λ-calculus

From Term Syntax to Monads in Haskell

2026

UDC 510.6
LBC 22.12

Elyukov A. S.

λ-calculus: From Term Syntax to Monads in Haskell.

A short book on the lambda calculus: term syntax, beta reduction, Church encoding, the Y combinator, typed calculi, and the Haskell concepts built on top of them — type classes, functors, applicatives, monoids, monads and monad transformers.

For students and developers familiar with the basics of mathematical logic and functional programming.

Contents

Chapter 1

Core concepts

Term syntax

Just as in mathematics an incredible number of fields — calculus, complex analysis, and so on — rest on a handful of axioms taken on faith, in the λ-calculus just three constructs form the foundation. The entire language is built from the variable, the abstraction, and the application; everything else — numbers, logic, data structures, and recursion — appears on top of them.

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

where x is a variable; $(\lambda x.\, M)$ is an *abstraction*: an anonymous function with parameter x and body M; $(M\, N)$ is an *application*: the function M applied to the argument N. Here N may be any term: a variable x, another application $(P\, Q)$, or a whole abstraction $(\lambda y.\, y)$. For example: $M\, x$, $M\, (P\, Q)$, $M\, (\lambda y.\, y)$.

A few important remarks: application is left-associative: $M\, N\, P \equiv ((M\, N)\, P)$; the body of an abstraction extends as far to the right as possible: $\lambda x.\, M\, N \equiv \lambda x.\, (M\, N)$; a chain of abstractions is abbreviated: $\lambda x\, y\, z.\, M \equiv \lambda x.\, \lambda y.\, \lambda z.\, M$.

A term is a tree: the leaves are variables, the inner nodes are abstractions and applications. It is worth training yourself to see a term as a tree rather than as a string.

Bound and free variables

In this chapter we look at some fairly trivial things, but they will be very useful in the material that follows. Let us introduce the notion of free variables, which we denote FV. Here are a few examples of applying this notion to all the terms we already know.

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

A single variable x has x itself free. In an abstraction $\lambda x.\, M$ the variable x is bound by the nearest lambda, so it must be removed from the free variables of the body M. In an application $M\, N$ the free variables are taken from both parts: everything free in M plus everything free in N. A term with no free variables is called closed, or a combinator; note that one and the same letter may occur both free and bound, for example, in $x\, (\lambda x.\, x)$.

Recall that a term is a tree, and variables are its leaves. Now, into those leaves one can in turn substitute subtrees, that is, terms. This operation is called substitution. Here are a few examples of such substitutions:

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

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

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

As you can see, some restrictions have already appeared. But why? There is an important subtlety here. For instance, if the condition $y \notin \mathrm{FV}(N)$ is violated, a free variable gets "captured" by a foreign abstraction:

$$(\lambda y.\, x)[x{:=}y] \neq \lambda y.\, y \quad \text{— the meaning has changed!}$$

The cure is α-conversion, but more on that a little later.

Alpha and eta conversions

In the previous chapter we ran into an important problem that arises during substitution. Here we look more closely at how to solve it.

α-conversion

The name of a bound variable is no more than a label: $\lambda x.\, x$ and $\lambda y.\, y$ are one and the same function.

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

The especially gifted folks skip the renaming hassle altogether and use de Bruijn indices: the lambdas themselves stay but become nameless, while every occurrence of a variable is written as a number — how many lambdas you must go up from it to reach its binder (counting from zero, 0 being the nearest enclosing lambda). Then α-equivalent terms get one and the same representation, and α-conversion disappears as a concept.

- $\lambda x.\, x \to \lambda.\, 0$ (the only variable points to its own lambda);
- $\lambda x.\, \lambda y.\, x\, y \to \lambda.\, \lambda.\, 1\, 0$ (x became 1, and y became 0).

η-conversion

This is really a small thing, but still worth a brief mention. η-conversion says that the "wrapper" $\lambda x.\, M\, x$ is no different from M itself.

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

η-reduction collapses such a wrapper to the left, while η-expansion unfolds it to the right.

In Haskell this is the familiar point-free style:

```
sum xs = foldr (+) 0 xs
-- becomes
sum = foldr (+) 0
```

Beta reduction

We keep pouring from one empty vessel into another, and now it is β-reduction's turn. Reduction, precisely — not conversion, since there can be no expansion here: there is only reduction. In general, the whole operational semantics of the λ-calculus is a single rewriting rule, nothing more.

A function call in any language is a special case of β-reduction.

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

A term of the form $(\lambda x.\, M)\, N$ is called a redex. Reduction may be applied to any redex anywhere in a term. The notation $\to_\beta$ denotes a single β-reduction step, $\twoheadrightarrow_\beta$ — zero or more steps, and $=_\beta$ — the equivalence generated by reduction. A normal form, or NF, is a term with no redexes: there is nothing left to compute. The result of a program is meant to be exactly a normal form.

If we forbid reduction under a λ — not looking inside the body of an abstraction — we get a weaker notion: *weak normal form*. In it every redex is burned away except those hidden under a λ. For instance, $\lambda x.\, (\lambda y.\, y)\, z$ is already in weak normal form, even though it has not yet reached the ordinary NF $\lambda x.\, z$. It is to this form (more precisely, the weak head one, WHNF) that real languages evaluate: they do not touch the body of a function until it is called — more on that in the chapter on reduction strategies.

This is how infinite structures live in Haskell: they are evaluated to weak head form, but not all the way.

```
nats = [1..]        -- an infinite list 1 : 2 : 3 : ...
-- already in WHNF: the outer constructor (:) is known, the tail is an unevaluated thunk;
-- there is no full NF - reaching it would mean building the whole list
take 3 nats         -- [1,2,3]: we take only the beginning, and it all works
```

Examples of β-reductions:

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

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

But, as always, there are subtleties, and the main one is that far from every term reduces to a normal form. Above, in Haskell, I already gave the example of an infinite list, but this is not about that one. The term Ω reduces to itself — the computation never terminates:

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

And the fun does not stop there: it can happen that the order in which redexes are evaluated decides whether the algorithm loops or not. In lazy languages this is all over the place, but the compiler usually sorts it out on its own, unless we impose our own will on it. In short, it only gets more interesting from here.

Currying

And what about a function of two variables? Do those even exist? Simple answer: no, they don't. All right, all right, I'm kidding. They do — they're just wrapped in layers like an onion, and that concept is called currying, after Haskell Curry. Yes, that very Haskell Curry, the one the Haskell language is named after too.

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

Formally this is an isomorphism, that is, a bijection between two sets of functions.

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

On the left of the formula are functions from the pair $A \times B$ to C; on the right, functions from A to functions from B to C. To each two-argument function $g\colon A \times B \to C$ corresponds exactly one curried $\hat{g}\colon A \to (B \to C)$ by the rule $\hat{g}(a)(b) = g(a, b)$, and conversely — $g(a, b) = \hat{g}(a)(b)$. These two passages are mutually inverse, losing and adding nothing, so the two sets of functions are simply indistinguishable.

A few examples from Haskell:

```
-- the type of a multi-argument function is a right-associative chain of arrows
f :: a -> b -> c   -- the same as a -> (b -> c)

-- partial application: add 2 is an "under-applied" function
add :: Int -> Int -> Int
add x y = x + y
add2 :: Int -> Int
add2 = add 2
map add2 [1, 2, 3]  -- [3, 4, 5]

-- curry and uncurry are the witnesses of the isomorphism
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c

-- operator sections are the same partial application
(+3)  :: Num a => a -> a
(*2)  :: Num a => a -> a
```

Church encoding

In pure λ-calculus there are no numbers, no booleans, no structures — only functions. Church encoding shows that nothing else is needed: data are their own processing operations.

Let's start with booleans. And notice how practically grounded everything is here. There is no truth or falsehood in itself: these notions make sense only when they presuppose strategies of action. Otherwise there is no point in truth and falsehood existing at all. Feel the formulas.

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

Here t and f are simply two arguments (mnemonic: the "true" branch and the "false" branch): **tru** returns the first, **fls** the second. So a boolean value is itself a choice between two — a ready-made **if**.

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

Here **if** does almost nothing on its own: it takes a boolean b and hands it two branches t and e, and b picks the right one. **not** is just as simple: it gives the boolean its branches in reverse order, so truth chooses falsehood and falsehood chooses truth.

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

In **and** the first boolean p decides what to return: if p is true, the result depends on q; if false, falsehood is returned at once. In **or** it is the opposite: if p is true, truth can be returned immediately; if false, we have to look at q.

Let's substitute truth ($1 = \mathbf{tru}$, $0 = \mathbf{fls}$) and perform the reductions:

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

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

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

Now let's talk about pairs: a very important type, widely used in Haskell. A pair is a function that holds two values and waits for a "receiver":

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

fst hands the pair **tru**, **snd** hands it **fls**, and the pair gives back the right component:

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

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

Notice how neat this is: **fst** and **snd** take a pair as their argument and hand control over to it — the pair itself then decides what to return. These are, so to speak, decorator functions.

Now for the most pressing thing — the natural numbers. A number n is "apply a function n times":

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

Important: f and x here are arbitrary — they are supplied by whoever uses the number. A numeral fixes only "how many times to apply", not "what to apply".

Feed the numeral a function f and an argument x — and it applies f exactly as many times as the number specifies:

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

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

Now let's learn to compute — all arithmetic grows from one trick: "apply f the required number of times".

Let's start with adding one: **succ** tacks one more application of f onto the number:

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

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

Addition continues the idea — it applies f first $\overline{n}$ times, then $\overline{m}$ more:

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

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

Multiplication is already repeated addition:

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

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

Exponentiation is repeated multiplication, and its notation is very short:

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

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

In **pow** the arguments simply swap places: $\overline{n}\,\overline{m}$ is $\overline{m}$ applied as a function n times, which yields $\overline{m^{\,n}}$.

Subtraction is surprisingly hard: **pred** is built through pairs — we run a pair $(i-1,\, i)$ along the number and take the first component:

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

By a well-known legend, Stephen Kleene came up with the predecessor function (**pred**, predecessor) — the key to subtracting Church numerals — right in the dentist's chair.

Everything "subtractive" rests on **pred**: subtraction itself is repeated **pred**, and division is repeated subtraction. We give no separate λ-definition for division: there is no new idea in it — the same **pred** in a loop, only a bulkier expression.

Let's check **pred** on a general numeral $\overline{n}$. Denote the step $\sigma = \lambda p.\, \mathbf{pair}\,(\mathbf{snd}\, p)\,(\mathbf{succ}\,(\mathbf{snd}\, p))$ — it shifts the pair: it drops the first component and grows the second:

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

Substitute $\overline{n}$ into the definition — that means applying σ exactly n times to the starting pair $\mathbf{pair}\,\overline{0}\,\overline{0}$:

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

The pairs run along a chain — the second component counts the steps, the first lags by one:

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

It only remains to take the first component:

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

For $\overline{0}$ the chain does not move, so $\mathbf{pred}\,\overline{0} \twoheadrightarrow_\beta \overline{0}$ — we take the predecessor of zero to be zero.

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

iszero feeds the number a function that answers **fls** at every step, with the start value **tru**: zero has no steps — **tru** remains; any other number has at least one step, which turns it into **fls**.

A list is its own right fold (the fold encoding):

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

The same trick in Haskell — the type of `foldr`: a list is fully determined by how it folds. A Church numeral is "a list of n identical applications".

```
-- a list is built by the constructor (:) (cons) and terminated by [] (nil):
[1, 2, 3]  ==  1 : (2 : (3 : []))   -- True

-- Church encoding is the same chain, only (:) and [] become parameters;
-- in Haskell this is exactly foldr:
foldr (:) []  [1, 2, 3]   -- 1 : (2 : (3 : [])) = [1,2,3]   (rebuilt the list)
foldr (+) 0   [1, 2, 3]   -- 1 + (2 + (3 + 0))  = 6         (folded to a sum)
```

But surely there's normal programming, with numbers and even floating point. Why cook all this up? I wanted declarative programming, where everything is simple, and instead… Actually the answer is simple: folks, if you don't like the circus, there's nothing for you here, don't come; but if you love the circus…

Curry's Y combinator

We have already built quite a lot on top of λ-terms — data and branching — yet loops are clearly missing. And in their classical form they will not appear at all: instead we get recursion. That is, we need to cook up a term that would spin any function forever. Finiteness we secure through branching, but that comes later.

The Earth's axis stays fixed while the planet itself rotates around it. The equation of the axis is the key characteristic of the rotation, and it is worth being able to derive it. With terms the analogy is the same, only we are after not an axis but a point that

is fixed relative to a term. There is a theorem (the fixed-point theorem); I will state it loosely: every λ-term F has a fixed point — a term X for which

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

Such a point is not hard to build explicitly. Take $X \equiv (\lambda x.\, F\,(x\,x))\,(\lambda x.\, F\,(x\,x))$; its single redex reduces in one β-step:

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

So $X \to_\beta F\,X$, and in particular $F\,X =_\beta X$.

The term X can be written as $(\lambda f.\,(\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x)))\,F$. We need to tear the fixed point out of the equation and spin the function directly with Curry's combinator; the combinator is written as:

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

Substituting $X = \mathrm{Y}\,F$ into the theorem, we get the fixed-point identity — the working form of recursion:

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

Identity (1.25) is precisely an equality, not a reduction: no chain of direct β-steps takes $\mathrm{Y}\,F$ to $F\,(\mathrm{Y}\,F)$. Indeed, the single redex gives $\mathrm{Y}\,F \to_\beta A \equiv (\lambda x.\, F\,(x\,x))\,(\lambda x.\, F\,(x\,x))$ — this is the familiar X from the proof of the theorem. From there the chain runs through $F\,(A)$, $F\,(F\,(A))$ and so on, and the term $F\,(\mathrm{Y}\,F)$ never appears in it: inside A the combinator Y no longer occurs. Meanwhile $F\,(\mathrm{Y}\,F)$ itself reduces to $F\,(A)$ in a single step:

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

The two sides of the identity are equal only because they converge to the common reduct $F\,(A)$; to obtain exactly $F\,(\mathrm{Y}\,F)$ from $\mathrm{Y}\,F$ one would have to step against the arrow (a conversion).

Let us try this on the classic example — the factorial. The recursive step is given by a template with an extra parameter f (the future recursive call):

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

The fixed point of this template is exactly the factorial. With Curry's combinator the unfolding goes through β-equality:

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

This scheme is native to lazy languages (call-by-name/call-by-need). In Haskell the fixed-point combinator is built in — `fix` from `Data.Function` — and it unfolds exactly as in (1.25), on demand:

```
fix :: (a -> a) -> a
fix f = let x = f x in x   -- the knot is tied by a lazy let: x refers to itself

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

For all its usefulness in Haskell, Curry's combinator has two problems — and Haskell sidesteps both. The first is the type system: the self-application $x\,x$ does not typecheck in Hindley–Milner, because it sends type inference into a loop. So the fixed point is defined with a recursive `let` (as in `fix` above) or the self-application is hidden behind a newtype. What this system is and why it works this way is the subject of a separate chapter further on.

The second is the evaluation order: in a strict (call-by-value) language the definition `let x = f x` would loop forever — to obtain x the runtime immediately evaluates $f\ x$, for which it again needs x, and so on without end, never reaching any useful work. Haskell is saved by laziness: x is unfolded only when its value is actually demanded, so the same definition works without looping.

The Turing combinator

Curry's combinator solves the problem of recursion, but not flawlessly. One of its shortcomings we have already noted: it merely declares recursion. The identity $\mathrm{Y}\,F =_\beta F\,(\mathrm{Y}\,F)$ rests on β-equality, and there is no direct reduction from $\mathrm{Y}\,F$ to $F\,(\mathrm{Y}\,F)$ — to close the loop one has to step "backward" (a conversion). Let us try to fix this: to build another fixed-point combinator that unfolds recursion by a genuine forward β-reduction rather than proclaiming it as an equality.

We obtain this reduction — $\Theta\,F \twoheadrightarrow_\beta F\,(\Theta\,F)$ — by "pushing" the function itself, as an argument, inside the self-application: take $G = \lambda x\,y.\,y\,(x\,x\,y)$ and set $\Theta = G\,G$. Then in two β-steps:

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

The function (in the role of y) is not lost here; on each turn it is applied anew to the recreated $\Theta\,F = G\,G\,F$, and that is why the equality turns into an honest reduction. Unfolding G, we get the final form of the Turing combinator:

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

Let us compute the same factorial as in the chapter on Curry's combinator (the same template F, the same numeral $\overline{3}$): now every turn of the recursion is a genuine β-reduction, not a β-equality:

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

Both give $\overline{6}$; the only difference is the status of the unfolding step — an equality for Curry versus a reduction for Turing.

In practice the Turing combinator is almost never used — its role is theoretical: it shows that recursion is reachable by an honest reduction, without a step "backward", and in this capacity it is the standard companion of Y in textbooks.

But both Curry and Turing are designed for normal order (call-by-name). Under strict, eager order (call-by-value) both diverge: the argument F — that is, $\mathsf{Y}\,F$ or $\Theta\,F$ itself — is evaluated in advance and unfolds forever, never reaching any useful work. This problem is solved by the next combinator — Z.

The Z combinator

There remains a problem that afflicts both Curry's and Turing's combinators alike: under eager order (call-by-value) both loop forever. The recursive argument — $\mathsf{Y}\,F$ or $\Theta\,F$ itself — is unwound by the runtime in advance, before the function is even called, never reaching any useful work. This can be fixed by delaying the recursive call — turning it into a value that unfolds only when it is actually needed.

The idea is dead simple: wrap a function around the self-application. Technically this is an η-expansion $x\,x \to \lambda v.\,x\,x\,v$. A call wrapped in λ is already a value and unfolds only on a real call. So from Curry's combinator (1.24) we get the Z combinator — the same construction, only each $x\,x$ is replaced by $\lambda v.\,x\,x\,v$:

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

Let us trace the first unfolding. Set $W = \lambda x.\,F\,(\lambda v.\,x\,x\,v)$; then in two β-steps:

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

The last equality folds $W\,W$ back into $Z\,F$. The key point is that the recursive call has come out wrapped in λv: under λ it stays a value and unfolds only when actually applied to an argument (whereas $\mathsf{Y}\,F$ and $\Theta\,F$ unfold immediately under eager order).

The same factorial as in the chapters on Curry and Turing (the same template F, the same numeral $\overline{3}$) is computed by genuine β-reductions, and the wrapped call opens up exactly when we move on to the next number:

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

So under eager order (call-by-value) it does not unfold in advance and does not loop — the recursive call fires at exactly the right moment.

None of Y, Θ, Z typechecks in the simply typed λ-calculus: the self-application $x\,x$ would require a recursive type.

In Haskell that recursive type is exactly what one introduces — by wrapping the self-application in a newtype; and on it one clearly sees that Z does not rely on laziness. The ordinary `fix` rests on the lazy knot `let x = f x`, whereas Z builds recursion by self-application and hides the recursive call under λv — under a value. Let us force evaluation to be strict (the argument is forced before the function is applied, as under call-by-value): the naive Y loops, whereas Z calmly computes the factorial:

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

-- $! forces the argument before applying f - that is exactly eager order (call-by-value)

-- naive Y: the recursive call (unRec x x) is not a value, forced in advance => loops
yStrict :: (a -> a) -> a
yStrict f = w (Rec w) where w x = f $! unRec x x

-- Z: the call is wrapped in (\v -> ...), already a value - $! does not unfold it
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  =>  loops
-- zStrict fac 5  =>  120   -- Z computed it without any laziness
```

Examples of combinators

We have gone through three fixed-point combinators — Curry's, Turing's and Z — and it is time to say what a combinator is in general. A combinator is a closed λ-term, one with no free variables: it takes nothing from its surroundings and only shuffles and glues together its own arguments. There are plenty of them; we will look at the most common:

- $\mathrm{I} = \lambda x.\, x$ — identity: returns its argument as is.
- $\mathrm{K} = \lambda x\, y.\, x$ — constant: takes the first argument and discards the second.
- $\mathrm{S} = \lambda f\, g\, x.\, f\, x\, (g\, x)$ — substitution: hands x to both f and g, then applies one to the other.
- $\mathrm{B} = \lambda f\, g\, x.\, f\, (g\, x)$ — composition: runs x through g, then through f.
- $\mathrm{C} = \lambda f\, x\, y.\, f\, y\, x$ — flip: swaps the two arguments.
- $\mathrm{W} = \lambda f\, x.\, f\, x\, x$ — duplication: feeds x to the function twice.

In Haskell the combinators look like this:

```
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)
```

The SKI basis

The λ-calculus itself can be built on just three of them — S, K and I. In combinatory logic they are taken as primitives:

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

And I is not even needed as a primitive — it is expressible through S and K:

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

Let us substitute and prove the formula — expand $\mathrm{S\,K\,K}$ on an arbitrary argument z using the definitions (1.28):

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

For any z we get z — exactly what I does, so $\mathrm{S\,K\,K} = \mathrm{I}$.

Any λ-term is translated into combinators mechanically. A term is built in three ways — a variable, an application and an abstraction; variables and applications already exist in combinatory logic, but abstraction has to be eliminated. This is done by bracket abstraction — the operation $[x]\,M$, "pull the variable x out of the term M". Its result is a combinator term in which x no longer occurs but which, if given x back, reduces to M again. It is defined by the shape of the body M:

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

- $[x]\,x = \mathrm{I}$: the body is x itself; returning the argument unchanged is exactly what I does.
- $[x]\,M = \mathrm{K}\,M$ when $x \notin \mathrm{FV}(M)$: the body does not depend on x, so the given argument is simply discarded.
- $[x]\,(M\,N) = \mathrm{S}\,([x]\,M)\,([x]\,N)$: in an application x may hide in both M and N, so the argument must be handed to both — which is exactly what S does.

If the body is itself an abstraction $\lambda y.\,M$, the inner variable is removed first and the outer one after: $[x]\,(\lambda y.\,M) = [x]\,([y]\,M)$; thus nested λ are stripped from the inside out until only variables and applications remain.

Take $\lambda x.\,f\,x\,x$. The body $f\,x\,x$ is an application $(f\,x)\,x$, so we unwind it with the application rule, reducing everything to S, K, I:

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

There are no variables in the result — only S, K, I and the free f: $\lambda x.\,f\,x\,x \;\rightsquigarrow\; \mathrm{S}\,(\mathrm{S}\,(\mathrm{K}\,f)\,\mathrm{I})\,\mathrm{I}$.

Church–Rosser theorem

All roads lead to Rome. Do they? The Church–Rosser theorem says so: it guarantees uniqueness of the normal form, whichever paths we take towards it. Admittedly, in a Turing-complete language we may never reach a normal form at all — but if we do reach one, it is unique: whichever way you go, you arrive precisely in Rome.

If $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta K$, then there is an L such that $N \twoheadrightarrow_\beta L$ and $K \twoheadrightarrow_\beta L$. In other words, $\twoheadrightarrow_\beta$ has the diamond property (confluence):

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

The proof is best carried out via parallel reduction. We need three reduction operators on terms:

- $\to_\beta$ — ordinary, *single-step* β-reduction: $M \to_\beta N$ means one redex in M is chosen and contracted (one step — one redex);
- $\Rightarrow$ — *parallel* reduction: $M \Rightarrow N$ means an arbitrary set of redexes already present in M is contracted (zero, one, two — any number), each at most once, and only the existing ones, not their future copies;
- $\twoheadrightarrow_\beta$ — *multi-step* reduction (the reflexive-transitive closure of $\to_\beta$): $M \twoheadrightarrow_\beta N$ means there is a chain $M \to_\beta \cdots \to_\beta N$ of any length, including zero.

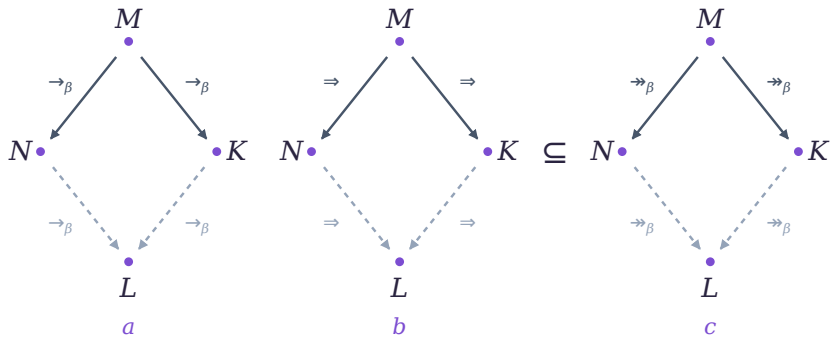

Fig. 1.1. The diamond property for the three reduction operators: a — single-step β-reduction $\to_\beta$; b — parallel reduction $\Rightarrow$; c — multi-step reduction $\twoheadrightarrow_\beta$.

Single-step reduction is a special case of parallel reduction, which in turn is a special case of multi-step reduction:

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

Why can't the diamond property be proved directly for single-step reduction $\to_\beta$ (fig. 1.1, a)? Redex copying gets in the way. Take a term with two redexes — an outer and an inner one:

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

Contract the outer redex — the substitution duplicates the inner one: we get $\big((\lambda y.y)\,z\big)\,\big((\lambda y.y)\,z\big)$. Contract the inner one — we get $(\lambda x.x\,x)\,z$. The paths have diverged by one step, but they can no longer meet in one step: the second term yields $z\,z$ immediately, while the first needs two steps, one per copy. So the single-step diamond property for $\to_\beta$ is simply false.

Parallel reduction $\Rightarrow$ cures exactly this disease: it contracts all the multiplied copies of a redex at once, in a single tick. For it the diamond property does hold (fig. 1.1, b), and it is proved by Takahashi's trick — via the complete development.

The complete development M^* is the term in which all redexes visible in M are contracted simultaneously — the greediest parallel step possible. It is defined by induction on the structure of the term:

The first rule: $x^* = x$ — a variable develops into itself.

The second rule: $(\lambda x.P)^* = \lambda x.P^*$ — the development goes under the abstraction.

The third rule: $(P\,Q)^* = P^*\,Q^*$ if $P\,Q$ is not a redex (P is not an abstraction): the parts are developed independently.

The fourth rule: $\big((\lambda x.P)\,Q\big)^* = P^*[x := Q^*]$ — the redex itself is contracted, and its parts are developed as well.

Let us develop the counterexample term by these rules:

- $\big((\lambda x.x\,x)\,((\lambda y.y)\,z)\big)^* = (x\,x)^*[x := ((\lambda y.y)\,z)^*]$ — the whole term is a redex, so the fourth rule fires: the outer redex is contracted and both its parts go into development;
- $(x\,x)^* = x^*\,x^* = x\,x$ — the application $x\,x$ is not a redex: the third rule, then the first for each variable;
- $((\lambda y.y)\,z)^* = y^*[y := z^*] = y[y := z] = z$ — the inner redex: the fourth rule, then the first twice;
- $(x\,x)[x := z] = z\,z$ — it remains to perform the substitution.

So $M^* = z\,z$: the development has contracted both the outer redex and the inner one — before the latter had a chance to multiply. The whole construction rests on the triangle lemma.

Triangle lemma. The complete development cannot be overtaken: whatever the parallel step $M \Rightarrow N$, exactly one parallel step leads from N to M^*:

$$M \Rightarrow N \quad\Longrightarrow\quad N \Rightarrow M^*$$

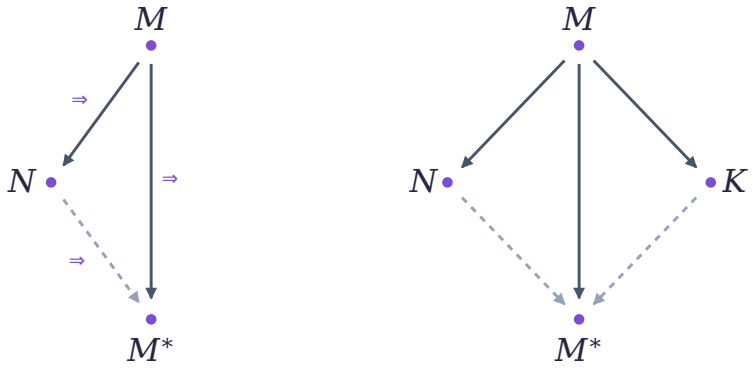

Fig. 1.2. The triangle lemma (left): any parallel step $M \Rightarrow N$ extends by one step to the complete development M^*. Right — the diamond property for $\Rightarrow$: two triangles sharing the bottom vertex M^*.

A parallel step is not obliged to contract all the redexes — from a single M the steps $\Rightarrow$ lead to a whole fan of terms, and the complete development is merely the extreme, greediest point of that fan. So the step $N \Rightarrow M^*$ is in general a real one; it is a zero step only when everything has already been contracted and $N = M^*$.

A reduction that contracts all visible redexes at once, in a single sweep, is called the Gross–Knuth reduction $M \Rightarrow_{\mathrm{GK}} M^*$. But it is of no interest to us: it has no variability — from a given term such a step leads to exactly one result. And the Church–Rosser theorem is precisely about variability: the paths $\twoheadrightarrow_\beta$ contract redexes in an arbitrary order, and they cannot be decomposed into rigid Gross–Knuth steps. What is needed is precisely the freedom to contract any subset.

The diamond property follows from the triangle immediately. Suppose $M \Rightarrow N$ and $M \Rightarrow K$. By the triangle lemma $N \Rightarrow M^*$ and $K \Rightarrow M^*$ — take $L = M^*$, and both sides close in one step (fig. 1.2, right). The common point is the same for all: it depends neither on N nor on K.

It remains to lift the diamond property from a single parallel step to multi-step reduction (fig. 1.1, c). The inclusions (1.32) let us read any chain of $\to_\beta$-steps as a chain of $\Rightarrow$-steps and vice versa. So let us write out $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta K$ as two chains of parallel steps, lay them along the sides of a grid and complete it cell by cell — each cell is closed by the diamond property for $\Rightarrow$.

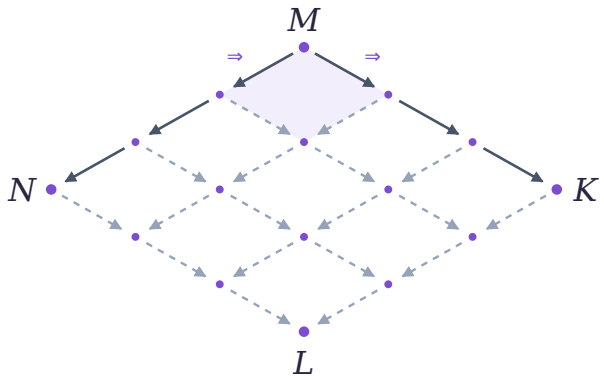

Fig. 1.3. A grid of diamond cells: the sides are chains of parallel steps $M \Rightarrow \cdots \Rightarrow N$ and $M \Rightarrow \cdots \Rightarrow K$; each cell (e.g. the shaded one) is the diamond property for $\Rightarrow$. The bottom vertex is the common reduct L.

Having filled the whole grid, we obtain at the bottom vertex a term L where both chains converge. All edges of the grid are parallel steps, and $\Rightarrow \subseteq \twoheadrightarrow_\beta$, so $N \twoheadrightarrow_\beta L$ and $K \twoheadrightarrow_\beta L$. The Church–Rosser theorem is proved.

The Church–Rosser theorem (1.31) immediately gives the existence of a common reduct: if $M =_\beta N$, there is an L with $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$ (induction on the definition of $=_\beta$). Hence the uniqueness of the normal form: a term has at most one β-NF — two paths would meet at a common L, and an NF does not reduce further, so $N_1 \equiv L \equiv N_2$. Next, reduction to NF: if N is a normal form of M, then $M \twoheadrightarrow_\beta N$ (not merely $M =_\beta N$). Finally, consistency: distinct normal forms are not β-equal — e.g. true and false — otherwise uniqueness of the NF would fail; this is how term "inequalities" are proved.

The order of contracting redexes, then, does not change the result — only *whether* the computation terminates and in how many steps. Choosing that order is the job of reduction strategies.

Reduction strategies

The rule that picks the next redex among all the redexes of a term is called a reduction strategy. Let us see where the choice arises. A term has one of three forms:

- a variable x — nothing to reduce;
- an abstraction $\lambda x.M$ — no choice: we reduce the body M;
- an application $M\,N$ — redexes may sit both in the left part and in the right one; choosing between them is exactly what a strategy does.

Let us unfold an application: its left part may itself be an application, the left part of that one too, and so on. Descending along the left branches to the first non-application — the head of the term — we get one of two shapes:

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

In the first shape the head is a variable x: there is nothing to contract on the left, redexes can hide only in the arguments N_i. In the second the head is an abstraction, and $(\lambda x.M)\,N_1$ is a redex; here is the fork: contract it right away or evaluate the argument first. This choice is what distinguishes the two classic strategies.

The normal strategy contracts the leftmost outermost redex — in our notation $(\lambda x.M)\,N_1$: arguments are substituted into the body unevaluated. *The applicative strategy* does the opposite: it first brings the argument of the redex to normal form and only then contracts the redex itself.

A term is conveniently drawn as a tree. For example, for $((\lambda x.M)\,N_1)\,N_2$ it looks like this (fig. 1.4): the @ nodes are applications, the λx node is an abstraction. The leftmost outermost redex is found on the tree by descending from the root along the left edges: it is the first @ whose left child is an abstraction.

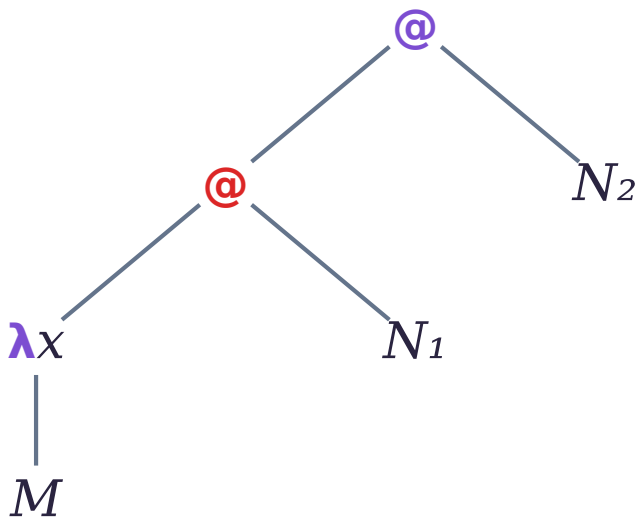

Fig. 1.4. The tree of the term $((\lambda x.M)\,N_1)\,N_2$: the @ nodes are applications, the λx node is an abstraction; the red @ is the leftmost outermost redex $(\lambda x.M)\,N_1$.

The same descent lets us introduce an important notion — the head normal form. Let us explain it with two examples:

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

The first term is a head normal form: the descent along the left branches ends at the variable y, called the head variable. The head of such a term will never change; redexes can hide only in the arguments N_i. The second term is not a head normal form: its head is the redex $(\lambda z.M)\, N_1$, called the head redex.

Let us state both strategies precisely, as rules of an operational semantics. As if we had not enough notions already — we need three more, three syntactic categories. The first is the non-abstraction NA:

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

Here everything seems clear: it is any term except an abstraction. M and N are arbitrary terms, so a non-abstraction may well be a redex — for example $(\lambda y.y)\, z$. The other two categories describe finished results: normal forms NF and their subclass, non-abstraction normal forms $NANF$:

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

The split is necessary: without it we could form an application with an abstraction on the left — and such a term is not a normal form any more, it is a redex. Examples of NF: $\lambda x.\, x$, $\lambda f.\lambda x.\, f\, x$. Examples of $NANF$: x, $f\, x$, $x\, (\lambda y.y)\, z$ — and each of them is at the same time an NF.

Now, at last, the strategies. At the heart of each lies its own redex-contraction rule. The normal strategy uses ordinary β-reduction: the redex is contracted right away, the argument N is an arbitrary term and is substituted into the body unevaluated:

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

The applicative strategy contracts a redex differently — only once the argument has been evaluated, that is, brought to normal form — to the left of the arrow stands an NF, and until the argument is one, contraction is forbidden:

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

Since we cannot contract a redex while the argument is not in normal form, the applicative strategy has a rule for the argument:

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

The remaining rules are shared by the normal strategy and the applicative one:

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

The first rule reduces the left part of an application while it remains a non-abstraction; the second moves on to the argument only when the left part is already a $NANF$; the

third takes reduction into the body of an abstraction. The categories *NA* and *NANF* in the premises are exactly what guarantees the leftmost outermost order: while there is something to contract on the left, we do not look to the right.

The choice of the leftmost outermost redex is no accident: the normalization theorem (Curry) guarantees that if a term has a normal form, then successively contracting exactly such a redex will reach it. For this the normal strategy is called complete.

Completeness is not free: an argument substituted into several places has to be recomputed. For a "big" N: $(\lambda x.\, F\, x\, (G\, x)\, x)\, N \to_\beta F\, N\, (G\, N)\, N$ — here N has to be reduced three times — once per copy. The applicative strategy evaluates N exactly once, but pays for this with completeness. Let $\mathrm{K} \equiv \lambda xy.x$, $\mathrm{I} \equiv \lambda x.x$, and let $\Omega \equiv (\lambda x.x\, x)\, (\lambda x.x\, x)$ — a term with no normal form. The normal strategy computes $\mathrm{K}\, \mathrm{I}\, \Omega \twoheadrightarrow_\beta \mathrm{I}$, discarding Ω without a glance; the applicative one first takes on the argument Ω — and loops forever.

In programming languages the two strategies live under different names. The applicative one is the call-by-value of strict languages such as C: arguments first, then the application. The normal one underlies lazy languages such as Haskell: call-by-name.

Turing completeness

Turing completeness is the presence of branching, repetition and data. All the ingredients have already been assembled in the previous chapters:

- data — Church encoding (section "Church encoding"): booleans, pairs, numerals, lists;
- branching — a Church boolean picks the branch itself: $\mathrm{if} \equiv \lambda b\, t\, e.\, b\, t\, e$;
- loops — the fixed-point combinator (section "Curry's Y combinator"): $\mathrm{Y}\, F =_\beta F\, (\mathrm{Y}\, F)$, recursion without names and without counters.

This is enough to model a Turing machine: the tape is a list, the state is a numeral, a step is branching, repeating the steps is Y. Turing completeness comes at a price — undecidability. There is no algorithm that, given a term, decides whether it has a normal form: this is the λ-counterpart of the halting problem.

Now, totality. A language is called total if every one of its programs terminates on every input. Turing completeness is incompatible with totality. In the λ-calculus infinite loops are possible:

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

Languages that chose totality do exist — for example, Agda. The Agda compiler checks every function: all input cases must be covered, and recursive calls must go to structurally smaller arguments. Addition of natural numbers passes the check — the recursive call is on n — a strictly smaller part of `suc` n:

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

_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)   -- the call is on n: strictly smaller than suc n
```

And this one Agda will reject:

```
loop : ℕ → ℕ
loop n = loop n           -- Termination checking failed: n does not decrease
```

The price is symmetric: Agda is not Turing complete — Ω cannot be written in it. In return, every program is a total function, and by the Curry–Howard correspondence (chapter "The Curry–Howard correspondence") it can be read as a proof. Haskell picks the middle ground: a typed core plus general recursion (`fix`, recursive let) — completeness returns, and with it infinite loops.

The Curry–Howard correspondence

So we have reached the most important part. A program in a language that takes the λ-calculus as its foundation and has solved the problem of non-termination is a theorem: its statement is the type, compilation is the checking of the proof, and execution is guaranteed to return a result of the promised type. The same is possible with imperative languages — program correctness has been proved ever since Hoare logic — but it is much harder: there the proof is not the program itself but a separate superstructure of preconditions, postconditions and invariants.

So: programs written in languages based on the λ-calculus are easier to prove correct than programs in other languages — not necessarily imperative ones; take Python. In this chapter we look at the Curry–Howard correspondence, a bridge between logic and programming: to write a program of type τ is the same as to prove the proposition τ. Curry noticed it back in the 1930s — in the types of combinators — and it took its precise form in Howard's 1969 manuscript.

Let us look at the types of familiar terms, reading the arrow as the implication "B follows from A". The identity function turns out to prove the tautology "A follows from A":

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

The combinator K, which discards its second argument, proves the proposition "if A holds, then A follows from anything":

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

The combinator S also proves a theorem — about distributing a premise over two consequences:

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

Let us, then, give the correspondence table:

Logic	Types / programs
implication $A \Rightarrow B$	function type $A \to B$
conjunction $A \wedge B$	pair (A, B)
disjunction $A \vee B$	`Either` A B
truth $\top$	unit type $()$
falsehood $\bot$	empty type `Void`
a proof	a term (a program)
proof simplification	β-reduction
second-order $\forall$	System F polymorphism
$\forall$, $\exists$ over objects	dependent types (LiquidHaskell)

A few clarifications. The unit type $()$ is the type with exactly one value. That value can always be constructed, trivially — hence $\top \leftrightarrow ()$. The empty type `Void`, by contrast, has no values at all: a program of this type cannot be written. This is falsehood itself: $\bot$ has no proofs.

"Second-order $\forall \to$ System F polymorphism" (we will come back to this later). A second-order quantifier ranges over propositions themselves (in the world of types — over types): "for every proposition A ...". In programs this is exactly parametric polymorphism: the type of the identity function $\forall a.\ a \to a$ reads as the theorem "for every A: A follows from A". System F is a λ-calculus with such quantification over types built into the language; Haskell's polymorphism is based on it.

"$\forall$, $\exists$ over objects $\to$ dependent types". First-order quantifiers range not over propositions but over objects — concrete values: "for every number n ...", "there exists a list x such that ...". To write this as a type, the type must depend on a value — this is what dependent types are. In Haskell itself $\forall$ ranges only over types, but LiquidHaskell attaches logical predicates to types — refinement types: the signature $\texttt{head}' : \{\, v : [a] \mid \texttt{len}\ v > 0 \,\} \to a$ reads as "for every list v whose length is greater than zero ...", and the implications between predicates are checked by an SMT solver. The quantifier $\exists$ is expressed by refining the result: $\{\, v : \texttt{Int} \mid \texttt{even}\ v \,\}$ is an even number presented together with the guarantee of its evenness. More on this in the "LiquidHaskell" chapter.

We are back to the problem of Turing completeness. Unrestricted recursion $\texttt{fix} : (A \to A) \to A$ reads logically as "if A follows from A, then A holds": a vicious circle that proves anything. That is why proof languages must be total: the totality check from the previous chapter is not an excess of caution but the condition of the logic's honesty. The Agda proof assistant stands on this correspondence: the specification is written as a type, a program of that type is the proof, and proof checking is ordinary type checking.

Typed lambda calculi

In the λ-calculus paradigm, types are a guarantee that a program finishes in finite time. The price is the loss of Turing completeness; the payoff is predictability, documentation and, as we have already seen, an entire logic.

The simplest such system is the simply typed λ-calculus ($\lambda^{\to}$). Every term in it is assigned a type, and the types themselves are built from atomic α by a single construction — the arrow:

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

Three inference rules — one per term construct:

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

Left to right: a variable takes its type from the context Γ; abstraction — if under the assumption $x{:}\sigma$ the body M has type τ, then $\lambda x.\, M$ is a function of type $\sigma \to \tau$; application — a function of type $\sigma \to \tau$ may only be applied to an argument of type σ, and the result gets type τ.

There are two important properties: the first is strong normalization: every typable term has a normal form, and every reduction order reaches it — as a consequence, Ω and Y are untypable and the language is not Turing-complete; the second is subject reduction: the type is preserved under reduction — "computation does not spoil the type".

What about polymorphism? Polymorphism in types, that is. $\lambda^{\to}$ has none: the identity function $\lambda x{:}\alpha.\, x$ is tied to one particular type α, and it has to be written anew for every type. Girard and Reynolds removed this restriction by allowing terms to abstract not only over values but also over types themselves — the result is System F, where the following holds:

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

Here $\Lambda\alpha$ is abstraction over a type: id first takes a type α, then a value x of that type, and returns it unchanged. The type $\forall\alpha.\, \alpha \to \alpha$ reads "for any type α, a function from α to α": one and the same id applies to a number, a boolean, a function — polymorphism became part of the calculus itself.

Type inference in full System F is undecidable, so Haskell uses a decidable fragment — the Hindley–Milner system (algorithm W), where all types are inferred without annotations; it is the subject of the next chapter.

The Hindley–Milner type system

The previous chapter left us with a dilemma: either we write all the types by hand (reliable but tedious), or we ask the machine to guess them itself (convenient but undecidable in general). The Hindley–Milner system (HM for short) is the golden mean: the compiler infers the most general type of any expression with no annotations at all.

How do we keep type polymorphism without sliding back into the undecidability of System F? Hindley and Milner solve this by restricting where a quantifier may stand, splitting types into two floors: downstairs — monotypes τ — types without quantifiers: atomic α, applications of type constructors $C\,\tau_1 \ldots \tau_n$ (this is how lists and pairs are built, for example) and arrows. Upstairs — schemes (polytypes) σ — monotypes capped with quantifiers:

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

Quantifiers are allowed only on the outside, at the very front of a type (prenex form): $\forall\alpha.\, \alpha \to \alpha$ is a legal scheme, while $(\forall\alpha.\, \alpha \to \alpha) \to \beta$ is not: here the quantifier has crept inside, to the left of an arrow. Allow this — and you get full System F with its undecidable inference; forbid it — and you lose a little expressiveness but turn type inference into an algorithm. The whole trick of HM lies in this restriction.

There are two transitions between the floors; let us look at both. Downwards — instantiation (Inst): a scheme becomes a monotype $\sigma[\alpha := \tau]$ by substituting any monotype for the bound variable. Upwards — generalization (Gen): if a variable α remains in the inferred type but does not occur free in any assumption of the context ($\alpha \notin \mathrm{free}(\Gamma)$), nothing outside constrains it — we may hang a quantifier on it:

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

Why these transitions are needed is seen in the rule for **let** — the heart of the whole system. Binding a name x to a value e_0, we generalize its type to a scheme σ, and then every use of x in the body e_1 instantiates this scheme anew, independently of the others:

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

The argument of a λ has no such privilege. Generalization happens only in the Let rule, and a function parameter never gets there: while we are typing the body, the function has not yet been applied to anything, and what will arrive as its input is unknown; all we can do is give the argument a single type variable, shared by all its uses in the body. The difference shows on two almost identical terms. The term $\mathbf{let}\ id = \lambda y.\,y\ \mathbf{in}\ (id\ \overline{3},\ id\ \mathbf{true})$ typechecks: id is bound by **let**, generalized to the scheme $\forall\alpha.\,\alpha \to \alpha$, and each of the two uses instantiates it with its own type. The term $(\lambda id.\,(id\ \overline{3},\ id\ \mathbf{true}))\ (\lambda y.\,y)$ — the same in meaning — no longer typechecks: here id is the argument of a λ, it has no scheme, and a single monotype cannot simultaneously be the type of a function on numbers and on booleans.

Haskell does have type annotations, of course — signatures can (and customarily are) written out. But they are optional: even if there were none at all, the compiler would reconstruct every type by itself — this is done by algorithm W (Damas–Milner). It walks the term, hands every subexpression a fresh type variable, and at every application records an equation: the type of the function must be an arrow from the argument type to the result type. The accumulated equations are solved by Robinson unification — a mechanical matching of two types: for $\alpha \to \beta$ to coincide with $(\gamma \to \gamma) \to \delta$, the substitution $\alpha := \gamma \to \gamma$, $\beta := \delta$ suffices. Bare code is enough for the compiler:

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

-- inferred without a single hint:
-- compose :: (b -> c) -> (a -> b) -> a -> c
```

Now we can see why the combinators Y, Θ and Z cannot be used in Haskell. They are all built on the self-application $x\,x$ — let us try to infer its type. Let $x : \alpha$; since x is applied to itself, α must be an arrow from α to some β — we get the equation $\alpha = \alpha \to \beta$. Substitute the right-hand side for α — and α is inside again: $(\alpha \to \beta) \to \beta$, $((\alpha \to \beta) \to \beta) \to \beta$, … — the substitution loops and the type grows without bound. This is exactly what the occurs check cuts short — the ban on a variable occurring in the type substituted for it: without recursive types the equation has no solution. That is why recursion in Haskell is introduced by the primitive `fix` and by recursive definitions.

Chapter 2

Haskell concepts

Type classes and kinds

Every abstraction of this part (monoid, functor, monad) is expressed in Haskell by a single mechanism — the type class: an overloadable interface plus laws that implementations must obey. A class with laws is an algebraic structure, not merely an interface as in, say, TypeScript.

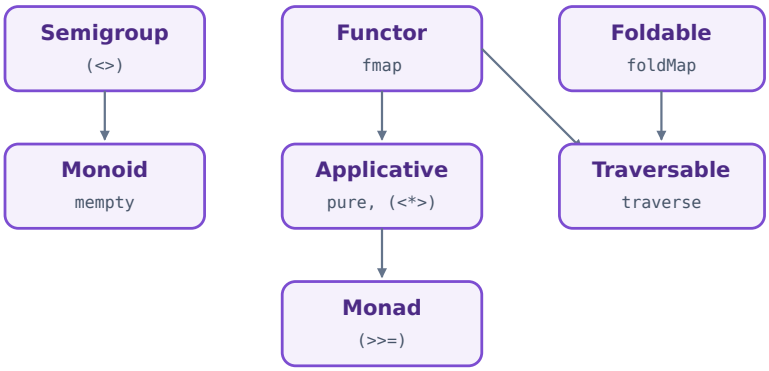

Fig. 2.1. The classes of this part and their hierarchy: an arrow goes from a superclass to a subclass; Traversable requires both Functor and Foldable.

As an example, consider the Functor type class: the class declaration defines the interface — the signature of the single method `fmap` — and an instance declaration implements this interface for a concrete type, here for `Maybe`.

```
class Functor f where
  -- fmap applies an ordinary function inside the context f
  fmap :: (a -> b) -> f a -> f b

instance Functor Maybe where
  -- if there is a value, apply the function to it
  fmap g (Just x) = Just (g x)
  -- if there is none, the Nothing context is preserved
  fmap g Nothing = Nothing
```

A method call is dispatched by type: the compiler sees that `fmap` is applied to a value of type `Maybe` and picks the instance above. Moreover, a type class can dispatch on the

return type as well: for example, `mempty` — the neutral element from the Monoid class — takes no arguments at all and "learns" its type from the call site. Class laws are not checked by the compiler — they are a contract on the conscience of the instance author: a class can be instantiated incorrectly — say, with an `fmap` that loses elements of the container — the compiler will accept such an instance, but code that relies on the laws will start producing wrong results. One more rule: a type gets at most one instance of a class (coherence).

Now, kinds: a kind tells how many parameters a constructor still lacks to become an ordinary type with values.

```
Int :: *
-- Int is an ordinary type: it has values

Maybe :: * -> *
-- Maybe is a type constructor with one parameter

Either :: * -> * -> *
-- Either is a type constructor with two parameters
```

The kind $*$ (a.k.a. `Type`) is the kind of ordinary types, the ones that have values. `Maybe` is not a type but a type constructor: a function at the type level that takes a type (say, `Int`) and returns a type (`Maybe Int`). Classes differ in the kind of their parameter: `Eq` — equality comparison — wants an ordinary type $*$, while `Functor` wants a constructor $* \to *$: a container can be a functor, a value cannot. Partial application works at the type level too: `Either` has kind $* \to * \to *$, but `Either e` $:: * \to *$ is already a legitimate functor.

Monoids

This little chapter is a pale shadow of Miss Monoid's lectures, but we'll do it our rough-and-ready way. A monoid is an abstraction equipped with an associative operation and an identity element. Associativity lets us split work into parts and group the results in different ways without changing the order of the data — hence parallel aggregation (map-reduce) and folds.

In Haskell, a monoid is represented by a pair of type classes: Semigroup provides an associative operation, and Monoid adds an identity element:

```
class Semigroup a where
  (<>) :: a -> a -> a          -- associative operation

class Semigroup a => Monoid a where
  mempty :: a                  -- identity element
```

Instances must obey two laws: associativity of the operation and the identity laws for `mempty`:

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

mempty <> x == x                 -- left identity
x <> mempty == x                 -- right identity
```

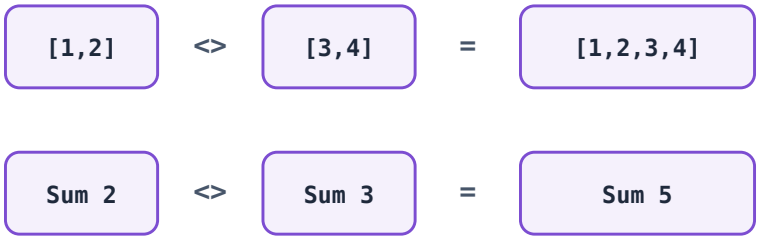

Fig. 2.2. Combining values in a monoid: for lists (<>) is concatenation; for `Sum` it is addition.

The standard library provides many instances:

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

-- numbers form a monoid in two ways, hence the wrappers:
Sum 2     <> Sum 3               -- Sum 5      (mempty = Sum 0)
Product 2 <> Product 3           -- Product 6  (mempty = Product 1)

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

-- minimum and maximum:
Min 5 <> Min 2                   -- Min 2
Max 5 <> Max 2                   -- Max 5
```

Now a few words about reduce: `mconcat` is its monoidal variant. It folds a list of monoidal values without knowing anything about the concrete type:

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

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

The Monoid laws let us safely split data into parts, process those parts in parallel, and then combine the partial results with (<>) while preserving the original order.

For example, each worker can independently accumulate its own piece of a log, after which the pieces are concatenated as lists. In the same way, we can sum separate chunks of data in parallel and then add the partial sums:

```
-- logs from three independent workers
workerLogs = ["worker 1: done\n", "worker 2: done\n", "worker 3: done\n"]
fullLog = mconcat workerLogs

-- sums computed independently for separate chunks of data
partialSums = [Sum 120, Sum 95, Sum 143]
total = getSum (mconcat partialSums)  -- 358
```

Associativity guarantees that the result does not depend on how the partial results are grouped: from left to right, pairwise as a tree, or according to the number of available cores.

Functors

Consider a list of natural numbers and a list of strings. Their element types differ, but the list structure and its structural operations remain the same. Let `T` denote the element type: the `List` constructor takes it and produces the concrete type `List T`. Therefore, the kind of `List` is $* \to *$.

A C++ programmer may reasonably point out that the templates `std` :: `vector`⟨`T`⟩ and `std` :: `list`⟨`T`⟩ already abstract over the element type, while iterators and ranges let the same algorithms work with different structures, such as lists and trees. Those algorithms, however, abstract over the traversal interface of an existing object rather than over a type constructor that preserves the shape of a context.

Haskell takes the next step. Both `List` and `Tree` have kind $* \to *$, so the `Functor` type class can abstract over the constructor `f` itself. The `fmap` operation takes a function `a` → `b`, transforms values inside `f a`, and returns `f b` while preserving the shape of the context. This is polymorphism not only over the element type but also over the context constructor. C++ can emulate it with templates and overloads, but the language has no direct standard counterpart to type classes and uniform higher-kinded polymorphism. The type class itself is declared as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b   -- apply a function inside context f
```

In this declaration, `f` is a variable for a type constructor of kind $* \to *$, not for a concrete type. The class specifies the common shape of `fmap`, while each instance defines how its context is preserved. The standard instances for lists and `Maybe` work as follows:

```
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)
```

As we can see, a `Functor` implementation depends on the context constructor but not on the concrete type of its elements. Now let us place the number 2 in a context. We begin with the key idea: a value in a context.

Fig. 2.3. A plain value and a value in the `Maybe` context: the "box" may contain a value (`Just 2`) or be empty (`Nothing`).

A context lets us define the rules for exceptional cases once and then apply ordinary functions without scattering checks throughout an algorithm. More powerful abstractions built on this idea can chain computations and specify execution strategies, but `Functor` itself guarantees only transformation through `fmap`. Suppose a JavaScript computation returns `undefined` instead of a value. We then have to write an `if`; if the result is passed onward, such checks begin to spread through the algorithm. Haskell represents absence explicitly with `Maybe`: its `Functor` instance leaves `Nothing` unchanged and applies the function to the value inside `Just`. Mapping over an empty list likewise returns an empty list. An ordinary function can be applied directly to a plain value; a value in a context would first have to be unpacked by hand. `fmap` takes care of that:

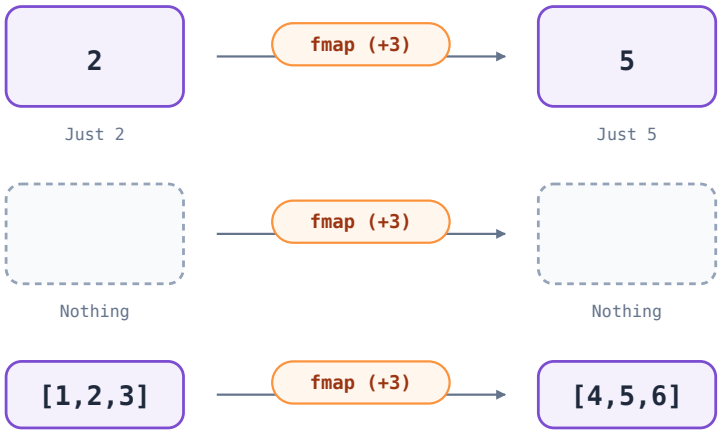

Fig. 2.4. `fmap` applies a function to the value inside a context while preserving the context itself: an empty box stays empty, and a list is mapped element by element.

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

`fmap` has an infix synonym: $g <\$> x = \texttt{fmap}\ g\ x$.

Every instance must obey two laws: identity and composition.

```
fmap id == id                      -- identity
fmap (g . h) == fmap g . fmap h    -- composition
```

These laws mean that `fmap` changes values but not the shape of the context: it neither rearranges a list nor turns `Just` into `Nothing`. They also state that mapping two functions in sequence is equivalent to mapping their composition.

The main instances:

```
-- Maybe: the value may be absent
fmap (+3) (Just 2)       -- Just 5

-- A list: zero or more values; fmap = map
fmap (+3) [1, 2, 3]      -- [4, 5, 6]

-- Either e: only Right is mapped
fmap (+3) (Right 2)      -- Right 5
fmap (+3) (Left "oops")  -- Left "oops"
```

A functor is a mapping between categories that preserves `id` and composition. In the usual model of Haskell, functors are endofunctors on the category of types and functions, and the `fmap` laws correspond exactly to the functor axioms.

Applicative functors

What if we need to apply a binary function to values from two lists? A C++ programmer would write a binary function or lambda and pass it to the chosen traversal algorithm. In Haskell, $(+)$ is curried: after receiving one argument, it returns a function waiting for the second. If we map $(+)$ over the first list, we get not sums but a list of partially applied functions: `fmap` $(+)$ $[1, 2, 3]$ has type `Num a` $\Rightarrow$ `[a` $\rightarrow$ `a]`. In other words, the functions are now inside the list context.

An ordinary `Functor` can apply a function `a` $\rightarrow$ `b` to values in `f a`, but it cannot apply functions from `f` (`a` $\rightarrow$ `b`) to values from another `f a`. That requires a stronger type class: `Applicative` and its $(<*>)$ operator.

The `Applicative` class extends `Functor` with two operations:

```
class Functor f => Applicative f where
  pure  :: a -> f a                  -- put a plain value into a context
  (<*>) :: f (a -> b) -> f a -> f b  -- apply a function from a context
```

In this declaration, `f` is a type constructor of kind $* \rightarrow *$. The `pure` operation puts a plain value into a context, while $(<*>)$ applies a function in a context to a value in the same context. Each instance defines the concrete semantics of these operations. The standard instances for lists and `Maybe` work as follows:

```
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
```

Fig. 2.5. Applicative application: both the function and the value are in a context; $(<*>)$ applies the former to the latter and returns the result in the same context.

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

Lifting is an important idea for applicative functors: an ordinary function of several arguments is "lifted" into a context so that it accepts arguments in contexts and returns its result in the same context. `liftA2` lifts a binary function, while `liftA3` lifts a ternary function:

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

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

There are four laws, and programmers are responsible for obeying them: an implementation may pass type checking while still breaking the laws. After defining an instance, we therefore need to verify all four conditions:

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

pure g <*> pure x == pure (g x)               -- homomorphism (a structure-preserving mapping)
u <*> pure y == pure ($ y) <*> u              -- interchange
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)  -- composition
```

The laws ensure that `pure` introduces no extra effect, contextual application agrees with ordinary function application, and regrouping a chain of $(<*>)$ does not change its result. They do not, however, permit effects to be reordered arbitrarily: order may still matter.

A couple of list examples, perhaps:

```
-- Every function is applied to every value
[(+1), (*2)] <*> [10, 20]
-- [11, 21, 20, 40]

-- Partially applied addition: again, all combinations
(+) <$> [1, 2] <*> [10, 20]
-- [11, 21, 12, 22]
```

Monads

Suppose we have a task: a function takes an integer and divides it by 2. If the result is odd, the function returns no value. If the result is even, it divides it by 2 once more and then multiplies it by 10 if the new result is even, and by 5 if it is odd. Let's first write such a function in 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;
}
```

Now let's rewrite this code in Haskell head-on. I'll say right away: this is not how things are done in Haskell.

```
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)
```

What clearly looks bad in the code above is the nesting. `TypeScript` has none, but to understand what the function returns you have to keep a careful eye on every `return`. Nesting, on the other hand, shows the direction of the computation unambiguously. In an imperative language it is clear anyway: code runs top to bottom. Let's rewrite the code, splitting the computation into two functions:

```
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
```

It may seem things only got worse, but they really haven't. Haskell already has a ready-made tool for this — the `Monad` class with its binding operator (`bind`):

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

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

Beautiful? That's the point. There is also the so-called `do` notation. Here is the same example written with it:

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

Does this remind you of anything? The code really does look like a program in an imperative language, although its semantics stays monadic. A monad lets you explicitly control the sequence of dependent computations — that is its defining feature. And it does so beautifully.

Just like functors, monads come with laws of their own, and keeping them is the programmer's responsibility:

```
pure a >>= f  ==  f a                          -- left identity
m >>= pure    ==  m                            -- right identity
(m >>= f) >>= g  ==  m >>= (\x -> f x >>= g)   -- associativity
```

So much for the binding operator. Now let's see which monads show up in real code. Almost every familiar "imperative" device — exceptions, a shared config, a log, mutable state, I/O — has a pure monadic counterpart.

Monad	Context / effect	Imperative counterpart
`Maybe`	the value may be absent	null + early return
`Either e`	an error with a cause	exceptions
`[]`	nondeterminism: zero or more results	nested loops
`Reader r`	shared read-only environment	config
`Writer w`	accumulating a log	logging
`State s`	mutable state	variables
`IO`	the outside world	system calls: files, network, console

At first glance these monads solve completely different problems. But Reader, Writer and State are built very much alike — they are wrappers around functions or values with an extra context:

```
newtype Reader r a = Reader (r -> a)        -- environment -> value
newtype Writer w a = Writer (a, w)          -- value + log
newtype State  s a = State  (s -> (a, s))   -- old state -> (value, new state)
```

The binding operator of the State monad threads the state through the chain by itself — no more writing $s_1, s_2, s_3 \ldots$ by hand. The primitives: `get` :: `State` s s, `put` :: $s \to$ `State` s ().

Every executable Haskell program runs inside the `IO` monad: its entry point has type `main` :: `IO` (), and all interaction with the outside world is performed as IO actions. A value of type `IO` a describes an action that, when executed, returns an a. You cannot simply extract a pure value out of `IO`: there is no function `IO` $a \to a$.

So all these monads share a single interface — which is why one and the same do code works with different effects. And Haskell stays pure: effects do not happen behind your back; they are explicitly reflected in the types and bound into a controlled sequence of computations.

Kleisli arrows

A Kleisli arrow is a function whose result is wrapped in a monad:

```
a -> m b
```

Monadic code is chains of functions of the form $a \to m\,b$: each one computes something and, along the way, produces a context. We would like to compose them as easily as pure functions with the dot (.). Kleisli arrows and their composition give exactly that. An ordinary function $a \to b$ is a special case of it (the context is trivial).

Two Kleisli arrows glue into one — this is Kleisli composition (its operator is nicknamed the "fish"):

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)   -- the reverse ‘‘fish’’

-- Kleisli composition is expressed through the monad’s bind:
(f >=> g) x = f x >>= g
```

The reversed order is written (<=<) — it reads like ordinary composition (.). The identity arrow is `pure` (also known as `return`):

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

Let's take the example from the previous section and rewrite it with arrows:

```
-- in the section on monads we had:
processNumber x = pure x >>= halveEven >>= halveAndScale

-- with Kleisli arrows - no argument and no pure:
processNumber = halveEven >=> halveAndScale
```

We will talk about categories in more detail soon; for now let's say just a couple of words, namely: take types as objects, Kleisli arrows as morphisms, (>=>) as composition and `pure` as the identity — and you get the Kleisli category. Its axioms are exactly the monad laws:

```
pure >=> f == f                          -- identity
f >=> pure == f
(f >=> g) >=> h == f >=> (g >=> h)       -- associativity
```

In other words, "monad" and "Kleisli category" are two names for the same thing. The laws that look cumbersome in terms of >>= become familiar here.

Monad transformers

Real code often needs several effects at once: state + errors + IO. But monads, unlike functors, do not compose automatically: $m\,(n\,a)$ is not a monad in general. A transformer is a version of a monad with a "slot" for another monad.

Transformers are defined as the same monads, but with a monad parameter inside:

```
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))
```

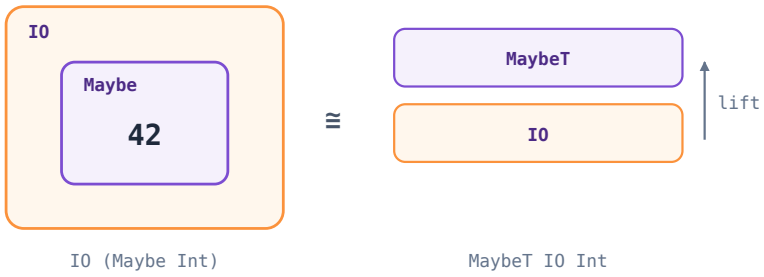

Fig. 2.6. Nested boxes are the transformer stack: `IO (Maybe Int)` ≅ `MaybeT IO Int`; `lift` raises a base-monad action up the stack.

There is a monad called `Identity`, defined like this:

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

instance Monad Identity where
  return           = Identity     -- wrap a value
  Identity x >>= f = f x          -- and immediately unwrap it
```

Ordinary monads are the same transformers over the identity monad:

```
type State s = StateT s Identity
MaybeT Identity a  ≅  Maybe a      -- Maybe is not a type synonym, but it is isomorphic
```

Actions of the base monad are raised through the layers of the stack by `lift`; its laws require consistency with `pure` and >>=:

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

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

The order of the stack is the semantics: the same set of effects, a different order — different behavior on error:

```
StateT s (Except e) a   ≅   s -> Either e (a, s)    -- an error eats the state
ExceptT e (State s) a   ≅   s -> (Either e a, s)     -- the state survives an error
```

Let's put it all together in a live example: a chain of IO actions, each of which may yield no result (the environment variable is missing, the file is missing). Without a transformer you would have to check `Nothing` after every step; `MaybeT IO` hides those checks inside do notation — the first `Nothing` aborts the whole chain.

```
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 if the file is missing
readConfig path = do
  exists <- doesFileExist path
  if exists then Just <$> readFile path else pure Nothing

-- IO + possible failure in a single do block:
loadConfig :: MaybeT IO String
loadConfig = do
  lift (putStrLn "reading config...")   -- lift an ordinary IO action
  path <- MaybeT (lookupEnv "CONFIG")   -- no variable -> the whole chain is Nothing
  MaybeT (readConfig path)              -- no file     -> the whole chain is 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 and Traversable

Folding is one of the most important integral characteristics in many areas of applied mathematics: sum, mean, maximum, norm — all of these fold a collection of data into a single number. In Haskell a fold can be pulled off on any type that has an instance of `Foldable`, which rests on a single method:

```
class Foldable t where
  foldMap :: Monoid w => (a -> w) -> t a -> w
```

```
  -- (the class has other methods too, but foldMap is enough)

-- a container t implements 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

-- fold the tree 1,2,3 with the Sum monoid:
t = Node (Leaf 1) 2 (Leaf 3)
foldMap Sum t   -- Sum 6   (the tree's shape collapsed into a single number)
```

In fact, `foldMap` alone is enough to derive `sum`, `length`, `maximum`, `elem`, `toList`... But `foldMap` reduces the whole structure to a single value. What if at every element we need to run a computation — one that can fail, hit a config, accumulate a log — and still collect the results without losing the shape of the structure? That is what `Traversable` does.

Note the classes `Functor` and `Foldable` in the declaration below: `traverse` is so general that `fmap` and `foldMap` are its special cases. Substitute the effect-free wrapper `Identity` for the context f — and `traverse` degenerates into `fmap`, so the container must be a `Functor`. Substitute `Const` w (an applicative whenever w is a monoid) — and `traverse` degenerates into `foldMap`, which means every `Traversable` is also a `Foldable`. That is why both classes end up as superclasses:

```
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 and sequenceA are interdefinable

-- f = Identity (an effect-free wrapper):
(a -> Identity b) -> t a -> Identity (t b)  ≅  (a -> b) -> t a -> t b   -- fmap

-- f = Const w (an Applicative whenever w is a monoid):
(a -> Const w b)  -> t a -> Const w (t b)   ≅  (a -> w) -> t a -> w     -- foldMap

-- half of an even number, failure otherwise:
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
```

Clearly, `fmap half` yields a list of boxes [`Maybe Int`], and `sequenceA` turns it inside out into a box with a list, `Maybe` [`Int`]; and as soon as a single element returns `Nothing`, the whole result is `Nothing`.

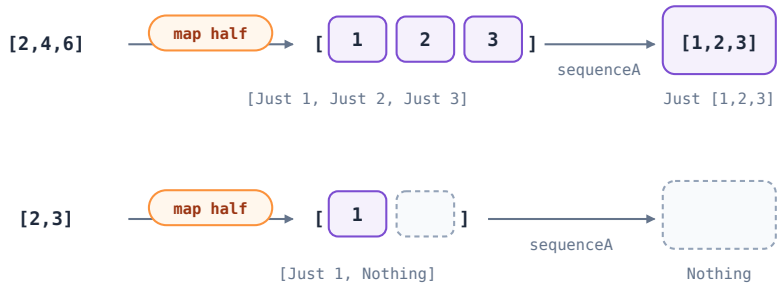

Fig. 2.7. `traverse half` turns a list into a `Maybe`-list: a single `Nothing` wipes out everything.

It remains to see why the signature of `traverse` demands precisely `Applicative`. In the example above `sequenceA` glued together the contexts of neighbouring elements: `Just` with `Just`, while any `Nothing` wiped out the result. Such gluing takes exactly two skills: `pure` — to start the traversal with an "empty" effect — and $<*>$ — to attach the effect of the next element to what has already been accumulated. That is precisely the `Applicative` interface: `Functor` is too weak here (`fmap` cannot merge two contexts into one), while `Monad` is overkill (`>>=` lets the next step depend on the result of the previous one, but the route of the traversal is rigidly fixed by the shape of the structure anyway).

`traverse` comes with laws, too. The most graphic one is identity: `traverse Identity` = `Identity` — an effect-free traversal changes neither the shape nor the values. The other two are composition (two consecutive traversals can be fused into one) and naturality (a traversal is consistent with swapping one applicative for another). Together they guarantee that `traverse` visits every element exactly once and preserves the shape of the structure.

And so the six classes close the loop: `Monoid` combines values, `Functor` / `Applicative` / `Monad` apply functions in ever more powerful contexts, `Foldable` / `Traversable` traverse structures — and practically all the data-processing boilerplate boils down to these six.

Categories

There is a branch of mathematics — category theory — and it fits what we are discussing like nothing else, so we simply have to talk about it a little. The idea at its core is defiantly simple: forget what lies inside mathematical structures and look only at the connections between them. The theory works with just two kinds of entities. The first are objects: $A, B, C, \ldots$; what they actually are, the category does not care. The second are morphisms, also known as arrows: each has a source and a target, $f : A \to B$.

Arrows can be glued together: for example, for $f : A \to B$ and $g : B \to C$ the composition $g \circ f : A \to C$ is defined. Then come two axioms. First: composition is associative, $h \circ (g \circ f) = (h \circ g) \circ f$. Second: every object has an identity arrow

$\mathrm{id}_A : A \to A$, neutral with respect to composition: $f \circ \mathrm{id} = \mathrm{id} \circ f = f$. Objects, arrows and two axioms — that is what a category is.

In Haskell this pattern is captured in a type class — `Category` from the `Control.Category` module:

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

-- the class laws are exactly the category axioms:
f . id == f
id . f == f
(f . g) . h == f . (g . h)
```

Here `cat` is the arrow itself, parameterised by source and target (kind $* \to * \to *$). Frankly, the most all-embracing instance of a category is *Hask*: objects are Haskell types, morphisms are functions between them, composition is (.) from the Prelude, the identity is `id`:

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

show   :: Int -> String      -- a morphism Int -> String
length :: String -> Int      -- a morphism String -> Int

length . show :: Int -> Int  -- their composition is a morphism too
```

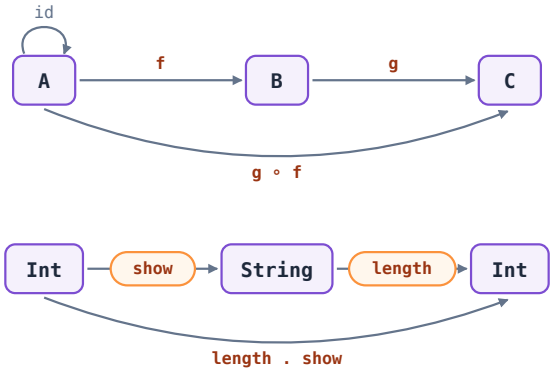

Fig. 2.8. A category sees only objects and arrows. Above — abstractly: f, g and their composition $g \circ f$; below — the same thing in *Hask*: types, functions and (.).

The real value of category theory for us is that every construction we discussed in this chapter can be stated in its language very cleanly and crisply:

- monoid — a category with a single object: the morphisms are the elements of the monoid, (<>) is the composition, `mempty` is the identity;

- functor — a mapping from category to category: objects go to objects, arrows go to arrows, and identity and composition are preserved (these are exactly the `fmap` laws); our functors map $Hask$ into itself, which is why they are called endofunctors — the prefix "endo-" is Greek for "within": a functor from a category into that very category;
- applicative functor — a monoidal functor: besides arrows it also carries pairs across, `liftA2` $(,)$ glues $(f\,a,\ f\,b)$ into $f\,(a,\,b)$, and `pure` provides the unit;
- monad — the category of Kleisli arrows $a \to m\,b$ with composition $(>=>)$ and identity `pure`;
- Foldable — a fold into a category with a single object: `foldMap` carries the elements of the structure into a monoid and glues them there;
- Traversable — swapping two endofunctors around: `sequenceA` $:: t\,(f\,a) \to f\,(t\,a)$.

The point about the monad can be touched with your hands: the Kleisli category is a newtype wrapper with a `Category` instance:

```
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)

-- the familiar half - now as a Kleisli arrow:
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          -- an ordinary dot glued effectful arrows together

runKleisli quarter 12   -- Just 3
runKleisli quarter 6    -- Nothing   (3 is odd - the second halving failed)
```

Checking the `Category` laws for this instance means checking the monad laws: the identity `pure` and the associativity of $(>=>)$ we have already seen in the section on Kleisli arrows. A monad and its Kleisli category can be recovered from each other: they are one construction written down in two ways.

We arrived at Haskell along the path of the λ-calculus: terms, reductions, types. But one could come from the other side as well: the typed λ-calculus and cartesian closed categories are one and the same subject in two languages. Then types are objects, programs are morphisms, and the monoids, functors and monads of this chapter are not Haskell inventions but notions of category theory, carried over into code almost verbatim.

Pipelines

The pipeline operator $(>>>)$: the same composition, but read left to right, the way the data flows. It is defined for any instance of `Category`:

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

-- arrows of the Hask category:
clean :: String -> String
clean = trim >>> lowercase >>> collapseSpaces

-- arrows of the Kleisli category:
pipeline :: Kleisli Maybe String User
pipeline = Kleisli parse >>> Kleisli validate >>> Kleisli build
```

So far our pipeline is a single pipe: (>>>) joins the stages strictly one after another. But since we have taken to steering the flow of data, why not run the goods through the pipes at full throttle: feed a pair in and process its halves separately. This is what the `Arrow` type class is for:

```
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')
```

There is no getting by without explanations here: `arr` lifts a pure function into the pipeline; `first` runs the first component of a pair through the arrow while carrying the second past it, untouched, along a bypass channel — this is the "data past a stage"; `second` is the mirror twin of `first`: it processes the second component while the first rides by; $(***)$ runs two arrows over the two halves of a pair in parallel; (&&&) forks the flow: a single input is duplicated into both arrows, and the output is the pair of results. In fact, two class methods are enough: `second`, $(***)$ and (&&&) are all derived from `arr` and `first`.

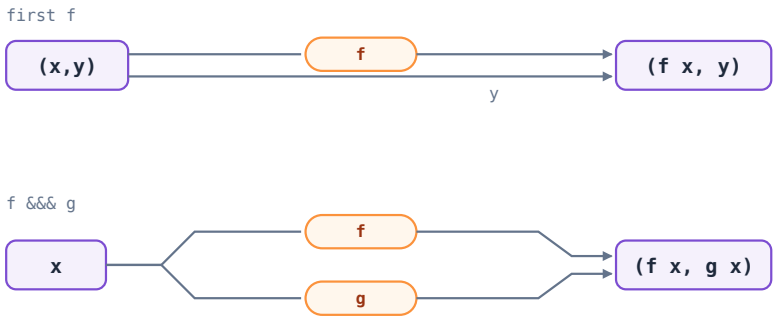

Fig. 2.9. Pipeline wiring: `first` f processes the first component of a pair and carries the second one past, f &&& g duplicates the input into two parallel flows.

Now a couple of instances — the plain function and the Kleisli arrow:

```
-- substitute a := (->) into the signature of 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))   -- prefix (->) x y is infix x -> y

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

-- the same for a Kleisli arrow, 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))      -- with the newtype wrapper removed

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))
```

A practical example — a summary of response times: the mean and the maximum in one pipeline. Fig. 2.10 shows how it flows — note that (&&&) appears twice: not only on the outside but also inside (`sum` &&& `genericLength`):

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

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

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

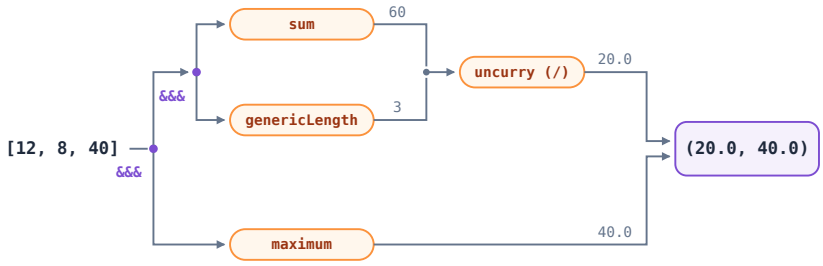

Fig. 2.10. The `summary` pipeline: the outer (&&&) forks the input into the mean and the maximum; inside the upper branch a second (&&&) computes the sum and the length.

Pure functions are not the whole story: the `Arrow` combinators work for any arrow. Let us move the `summary` pipeline into the Kleisli category: the response times no longer sit in a ready-made list but are read from a file — the pipeline now lives in IO. The first step is a genuine Kleisli arrow, then `arr` lifts the pure `mean` and `maximum` into the same category, and the (&&&) fork repeats the `summary` line word for word:

```
-- the same summary, but the response times now come from a file
readTimes :: Kleisli IO FilePath [Double]
readTimes = Kleisli readFile >>> arr (lines >>> map read)

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

runKleisli summaryIO "times.log"   -- a file with lines 12, 8, 40 -> (20.0, 40.0)
```

Now a few words not about the pair but about `Either`; working with it is moved out into a separate class, `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
```

Each method is a twin of an Arrow method: `left` and `right` mirror `first` and `second`, only they process one branch of an `Either` instead of one half of a pair, letting the other branch pass through untouched; if $(***)$ processes *both* halves of a pair, its twin $(+++)$ processes only *the* branch that actually arrived, while $(|||)$ — the twin of $(\&\&\&)$ — additionally merges the two branches into a common output, routing the flow. The instance for the plain function arrow $(\rightarrow)$:

```
-- substitute a := (->) into the signature of 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)   -- the left branch arrived - process it
  left f (Right y) = Right y      -- the right one passes through untouched

-- the remaining methods are derived from left:
f +++ g = left f >>> right g
f ||| g = (f +++ g) >>> arr (either id id)
```

A practical example — the same summary, now guarded against an empty log: $(+++)$ sends each branch down its own pipeline, and $(|||)$ merges them into a common output:

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

report :: [Double] -> String
report = checkLog >>> (("no summary: " ++) +++ summary) >>> (id ||| show)

report [12, 8, 40]   -- "(20.0,40.0)"
report []            -- "no summary: empty log"
```

LiquidHaskell

Well then, time to let in the liquid one — as in Terminator 2, where a robot of liquid metal appears. The type `Int` guarantees the shape but says nothing about the content: that a number is non-zero, that a list is non-empty, that an index stays in bounds. LiquidHaskell hangs logical predicates onto ordinary Haskell types — refinement types — and checks them at compile time with an SMT solver (Z3). The "decorator" annotations

live in special comments {-@ ... @-}, so the code remains plain Haskell and builds with plain GHC. The decorator syntax:

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

— "values of type τ for which the predicate p holds". The basic example is a division that cannot be called with zero:

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

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

ok  = safeDiv 10 2    -- passes: the solver sees for itself that 2 /= 0
bad = safeDiv 10 0    -- compile error: 0 does not fit into NonZero
```

Under the hood this is subtyping — a comparison of two refined types, each with its own predicate: p on the argument, q on the parameter. The type $\{\, v : \tau \mid p \,\}$ is a subtype of $\{\, v : \tau \mid q \,\}$ when every value satisfying p satisfies q as well — and this implication is exactly what the SMT solver checks. For the call `safeDiv` 10 2 the argument has $p \equiv (v = 2)$ — the singleton type of the literal — the parameter `NonZero` has $q \equiv (v \neq 0)$, and the implication $v = 2 \Rightarrow v \neq 0$ holds:

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

Predicates can use measures — simple functions over data that LiquidHaskell knows how to unfold in the logic. The classic is `head`, which stops being partial:

```
{-@ 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    -- no [] branch needed: the type rules it out
```

LiquidHaskell also checks recursion for termination — it looks for a decreasing metric (by default, the first numeric argument). The type `Nat` in the signature is LiquidHaskell's built-in alias $\{\, v : \texttt{Int} \mid v \geq 0 \,\}$, declared just like our `NonZero`. The factorial from the section on the Y combinator is total here:

```
{-@ fac :: Nat -> Nat @-}
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)   -- n decreases and never drops below zero - the solver is satisfied
```

In reflection mode, decorators prove genuine theorems: the property is written as a type, the proof as a program. This is the Curry–Howard correspondence:

```
{-@ 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
```

This is a step toward types that depend on terms, but not full dependent types as in Agda: the predicates are restricted to SMT-decidable theories (linear arithmetic, equality, sets) — in return, everything is checked automatically, with no manual proofs.