x{:=}N], \qquad y \neq x,\; y \notin \mathrm{FV}(N) \tag{1.5}$$

Вот видите, появились уже какие-то ограничения. А почему? Тут есть важная тонкость. Например, если условие $y \notin \mathrm{FV}(N)$ нарушено, свободная переменная «захватывается» чужой абстракцией:

$$(\lambda y.\,x)[x{:=}y] \neq \lambda y.\,y \quad \text{— смысл изменился!}$$

Лекарство — α-конверсия, но об этом чуть позже.

α- и η-конверсии

В предыдущей главе мы осознали важную проблему, возникающую при подстановке. Тут поговорим подробнее о методе решения этой проблемы.

α-конверсия

Имя связанной переменной — не более чем метка: $\lambda x.\,x$ и $\lambda y.\,y$ — одна и та же функция.

$$\lambda x.\,M \;=_\alpha\; \lambda y.\,M[x{:=}y], \qquad y \notin \mathrm{FV}(M) \tag{1.6}$$

Особо одарённые товарищи от возни с переименованием избавляются и используют индексы де Брёйна: сами лямбды остаются, но становятся безымянными, а каждое вхождение переменной записывается числом — сколько лямбд нужно пройти от него вверх до связывателя (отсчёт с нуля, 0 — ближайшая охватывающая лямбда). Тогда α-эквивалентные термы получают одинаковую запись, и α-конверсия как понятие исчезает.

- $\lambda x.\,x \to \lambda.\,0$ (единственная переменная указывает на свою лямбду);
- $\lambda x.\,\lambda y.\,x\,y \to \lambda.\,\lambda.\,1\,0$ (x стало 1, а y — 0).

η-конверсия

На самом деле это совсем мелочь, но всё равно надо коротко сказать. η-конверсия говорит, что «обёртка» $\lambda x.\,M\,x$ ничем не отличается от самого M.

$$\lambda x.\,(M\,x) \;=_\eta\; M, \qquad x \notin \mathrm{FV}(M) \tag{1.7}$$

η-редукция сворачивает такую обёртку влево, а η-экспансия разворачивает её вправо.

В Haskell это знакомый point-free стиль:

```
sum xs = foldr (+) 0 xs
-- превращается в
sum = foldr (+) 0
```

β-редукция

Переливание из пустого в порожнее продолжается, и сейчас поговорим о β-редукции. Именно не о конверсии, так как экспансии тут быть не может: есть только редукция. Вообще вся операционная семантика λ-исчисления — одно правило переписывания, не более чем.

Вызов функции в любом языке — частный случай β-редукции.

$$(\lambda x.\, M)\, N \;\to_\beta\; M[x{:=}N] \tag{1.8}$$

Терм вида $(\lambda x.\, M)\, N$ называется редексом. Редукция может применяться к любому редексу в любом месте терма. Обозначение $\to_\beta$ означает один шаг β-редукции, $\twoheadrightarrow_\beta$ — ноль и более шагов, а $=_\beta$ — эквивалентность, порождённая редукцией. Нормальная форма, или НФ, — это терм без редексов: вычислять больше нечего. Результатом работы программы как раз и должна быть нормальная форма.

Если запретить редукцию под знаком λ — не заглядывать в тело абстракции, — получается более слабое понятие: *слабая нормальная форма*. В ней стянуты все редексы, кроме спрятанных под λ. Скажем, $\lambda x.\, (\lambda y.\, y)\, z$ уже в слабой нормальной форме, хотя обычной НФ $\lambda x.\, z$ ещё не достиг. Именно до такой формы (точнее, слабой головной, WHNF) и вычисляют реальные языки: тело функции не трогают, пока её не позвали — подробнее в главе о стратегиях редукции.

Так в Haskell живут бесконечные структуры: их вычисляют до слабой головной формы, но не до конца.

```
nats = [1..]        -- бесконечный список 1 : 2 : 3 : ...
-- уже в WHNF: виден внешний конструктор (:), хвост - невычисленный thunk;
-- полной НФ нет - чтобы её достичь, пришлось бы построить весь список
take 3 nats         -- [1,2,3]: берём только начало, и всё работает
```

Примеры β-редукций:

$$(\lambda x.\, x)\, y \to_\beta y$$

$$(\lambda f.\, f\,(f\, z))\,(\lambda x.\, x) \to_\beta (\lambda x.\, x)\,((\lambda x.\, x)\, z) \twoheadrightarrow_\beta z$$

Но, как всегда, есть нюансы, и главный нюанс заключается в том, что далеко не все термы редуцируются до нормальной формы. Выше на Haskell я уже дал пример бесконечного списка, но тут будет не о нём. Терм Ω редуцируется сам в себя — вычисление не завершается никогда:

$$\Omega = (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \to_\beta \Omega \to_\beta \cdots$$

Но на этом приколы не заканчиваются: бывает так, что порядок вычисления редексов влияет на то, зациклится алгоритм или нет. В ленивых языках такое сплошь и рядом, но компилятор обычно умеет сам это разруливать, если мы не навяжем ему свою волю. В общем, дальше только интереснее будет.

Каррирование

А что, если функция двух переменных? А такие вообще бывают? Ответ простой: нет, не бывают. Ладно-ладно, шучу. Бывают, просто они, как луковица, завёрнуты в оболочки, а такой концепт называется каррированием в честь Хаскелла Карри. Да, это тот самый Haskell Curry, в честь которого назван и язык Haskell.

$$f(x, y) \quad \rightsquigarrow \quad f = \lambda x.\, \lambda y.\, M, \qquad f\, a\, b = (f\, a)\, b \tag{1.9}$$

Формально это изоморфизм, то есть биекция между двумя множествами функций.

$$\mathrm{Hom}(A \times B,\, C) \;\cong\; \mathrm{Hom}(A,\, \mathrm{Hom}(B,\, C)) \tag{1.10}$$

В формуле слева — функции из пары $A \times B$ в C, справа — функции из A в функции из B в C. Каждой функции двух аргументов $g\colon A \times B \to C$ отвечает ровно одна каррированная $\hat{g}\colon A \to (B \to C)$ по правилу $\hat{g}(a)(b) = g(a, b)$, и наоборот — $g(a, b) = \hat{g}(a)(b)$. Эти два перехода взаимно обратны, ничего не теряя и не добавляя, поэтому оба множества функций попросту неразличимы.

Несколько примеров из Haskell:

```
-- тип функции нескольких аргументов - правоассоциативная цепочка стрелок
f :: a -> b -> c   -- то же самое, что a -> (b -> c)

-- частичное применение: add 2 - «недоданная» функция
add :: Int -> Int -> Int
add x y = x + y
add2 :: Int -> Int
add2 = add 2
map add2 [1, 2, 3]  -- [3, 4, 5]

-- переходники curry и uncurry - свидетели изоморфизма
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c

-- сечения операторов - то же частичное применение
(+3)  :: Num a => a -> a
(*2)  :: Num a => a -> a
```

Кодирование Чёрча

В чистом λ-исчислении нет ни чисел, ни булевых значений, ни структур — только функции. Кодирование Чёрча показывает, что ничего другого и не нужно: данные — это их собственные операции обработки.

Начнём с булевых значений. И заметьте, как тут всё практично привязано. Не существует правды и лжи самих по себе: эти понятия имеют смысл только тогда, когда предполагают стратегии действия. В противном случае смысла в существовании правды и лжи нет. Прочувствуйте формулы.

$$\mathbf{tru} = \lambda t\, f.\, t, \qquad \mathbf{fls} = \lambda t\, f.\, f \tag{1.11}$$

Здесь t и f — это просто два аргумента (мнемоника: ветка «true» и ветка «false»): **tru** возвращает первый, а **fls** — второй. То есть само булево значение — это уже выбор одного из двух, готовый **if**.

$$\mathbf{if} = \lambda b\, t\, e.\, b\, t\, e, \qquad \mathbf{not} = \lambda b.\, b\, \mathbf{fls}\, \mathbf{tru} \tag{1.12}$$

Здесь **if** почти ничего не делает сам: он берёт булево значение b и отдаёт ему две ветки t и e, а b уже выбирает нужную. **not** устроен так же просто: он даёт булевому значению ветки в обратном порядке, поэтому правда выбирает ложь, а ложь — правду.

$$\mathbf{and} = \lambda p\, q.\, p\, q\, p, \qquad \mathbf{or} = \lambda p\, q.\, p\, p\, q \tag{1.13}$$

В **and** первое булево значение p решает, что вернуть: если p истинно, результат зависит от q, а если ложно, сразу возвращается ложь. В **or** наоборот: если p истинно, можно сразу вернуть правду, а если ложно, приходится смотреть на q.

Подставим истину ($1 = \mathbf{tru}$, $0 = \mathbf{fls}$) и произведём редукции:

$$\mathbf{not\ tru} \;\to_\beta\; \mathbf{tru\ fls\ tru} \;\twoheadrightarrow_\beta\; \mathbf{fls} \qquad (\neg 1 = 0)$$

$$\mathbf{and\ tru\ tru} \;\twoheadrightarrow_\beta\; \mathbf{tru\ tru\ tru} \;\twoheadrightarrow_\beta\; \mathbf{tru} \qquad (1 \wedge 1 = 1)$$

$$\mathbf{or\ tru\ tru} \;\twoheadrightarrow_\beta\; \mathbf{tru\ tru\ tru} \;\twoheadrightarrow_\beta\; \mathbf{tru} \qquad (1 \vee 1 = 1)$$

Теперь поговорим о парах: это очень важный тип, который широко используется в Haskell. Пара — функция, хранящая два значения и ждущая «получателя»:

$$\mathbf{pair} = \lambda x\, y\, f.\, f\, x\, y, \qquad \mathbf{fst} = \lambda p.\, p\, \mathbf{tru}, \qquad \mathbf{snd} = \lambda p.\, p\, \mathbf{fls} \tag{1.14}$$

fst передаёт паре **tru**, а **snd** — **fls**, и та отдаёт нужную компоненту:

$$\mathbf{fst}\,(\mathbf{pair}\ a\ b) \;\twoheadrightarrow_\beta\; \mathbf{tru}\ a\ b \;\twoheadrightarrow_\beta\; a$$

$$\mathbf{snd}\,(\mathbf{pair}\ a\ b) \;\twoheadrightarrow_\beta\; \mathbf{fls}\ a\ b \;\twoheadrightarrow_\beta\; b$$

Заметьте, как интересно: функции **fst** и **snd** принимают аргумент-пару и передают ей управление — дальше она сама решает, что возвращать. Это, так сказать, функции-декораторы.

Теперь самое насущное — натуральные числа. Число n — это «применить функцию n раз»:

$$\overline{n} = \lambda f\, x.\, f^{\,n}(x): \qquad \overline{0} = \lambda f\, x.\, x, \quad \overline{1} = \lambda f\, x.\, f\, x, \quad \overline{2} = \lambda f\, x.\, f\,(f\, x) \tag{1.15}$$

Важно: f и x здесь произвольные — их подставляет тот, кто пользуется числом. Нумерал фиксирует только «сколько раз применить», а не «что применять».

Скормим нумералу функцию f и аргумент x — и он применит f ровно столько раз, сколько задаёт число:

$$\overline{0}\ f\ x\ \twoheadrightarrow_\beta\ x$$

$$\overline{3}\ f\ x\ \twoheadrightarrow_\beta\ f\,(f\,(f\,x))$$

Теперь научимся считать — вся арифметика вырастает из одного приёма «применить f нужное число раз».

Начнём с прибавления единицы: **succ** навешивает на число ещё одно применение f:

$$\mathbf{succ} = \lambda n\, f\, x.\, f\,(n\, f\, x) \tag{1.16}$$

$$\mathbf{succ}\,\overline{n}\ \twoheadrightarrow_\beta\ \lambda f\, x.\, f\,(\overline{n}\, f\, x)\ \twoheadrightarrow_\beta\ \lambda f\, x.\, f\big(f^{\,n}(x)\big)\ =\ \lambda f\, x.\, f^{\,n+1}(x) = \overline{n+1}$$

Сложение продолжает эту идею — применяет f сначала $\overline{n}$ раз, а потом ещё $\overline{m}$:

$$\mathbf{add} = \lambda m\, n\, f\, x.\, m\, f\,(n\, f\, x) \tag{1.17}$$

$$\mathbf{add}\,\overline{m}\,\overline{n}\ \twoheadrightarrow_\beta\ \lambda f\, x.\, \overline{m}\, f\,(\overline{n}\, f\, x)\ \twoheadrightarrow_\beta\ \lambda f\, x.\, f^{\,m}\big(f^{\,n}(x)\big)\ =\ \overline{m+n}$$

Умножение — это уже повторное сложение:

$$\mathbf{mul} = \lambda m\, n\, f.\, m\,(n\, f) \tag{1.18}$$

$$\mathbf{mul}\,\overline{m}\,\overline{n}\ \twoheadrightarrow_\beta\ \lambda f.\, \overline{m}\,(\overline{n}\, f)\ \twoheadrightarrow_\beta\ \lambda f\, x.\, \big(f^{\,n}\big)^{m}(x)\ =\ \lambda f\, x.\, f^{\,mn}(x) = \overline{mn}$$

Возведение в степень — повторное умножение, и в записи оно совсем короткое:

$$\mathbf{pow} = \lambda m\, n.\, n\, m \tag{1.19}$$

$$\mathbf{pow}\,\overline{m}\,\overline{n}\ \to_\beta\ \overline{n}\,\overline{m}\ \twoheadrightarrow_\beta\ \overline{m^{\,n}}$$

В **pow** аргументы просто меняются местами: $\overline{n}\,\overline{m}$ — это $\overline{m}$, применённое как функция n раз, что и даёт $\overline{m^{\,n}}$.

Вычитание неожиданно трудно: **pred** строится через пары — прогоняем по числу пару $(i-1,\ i)$ и берём первую компоненту:

$$\mathbf{pred} = \lambda n.\, \mathbf{fst}\,\big(n\ (\lambda p.\, \mathbf{pair}\,(\mathbf{snd}\, p)\,(\mathbf{succ}\,(\mathbf{snd}\, p)))\ (\mathbf{pair}\,\overline{0}\,\overline{0})\big) \tag{1.20}$$

По распространённой легенде Стивен Клини придумал функцию-предшественник (**pred**, predecessor) — ключ к вычитанию нумералов Чёрча — прямо в кресле у стоматолога.

На **pred** держится всё «вычитательное»: само вычитание — это многократный **pred**, а деление — многократное вычитание. Отдельного λ-определения для деления не приводим: новой идеи там нет, тот же **pred** в цикле, только запись громоздкая.

Проверим **pred** на общем нумерале $\overline{n}$. Обозначим шаг $\sigma = \lambda p.\, \mathbf{pair}\,(\mathbf{snd}\,p)\,(\mathbf{succ}\,(\mathbf{snd}\,p))$ — он сдвигает пару: роняет первую компоненту и наращивает вторую:

$$\sigma\,(\mathbf{pair}\;a\;b) \;\twoheadrightarrow_\beta\; \mathbf{pair}\;b\;(\mathbf{succ}\;b)$$

Подставляем $\overline{n}$ в определение — это значит применить σ ровно n раз к стартовой паре $\mathbf{pair}\,\overline{0}\,\overline{0}$:

$$\mathbf{pred}\;\overline{n} \;\to_\beta\; \mathbf{fst}\,\big(\overline{n}\,\sigma\,(\mathbf{pair}\,\overline{0}\,\overline{0})\big) \;\twoheadrightarrow_\beta\; \mathbf{fst}\,\big(\sigma^{\,n}(\mathbf{pair}\,\overline{0}\,\overline{0})\big)$$

Пары бегут по цепочке — вторая компонента считает шаги, первая отстаёт на единицу:

$$\mathbf{pair}\,\overline{0}\,\overline{0} \;\to\; \mathbf{pair}\,\overline{0}\,\overline{1} \;\to\; \mathbf{pair}\,\overline{1}\,\overline{2} \;\to\; \cdots \;\to\; \mathbf{pair}\,\overline{n-1}\,\overline{n}$$

Осталось взять первую компоненту:

$$\mathbf{pred}\;\overline{n} \;\twoheadrightarrow_\beta\; \mathbf{fst}\,(\mathbf{pair}\,\overline{n-1}\,\overline{n}) \;\twoheadrightarrow_\beta\; \overline{n-1}$$

У $\overline{0}$ цепочка не двигается, поэтому $\mathbf{pred}\,\overline{0} \twoheadrightarrow_\beta \overline{0}$ — предшественником нуля считаем нуль.

$$\mathbf{iszero} = \lambda n.\, n\,(\lambda z.\, \mathbf{fls})\,\mathbf{tru} \tag{1.21}$$

iszero скармливает числу функцию, которая на каждом шаге отвечает **fls**, и старт **tru**: у нуля шагов нет — остаётся **tru**, а у любого другого числа хотя бы один шаг даёт **fls**.

Список — его собственная правая свёртка (fold-кодирование):

$$[x_1, \ldots, x_n] \;\rightsquigarrow\; \lambda c\, n.\, c\, x_1\,(c\, x_2\,(\cdots(c\, x_n\, n))) \tag{1.22}$$

Тот же приём в Haskell — тип `foldr`: список полностью определяется тем, как он сворачивается. Нумерал Чёрча — «список из n одинаковых применений».

```
-- список строится конструктором (:) (cons) и завершается [] (nil):
[1, 2, 3]  ==  1 : (2 : (3 : []))   -- True

-- Чёрч-кодирование - та же цепочка, только (:) и [] вынесены в параметры;
-- в Haskell это ровно foldr:
foldr (:) []  [1, 2, 3]   -- 1 : (2 : (3 : [])) = [1,2,3]   (собрали список обратно)
foldr (+) 0   [1, 2, 3]   -- 1 + (2 + (3 + 0))  = 6         (свернули в сумму)
```

Ну ведь есть же нормальное программирование, в котором есть и числа, и даже с плавающей точкой. Зачем всё это городить? Я хотел декларативного программирования, где всё просто, а тут... На самом деле ответ простой: ребята, если вы не любите цирк, вам там нечего делать, не приходите, но если вы любите цирк...

Комбинатор Карри

Мы уже многое построили на базе λ-термов, а именно: данные, ветвление, но вот явно не хватает циклов. Которых в своём классическом понимании и не будет вообще. Вместо этого будет рекурсия. То есть надо придумать такой терм, который бы крутил любую функцию бесконечно. Конечность мы обеспечим за счёт ветвления, но это будет потом.

Ось Земли является неподвижной, в то время как сама планета вращается вокруг этой оси. Уравнение оси — важнейшая характеристика вращения, и его хорошо бы уметь выводить. В случае с термами аналогия такая же, только мы ищем не ось, а точку, которая неподвижна относительно терма. Есть такая теорема (о неподвижной точке), сформулирую вольно: у любого λ-терма F есть неподвижная точка — терм X, для которого

$$F\,X =_\beta X. \tag{1.23}$$

Такую точку нетрудно построить явно. Возьмём $X \equiv (\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x))$; единственный редекс сводится одной β-редукцией:

$$X = (\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x)) \;\to_\beta\; F\big((\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x))\big) = F\,X.$$

Итак, $X \to_\beta F\,X$, в частности $F\,X =_\beta X$.

Терм X можно представить как $(\lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)))\,F$. Нам надо вырвать неподвижную точку из уравнения, а функцию крутить непосредственно комбинатором Карри, который обозначается так:

$$\mathrm{Y} = \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)) \tag{1.24}$$

Подставив $X = \mathrm{Y}\,F$ в теорему, получаем тождество неподвижной точки — рабочую форму рекурсии:

$$\mathrm{Y}\,F \;=_\beta\; F\,(\mathrm{Y}\,F) \tag{1.25}$$

Тождество (1.25) — именно равенство, а не редукция: никакая цепочка прямых β-шагов не приведёт $\mathrm{Y}\,F$ к $F\,(\mathrm{Y}\,F)$. В самом деле, единственный редекс даёт $\mathrm{Y}\,F \to_\beta A \equiv (\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x))$ — это знакомый нам X из доказательства теоремы. Дальше цепочка идёт через $F\,(A)$, $F\,(F\,(A))$ и так далее, и терма $F\,(\mathrm{Y}\,F)$ в ней нет: внутри A комбинатор Y уже не встречается. Зато $F\,(\mathrm{Y}\,F)$ сам за один шаг редуцируется в $F\,(A)$:

$$\mathrm{Y}\,F \;\to_\beta\; A \;\to_\beta\; F\,(A), \qquad F\,(\mathrm{Y}\,F) \;\to_\beta\; F\,(A).$$

Стороны тождества равны лишь потому, что сходятся к общему редукту $F\,(A)$, а чтобы из $\mathrm{Y}\,F$ получить именно $F\,(\mathrm{Y}\,F)$, пришлось бы шагнуть против стрелки (конверсия).

Опробуем это на классическом примере — факториале. Рекурсивный шаг задаём шаблоном с дополнительным параметром f (будущим рекурсивным вызовом):

$$F = \lambda f\,n.\,\mathbf{if}\;(\mathbf{iszero}\,n)\;\overline{1}\;(\mathbf{mul}\,n\,(f\,(\mathbf{pred}\,n)))$$

Неподвижная точка этого шаблона и есть факториал. С комбинатором Карри развёртка идёт через β-равенство:

$$\begin{aligned}\mathbf{fac}\,\overline{3} = \mathrm{Y}\,F\,\overline{3} &=_\beta F\,(\mathrm{Y}\,F)\,\overline{3}\\ &=_\beta \mathbf{if}\,(\mathbf{iszero}\,\overline{3})\,\overline{1}\,(\mathbf{mul}\,\overline{3}\,(\mathrm{Y}\,F\,\overline{2})) =_\beta \mathbf{mul}\,\overline{3}\,(\mathrm{Y}\,F\,\overline{2})\\ &=_\beta \mathbf{mul}\,\overline{3}\,(\mathbf{mul}\,\overline{2}\,(\mathbf{mul}\,\overline{1}\,(\mathrm{Y}\,F\,\overline{0})))\\ &=_\beta \mathbf{mul}\,\overline{3}\,(\mathbf{mul}\,\overline{2}\,(\mathbf{mul}\,\overline{1}\,\overline{1})) = \overline{6}\end{aligned}$$

Эта схема — родная для ленивых языков (call-by-name/call-by-need). В Haskell комбинатор неподвижной точки встроен — `fix` из `Data.Function`, и развёртка идёт ровно как в (1.25), по мере надобности:

```
fix :: (a -> a) -> a
fix f = let x = f x in x   -- узел завязан ленивым let: x ссылается сам на себя

fac :: Integer -> Integer
fac = fix (\f n -> if n == 0 then 1 else n * f (n - 1))
```

У комбинатора Карри, при всей его применимости в Haskell, есть две проблемы — и Haskell обе обходит. Первая — система типов: самоаппликация $x\,x$ в системе Хиндли — Милнера не типизируется, потому что приводит к зацикливанию вывода типов. Поэтому неподвижную точку задают рекурсивным `let` (как в `fix` выше) или прячут самоаппликацию за newtype. Что это за система и почему она так устроена — тема отдельной главы дальше.

Вторая — энергичный порядок вычислений: в строгом (call-by-value) языке определение `let x = f x` крутилось бы вечно — чтобы получить x, среда сразу вычисляет $f\ x$, для этого снова x, и так без конца, не добираясь до полезной работы. Haskell спасает ленью: x раскрывается лишь тогда, когда его значение действительно понадобится, поэтому та же запись работает без зацикливания.

Комбинатор Тьюринга

Комбинатор Карри решает задачу рекурсии, но не безупречно. Один из его изъянов мы уже отметили: рекурсию он лишь декларирует. Тождество $\mathrm{Y}\,F =_\beta F\,(\mathrm{Y}\,F)$ держится на β-равенстве, а прямой редукции из $\mathrm{Y}\,F$ в $F\,(\mathrm{Y}\,F)$ нет — чтобы замкнуть петлю, приходится шагать «назад» (конверсия). Попробуем это исправить: построить другой комбинатор неподвижной точки, который разворачивает рекурсию настоящей β-редукцией вперёд, а не провозглашает её равенством.

Добудем эту редукцию — $\Theta\,F \twoheadrightarrow_\beta F\,(\Theta\,F)$ — «протолкнув» саму функцию аргументом внутрь самоаппликации: возьмём $G = \lambda x\,y.\,y\,(x\,x\,y)$ и положим $\Theta = G\,G$. Тогда за два шага β-редукции:

$$\Theta\,F = (G\,G)\,F \ \to_\beta\ (\lambda y.\,y\,(G\,G\,y))\,F \ \to_\beta\ F\,(G\,G\,F) = F\,(\Theta\,F).$$

Функция (в роли y) здесь не теряется, а на каждом витке заново применяется к воссозданному $\Theta\,F = G\,G\,F$; поэтому равенство и превращается в честную редукцию. Разворачивая G, получаем окончательный вид комбинатора Тьюринга:

$$\Theta = (\lambda x\, y.\, y\,(x\,x\,y))\,(\lambda x\, y.\, y\,(x\,x\,y)), \qquad \Theta\, F \twoheadrightarrow_\beta F\,(\Theta\, F) \tag{1.26}$$

Посчитаем тот же факториал, что и в главе про комбинатор Карри (тот же шаблон F, то же число $\overline{3}$), только теперь каждый виток рекурсии — настоящая β-редукция, а не β-равенство:

$$\begin{aligned}\mathbf{fac}\ \overline{3} = \Theta\, F\ \overline{3} &\twoheadrightarrow_\beta F\,(\Theta\, F)\ \overline{3} \twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ (\Theta\, F\ \overline{2}) \\ &\twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ (\mathbf{mul}\ \overline{2}\ (\mathbf{mul}\ \overline{1}\ (\Theta\, F\ \overline{0}))) \\ &\twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ (\mathbf{mul}\ \overline{2}\ (\mathbf{mul}\ \overline{1}\ \overline{1})) = \overline{6}\end{aligned}$$

Оба дают $\overline{6}$; разница лишь в статусе шага развёртки — равенство у Карри против редукции у Тьюринга.

На практике комбинатор Тьюринга почти не встречается — его роль теоретическая: он показывает, что рекурсия достижима честной редукцией, без шага «назад», и в этом качестве он стандартный спутник Y в учебниках.

Но и Карри, и Тьюринг рассчитаны на нормальный порядок (call-by-name). При строгом, энергичном порядке (call-by-value) оба расходятся: аргумент F — то есть сам $\mathrm{Y}\,F$ или $\Theta\,F$ — вычисляется заранее и разворачивается бесконечно, так и не дойдя до полезной работы. Эту проблему решает следующий комбинатор — Z.

Z-комбинатор

Осталась проблема, которой одинаково подвержены и комбинатор Карри, и комбинатор Тьюринга: в энергичном порядке (call-by-value) оба зацикливаются. Рекурсивный аргумент — сам $\mathrm{Y}\,F$ или $\Theta\,F$ — среда раскручивает заранее, ещё до вызова функции, так и не дойдя до полезной работы. Исправить это можно, отложив рекурсивный вызов — превратив его в значение, которое раскроется лишь тогда, когда действительно понадобится.

Идея до боли простая: над самоаппликацией надо навесить обёртку-функцию. Технически это η-развёртка $x\,x \to \lambda v.\, x\,x\,v$. Завёрнутый в λ вызов — уже значение и разворачивается только при реальном обращении. Так из комбинатора Карри (1.24) получается Z-комбинатор — та же конструкция, только каждое $x\,x$ заменено на $\lambda v.\, x\,x\,v$:

$$Z = \lambda f.\, (\lambda x.\, f\,(\lambda v.\, x\,x\,v))\,(\lambda x.\, f\,(\lambda v.\, x\,x\,v)) \tag{1.27}$$

Проследим первый разворот. Обозначим $W = \lambda x.\, F\,(\lambda v.\, x\,x\,v)$; тогда за два шага β-редукции:

$$Z\,F \ \to_\beta\ W\,W \ \to_\beta\ F\,(\lambda v.\, W\,W\,v) \ =\ F\,(\lambda v.\, Z\,F\,v).$$

Последнее равенство сворачивает $W\,W$ обратно в $Z\,F$. Главное — рекурсивный вызов вышел наружу завёрнутым в λv: под λ он остаётся значением и разворачивается лишь при реальном применении к аргументу (тогда как $\mathrm{Y}\,F$ и $\Theta\,F$ в энергичном порядке разворачиваются сразу).

Тот же факториал, что и в главах про Карри и Тьюринга (тот же шаблон F, то же число $\overline{3}$), считается настоящими β-редукциями, а завёрнутый вызов раскрывается ровно при переходе к следующему числу:

$$\begin{aligned}\mathbf{fac}\ \overline{3} = Z\,F\,\overline{3} &\twoheadrightarrow_\beta F\,(\lambda v.\,Z\,F\,v)\,\overline{3} \twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ ((\lambda v.\,Z\,F\,v)\ \overline{2}) \\ &\twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ (Z\,F\ \overline{2}) \twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ (\mathbf{mul}\ \overline{2}\ (\mathbf{mul}\ \overline{1}\ (Z\,F\ \overline{0}))) \\ &\twoheadrightarrow_\beta \mathbf{mul}\ \overline{3}\ (\mathbf{mul}\ \overline{2}\ (\mathbf{mul}\ \overline{1}\ \overline{1})) = \overline{6}\end{aligned}$$

Поэтому под энергичным порядком (call-by-value) он не разворачивается заранее и не зацикливается — рекурсивный вызов срабатывает точно в нужный момент.

Ни Y, ни Θ, ни Z не типизируются в простом типизированном λ-исчислении: самоаппликация $x\,x$ потребовала бы рекурсивного типа.

В Haskell такой рекурсивный тип и заводят — оборачивая самоаппликацию в newtype; на нём хорошо видно, что Z не опирается на лень. Обычный `fix` держится на ленивом узле `let x = f x`, а Z строит рекурсию самоприменением и прячет рекурсивный вызов под λv — под значение. Заставим вычисление быть строгим (аргумент форсится перед применением функции, как при call-by-value): наивный Y зацикливается, а Z спокойно считает факториал:

```
newtype Rec a = Rec { unRec :: Rec a -> a }

-- $! форсит аргумент до применения f - это и есть энергичный порядок (call-by-value)

-- наивный Y: рекурсивный вызов (unRec x x) не значение, форсится заранее => зацикливается
yStrict :: (a -> a) -> a
yStrict f = w (Rec w) where w x = f $! unRec x x

-- Z: вызов завёрнут в (\v -> ...), это уже значение - $! его не разворачивает
zStrict :: ((b -> c) -> b -> c) -> b -> c
zStrict f = w (Rec w) where w x = f $! \v -> unRec x x v

fac :: (Int -> Int) -> Int -> Int
fac rec n = if n == 0 then 1 else n * rec (n - 1)

-- yStrict fac 5  =>  зацикливается
-- zStrict fac 5  =>  120   -- Z досчитал без всякой ленивости
```

Примеры комбинаторов

Мы разобрали три комбинатора неподвижной точки — Карри, Тьюринга и Z — и пора сказать, что вообще такое комбинатор. Комбинатор — это замкнутый λ-терм, без свободных переменных: он ничего не берёт из окружения, а только тасует и склеивает собственные аргументы. Таких термов множество, мы посмотрим наиболее часто встречающиеся:

- $\mathrm{I} = \lambda x.\,x$ — тождество: отдаёт аргумент как есть.
- $\mathrm{K} = \lambda x\,y.\,x$ — константа: берёт первый аргумент, второй отбрасывает.
- $\mathrm{S} = \lambda f\,g\,x.\,f\,x\,(g\,x)$ — подстановка: раздаёт x и в f, и в g, затем применяет одно к другому.
- $\mathrm{B} = \lambda f\,g\,x.\,f\,(g\,x)$ — композиция: прогоняет x через g, потом через f.
- $\mathrm{C} = \lambda f\,x\,y.\,f\,y\,x$ — перестановка: меняет местами два аргумента.
- $\mathrm{W} = \lambda f\,x.\,f\,x\,x$ — удвоение: скармливает x функции дважды.

На Haskell комбинаторы выглядят так:

```
id 5              -- => 5            (I)
const 1 2         -- => 1            (K)
ap (+) (*2) 3     -- => 9  = 3 + 3*2 (S)
((+1) . (*2)) 3   -- => 7  = 3*2 + 1 (B)
flip (-) 3 10     -- => 7  = 10 - 3  (C)
join (*) 4        -- => 16 = 4*4     (W)
```

SKI базис

Само λ-исчисление можно построить всего на трёх из них — S, K, I. В комбинаторной логике их берут за примитивы:

$$\mathrm{S} = \lambda x\, y\, z.\, x\, z\, (y\, z), \qquad \mathrm{K} = \lambda x\, y.\, x, \qquad \mathrm{I} = \lambda x.\, x \tag{1.28}$$

Причём I даже не обязателен как примитив — он выражается через S и K:

$$\mathrm{S}\,\mathrm{K}\,\mathrm{K} = \mathrm{I} \tag{1.29}$$

Давайте подставим и докажем формулу — раскроем $\mathrm{S}\,\mathrm{K}\,\mathrm{K}$ на произвольном аргументе z по определениям (1.28):

$$\mathrm{S}\,\mathrm{K}\,\mathrm{K}\,z \;\to_\beta\; \mathrm{K}\,z\,(\mathrm{K}\,z) \;\to_\beta\; z \;=\; \mathrm{I}\,z.$$

На любом z получаем z — ровно то, что делает I, значит $\mathrm{S}\,\mathrm{K}\,\mathrm{K} = \mathrm{I}$.

Любой λ-терм механически переводится в комбинаторы. Терм строится тремя способами — переменная, применение и абстракция; переменные и применение в комбинаторной логике уже есть, а вот абстракцию нужно устранить. Это делает bracket abstraction — операция $[x]\,M$, «вынести переменную x из терма M». Её результат — комбинаторный терм, в котором x уже не встречается, но который, если подать ему x обратно, снова сводится к M. Определяется она по виду тела M:

$$[x]\,x = \mathrm{I}, \qquad [x]\,M = \mathrm{K}\,M \;\; (x \notin \mathrm{FV}(M)), \qquad [x]\,(M\,N) \\ = \mathrm{S}\,([x]\,M)\,([x]\,N) \tag{1.30}$$

- $[x]\,x = \mathrm{I}$: тело — это сама x; вернуть аргумент как есть умеет I.
- $[x]\,M = \mathrm{K}\,M$ при $x \notin \mathrm{FV}(M)$: тело от x не зависит, поэтому переданный аргумент надо просто выбросить.
- $[x]\,(M\,N) = \mathrm{S}\,([x]\,M)\,([x]\,N)$: в применении x может прятаться и в M, и в N, поэтому аргумент нужно раздать обоим — этим и занимается S.

Если же тело само — абстракция $\lambda y.\, M$, сперва убирают внутреннюю переменную, а потом внешнюю: $[x]\,(\lambda y.\, M) = [x]\,([y]\,M)$; так вложенные λ снимаются изнутри наружу, пока не останутся только переменные и применения.

Разберём $\lambda x.\, f\, x\, x$. Тело $f\, x\, x$ — это применение $(f\, x)\, x$, так что раскручиваем правилом для применения, сводя всё к S, K, I:

$$[x]\,(f\,x\,x) = \mathrm{S}\,([x]\,(f\,x))\,([x]\,x) = \mathrm{S}\,\big(\mathrm{S}\,([x]\,f)\,([x]\,x)\big)\,\mathrm{I}$$
$$= \mathrm{S}\,(\mathrm{S}\,(\mathrm{K}\,f)\,\mathrm{I})\,\mathrm{I}$$

Переменных в ответе нет — только S, K, I и свободная f: $\lambda x.\, f\,x\,x \quad \rightsquigarrow \mathrm{S}\,(\mathrm{S}\,(\mathrm{K}\,f)\,\mathrm{I})\,\mathrm{I}$.

Теорема Чёрча — Россера

Все дороги ведут в Рим. Точно? На это есть теорема Чёрча — Россера, которая гарантирует единственность нормальной формы, какими бы путями мы к ней ни шли. Конечно, надо заметить, что в Тьюринг-полном языке мы можем такой нормальной формы и не достичь, но вот если достигаем, то она единственна, то есть каким путём ни иди, всегда придёшь именно в Рим.

Если $M \twoheadrightarrow_\beta N$ и $M \twoheadrightarrow_\beta K$, то найдётся L такой, что $N \twoheadrightarrow_\beta L$ и $K \twoheadrightarrow_\beta L$. Иначе говоря, $\twoheadrightarrow_\beta$ обладает свойством ромба (сходимостью):

$$M \twoheadrightarrow_\beta N,\;\; M \twoheadrightarrow_\beta K \;\;\Longrightarrow\;\; \exists L:\; N \twoheadrightarrow_\beta L,\;\; K \twoheadrightarrow_\beta L \tag{1.31}$$

Доказательство удобно вести через параллельную редукцию. Нам понадобятся три оператора редукции над термами:

- $\to_\beta$ — обычная, *одношаговая* β-редукция: $M \to_\beta N$ значит, что в M выбрали ровно один редекс и стянули его (один шаг — один редекс);
- $\Rightarrow$ — *параллельная* редукция: $M \Rightarrow N$ значит, что стянули произвольный набор редексов, уже присутствующих в M (нуль, один, два — сколько угодно), каждый не более одного раза, и только имеющиеся, а не их будущие копии;
- $\twoheadrightarrow_\beta$ — *многошаговая* редукция (рефлексивно-транзитивное замыкание $\to_\beta$): $M \twoheadrightarrow_\beta N$ значит, что есть цепочка $M \to_\beta \cdots \to_\beta N$ любой длины, в том числе нулевой.

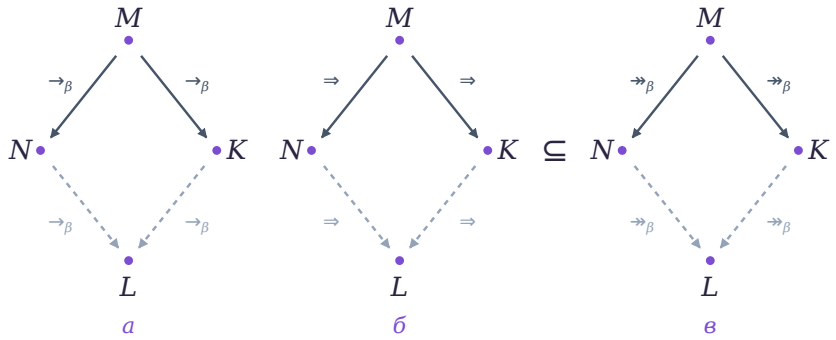

Рис. 1.1. Свойство ромба для трёх операторов редукции: а — одношаговая β-редукция $\to_\beta$; б — параллельная редукция $\Rightarrow$; в — многошаговая редукция $\twoheadrightarrow_\beta$.

Одношаговая редукция — частный случай параллельной, а та — частный случай многошаговой:

$$\to_\beta \subseteq \Rightarrow \subseteq \twoheadrightarrow_\beta \tag{1.32}$$

Почему свойство ромба не доказывается прямо для одношаговой редукции $\to_\beta$ (рис. 1.1, а)? Мешает копирование редексов. Возьмём терм, в котором два редекса — внешний и внутренний:

$$M \equiv (\lambda x.x\,x)\,\big((\lambda y.y)\,z\big)$$

Стянем внешний редекс — внутренний при подстановке удвоится: получим $\big((\lambda y.y)\,z\big)\,\big((\lambda y.y)\,z\big)$. Стянем внутренний — получим $(\lambda x.x\,x)\,z$. Пути разошлись на один шаг, а вот сойтись за один шаг уже не могут: из второго терма $z\,z$ получается сразу, а из первого — только за два шага, по шагу на каждую копию. Значит, одношаговое свойство ромба для $\to_\beta$ попросту ложно.

Параллельная редукция $\Rightarrow$ лечит ровно эту болезнь: размножившиеся копии редекса она стягивает все разом, за один тик. Для неё свойство ромба уже верно (рис. 1.1, б), и доказывается оно приёмом Такахаси — через полную развёртку.

Полной развёрткой M^* назовём терм, в котором одновременно стянуты все редексы, видимые в M, — самый жадный из возможных параллельных шагов. Определение — индукцией по строению терма:

Первое правило: $x^* = x$ — переменная развёртывается в себя.

Второе правило: $(\lambda x.P)^* = \lambda x.P^*$ — развёртка проходит под абстракцию.

Третье правило: $(P\,Q)^* = P^*\,Q^*$, если $P\,Q$ — не редекс (P — не абстракция): части развёртываются независимо.

Четвёртое правило: $\big((\lambda x.P)\,Q\big)^* = P^*[x := Q^*]$ — сам редекс стягивается, а его части ещё и развёртываются.

Развернём терм из контрпримера по этим правилам:

- $\big((\lambda x.x\,x)\,((\lambda y.y)\,z)\big)^* = (x\,x)^*[x := ((\lambda y.y)\,z)^*]$ — весь терм есть редекс, работает четвёртое правило: внешний редекс стянут, обе его части ушли в развёртку;
- $(x\,x)^* = x^*\,x^* = x\,x$ — применение $x\,x$ — не редекс: третье правило, затем первое для каждой переменной;
- $((\lambda y.y)\,z)^* = y^*[y := z^*] = y[y := z] = z$ — внутренний редекс: четвёртое правило, затем дважды первое;
- $(x\,x)[x := z] = z\,z$ — осталось выполнить подстановку.

Итого $M^* = z\,z$: развёртка стянула и внешний редекс, и внутренний — ещё до того, как тот успел размножиться. Всю конструкцию держит лемма о треугольнике.

Лемма о треугольнике. Полную развёртку нельзя обогнать: каким бы ни был параллельный шаг $M \Rightarrow N$, из N ровно один параллельный шаг ведёт в M^*:

$$M \Rightarrow N \implies N \Rightarrow M^*$$

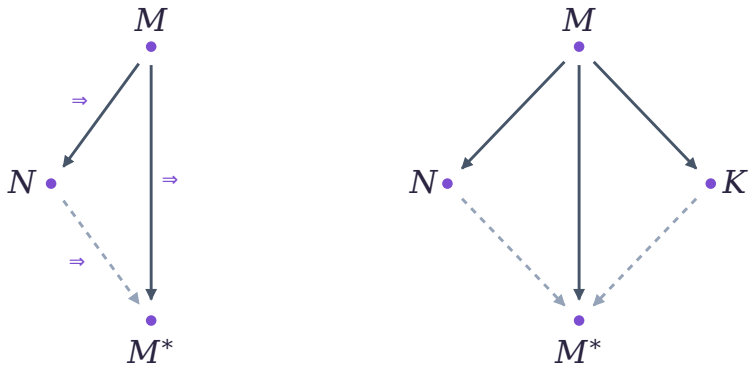

Рис. 1.2. Лемма о треугольнике (слева): любой параллельный шаг $M \Rightarrow N$ достраивается одним шагом до полной развёртки M^*. Справа — свойство ромба для $\Rightarrow$: два треугольника с общей нижней вершиной M^*.

Параллельный шаг не обязан стягивать все редексы — из одного M шаги $\Rightarrow$ ведут в целый веер термов, и полная развёртка — лишь крайняя, самая жадная точка этого веера. Поэтому шаг $N \Rightarrow M^*$ в общем случае настоящий; нулевым он оказывается, только когда всё уже стянуто и $N = M^*$.

Редукция, которая сразу стягивает все видимые редексы одним разом, называется редукцией Гросса — Кнута $M \Rightarrow_{\mathrm{GK}} M^*$. Но она нам неинтересна: в ней нет никакой вариативности — из терма такой шаг ведёт ровно в один результат. А теорема Чёрча — Россера как раз про вариативность: пути $\twoheadrightarrow_\beta$ стягивают редексы в произвольном порядке, и в жёсткие шаги Гросса — Кнута их не разложить. Нужна именно свобода стягивать любое подмножество.

Свойство ромба из треугольника получается очевидно. Пусть $M \Rightarrow N$ и $M \Rightarrow K$. По лемме о треугольнике $N \Rightarrow M^*$ и $K \Rightarrow M^*$ — берём $L = M^*$, и обе стороны замыкаются одним шагом (рис. 1.2, справа). Общая точка одна на всех: она не зависит ни от N, ни от K.

Осталось поднять свойство ромба с одного параллельного шага на многошаговую редукцию (рис. 1.1, в). Вложения (1.32) позволяют читать любую цепочку $\to_\beta$-шагов как цепочку $\Rightarrow$-шагов и наоборот. Поэтому распишем $M \twoheadrightarrow_\beta N$ и $M \twoheadrightarrow_\beta K$ как две цепочки параллельных шагов, выложим их по сторонам сетки и достроим её клетка за клеткой — каждую клетку замыкает свойство ромба для $\Rightarrow$.

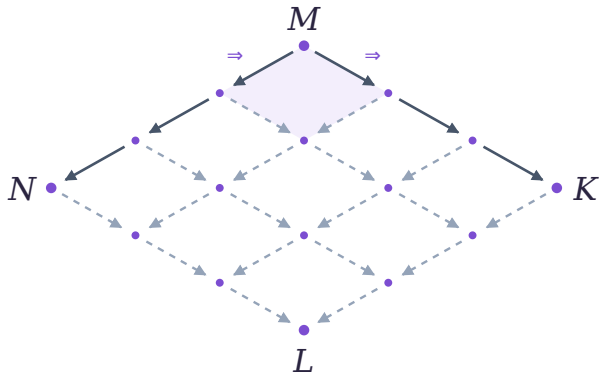

Рис. 1.3. Сетка из клеток-ромбов: стороны — цепочки параллельных шагов $M \Rightarrow \cdots \Rightarrow N$ и $M \Rightarrow \cdots \Rightarrow K$; каждая клетка (например, закрашенная) — свойство ромба для $\Rightarrow$. Нижняя вершина — общий редукт L.

Заполнив всю сетку, в нижней вершине получаем терм L, в который сходятся обе цепочки. Все рёбра сетки — параллельные шаги, а $\Rightarrow \subseteq \twoheadrightarrow_\beta$, поэтому $N \twoheadrightarrow_\beta L$ и $K \twoheadrightarrow_\beta L$. Теорема Чёрча — Россера доказана.

Из теоремы Чёрча — Россера (1.31) сразу вытекает существование общего редукта: если $M =_\beta N$, то найдётся L с $M \twoheadrightarrow_\beta L$ и $N \twoheadrightarrow_\beta L$ (индукция по определению $=_\beta$). Отсюда единственность нормальной формы: у терма не больше одной β-НФ — два пути сошлись бы к общему L, а НФ дальше не редуцируется, поэтому $N_1 \equiv L \equiv N_2$. Далее, редуцируемость к НФ: если N — нормальная форма M, то $M \twoheadrightarrow_\beta N$ (а не просто $M =_\beta N$). Наконец, непротиворечивость: разные нормальные формы не β-равны — например true и false, — иначе нарушилась бы единственность НФ; так доказываются «неравенства» термов.

Порядок стягивания редексов, стало быть, на результат не влияет — только на то, *завершится ли* вычисление и за сколько шагов. Выбором этого порядка и занимаются стратегии редукции.

Стратегии редукции

Правило, по которому из всех редексов терма выбирается очередной, называется стратегией редукции. Разберёмся, где возникает выбор. Терм бывает трёх видов:

- переменная x — редуцировать нечего;
- абстракция $\lambda x.M$ — выбора нет: редуцируем тело M;
- аппликация $M\,N$ — редексы могут быть и в левой части, и в правой; выбор между ними и делает стратегия.

Раскроем аппликацию: её левая часть может снова быть аппликацией, левая часть той — тоже, и так далее. Спустившись по левым ветвям до первой не-аппликации — до головы терма, — получим один из двух видов:

$$(\ldots((x\,N_1)\,N_2)\ldots N_k),$$
$$(\ldots(((\lambda x.M)\,N_1)\,N_2)\ldots N_k).$$

В первом виде голова — переменная x: слева стягивать нечего, редексы могут прятаться только в аргументах N_i. Во втором голова — абстракция, и $(\lambda x.M)\,N_1$ — редекс; вот и развилка: стянуть его сразу или сначала вычислить аргумент. Этот выбор и различает две классические стратегии.

Нормальная стратегия стягивает самый левый внешний редекс — в нашей записи это $(\lambda x.M)\,N_1$: аргументы подставляются в тело невычисленными. *Аппликативная стратегия* поступает наоборот: сперва доводит аргумент редекса до нормальной формы и лишь потом стягивает сам редекс.

Терм удобно изображать деревом. Например, для $((\lambda x.M)\,N_1)\,N_2$ оно устроено так (рис. 1.4): узлы @ — аппликации, узел λx — абстракция. Самый левый внешний редекс на дереве ищется спуском от корня по левым рёбрам: это первый @, у которого левый ребёнок — абстракция.

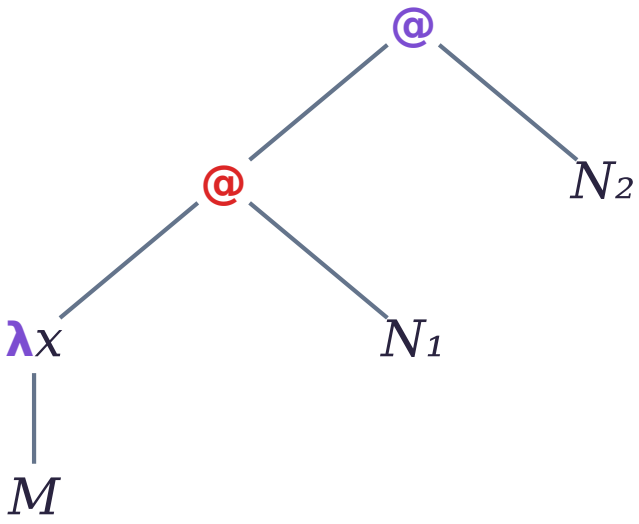

Рис. 1.4. Дерево терма $((\lambda x.M)\,N_1)\,N_2$: узлы @ — аппликации, узел λx — абстракция; красный @ — самый левый внешний редекс $(\lambda x.M)\,N_1$.

Этот же спуск позволяет ввести важное понятие — головную нормальную форму. Поясним его на двух примерах:

$$\lambda x_1 \ldots x_n.\, y\, N_1 \ldots N_k, \quad n, k \geq 0,$$
$$\lambda x_1 \ldots x_n.\, (\lambda z.M)\, N_1 \ldots N_k, \quad n \geq 0,\ k > 0.$$

Первый терм — головная нормальная форма: спуск по левым ветвям упирается в переменную y, она называется головной переменной. Голова такого терма уже не изменится, редексы могут прятаться лишь в аргументах N_i. Второй терм головной нормальной формой не является: его голова — редекс $(\lambda z.M)\,N_1$, он называется головным редексом.

Сформулируем обе стратегии точно — правилами операционной семантики. Мало нам понятий — надо ещё: понадобятся три синтаксические категории. Первая — не-абстракция NA:

$$NA ::= x \mid M\,N.$$

Тут, кажется, всё понятно: это любой терм, только не абстракция. M и N — произвольные термы, так что не-абстракция вполне может быть редексом — например, $(\lambda y.y)\,z$. Две другие категории описывают готовые результаты: нормальные формы NF и их подкласс — нормальные формы-не-абстракции $NANF$:

$$NF ::= \lambda x.NF \mid NANF,$$
$$NANF ::= x \mid NANF\ \ NF.$$

Разделение необходимо: без него была бы возможна аппликация, у которой слева стоит абстракция, — а такой терм уже не нормальная форма, это редекс. Примеры NF: $\lambda x.\,x$, $\lambda f.\lambda x.\,f\,x$. Примеры $NANF$: x, $f\,x$, $x\,(\lambda y.y)\,z$ — причём каждый из них одновременно и NF.

Теперь наконец-то стратегии. В основе каждой лежит своё правило стягивания редекса. Нормальная стратегия использует обычную β-редукцию: редекс стягивается сразу, аргумент N — произвольный терм, он подставляется в тело невычисленным:

$$(\lambda x.M)\,N \to M[x := N] \tag{1.33}$$

Аппликативная стратегия стягивает редекс иначе — лишь когда аргумент уже вычислен, то есть доведён до нормальной формы — слева от стрелки стоит NF, и пока аргумент не таков, стягивание запрещено:

$$(\lambda x.M)\,NF \to M[x := NF] \tag{1.34}$$

Так как мы не можем стянуть редекс, пока аргумент не в нормальной форме, в аппликативной стратегии есть правило для аргумента:

$$\frac{N \to N'}{(\lambda x.M)\,N \to (\lambda x.M)\,N'} \tag{1.35}$$

Оставшиеся правила — общие для нормальной стратегии и аппликативной:

$$\frac{NA \to NA'}{NA\,N \to NA'\,N}, \quad \frac{N \to N'}{NANF\,N \to NANF\,N'}, \quad \frac{M \to M'}{\lambda x.M \to \lambda x.M'} \tag{1.36}$$

Первое правило редуцирует левую часть аппликации, пока та остаётся неабстракцией; второе переходит к аргументу, лишь когда слева уже стоит $NANF$; третье заводит редукцию в тело абстракции. Категории NA и $NANF$ в посылках как раз и гарантируют порядок «самый левый внешний»: пока слева есть что стягивать, вправо не смотрим.

Выбор самого левого внешнего редекса неслучаен: теорема о нормализации (Карри) гарантирует, что если у терма есть нормальная форма, то последовательное стягивание именно такого редекса её достигнет. За это нормальную стратегию называют полной.

Полнота не бесплатна: аргумент, подставленный в несколько мест, приходится пересчитывать. Для «большого» N: $(\lambda x.\,F\,x\,(G\,x)\,x)\,N \to_\beta F\,N\,(G\,N)\,N$ — здесь N придётся редуцировать трижды — по разу на копию. Аппликативная стратегия вычислит N ровно один раз, но платит за это полнотой. Пусть $\mathrm{K} \equiv \lambda xy.x$, $\mathrm{I} \equiv \lambda x.x$, а $\Omega \equiv (\lambda x.x\,x)\,(\lambda x.x\,x)$ — терм без нормальной формы. Нормальная стратегия вычисляет $\mathrm{K\,I}\,\Omega \twoheadrightarrow_\beta \mathrm{I}$, отбросив Ω не глядя; аппликативная же сперва берётся за аргумент Ω — и зацикливается.

В языках программирования обе стратегии живут под другими именами. Аппликативная — это вызов по значению (call-by-value) строгих языков вроде Си: сначала аргументы, потом применение. Нормальная — основа ленивых вроде Haskell: вызов по имени (call-by-name).

Полнота по Тьюрингу

Полнота по Тьюрингу — это наличие конструкций ветвления, повтора и данных. Все ингредиенты уже собраны в предыдущих главах:

- данные — кодирование Чёрча (раздел «Кодирование Чёрча»): булевы значения, пары, нумералы, списки;

- ветвление — булево значение Чёрча само выбирает ветку: $\mathrm{if} \equiv \lambda b\, t\, e.\, b\, t\, e$;
- циклы — комбинатор неподвижной точки (раздел «Комбинатор Карри»): $\mathrm{Y}\, F =_\beta F\, (\mathrm{Y}\, F)$, рекурсия без имён и без счётчиков.

Этого хватает, чтобы смоделировать машину Тьюринга: лента — список, состояние — нумерал, шаг — ветвление, повторение шагов — Y. У полноты по Тьюрингу есть цена — неразрешимость. Нет алгоритма, который по терму определяет, есть ли у него нормальная форма: это λ-аналог проблемы останова.

Теперь про тотальность. Язык называется тотальным, если каждая его программа завершается на каждом входе. Полнота по Тьюрингу с тотальностью несовместима. В λ-исчислении возможны бесконечные циклы:

$$\Omega \equiv (\lambda x.x\, x)\, (\lambda x.x\, x) \to_\beta \Omega \to_\beta \Omega \to_\beta \ldots$$

Языки, выбравшие тотальность, существуют — например, Agda. Компилятор Agda проверяет каждую функцию: все случаи входа должны быть разобраны, а рекурсивные вызовы — идти по структурно меньшим аргументам. Сложение натуральных чисел проверку проходит — рекурсивный вызов идёт на n — строго меньшую часть `suc` n:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)   -- вызов на n: строго меньше, чем suc n
```

А вот это Agda отвергнет:

```
loop : ℕ → ℕ
loop n = loop n           -- Termination checking failed: n не убывает
```

Плата симметрична: Agda не Тьюринг-полна — Ω в ней не написать. Зато каждая программа — тотальная функция, и по соответствию Карри — Ховарда (глава «Соответствие Карри — Ховарда») её можно читать как доказательство. Haskell выбирает середину: типизированное ядро плюс общая рекурсия (`fix`, рекурсивный let) — полнота возвращается, а с ней и вечные циклы.

Соответствие Карри — Ховарда

Вот мы и добрались до самого важного. Программа на языке, взявшем за основу λ-исчисление и решившем проблему зацикливания, — это теорема: её формулировка — тип, компиляция — проверка доказательства, а исполнение гарантированно возвращает результат обещанного типа. С императивными языками такое тоже возможно — корректность программ доказывают со времён логики Хоара, — но куда сложнее: там доказательство не сама программа, а отдельная надстройка из предусловий, постусловий и инвариантов.

Итак, программы, написанные на языках, базирующихся на λ-исчислении, проще доказывать, чем программы на других языках — не обязательно императивных, скажем, на Python. В этой главе мы рассмотрим соответствие Карри — Ховарда, мост между логикой и программированием: написать программу типа τ — то же самое, что доказать высказывание τ. Заметил его Карри ещё в 1930-х — по типам комбинаторов, — а точную форму оно получило в рукописи Ховарда 1969 года.

Присмотримся к типам знакомых термов, читая стрелку как импликацию «из A следует B». Тождественная функция оказывается доказательством тавтологии «из A следует A»:

$$\lambda x.\, x \;:\; A \to A$$

Комбинатор K, отбрасывающий второй аргумент, — доказательство высказывания «если верно A, то A следует из чего угодно»:

$$\lambda x\, y.\, x \;:\; A \to (B \to A)$$

Комбинатор S тоже доказывает теорему — про раздачу посылки двум следствиям:

$$\lambda x\, y\, z.\, x\, z\, (y\, z) \;:\; (A \to B \to C) \to (A \to B) \to (A \to C)$$

Приведём, пожалуй, таблицу соответствия:

Логика	Типы / программы
импликация $A \Rightarrow B$	тип функции $A \to B$
конъюнкция $A \wedge B$	пара $(A,\, B)$
дизъюнкция $A \vee B$	`Either` $A\ B$
истина $\top$	единичный тип $()$
ложь $\bot$	пустой тип `Void`
доказательство	терм (программа)
упрощение доказательства	β-редукция
$\forall$ второго порядка	полиморфизм System F
$\forall$, $\exists$ над объектами	зависимые типы (LiquidHaskell)

Немного пояснений. Единичный тип $()$ — тип ровно с одним значением. Построить его значение можно всегда и тривиально, поэтому $\top \leftrightarrow ()$. А вот пустой тип `Void` — тип без единого значения: программу такого типа написать невозможно. Это и есть ложь: у $\bot$ нет доказательств.

«$\forall$ второго порядка $\to$ полиморфизм System F» (мы это ещё обсудим дальше). Квантор второго порядка пробегает по самим высказываниям (в мире типов — по типам): «для любого высказывания A верно $\ldots$». В программах это в точности параметрический полиморфизм: тип тождественной функции $\forall a.\ a \to a$ читается как теорема «для любого A: из A следует A». System F — это λ-исчисление, где такая квантификация по типам встроена в язык; на нём основан полиморфизм Haskell.

«$\forall$, $\exists$ над объектами $\to$ зависимые типы». Кванторы первого порядка пробегают уже не по высказываниям, а по объектам — конкретным значениям: «для любого числа n $\ldots$», «существует список x, такой что $\ldots$». Чтобы записать такое типом, тип должен зависеть от значения — это и есть зависимые типы. В самом Haskell $\forall$ есть только по типам, зато LiquidHaskell навешивает на типы логические предикаты — уточняющие типы: сигнатура $\texttt{head}' : \{\, v : [a] \mid \texttt{len}\ v > 0 \,\} \to a$ читается

как «для любого списка v, длина которого больше нуля …», а импликации между предикатами проверяет SMT-решатель. Квантор $\exists$ выражается уточнением результата: $\{\, v : \texttt{Int} \mid \texttt{even}\ v \,\}$ — предъявленное чётное число вместе с гарантией чётности. Подробнее — в главе «LiquidHaskell».

Мы возвращаемся к проблеме полноты по Тьюрингу. Неограниченная рекурсия $\texttt{fix} : (A \to A) \to A$ в логическом прочтении — «если из A следует A, то A верно»: порочный круг, доказывающий что угодно. Поэтому языки-доказатели обязаны быть тотальными: проверка тотальности из прошлой главы — не перестраховка, а условие честности логики. На этом соответствии стоит пруф-ассистент Agda: спецификация записывается типом, программа этого типа и есть доказательство, а проверка доказательства — обычная проверка типов.

Типизированные λ-исчисления

Типы — это, в парадигме λ-исчисления, гарантия конечности времени работы программы. Плата — потеря полноты по Тьюрингу; выигрыш — предсказуемость, документация и, как мы уже видели, целая логика.

Простейшая такая система — просто типизированное λ-исчисление ($\lambda^{\to}$). Каждый терм в ней получает тип, а сами типы строятся из атомарных α единственной конструкцией — стрелкой:

$$\tau \ ::= \ \alpha \ \mid \ \tau \to \tau \tag{1.37}$$

Три правила вывода — по одному на конструкцию терма:

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma,\ x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.\, M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N : \tau} \tag{1.38}$$

Слева направо: переменная берёт свой тип из контекста Γ; абстракция — если в предположении $x{:}\sigma$ тело M имеет тип τ, то $\lambda x.\, M$ — функция типа $\sigma \to \tau$; аппликация — функцию $\sigma \to \tau$ можно применить только к аргументу типа σ, и результат получает тип τ.

Есть два важных свойства: первое — сильная нормализация: всякий типизируемый терм имеет НФ, и любой порядок редукции её достигает — как следствие, Ω и Y нетипизируемы, а язык не полон по Тьюрингу; второе — subject reduction: тип сохраняется при редукции — «вычисление не портит тип».

А что насчёт полиморфизма? Причём по типам. В $\lambda^{\to}$ его нет: тождественная функция $\lambda x{:}\alpha.\, x$ привязана к одному конкретному типу α, и для каждого нового типа её приходится писать заново. Жирар и Рейнольдс устранили это ограничение, разрешив термам абстрагироваться не только по значениям, но и по самим типам, — получилась System F, для которой справедливо следующее:

$$\mathrm{id} = \Lambda\alpha.\, \lambda x{:}\alpha.\, x \ : \ \forall\alpha.\, \alpha \to \alpha$$

Здесь $\Lambda\alpha$ — абстракция по типу: id принимает сначала тип α, затем значение x этого типа и возвращает его же. Тип $\forall\alpha.\, \alpha \to \alpha$ читается «для любого типа α — функция из α в α»: одна и та же id применима к числу, булеву значению, функции — полиморфизм стал частью самого исчисления.

Вывод типов в полной System F неразрешим, поэтому Haskell использует разрешимый фрагмент — систему Хиндли — Милнера (алгоритм W), где все типы выводятся без аннотаций; ей посвящена следующая глава.

Система Хиндли — Милнера

В предыдущей главе мы упёрлись в дилемму: либо пишем все типы руками (надёжно, но занудно), либо просим машину догадаться саму (удобно, но в общем случае неразрешимо). Система Хиндли — Милнера (сокращённо HM) — золотая середина: компилятор выводит самый общий тип любого выражения вообще без аннотаций.

Как удержать полиморфизм типов и не свалиться обратно в неразрешимость System F? Хиндли и Милнер решают это ограничением мест, где может стоять квантор, разнося типы на два этажа: внизу — монотипы τ — типы без кванторов: атомарные α, применения конструкторов типов $C\,\tau_1 \ldots \tau_n$ (так строятся, например, списки и пары) и стрелки. Наверху — схемы (политипы) σ — монотипы, накрытые сверху кванторами:

$$\tau ::= \alpha \mid C\,\tau_1 \ldots \tau_n \mid \tau \to \tau, \qquad \sigma ::= \tau \mid \forall\alpha.\,\sigma \tag{1.39}$$

Кванторы разрешены только снаружи, в самом начале типа (пренексная форма): $\forall\alpha.\,\alpha \to \alpha$ — законная схема, а $(\forall\alpha.\,\alpha \to \alpha) \to \beta$ — уже нет: здесь квантор забрался внутрь, слева от стрелки. Разрешить такое — получить полную System F с неразрешимым выводом; запретить — потерять немного выразительности, но превратить вывод типов в алгоритм. Весь фокус HM — в этом ограничении.

Между этажами два перехода, рассмотрим оба. Вниз — инстанцирование (Inst): из схемы делают монотип $\sigma[\alpha := \tau]$, подставляя вместо связанной переменной любой монотип. Вверх — обобщение (Gen): если переменная α осталась в выведенном типе, но не встречается свободно ни в одном предположении контекста ($\alpha \notin \mathrm{free}(\Gamma)$), то ничто снаружи её не ограничивает — на неё можно навесить квантор:

$$\frac{\Gamma \vdash e : \forall\alpha.\,\sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]}\ (\text{Inst}), \qquad \frac{\Gamma \vdash e : \sigma \quad \alpha \notin \mathrm{free}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\,\sigma}\ (\text{Gen}) \tag{1.40}$$

Зачем эти переходы, видно из правила для **let** — сердца всей системы. Связывая имя x значением e_0, мы обобщаем его тип до схемы σ, а дальше каждое использование x в теле e_1 инстанцирует эту схему заново и независимо от остальных:

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x{:}\sigma \vdash e_1 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 : \tau}\ (\text{Let}) \tag{1.41}$$

У аргумента λ такой привилегии нет. Обобщение происходит только в правиле Let, а параметр функции в него не попадает: когда мы типизируем тело, функция ещё ни к чему не применена, и что придёт ей на вход — неизвестно; всё, что можно сделать, — выдать аргументу одну типовую переменную, общую для всех его использований в теле. Разница видна на двух почти одинаковых термах. Терм $\mathbf{let}\ id = \lambda y.\,y\ \mathbf{in}\ (id\ \overline{3},\ id\ \mathbf{true})$ типизируется: id связан через **let**, обобщён до схемы $\forall\alpha.\,\alpha \to \alpha$, и каждое из двух использований инстанцирует её своим типом. Терм $(\lambda id.\,(id\ \overline{3},\ id\ \mathbf{true}))\,(\lambda y.\,y)$ — тот же по смыслу — уже не типизируется: здесь id — аргумент λ, схемы у него нет, а один монотип не может быть одновременно типом функции над числами и над булевыми значениями.

В Haskell аннотации типов, конечно, есть — сигнатуры можно (и принято) выписывать. Но они необязательны: даже если бы их не было вовсе, компилятор восстановил бы все типы сам — это делает алгоритм W (Дамас — Милнер). Он обходит терм, выдаёт каждому подвыражению свежую типовую переменную, а на каждой аппликации записывает уравнение: тип функции обязан быть стрелкой из типа аргумента в тип результата. Накопленные уравнения решает унификация Робинсона — механическое сопоставление двух типов: чтобы $\alpha \to \beta$ совпало с $(\gamma \to \gamma) \to \delta$, достаточно подстановки $\alpha := \gamma \to \gamma$, $\beta := \delta$. Компилятору хватает голого кода:

```
compose f g x = f (g x)

-- вывод без единой подсказки:
-- compose :: (b -> c) -> (a -> b) -> a -> c
```

Теперь мы понимаем, почему не получается использовать комбинаторы Y, Θ и Z в языке Haskell. Все они построены на самоаппликации $x\,x$ — попробуем вывести её тип. Пусть $x : \alpha$; раз x применяется к самому себе, α обязан быть стрелкой из α в какой-то β — получаем уравнение $\alpha = \alpha \to \beta$. Подставим правую часть вместо α — и α снова внутри: $(\alpha \to \beta) \to \beta$, $((\alpha \to \beta) \to \beta) \to \beta$, … — подстановка зацикливается, тип растёт до бесконечности. Именно это пресекает occurs check — запрет переменной встречаться в типе, который за неё подставляют: без рекурсивных типов у уравнения решения нет. Поэтому рекурсию в Haskell заводят примитивом `fix` и рекурсивными определениями.

Глава 2

Концепты Haskell

Классы типов и роды

Все абстракции этого раздела (моноид, функтор, монада) оформлены в Haskell одним механизмом — классом типов: перегружаемый интерфейс и законы, которым обязаны подчиняться реализации. Класс с законами — это алгебраическая структура, а не просто интерфейс, как, скажем, в TypeScript.

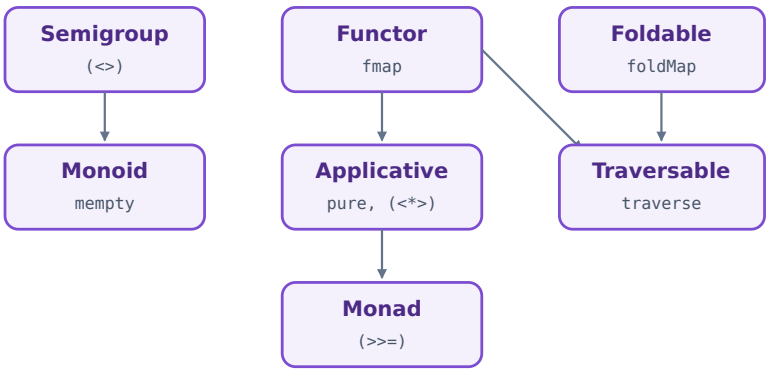

Рис. 2.1. Классы раздела и их иерархия: стрелка ведёт от суперкласса к подклассу; Traversable требует и Functor, и Foldable.

В качестве примера рассмотрим класс типов Functor: объявление class задаёт интерфейс — сигнатуру единственного метода `fmap`, а объявление instance реализует этот интерфейс для конкретного типа — здесь для `Maybe`.

```
class Functor f where
  -- fmap применяет обычную функцию внутри контекста f
  fmap :: (a -> b) -> f a -> f b

instance Functor Maybe where
  -- Если значение есть, применяем функцию к нему
  fmap g (Just x) = Just (g x)
  -- Если значения нет, контекст Nothing сохраняется
  fmap g Nothing = Nothing
```

Вызов метода класса диспетчеризуется по типу: компилятор видит, что `fmap` применяется к значению типа `Maybe`, и подставляет экземпляр выше. Причём класс типов умеет диспетчеризоваться и по возвращаемому типу: например, `mempty` — нейтральный элемент из класса Monoid — аргументов не имеет вовсе и «узнаёт» свой тип из контекста вызова. Законы классов компилятор не проверяет — это контракт на совести автора экземпляра: класс можно инстанцировать неверно — скажем, написать `fmap`, который теряет элементы контейнера, — компилятор такой экземпляр примет, но код, полагающийся на законы, начнёт выдавать неверные результаты. И ещё одно правило: на один тип приходится не больше одного экземпляра класса (когерентность).

Теперь про род: он говорит, скольких параметров конструктору не хватает до обычного типа со значениями.

```
Int :: *
-- Int - обычный тип: у него есть значения

Maybe :: * -> *
-- Maybe - конструктор типов с одним параметром

Either :: * -> * -> *
-- Either - конструктор типов с двумя параметрами
```

Род $*$ (он же `Type`) — род обычных типов, у которых есть значения. `Maybe` — не тип, а конструктор типов: функция на уровне типов, которая принимает тип (скажем, `Int`) и возвращает тип (`Maybe Int`). Классы различаются родом параметра: классу `Eq` — сравнению на равенство — нужен обычный тип $*$, а классу `Functor` — конструктор $* \to *$: функтором бывает контейнер, но не значение. Частичное применение работает и на уровне типов: `Either` имеет род $* \to * \to *$, но `Either e` :: $* \to *$ — уже законный функтор.

Моноиды

Эта главейка — блёклая тень лекций мисс-моноид, но мы провернём всё поколхозному. Моноид — это абстракция, для которой заданы ассоциативная операция и нейтральный элемент. Ассоциативность разрешает разбивать работу на части и группировать результаты разными способами, не меняя порядок данных, — отсюда параллельная агрегация (map-reduce) и свёртки.

В Haskell моноид задаётся парой классов: Semigroup даёт ассоциативную операцию, а Monoid добавляет к ней нейтральный элемент:

```
class Semigroup a where
  (<>) :: a -> a -> a          -- ассоциативная операция

class Semigroup a => Monoid a where
  mempty :: a                  -- нейтральный элемент
```

Экземпляры обязаны соблюдать два закона — ассоциативность операции и нейтральность `mempty`:

```
(x <> y) <> z == x <> (y <> z)   -- ассоциативность

mempty <> x == x                -- нейтральный элемент
x <> mempty == x                -- с обеих сторон
```

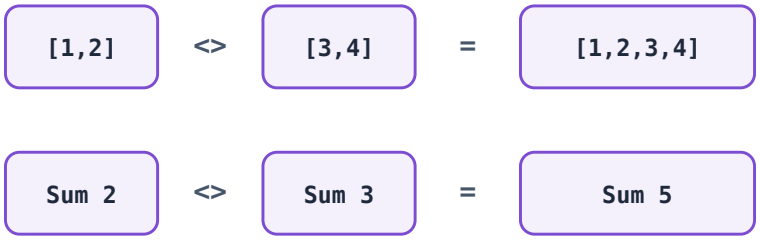

Рис. 2.2. Комбинирование в моноиде: для списков (<>) — конкатенация, для `Sum` — сложение.

Экземпляров в стандартной библиотеке множество:

```
-- списки: (<>) = (++), mempty = []
[1, 2] <> [3]                   -- [1, 2, 3]

-- числа - моноид двумя способами, поэтому нужны обёртки:
Sum 2     <> Sum 3              -- Sum 5      (mempty = Sum 0)
Product 2 <> Product 3          -- Product 6  (mempty = Product 1)

-- булевы значения:
Any True <> Any False           -- Any True   (||, mempty = Any False)
All True <> All False           -- All False  (&&, mempty = All True)

-- минимум и максимум:
Min 5 <> Min 2                  -- Min 2
Max 5 <> Max 2                  -- Max 5
```

Теперь пару слов о reduce: `mconcat` — его моноидный вариант. Он сворачивает список моноидных значений, ничего не зная о конкретном типе:

```
mconcat :: Monoid a => [a] -> a
mconcat = foldr (<>) mempty

mconcat [[1], [2, 3], []]       -- [1, 2, 3]
```

Законы Monoid позволяют безопасно разбить данные на части, обработать части параллельно, а затем собрать частичные результаты через (<>), сохранив исходный порядок.

Например, каждый рабочий поток может независимо накопить свой кусок лога, после чего куски склеиваются как списки. Точно так же можно параллельно посчитать суммы отдельных порций данных, а затем сложить частичные суммы:

```
-- логи трёх независимых рабочих потоков
workerLogs = ["worker 1: done\n", "worker 2: done\n", "worker 3: done\n"]
fullLog = mconcat workerLogs

-- суммы, независимо посчитанные на разных порциях данных
partialSums = [Sum 120, Sum 95, Sum 143]
total = getSum (mconcat partialSums)  -- 358
```

Ассоциативность гарантирует, что результат не зависит от группировки частичных результатов: слева направо, попарно деревом или по числу доступных ядер.

Функторы

Представим список натуральных чисел и список строк. Типы элементов различаются, но устройство списка и операции над его структурой остаются общими. Обозначим тип элемента как `T`: конструктор `List` принимает его и создаёт конкретный тип `List T`. Поэтому род `List` — $* \to *$.

Любитель C++ справедливо заметит, что шаблоны `std::vector`$\langle$`T`$\rangle$ и `std::list`$\langle$`T`$\rangle$ уже позволяют абстрагироваться от типа элемента, а итераторы и ranges — применять одни и те же алгоритмы к разным структурам, например к списку и дереву. Однако такие алгоритмы обобщаются по интерфейсу обхода готового объекта, а не по конструктору типа, сохраняющему форму контекста.

Haskell делает следующий шаг. И `List`, и `Tree` имеют род $* \to *$, поэтому класс типов `Functor` может абстрагироваться над самим конструктором `f`. Операция `fmap` принимает функцию `a -> b`, преобразует значения внутри `f a` и возвращает `f b`, сохраняя форму контекста. Это полиморфизм не только по типу элемента, но и по конструктору контекста. В C++ его можно имитировать шаблонами и перегрузками, однако прямого стандартного аналога классов типов и единообразного полиморфизма высших родов в языке нет. Сам класс типов объявляется так:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b   -- применить функцию внутри контекста f
```

В этом объявлении `f` — переменная не конкретного типа, а конструктора типа рода $* \to *$. Класс задаёт общую форму операции `fmap`, а способ сохранения контекста определяет каждый экземпляр. Для списка и `Maybe` стандартные экземпляры устроены так:

```
instance Functor [] where
  fmap _ []       = []
  fmap g (x : xs) = g x : fmap g xs

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just x) = Just (g x)
```

Как видим, реализация `Functor` зависит от конструктора контекста, но не от конкретного типа его элементов. Теперь рассмотрим число 2 в некотором контексте. Начнём с ключевого понятия — значения в контексте:

Рис. 2.3. Обычное значение и значение в контексте `Maybe`: «коробка» может содержать значение (`Just 2`) или быть пустой (`Nothing`).

Контекст позволяет один раз определить правила работы с особыми случаями и затем применять обычные функции, не размазывая проверки по алгоритму. Более мощные абстракции, построенные поверх этой идеи, позволяют связывать вычисления в цепочки и задавать стратегии их выполнения, но сам `Functor` гарантирует только преобразование через `fmap`. Допустим, вычисление в JavaScript вернуло не значение, а `undefined`. Тогда приходится писать `if`; если передать результат дальше, проверки начинают расползаться по всему алгоритму. В Haskell отсутствие представляют явно с помощью `Maybe`: экземпляр `Functor` оставляет `Nothing` без изменений, а к значению внутри `Just` применяет функцию. Точно так же отображение по пустому списку возвращает пустой список. К обычному значению функцию можно применить напрямую; значение в контексте сначала пришлось бы разбирать вручную. `fmap` берёт эту работу на себя:

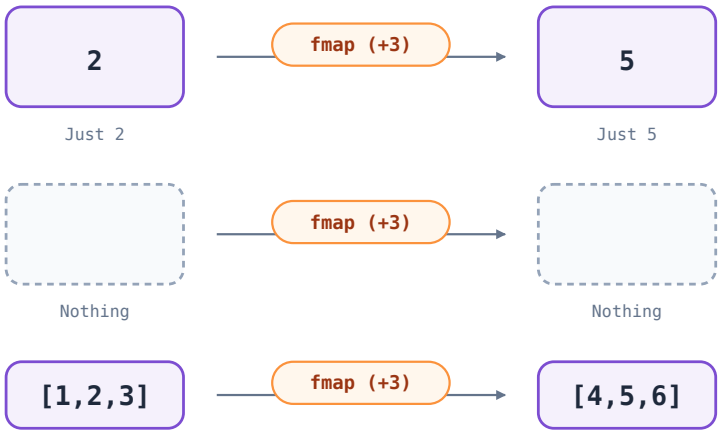

Рис. 2.4. `fmap` применяет функцию к значению внутри контекста и сохраняет сам контекст: пустая коробка остаётся пустой, а список отображается поэлементно.

```
fmap (+3) (Just 2)    -- Just 5
fmap (+3) Nothing     -- Nothing
```

У `fmap` есть инфиксный синоним: $g <\$> x = \texttt{fmap}\ g\ x$.

Экземпляры должны соблюдать два закона — тождество и композицию:

```
fmap id == id                      -- тождество
fmap (g . h) == fmap g . fmap h    -- композиция
```

Эти законы означают, что `fmap` меняет только значения, но не форму контекста: не перестраивает список и не превращает `Just` в `Nothing`. Кроме того, последовательное отображение двух функций эквивалентно отображению их композиции.

Главные экземпляры:

```
-- Maybe - контекст «значение может отсутствовать»
fmap (+3) (Just 2)       -- Just 5

-- список - контекст «ноль или несколько значений»: fmap = map
fmap (+3) [1, 2, 3]      -- [4, 5, 6]

-- Either e - отображается только Right
fmap (+3) (Right 2)      -- Right 5
fmap (+3) (Left "oops")  -- Left "oops"
```

Функтор — отображение между категориями, сохраняющее `id` и композицию. В принятой для Haskell модели функторы являются эндофункторами категории типов и функций, а законы `fmap` в точности соответствуют аксиомам функтора.

Аппликативные функторы

А что, если нужно применить функцию с двумя аргументами к значениям из двух списков? Программист C++ напишет функцию с двумя аргументами или лямбду и передаст её выбранному алгоритму обхода. В Haskell операция $(+)$ каррирована: получив один аргумент, она возвращает функцию, которая ожидает второй. Если отобразить $(+)$ по первому списку, мы получим не суммы, а список частично применённых функций: `fmap` $(+)$ $[1, 2, 3]$ имеет тип `Num a` $\Rightarrow$ `[a` $\to$ `a]`. Иными словами, функции оказались внутри контекста списка.

Обычный `Functor` умеет применять функцию `a` $\to$ `b` к значениям в `f a`, но не умеет применять функции из `f` (`a` $\to$ `b`) к значениям из другого `f a`. Для этого нужен более сильный класс типов — `Applicative` с оператором $(<*>)$.

Класс `Applicative` расширяет `Functor` двумя операциями:

```
class Functor f => Applicative f where
  pure  :: a -> f a                  -- завернуть обычное значение
  (<*>) :: f (a -> b) -> f a -> f b  -- применить функцию из контекста
```

В этом объявлении `f` — конструктор типа рода $* \to *$. Операция `pure` помещает обычное значение в контекст, а $(<*>)$ применяет функцию в контексте к значению в том же контексте. Конкретную семантику этих операций задаёт каждый экземпляр. Для списка и `Maybe` стандартные экземпляры устроены так:

```
instance Applicative [] where
  pure x    = [x]
  fs <*> xs = [g x | g <- fs, x <- xs]

instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just g  <*> x = fmap g x
```

Рис. 2.5. Аппликативное применение: функция и значение находятся в контексте; $(< * >)$ применяет первую ко второму и возвращает результат в том же контексте.

```
Just (+3) <*> Just 2   -- Just 5
```

Важное для аппликативных функторов понятие — лифтинг: обычную функцию нескольких аргументов «поднимают» в контекст, чтобы она принимала аргументы в контекстах и возвращала результат в том же контексте. `liftA2` поднимает функцию с двумя аргументами, а `liftA3` — функцию с тремя аргументами:

```
liftA2 (+)
  (Just 2)
  (Just 3)
-- Just 5

liftA3 (\x y z -> x + y + z)
  (Just 1)
  (Just 2)
  (Just 3)
-- Just 6
```

Законов четыре, и их соблюдение лежит на плечах программиста: можно написать реализацию, которая успешно проходит проверку типов, но нарушает законы. Поэтому после определения экземпляра необходимо убедиться, что выполнены все четыре условия:

```
pure id <*> v == v                            -- тождество

pure g <*> pure x == pure (g x)               -- гомоморфизм (отображение, сохраняющее структуру)
u <*> pure y == pure ($ y) <*> u              -- обмен
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)  -- композиция
```

Законы гарантируют, что `pure` не добавляет лишнего эффекта, применение в контексте согласовано с обычным применением функций, а изменение группировки цепочки $(< * >)$ не меняет результата. При этом они не разрешают произвольно переставлять эффекты: порядок может оставаться существенным.

Парочка примеров со списками, пожалуй:

```
-- каждая функция применяется к каждому значению
[(+1), (*2)] <*> [10, 20]
-- [11, 21, 20, 40]

-- частично применённое сложение: также все сочетания
(+) <$> [1, 2] <*> [10, 20]
-- [11, 21, 12, 22]
```

Монады

Пусть у нас есть задача: функция принимает целое число и делит его на 2. Если результат нечётный, функция не возвращает значения. Если результат чётный, она делит его на 2 ещё раз, после чего умножает на 10, если новый результат чётный, и на 5, если нечётный. Сначала напишем такую функцию на TypeScript.

```
function processNumber(value: number): number | undefined {
  const halved = Math.floor(value / 2);

  if (halved % 2 !== 0) {
    return undefined;
  }

  const halvedAgain = halved / 2;

  return halvedAgain % 2 === 0
    ? halvedAgain * 10
    : halvedAgain * 5;
}
```

Теперь перепишем этот код на Haskell в лоб. Сразу скажу, что на Haskell так не делают.

```
processNumber :: Int -> Maybe Int
processNumber x =
  let halved = x `div` 2
  in case even halved of
    False -> Nothing
    True  ->
      let halvedAgain = halved `div` 2
      in case even halvedAgain of
        True  -> Just (halvedAgain * 10)
        False -> Just (halvedAgain * 5)
```

В приведённом выше коде явно плохо смотрится вложенность. В `TypeScript` её нет, но, чтобы понять, что возвращает функция, приходится внимательно следить за `return`. При этом вложенность однозначно показывает направление вычислений. В императивном языке оно и так понятно: код выполняется сверху вниз. Давайте перепишем код, разделив вычисление на две функции:

```
halveEven :: Int -> Maybe Int
halveEven x =
  let halved = x `div` 2
  in case even halved of
    False -> Nothing
    True  -> Just halved

halveAndScale :: Int -> Maybe Int
```

```
halveAndScale x =
  let halved = x `div` 2
  in case even halved of
    True  -> Just (halved * 10)
    False -> Just (halved * 5)

processNumber :: Int -> Maybe Int
processNumber x =
  case halveEven x of
    Nothing     -> Nothing
    Just halved -> halveAndScale halved
```

Кажется, стало только хуже, но на самом деле это не так. В Haskell для этого уже есть готовый инструмент — класс `Monad` с его оператором связывания (`bind`):

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

processNumber :: Int -> Maybe Int
processNumber x = pure x >>= halveEven >>= halveAndScale
```

Красиво? То-то же. Существует ещё так называемая `do`-нотация. Вот тот же пример с её использованием:

```
processNumber :: Int -> Maybe Int
processNumber x = do
  halved <- halveEven x
  result <- halveAndScale halved
  pure result
```

Ничего не напоминает? Запись действительно похожа на программу на императивном языке, хотя её семантика остаётся монадической. Монада позволяет явно управлять последовательностью зависимых вычислений — в этом её главная особенность. И делает она это красиво.

Аналогично функторам, у монад есть свои законы, соблюдение которых ложится на плечи программиста:

```
pure a >>= f  ==  f a                              -- левая единица
m >>= pure    ==  m                                -- правая единица
(m >>= f) >>= g  ==  m >>= (\x -> f x >>= g)     -- ассоциативность
```

С оператором связывания разобрались. Теперь посмотрим, какие монады встречаются в реальном коде. Почти у каждого привычного «императивного» приёма — исключений, общего конфига, лога, изменяемого состояния или ввода-вывода — есть чистый монадный двойник.

Монада	Контекст / эффект	Императивный аналог
`Maybe`	значения может не быть	null + ранний выход
`Either e`	ошибка с причиной	исключения
`[]`	недетерминизм: ноль или несколько результатов	вложенные циклы
`Reader r`	общее окружение только на чтение	конфиг
`Writer w`	накопление лога	логирование
`State s`	изменяемое состояние	переменные
`IO`	внешний мир	системные вызовы: файлы, сеть, консоль

На первый взгляд эти монады решают совсем разные задачи. Но Reader, Writer и State устроены очень похоже — это обёртки над функциями или значениями с дополнительным контекстом:

```
newtype Reader r a = Reader (r -> a)        -- окружение -> значение
newtype Writer w a = Writer (a, w)          -- значение + лог
newtype State  s a = State  (s -> (a, s))   -- старое состояние -> (значение, новое)
```

Оператор связывания монады State сам протаскивает состояние сквозь цепочку — вручную писать $s_1, s_2, s_3 \ldots$ больше не нужно. Примитивы: `get`::`State` s s, `put`::$s \to$ `State` s ().

Любая исполняемая программа на Haskell работает внутри монады `IO`: её точка входа имеет тип `main` :: `IO` (), а всё взаимодействие с внешним миром выполняется как IO-действия. Значение `IO` a описывает действие, которое при выполнении вернёт a. Из `IO` нельзя просто извлечь чистое значение: функции `IO` $a \to a$ не существует.

Итак, у всех этих монад один интерфейс — поэтому один и тот же do-код работает с разными эффектами. При этом Haskell остаётся чистым: эффекты не выполняются незаметно, а явно отражаются в типах и связываются в контролируемую последовательность вычислений.

Стрелки Клейсли

Стрелка Клейсли — это функция, чей результат завёрнут в монаду:

```
a -> m b
```

Монадический код — это цепочки функций вида $a \to m\,b$: каждая что-то считает и попутно порождает контекст. Хочется складывать их так же легко, как чистые функции точкой (.). Стрелки Клейсли и их композиция дают ровно это. Обычная функция $a \to b$ — её частный случай (контекст тривиален).

Две стрелки Клейсли склеиваются в одну — это композиция Клейсли (её оператор зовут «рыбкой»):

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)   -- обратная «рыбка»

-- композиция Клейсли выражается через bind монады:
(f >=> g) x = f x >>= g
```

Обратный порядок пишут (<=<) — он читается как обычная композиция (.). Тождественная стрелка — `pure` (она же `return`):

```
pure :: a -> m a
```

Возьмём пример из предыдущего раздела и перепишем его на стрелках:

```
-- в разделе про монады было:
processNumber x = pure x >>= halveEven >>= halveAndScale

-- на стрелках Клейсли - без аргумента и без pure:
processNumber = halveEven >=> halveAndScale
```

Скоро мы поговорим детальнее про категории, а тут пока скажем пару слов, а именно, возьмём типы за объекты, стрелки Клейсли за морфизмы, (>=>) за композицию, а `pure` за тождество — получится категория Клейсли. Её аксиомы — это в точности законы монады:

```
pure >=> f == f                          -- единица
f >=> pure == f
(f >=> g) >=> h == f >=> (g >=> h)       -- ассоциативность
```

То есть «монада» и «категория Клейсли» — два названия одного и того же. Законы, которые через >>= выглядят громоздко, здесь становятся привычными.

Трансформеры монад

Реальному коду зачастую нужно несколько эффектов сразу: состояние + ошибки + IO. Но монады, в отличие от функторов, не композируются автоматически: $m\,(n\,a)$ в общем случае не монада. Трансформер — версия монады со «слотом» для другой монады.

Определяются трансформеры как те же монады, но с параметром-монадой внутри:

```
newtype MaybeT  m a   = MaybeT  (m (Maybe a))
newtype ExceptT e m a = ExceptT (m (Either e a))
newtype StateT  s m a = StateT  (s -> m (a, s))
```

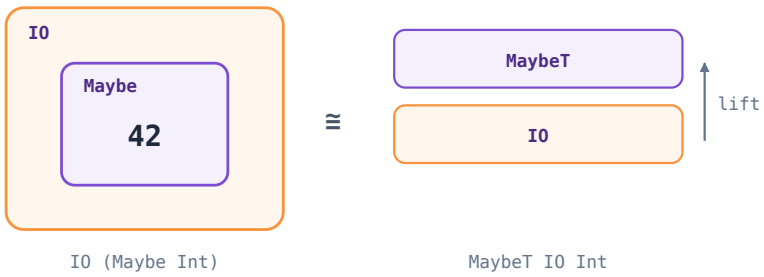

Рис. 2.6. Вложенные коробки — это стек трансформеров: `IO (Maybe Int)` ≅ `MaybeT IO Int`; `lift` поднимает действие базовой монады вверх по стеку.

Существует такая монада `Identity`, определяется она так:

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
  return           = Identity     -- обернуть значение
  Identity x >>= f = f x          -- и сразу развернуть
```

Обычные монады — те же трансформеры над тождественной монадой:

```
type State s = StateT s Identity
MaybeT Identity a  ≅  Maybe a       -- Maybe не type-синоним, но изоморфна
```

Через слои стека действия базовой монады поднимает `lift`; его законы требуют согласованности с `pure` и `>>=`:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a

lift . pure == pure
lift (m >>= f) == lift m >>= (lift . f)
```

Порядок стека — это семантика: один и тот же набор эффектов, разный порядок — разное поведение при ошибке:

```
StateT s (Except e) a   ≅   s -> Either e (a, s)    -- ошибка съедает состояние
ExceptT e (State s) a   ≅   s -> (Either e a, s)     -- состояние переживает ошибку
```

Соберём сказанное на живом примере: цепочка IO-действий, каждое из которых может не дать результата (переменной окружения нет, файла нет). Без трансформера пришлось бы проверять `Nothing` после каждого шага; `MaybeT IO` прячет эти проверки в do-нотацию — первый же `Nothing` обрывает всю цепочку.

```
import Control.Monad.Trans.Maybe (MaybeT(..), runMaybeT)
import Control.Monad.Trans.Class (lift)
import System.Environment        (lookupEnv)      -- :: String -> IO (Maybe String)
import System.Directory          (doesFileExist)

readConfig :: FilePath -> IO (Maybe String)       -- Nothing, если файла нет
readConfig path = do
  exists <- doesFileExist path
  if exists then Just <$> readFile path else pure Nothing

-- IO + возможный провал в одной do-нотации:
loadConfig :: MaybeT IO String
loadConfig = do
  lift (putStrLn "reading config...")   -- обычное IO поднимаем через lift
  path <- MaybeT (lookupEnv "CONFIG")   -- нет переменной -> вся цепочка Nothing
  MaybeT (readConfig path)              -- нет файла      -> вся цепочка Nothing

main :: IO ()
main = do
  result <- runMaybeT loadConfig        -- runMaybeT :: MaybeT IO a -> IO (Maybe a)
  case result of
    Just cfg -> putStrLn ("ok: " ++ cfg)
    Nothing  -> putStrLn "config not found"
```

Foldable и Traversable

Свёртка — одна из самых важных интегральных характеристик во многих областях прикладной математики: сумма, среднее, максимум, норма — всё это свёртки коллекции данных в одно число. В Haskell свёртку можно провернуть на любом типе, обладающем экземпляром `Foldable`, который держится на одном методе:

```
class Foldable t where
  foldMap :: Monoid w => (a -> w) -> t a -> w
  -- (в классе есть и другие методы, но foldMap достаточно)

-- контейнер t реализует Foldable
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
instance Semigroup (Tree a) where
  Empty      <> s     = s
  l          <> Empty = l
  Leaf x     <> s     = Node Empty x s
  Node l x r <> s     = Node l x (r <> s)
instance Monoid (Tree a) where mempty = Empty
instance Foldable Tree where
  foldMap _ Empty        = mempty
  foldMap f (Leaf x)     = f x
  foldMap f (Node l x r) = foldMap f l <> f x <> foldMap f r

newtype Sum a = Sum a
instance Num a => Semigroup (Sum a) where Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid    (Sum a) where mempty = Sum 0

-- свернём дерево 1,2,3 моноидом Sum:
t = Node (Leaf 1) 2 (Leaf 3)
foldMap Sum t   -- Sum 6   (форма дерева схлопнулась в одно число)
```

На самом деле одного `foldMap` достаточно для вывода: `sum`, `length`, `maximum`, `elem`, `toList`… Но `foldMap` сводит всю структуру к одному значению. А если на каждом элементе нужно выполнить вычисление — которое может провалиться, сходить в конфиг, накопить лог — и при этом собрать результаты, не потеряв форму структуры? Это умеет `Traversable`.

Обратите внимание на классы `Functor` и `Foldable` в объявлении ниже: `traverse` настолько общий, что `fmap` и `foldMap` — его частные случаи. Подставьте вместо контекста f обёртку без эффекта `Identity` — и `traverse` вырождается в `fmap`, так что контейнер обязан быть `Functor`. Подставьте `Const` w (аппликатив, когда w — моноид) — и `traverse` вырождается в `foldMap`, значит, всякий `Traversable` заодно и `Foldable`. Оттого оба класса и вынесены в суперклассы:

```
class (Functor t, Foldable t) => Traversable t where
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  traverse g = sequenceA . fmap g   -- traverse и sequenceA взаимовыразимы

-- f = Identity (обёртка без эффекта):
(a -> Identity b) -> t a -> Identity (t b)  ≅  (a -> b) -> t a -> t b   -- fmap

-- f = Const w (Applicative, когда w - моноид):
(a -> Const w b)  -> t a -> Const w (t b)   ≅  (a -> w) -> t a -> w     -- foldMap

-- половинка чётного, иначе провал:
half :: Int -> Maybe Int
half x = if even x then Just (x `div` 2) else Nothing
```

```
traverse half [2, 4, 6]   -- Just [1,2,3]
traverse half [2, 3]      -- Nothing
```

Понятно, что `fmap half` даёт список коробок `[Maybe Int]`, а `sequenceA` выворачивает его в коробку списка `Maybe [Int]`; и стоит одному элементу вернуть `Nothing`, как весь результат — `Nothing`.

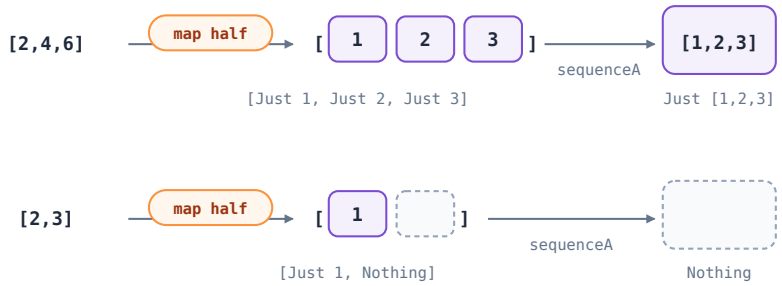

Рис. 2.7. `traverse half` превращает список в `Maybe`-список: один `Nothing` обнуляет всё.

Осталось понять, откуда в сигнатуре `traverse` взялся именно `Applicative`. В примере выше `sequenceA` склеивала контексты соседних элементов: `Just` с `Just`, а любой `Nothing` обнулял результат. Для такой склейки нужны ровно два умения: `pure` — начать обход с «пустого» эффекта — и $<*>$ — присоединить эффект очередного элемента к уже накопленному. Это в точности интерфейс `Applicative`: `Functor` здесь слаб (`fmap` не умеет сливать два контекста в один), а `Monad` избыточен (`>>=` разрешает следующему шагу зависеть от результата предыдущего, но маршрут обхода и так жёстко задан формой структуры).

У `traverse` есть и законы. Самый наглядный — тождество: `traverse Identity` = `Identity`, обход без эффекта не меняет ни форму, ни значения. Остальные два — композиция (два обхода подряд можно слить в один) и натуральность (обход согласован с заменой одного аппликатива на другой). Вместе они гарантируют, что `traverse` обходит каждый элемент ровно один раз и сохраняет форму структуры.

Так шестёрка классов и замыкается: `Monoid` комбинирует значения, `Functor` / `Applicative` / `Monad` применяют функции во всё более сильных контекстах, `Foldable` / `Traversable` обходят структуры — и практически весь «бойлерплейт» обработки данных сводится к ним.

Категории

Есть такой раздел математики — теория категорий, и ложится она на всё, что мы обсуждаем, как нельзя лучше, — так что мы просто обязаны немного о ней поговорить. Мысль в её основе вызывающе проста: забыть, что лежит внутри математических структур, и смотреть только на связи между ними. Работает теория всего с двумя видами сущностей. Первые — объекты: $A, B, C, \ldots$; что они такое, категорию не интересует. Вторые — морфизмы, они же стрелки: у каждой есть источник и цель, $f : A \to B$.

Стрелки могут склеиваться: например, для $f : A \to B$ и $g : B \to C$ определена композиция $g \circ f : A \to C$. Дальше — две аксиомы. Первая: композиция ассоциативна, $h \circ (g \circ f) = (h \circ g) \circ f$. Вторая: у каждого объекта есть тождественная стрелка $\mathrm{id}_A : A \to A$, нейтральная к композиции: $f \circ \mathrm{id} = \mathrm{id} \circ f = f$. Объекты, стрелки и две аксиомы — это и есть категория.

В Haskell этот паттерн выделен в класс типов — `Category` из модуля `Control.Category`:

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c

-- законы класса - в точности аксиомы категории:
f . id == f
id . f == f
(f . g) . h == f . (g . h)
```

Здесь `cat` — сама стрелка, параметризованная источником и целью (род $* \to * \to *$). В общем-то самый всеобъемлющий инстанс категории — это $Hask$: объекты — типы Haskell, морфизмы — функции между ними, композиция — (.) из Prelude, тождество — `id`:

```
instance Category (->) where
  id x = x
  (g . f) x = g (f x)

show   :: Int -> String      -- морфизм Int -> String
length :: String -> Int      -- морфизм String -> Int

length . show :: Int -> Int  -- их композиция - тоже морфизм
```

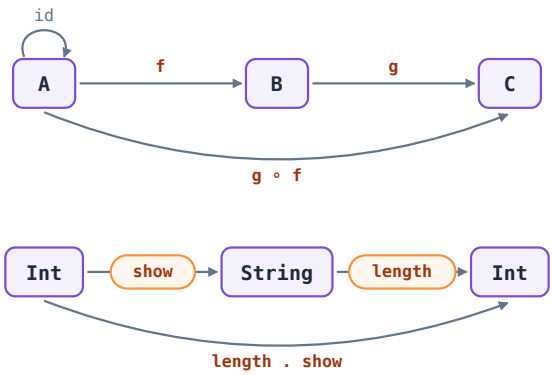

Рис. 2.8. Категория видит только объекты и стрелки. Вверху — абстрактно: f, g и их композиция $g \circ f$; внизу — то же самое в $Hask$: типы, функции и (.).

Настоящая ценность теории категорий для нас — то, что все конструкции, которые мы обсудили в этой главе, формулируются на её языке очень чисто и чётко:

- моноид — категория с одним объектом: морфизмы — элементы моноида, (<>) — композиция, `mempty` — тождество;

- функтор — отображение категории в категорию: объекты переходят в объекты, стрелки — в стрелки, а тождество и композиция сохраняются (это в точности законы `fmap`); наши функторы переводят $Hask$ в него же, поэтому зовутся эндофункторами — приставка «эндо-» по-гречески значит «внутри»: функтор из категории в неё же саму;
- аппликативный функтор — моноидальный функтор: помимо стрелок он переносит и пары, `liftA2` $(,)$ склеивает $(f\,a,\ f\,b)$ в $f\,(a,\,b)$, а `pure` даёт единицу;
- монада — категория стрелок Клейсли $a \to m\,b$ с композицией $(>=>)$ и тождеством `pure`;
- Foldable — свёртка в категорию с одним объектом: `foldMap` увозит элементы структуры в моноид и там склеивает;
- Traversable — перестановка двух эндофункторов местами: `sequenceA` $:: t\,(f\,a) \to f\,(t\,a)$.

Пункт про монаду можно потрогать руками: категория Клейсли — это newtype-обёртка с инстансом `Category`:

```
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Category (Kleisli m) where
  id                    = Kleisli pure
  Kleisli g . Kleisli f = Kleisli (f >=> g)

-- знакомый half - теперь как стрелка Клейсли:
half :: Kleisli Maybe Int Int
half = Kleisli (\x -> if even x then Just (x `div` 2) else Nothing)

quarter :: Kleisli Maybe Int Int
quarter = half . half          -- обычная точка склеила эффектные стрелки

runKleisli quarter 12   -- Just 3
runKleisli quarter 6    -- Nothing   (3 нечётно - вторая половинка не взялась)
```

Проверить законы `Category` для этого инстанса — значит проверить законы монады: тождество `pure` и ассоциативность $(>=>)$ мы уже видели в разделе о стрелках Клейсли. Монада и её категория Клейсли восстанавливаются друг из друга: это одна конструкция, записанная двумя способами.

К Haskell мы пришли тропой λ-исчисления: термы, редукции, типы. Но можно было идти и с другой стороны: типизированное λ-исчисление и декартово замкнутые категории — один и тот же предмет на двух языках. Тогда типы — это объекты, программы — морфизмы, а моноиды, функторы и монады этой главы — не изобретения Haskell, а понятия теории категорий, перенесённые в код почти дословно.

Конвейеры

Оператор конвейера $(>>>)$: та же композиция, но читается слева направо, по движению данных. Определён он для любого инстанса `Category`:

```
(>>>) :: Category cat => cat a b -> cat b c -> cat a c
f >>> g = g . f

-- стрелки категории Hask:
clean :: String -> String
clean = trim >>> lowercase >>> collapseSpaces

-- стрелки категории Клейсли:
pipeline :: Kleisli Maybe String User
pipeline = Kleisli parse >>> Kleisli validate >>> Kleisli build
```

Пока наш конвейер — одна труба: (>>>) соединяет шаги строго друг за другом. Но раз уж мы взялись управлять потоком данных, почему бы не пустить добро по трубам на полную катушку: подать на вход пару и обработать её половинки порознь. Для этого существует класс типов `Arrow`:

```
class Category a => Arrow a where
  arr    :: (b -> c) -> a b c
  first  :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***)  :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&)  :: a b c -> a b c'  -> a b (c, c')
```

Тут никуда без пояснений: `arr` поднимает чистую функцию в конвейер; `first` пропускает через стрелку первую компоненту пары, а вторую ведёт обводным каналом нетронутой — это и есть «данные мимо шага»; `second` — зеркальный близнец `first`: обрабатывает вторую компоненту, первая едет мимо; $(***)$ запускает две стрелки по половинкам пары параллельно; (&&&) раздваивает поток: один вход дублируется в обе стрелки, на выходе — пара результатов. При этом двух методов класса достаточно: `second`, $(***)$ и (&&&) выводятся из `arr` и `first`.

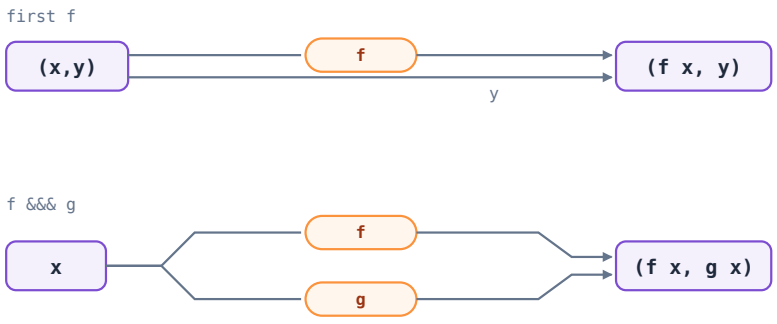

Рис. 2.9. Провода конвейера: `first` f обрабатывает первую компоненту пары и протаскивает вторую мимо, f &&& g дублирует вход в два параллельных потока.

Теперь пара инстансов — обычная функция и стрелка Клейсли:

```
-- подставим a := (->) в сигнатуру first:
first :: a    b c -> a    (b, d) (c, d)
first :: (->) b c -> (->) (b, d) (c, d)   -- a := (->)
```

```
first :: (b -> c) -> ((b, d) -> (c, d))   -- префиксное (->) x y = инфиксное x -> y

instance Arrow (->) where
  arr f          = f
  first f (x, y) = (f x, y)

-- то же для стрелки Клейсли, a := Kleisli m:
first :: a         b c -> a         (b, d) (c, d)
first :: Kleisli m b c -> Kleisli m (b, d) (c, d)   -- a := Kleisli m
first :: (b -> m c)    -> ((b, d) -> m (c, d))      -- сняли newtype-обёртку

instance Monad m => Arrow (Kleisli m) where
  arr f              = Kleisli (pure . f)
  first (Kleisli f)  = Kleisli (\(x, y) -> do c <- f x; pure (c, y))
```

Практический пример — сводка по временам ответа: среднее и максимум одним конвейером. На рис. 2.10 покажем, как он идёт, — заметьте, что (&&&) там дважды: не только снаружи, но и внутри (`sum` &&& `genericLength`):

```
mean :: [Double] -> Double
mean = (sum &&& genericLength) >>> uncurry (/)

summary :: [Double] -> (Double, Double)
summary = mean &&& maximum

summary [12, 8, 40]   -- (20.0, 40.0)
```

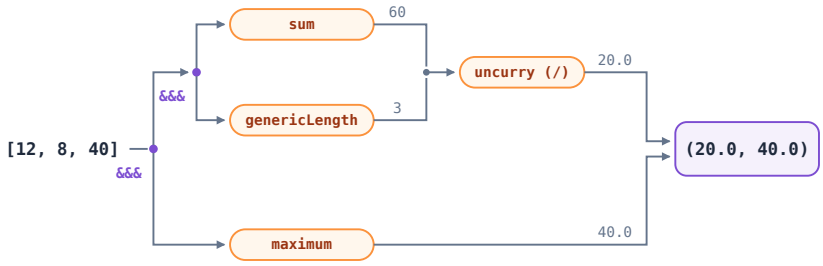

Рис. 2.10. Конвейер `summary`: внешний (&&&) раздваивает вход на среднее и максимум, внутри верхней ветки второй (&&&) считает сумму и длину.

На чистых функциях свет клином не сошёлся: комбинаторы класса `Arrow` работают для любой стрелки. Перенесём сводку `summary` в категорию Клейсли: времена ответа теперь не лежат готовым списком, а читаются из файла — конвейер живёт в IO. Первый шаг — настоящая стрелка Клейсли, дальше `arr` поднимает чистые `mean` и `maximum` в ту же категорию, и развилка (&&&) повторяет строчку `summary` один в один:

```
-- та же сводка, но времена ответа теперь читаются из файла
readTimes :: Kleisli IO FilePath [Double]
readTimes = Kleisli readFile >>> arr (lines >>> map read)

summaryIO :: Kleisli IO FilePath (Double, Double)
summaryIO = readTimes >>> arr mean &&& arr maximum   -- ср.: summary = mean &&& maximum

runKleisli summaryIO "times.log"   -- файл со строками 12, 8, 40 -> (20.0, 40.0)
```

Теперь несколько слов скажем не про пару, а про `Either` — работа с ним вынесена в отдельный класс `ArrowChoice`:

```
class Arrow a => ArrowChoice a where
  left  :: a b c -> a (Either b d) (Either c d)
  right :: a b c -> a (Either d b) (Either d c)
  (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
  (|||) :: a b d -> a c d   -> a (Either b c) d
```

Каждый метод — двойник метода из Arrow: `left` и `right` — как `first` и `second`, только вместо половинки пары обрабатывают одну ветвь `Either`, а другую пропускают нетронутой; если $(***)$ обрабатывает *обе* половинки пары, то его двойник $(+++)$ — лишь *my* ветвь, которая реально пришла, а $(|||)$ — двойник $(\&\&\&)$ — вдобавок сливает обе ветви в общий выход, маршрутизируя поток. Инстанс для обычной стрелки $(\to)$:

```
-- подставим a := (->) в сигнатуру left:
left :: a    b c -> a    (Either b d) (Either c d)
left :: (->) b c -> (->) (Either b d) (Either c d)   -- a := (->)
left :: (b -> c) -> (Either b d -> Either c d)

instance ArrowChoice (->) where
  left f (Left  x) = Left (f x)   -- пришла левая ветвь - обрабатываем
  left f (Right y) = Right y      -- правая проходит мимо нетронутой

-- остальные методы выводятся из left:
f +++ g = left f >>> right g
f ||| g = (f +++ g) >>> arr (either id id)
```

Практический пример — та же сводка, но с защитой от пустого лога: $(+++)$ ведёт каждую ветвь своим конвейером, а $(|||)$ сливает их в общий выход:

```
checkLog :: [Double] -> Either String [Double]
checkLog xs = if null xs then Left "пустой лог" else Right xs

report :: [Double] -> String
report = checkLog >>> (("нет сводки: " ++) +++ summary) >>> (id ||| show)

report [12, 8, 40]   -- "(20.0,40.0)"
report []            -- "нет сводки: пустой лог"
```

LiquidHaskell

Ну что, пора пустить жиденького — как в «Терминаторе 2», где появляется робот из жидкого металла. Тип `Int` гарантирует форму, но молчит о содержании: что число не ноль, что список не пуст, что индекс в границах. LiquidHaskell навешивает на обычные типы Haskell логические предикаты — уточняющие типы (refinement types) — и проверяет их при компиляции SMT-решателем (Z3). Аннотации-«декораторы»

живут в специальных комментариях {-@ … @-}, поэтому код остаётся обычным Haskell и собирается обычным GHC. Синтаксис декораторов:

$$\{\, v : \tau \mid p(v) \,\} \tag{2.1}$$

— «значения типа τ, для которых верен предикат p». Базовый пример — деление, которое невозможно вызвать с нулём:

```
{-@ type NonZero = {v:Int | v /= 0} @-}

{-@ safeDiv :: Int -> NonZero -> Int @-}
safeDiv :: Int -> Int -> Int
safeDiv x y = x `div` y

ok  = safeDiv 10 2    -- проходит: что 2 /= 0, решатель видит сам
bad = safeDiv 10 0    -- ошибка компиляции: 0 не влезает в NonZero
```

Под капотом это подтипирование — сравнение двух уточнённых типов, у каждого свой предикат: p у аргумента, q у параметра. Тип $\{\, v : \tau \mid p \,\}$ — подтип $\{\, v : \tau \mid q \,\}$, когда любое значение, удовлетворяющее p, удовлетворяет и q, — эту импликацию и проверяет SMT-решатель. Для вызова `safeDiv 10 2` у аргумента $p \equiv (v = 2)$ — синглтонный тип литерала, у параметра `NonZero` — $q \equiv (v \neq 0)$, импликация $v = 2 \Rightarrow v \neq 0$ верна:

$$\frac{\forall v.\; p(v) \Rightarrow q(v)}{\{\, v : \tau \mid p \,\} \;\preceq\; \{\, v : \tau \mid q \,\}} \tag{2.2}$$

Предикатам доступны меры (measures) — простые функции над данными, которые LiquidHaskell умеет разворачивать в логике. Классика — `head`, который перестаёт быть частичным:

```
{-@ measure len @-}
len :: [a] -> Int
len []     = 0
len (_:xs) = 1 + len xs

{-@ head' :: {v:[a] | len v > 0} -> a @-}
head' :: [a] -> a
head' (x:_) = x    -- ветка [] не нужна: она невозможна по типу
```

Рекурсию LiquidHaskell проверяет на завершимость — ищет убывающую метрику (по умолчанию — первый числовой аргумент). Тип `Nat` в сигнатуре — встроенный в LiquidHaskell алиас $\{\, v : \texttt{Int} \mid v \geq 0 \,\}$, объявленный так же, как наш `NonZero`. Факториал из раздела про Y-комбинатор здесь тотален:

```
{-@ fac :: Nat -> Nat @-}
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)   -- n убывает и не уходит ниже нуля - решатель доволен
```

В режиме рефлексии на декораторах доказывают настоящие теоремы: свойство записывается типом, доказательство — программой. Это соответствие Карри — Ховарда:

```
{-@ LIQUID "--reflection" @-}
import Language.Haskell.Liquid.ProofCombinators

{-@ reflect double @-}
double :: Int -> Int
double x = x + x

{-@ doubleIsTwice :: x:Int -> { double x == 2 * x } @-}
doubleIsTwice :: Int -> Proof
doubleIsTwice x = double x === x + x === 2 * x *** QED
```

Это шаг в сторону типов, зависящих от термов, но не полноценные зависимые типы, как в Agda: предикаты ограничены разрешимыми теориями SMT (линейная арифметика, равенство, множества), зато всё проверяется автоматически, без ручных доказательств.