COMP0009 Logic

Lecture notes · 2026–27

Part 1: Type Theory

---

*Chapter 1*

# The Combinator Calculus

---

Alexander V. Gheorghiu

Department of Computer Science, University College London

This chapter is issued on its own from the complete lecture notes for COMP0009 Logic. Its section, theorem, exercise and equation numbers are those of the complete notes, so that the two can be read side by side; references to other chapters and parts point into the complete notes.

Contents

1. The Combinator Calculus 1
1.1. Combinator Algebra 2
1.2. Confluence 5
1.3. Evaluation Strategies 9
1.4. Programming 10
1.5. The Monster 15
1.6. Types 17
1.7. Subject Reduction 22
History 24
Exercises 28

# 1. The Combinator Calculus

Logic is the study of what it is for one thing to follow from another. In the 1930s logicians asked what a mechanical procedure could and couldn't decide, and by 1936 they had arrived at the idea of a machine that can carry out any computation. That machine is the computer.[1]

Plenty of courses stay with the older part of this story, and it's a fine story. This one starts from the computer instead, because over the last century logic has found its biggest home in computer science. The quickest way to see why, I think, is to start where you've already met it: a red line.

At some point your compiler has refused to run your code. Before the program ever started, a red line appeared under something, and the compiler wouldn't go any further. The function hadn't been called and no input had been tried, and still the compiler was sure enough to refuse. Sure of what? And what entitles it to be sure? Certainty that has cost nothing is either a swindle or a theorem, and this chapter is an attempt to find out which.

Inside the machine everything is bits: 1s and 0s, "on" and "off". We use patterns of them, thirty-two at a time, to stand for information: an integer, a few letters of a string, a boolean. The meaning comes from us; the processor doesn't know and doesn't care. Asked to combine a string and a number, it will do so without hesitation. The little red line is a sanity check that stops it when the result would be unintended or, worse, dangerous.

In 1999, after years of effort, NASA's Mars Climate Orbiter, built to be the first interplanetary weather satellite, reached Mars and promptly vanished. It passed behind the planet and radio contact was never restored, so we don't know exactly what became of it. This was not meant to happen. The orbiter hadn't followed the path intended, and yet every calculation was correct. What had gone wrong? Not the physics, and not the arithmetic, but something subtler: one program reported a figure in pound-force seconds, and another read it as newton-seconds (Mars Climate Orbiter Mishap Investigation Board, 1999). No test caught the fault. A checker that knew the *kind* of each number would have refused the hand-over, and that is what the little red line in your code is.[2]

Today almost every language you use has such a checker.[3]

Put like that, the job sounds modest: a spell-checker for numbers. But there is a more worrying sort of failure. A few lines of code can, quite by accident, run for ever, using up resources as they go. Such a program never outputs anything, so it is never caught in an error; it just goes on, like a loop with no exit. I call such programs *monsters*, because they are as dangerous

[1] The longer story: for two thousand years, from Aristotle on, logic lived in philosophy. For three hundred more, from Descartes and Leibniz, it lived in mathematics as well. Then, from 1879 into the 1920s, Frege, Russell and Hilbert rebuilt it almost from scratch. The logicians of the 1930s who arrived at the machine were Gödel, Church and Turing.

[2] A gentler example of the same bug: in 1983 an Air Canada Boeing 767 ran out of fuel in flight because its fuel load had been worked out in pounds instead of kilograms. The pilot glided it down onto a disused air-force strip at Gimli, Manitoba, then in use for drag racing, and everyone on board survived.

[3] Java and Rust were born with theirs. JavaScript grew one, called TypeScript, when the untyped web became unmanageable, which may be the most polite sentence anyone has written about JavaScript. Python, in 2015, acquired the means for one: type hints.

as they are hard to spot. The little red line was invented to keep them out too. How can a checker that looks only at the *kinds* of things do that? That is the question this chapter is built around.

Part 1 is about type-checking: the red line the compiler draws. There are hundreds of languages, each with a checker of its own, and no two of them look alike on the page. They all work on the same principles, though, and those are what we want to study. So we'll set aside any particular language and work at the level of computation itself.

For that we take up a proposal Haskell Curry made in the 1920s, before there were any computers. Curry wanted to understand the most mechanical step in all of logic, putting one thing in place of a variable in another. In effect, he wanted to understand what it is to compute with functions. His proposal comes in three parts.

- Everything is a function, and the only thing to do with a function is to apply it to something.
- To compute is to rewrite: a function applied to its arguments is replaced, by a fixed rule, with what it gives back.
- Two functions are enough to start from: one that throws an argument away, and one that shares an argument out.

What comes out is a language with no features at all, and it computes everything that any language you care to mention can. Curry's own reasons were logical rather than practical, and the chapter ends with them (§1.7).

1.1. **Combinator Algebra.** Our job in this section is to make Curry's outline exact: to say precisely what the terms are, and precisely how they are rewritten.

**Definition 1.1** (terms, and combinators)**.** Fix two *constants*, $\mathsf{K}$ and $\mathsf{S}$, and a stock of *variables* $x, y, z, f, g, \ldots$. The *terms* are defined inductively:

1. Each constant, $\mathsf{K}$ and $\mathsf{S}$, is a term.
2. Each variable $x$ is a term.
3. If $M$ and $N$ are terms, then $(M\,N)$ is a term: the *application* of $M$ to $N$.
4. Nothing else is a term.

A term without any variables is a *combinator*.

In one line, with $M$ and $N$ standing for terms already built,

$$M, N ::= \mathsf{K} \mid \mathsf{S} \mid x \mid (M\,N).$$

Writing out every bracket soon gets tiresome, and it makes a term hard for the human eye to read. Here, for instance, is a term with every bracket written:

$$(((((\mathsf{S}\,(\mathsf{K}\,\mathsf{S}))\,\mathsf{K})\,f)\,g)\,x).$$

One gets lost matching each opening bracket to its partner. So we adopt the usual convention, which lets us drop most of them: application groups to the left. Thus $\mathsf{K}\,x\,y$ means $((\mathsf{K}\,x)\,y)$, and we need to write brackets only when the grouping is *not* to the left: $f\,(g\,x)$ and $f\,g\,x$ are different terms,

the first being $f$ applied to $g\,x$, the second $f\,g$ applied to $x$. Under this convention the term above comes out as

$$\mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}\,f\,g\,x.$$

We use the convention throughout, and it soon becomes second nature.

On their own, terms just sit there. We want them to *do* something, and we allow them exactly one kind of action: *rewriting*. We write $M \rightsquigarrow N$, with a squiggly arrow, to say that $M$ *steps to* $N$, that is, that one rewrite turns $M$ into $N$. There are infinitely many terms, and we can't say what each of them does. Nor do we need to. It is enough to say what $\mathsf{K}$ and $\mathsf{S}$ do, and how a step taken in one part of a term becomes a step of the whole.

**Definition 1.2** (reduction)**.** The relation $M \rightsquigarrow N$, read "$M$ steps to $N$", is the one given by these rules:

$$\frac{}{((\mathsf{K}\,P)\,Q) \;\rightsquigarrow\; P}\;\mathsf{K} \qquad \frac{}{(((\mathsf{S}\,P)\,Q)\,R) \;\rightsquigarrow\; ((P\,R)\,(Q\,R))}\;\mathsf{S}$$

$$\frac{P \rightsquigarrow P'}{(P\,Q) \;\rightsquigarrow\; (P'\,Q)}\;\mathrm{app}_1 \qquad \frac{Q \rightsquigarrow Q'}{(P\,Q) \;\rightsquigarrow\; (P\,Q')}\;\mathrm{app}_2$$

A rule says: whenever what is above the bar holds, so does what is below. The rules $(\mathsf{K})$ and $(\mathsf{S})$ have nothing above the bar, so they hold outright, whenever a term has their shape; they are the two ways to rewrite. In words, $\mathsf{K}$ keeps the first of two arguments and drops the second, and $\mathsf{S}$ hands its third argument to both of the others and applies the first result to the second.

The rule $(\mathrm{app}_1)$ says that if $P \rightsquigarrow P'$, then $P\,Q \rightsquigarrow P'\,Q$: a step in the function is a step of the application. The rule $(\mathrm{app}_2)$ says the same of the argument. Since every term is built by application, between them they reach every part of a term. To show that $M \rightsquigarrow N$ is to find rules that stack up to it, each bar's conclusion standing above the next bar.

The rules are written with every bracket, because the brackets say which term is meant. With the convention, $(\mathsf{K})$ reads $\mathsf{K}\,P\,Q \rightsquigarrow P$ and $(\mathsf{S})$ reads $\mathsf{S}\,P\,Q\,R \rightsquigarrow P\,R\,(Q\,R)$; the pair round $Q\,R$ stays because that grouping is not to the left. Here are four steps, each written first with its brackets, as the rules see it, and then as we write it.

| **With the brackets** | **As written** | **Rules** |
|---|---|---|
| $((\mathsf{K}\,x)\,y) \rightsquigarrow x$ | $\mathsf{K}\,x\,y \rightsquigarrow x$ | $(\mathsf{K})$ |
| $(((\mathsf{K}\,x)\,y)\,z) \rightsquigarrow (x\,z)$ | $\mathsf{K}\,x\,y\,z \rightsquigarrow x\,z$ | $(\mathsf{K})$, then $(\mathrm{app}_1)$ |
| $(((\mathsf{S}\,x)\,y)\,z) \rightsquigarrow ((x\,z)\,(y\,z))$ | $\mathsf{S}\,x\,y\,z \rightsquigarrow x\,z\,(y\,z)$ | $(\mathsf{S})$ |
| $(x\,((\mathsf{K}\,y)\,z)) \rightsquigarrow (x\,y)$ | $x\,(\mathsf{K}\,y\,z) \rightsquigarrow x\,y$ | $(\mathsf{K})$, then $(\mathrm{app}_2)$ |

The first row is the rule $(\mathsf{K})$ itself, with $x$ for $P$ and $y$ for $Q$. In the second, the brackets show that $\mathsf{K}$ has two arguments, not three, and that $((\mathsf{K}\,x)\,y)$ is the function in a larger application whose argument is $z$; so $(\mathrm{app}_1)$ carries the step. In the fourth, the part rewritten is the argument, and $(\mathrm{app}_2)$ carries it. Keep the hidden brackets in mind in everything that follows: most of the mistakes you can make in this chapter are mistakes about where they go.

**Example 1.3** (a run, rule by rule)**.** Run the term $\mathsf{S}\,(\mathsf{K}\,x\,y)\,(\mathsf{K}\,u)\,w$, taking the step inside it first. (It could also step by $(\mathsf{S})$ straight away; §1.2 is about

such choices.) The first step rewrites $\mathsf{K}\,x\,y$, deep inside the term, and four rules stack up to it:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{K}\,x\,y \;\rightsquigarrow\; x}\,\mathsf{K}}{\mathsf{S}\,(\mathsf{K}\,x\,y) \;\rightsquigarrow\; \mathsf{S}\,x}\,\mathrm{app}_2}{\mathsf{S}\,(\mathsf{K}\,x\,y)\,(\mathsf{K}\,u) \;\rightsquigarrow\; \mathsf{S}\,x\,(\mathsf{K}\,u)}\,\mathrm{app}_1}{\mathsf{S}\,(\mathsf{K}\,x\,y)\,(\mathsf{K}\,u)\,w \;\rightsquigarrow\; \mathsf{S}\,x\,(\mathsf{K}\,u)\,w}\,\mathrm{app}_1$$

Read it from the top. ($\mathsf{K}$) rewrites $\mathsf{K}\,x\,y$ to $x$. Then ($\mathrm{app}_2$) carries the step into the argument of $\mathsf{S}$, and ($\mathrm{app}_1$), twice, carries it out past $\mathsf{K}\,u$ and then past $w$, one application at a time. Each bar below the first puts back one pair of the hidden brackets.

The second step needs one rule, since the whole term now has the shape of the left-hand side of ($\mathsf{S}$), with $x$ for $P$, $\mathsf{K}\,u$ for $Q$ and $w$ for $R$:

$$\dfrac{}{\mathsf{S}\,x\,(\mathsf{K}\,u)\,w \;\rightsquigarrow\; x\,w\,(\mathsf{K}\,u\,w)}\,\mathsf{S}$$

The third rewrites the argument $\mathsf{K}\,u\,w$, so ($\mathrm{app}_2$) carries it:

$$\dfrac{\dfrac{}{\mathsf{K}\,u\,w \;\rightsquigarrow\; u}\,\mathsf{K}}{x\,w\,(\mathsf{K}\,u\,w) \;\rightsquigarrow\; x\,w\,u}\,\mathrm{app}_2$$

Now $x\,w\,u$ has nothing left to rewrite, and the run is over.

Nobody draws stacks like these: a step is a step, and the rules only say which ones there are. We write a run as a chain instead, one step per line, with the rules it used on the right:

$$\begin{aligned}
\mathsf{S}\,(\mathsf{K}\,x\,y)\,(\mathsf{K}\,u)\,w &\;\rightsquigarrow\; \mathsf{S}\,x\,(\mathsf{K}\,u)\,w && (\mathsf{K}),\ \text{then } (\mathrm{app}_2),\ \text{then } (\mathrm{app}_1)\ \text{twice}\\
&\;\rightsquigarrow\; x\,w\,(\mathsf{K}\,u\,w) && (\mathsf{S})\\
&\;\rightsquigarrow\; x\,w\,u && (\mathsf{K}),\ \text{then } (\mathrm{app}_2).
\end{aligned}$$

Often even that is more than we need, and we name only the step at the top of the stack: "the $\mathsf{K}$ step", "the $\mathsf{S}$ step". The rules for getting inside a term are always ($\mathrm{app}_1$) and ($\mathrm{app}_2$), as many times as it takes.

Such a chain is a *run*: a sequence of steps

$$M \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \cdots \rightsquigarrow M_n.$$

We write $M \rightsquigarrow^* N$, read "$M$ runs to $N$", when some run from $M$ ends at $N$. The run may be of any length, zero included, so $M \rightsquigarrow^* M$ always. The run above shows that $\mathsf{S}\,(\mathsf{K}\,x\,y)\,(\mathsf{K}\,u)\,w \rightsquigarrow^* x\,w\,u$.

Here's a way to picture a term: it is a machine with a single slot. You put one thing in, and something may come out. Often what comes out is another machine, with a slot of its own, waiting for the next thing. The constant $\mathsf{K}$, for instance, is a machine that wants two things, but it can take them only one at a time. Put $f$ in, and out comes a new machine, $\mathsf{K}\,f$, which ignores whatever it's given next and hands back $f$. Put $g$ into that, and out comes $f$. So $\mathsf{K}\,f\,g$ is two machines used one after the other, $(\mathsf{K}\,f)\,g$, and that is the bracketing the convention supplies.

In the same way $\mathsf{S}$ is a machine that wants three things: $\mathsf{S}\,M$ is a machine waiting for two more, $\mathsf{S}\,M\,N$ a machine waiting for one, and $\mathsf{S}\,M\,N\,P$ is the moment it has all three and does its work. Nothing is ever fed two things

at once. Taking arguments one at a time is called *currying*,[4] and it's why application groups to the left: a call with several arguments is a chain of single applications, $f\,x\,y\,z$ is the commonest shape there is, and grouping to the left lets it go without brackets.

A machine still short of an input does nothing: it waits, and there is no step to take in it. The term $\mathsf{K}\,x$ is one, and so are $\mathsf{S}\,x\,y$ and $\mathsf{K}$ on its own. None of them is wrong; each is a function waiting for another argument. Why not simply say in the grammar that $\mathsf{K}$ takes two arguments and $\mathsf{S}$ three? One could, but it would bring a tiresome bureaucracy for no gain, and it would hide something the machines make plain: $\mathsf{K}\,f$, a function with one argument still to come, is a perfectly good term in its own right.

The function that returns its argument unchanged isn't one of the two constants, but the language has it all the same. Write $\mathsf{I}$ for $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$, a name for a term and not a new constant. Whatever $x$ is, $\mathsf{I}\,x \rightsquigarrow^* x$, by an $\mathsf{S}$ step and then a $\mathsf{K}$ step.

**Example 1.4** (the identity)**.** Here is the run of $\mathsf{I}$ on an argument:

$$\begin{aligned} \mathsf{S}\,\mathsf{K}\,\mathsf{K}\,x &\rightsquigarrow \mathsf{K}\,x\,(\mathsf{K}\,x) && \text{the } \mathsf{S} \text{ step, with } P = Q = \mathsf{K} \text{ and } R = x \\ &\rightsquigarrow x && \text{the } \mathsf{K} \text{ step: keep } x, \text{ drop } \mathsf{K}\,x. \end{aligned}$$

**Example 1.5** (a puzzle about brackets)**.** The bracket convention has to be taken seriously: applied carelessly, it seems to give one term two different results. Run $\mathsf{K}\,\mathsf{I}\,f\,x$, and to keep the runs short count the two steps of $\mathsf{I}$ as one. Two runs seem to be on offer:

$$\begin{aligned} \mathsf{K}\,\mathsf{I}\,f\,x &\rightsquigarrow \mathsf{I}\,x \rightsquigarrow x && \text{the } \mathsf{K} \text{ step, then the } \mathsf{I} \text{ step,} \\ \mathsf{K}\,\mathsf{I}\,f\,x &\rightsquigarrow \mathsf{K}\,f\,x \rightsquigarrow f && \text{the } \mathsf{I} \text{ step, then the } \mathsf{K} \text{ step.} \end{aligned}$$

One of these is not a run at all, and it is the second. To see why, put the brackets in. The term is $(((\mathsf{K}\,\mathsf{I})\,f)\,x)$: $\mathsf{K}$ is applied to $\mathsf{I}$, the result to $f$, and that result to $x$.

Nowhere in the term is $\mathsf{I}$ applied to $f$. They stand next to each other on the page, but $\mathsf{I}$ is one argument of $\mathsf{K}$ and $f$ is another, so the $\mathsf{I}$ step of the second run has nothing to rewrite. Only one step is available, and $\mathsf{K}\,\mathsf{I}\,f\,x \rightsquigarrow^* x$. The second run was really about a different term, $\mathsf{K}\,(\mathsf{I}\,f)\,x$, in which the brackets put $\mathsf{I}$ and $f$ together; Exercise 1.8 runs it.

Terms built from constants by application, together with rules for rewriting them, make a *combinatory algebra*, and $\mathsf{K}$ and $\mathsf{S}$ are its basic combinators.

1.2. **Confluence.** The rules ($\text{app}_1$) and ($\text{app}_2$) let a step happen anywhere inside a term, so a term can often step in more than one way. Take

$$E = \mathsf{K}\,x\,y\,(\mathsf{S}\,x\,y\,z),$$

[4] After Haskell Curry, who also has a programming language named after him, and who used the idea so much that the name stuck. Schönfinkel did it before Curry (and Frege, in passing, before both); "schönfinkeling" never caught on.

which is $\mathsf{K}\,x\,y$ applied to $\mathsf{S}\,x\,y\,z$. A run can start at the front or in the argument, and here are both:

$$\begin{aligned} E &\rightsquigarrow x\,(\mathsf{S}\,x\,y\,z) && (\mathsf{K}),\ \text{then } (\text{app}_1)\\ &\rightsquigarrow x\,(x\,z\,(y\,z)) && (\mathsf{S}),\ \text{then } (\text{app}_2),\\ E &\rightsquigarrow \mathsf{K}\,x\,y\,(x\,z\,(y\,z)) && (\mathsf{S}),\ \text{then } (\text{app}_2)\\ &\rightsquigarrow x\,(x\,z\,(y\,z)) && (\mathsf{K}),\ \text{then } (\text{app}_1). \end{aligned}$$

Both runs end at $x\,(x\,z\,(y\,z))$, and there they stop: nothing in that term can be rewritten.

The term $E$ has two first steps because it contains two parts that can each be rewritten on their own. Such a part is a *redex*, short for *reducible expression*: a term with the shape of the left-hand side of $(\mathsf{K})$ or of $(\mathsf{S})$, that is, $\mathsf{K}$ with two arguments or $\mathsf{S}$ with three. What the rule puts in its place is the *reduct*. The redexes of $E$ are $\mathsf{K}\,x\,y$ and $\mathsf{S}\,x\,y\,z$, and their reducts are $x$ and $x\,z\,(y\,z)$. The term $E$ itself is not a redex: it is $\mathsf{K}\,x\,y$ applied to something more.

Whether a part of a term is a redex depends on the hidden brackets, so it pays to have a routine for finding them.

1. Read the term from the left. Its first symbol is its *head*, and the terms that follow it, each a single symbol or a bracketed group, are the head's arguments.
2. If the head is $\mathsf{K}$ with at least two arguments, or $\mathsf{S}$ with at least three, then the head and its first two or three arguments make a redex.
3. Then do the same inside every bracketed group.

Here is the routine run on a few terms.

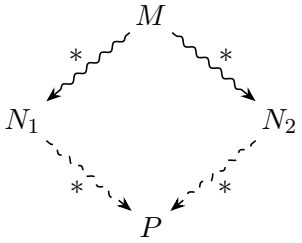
| **Term** | **Its redexes** |
|---|---|
| $\mathsf{K}\,x\,y$ | the whole term; the step gives $x$ |
| $\mathsf{K}\,(x\,y)$ | none: $\mathsf{K}$ has only one argument |
| $\mathsf{K}\,x\,y\,z$ | $\mathsf{K}\,x\,y$, at the front; the step gives $x\,z$ |
| $x\,\mathsf{K}\,y\,z$ | none: the head is $x$, and $\mathsf{K}$ is just one of its arguments |
| $x\,(\mathsf{K}\,y\,z)$ | $\mathsf{K}\,y\,z$, inside the brackets; the step gives $x\,y$ |
| $\mathsf{S}\,(\mathsf{K}\,x)\,(\mathsf{K}\,y)$ | none: $\mathsf{S}$ has two arguments, and each $\mathsf{K}$ has one |
| $\mathsf{K}\,(\mathsf{K}\,x\,y)\,(\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,z)$ | three: the whole term, $\mathsf{K}\,x\,y$, and $\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,z$ |

The fourth and fifth rows differ by one pair of brackets. The third shows that a head may have more arguments than it needs: the redex is then the head with its first few, and the rest, here $z$, are carried along unchanged.

The two runs of $E$ passed through different terms and arrived at the same one. Is that luck? It had better not be. If a term could run to two different results, "the result of running $M$" would mean nothing until one also said in what order, and the answer to a computation would depend on the implementation. It doesn't, and the reason is a theorem.

**Theorem 1.6** (Church–Rosser; on loan)**.** *If $M \rightsquigarrow^* N_1$ and $M \rightsquigarrow^* N_2$, then there is a term $P$ such that $N_1 \rightsquigarrow^* P$ and $N_2 \rightsquigarrow^* P$.*

It's usually remembered as a picture. The two runs at the top are given; the theorem promises the two dashed runs at the bottom.




This property is called *confluence*. In words: however far two runs of the same term have gone apart, each can be continued so that they come back together.[5]

Some terms can step in more than one way; others can't step at all. A term with no redex is in *normal form*. The machines waiting for inputs, such as $\mathsf{K}\,x$ and $\mathsf{S}\,x\,y$, are in normal form, and so are $\mathsf{K}$, $\mathsf{S}$ and the variables on their own. A run that reaches a normal form is over, and the normal form is its *result*: what the run hands back. What's normal about it? Nothing deep: it is the standard form a term settles into, if it settles at all.

The routine for finding redexes also says which terms have none. A head that cannot fire, applied to arguments that have no redexes of their own, gives a normal form, and nothing else does.

**Lemma 1.7** (the shape of a normal form)**.** *A term is in normal form exactly when it has one of the following shapes, where $k \geq 0$ and $N_1, \ldots, N_k$ are terms in normal form:*

*(i)* $x\,N_1 \cdots N_k$*, for a variable* $x$*;*
*(ii)* $\mathsf{K}$*, or* $\mathsf{K}\,N_1$*;*
*(iii)* $\mathsf{S}$*, or* $\mathsf{S}\,N_1$*, or* $\mathsf{S}\,N_1\,N_2$*.*

*In particular, a combinator is in normal form exactly when it is one of* $\mathsf{K}$*,* $\mathsf{K}\,N$*,* $\mathsf{S}$*,* $\mathsf{S}\,N$ *and* $\mathsf{S}\,N\,N'$*, where* $N$ *and* $N'$ *are combinators in normal form.*

In words: a variable never fires, however many arguments it has; $\mathsf{K}$ fires on two and $\mathsf{S}$ on three; so a normal form is a head with too few arguments to fire, or a variable, applied to normal forms.

*Proof.* Every term is its head $h$, a constant or a variable, applied to its arguments: $h\,M_1 \cdots M_k$. Its parts are the terms $h\,M_1 \cdots M_j$ for $j \leq k$, which run from the head along the arguments, and the parts of the arguments $M_1, \ldots, M_k$. So its redexes are of two kinds.

- *At the head.* A part $h\,M_1 \cdots M_j$ is a redex exactly when $h$ is $\mathsf{K}$ and $j = 2$, or $h$ is $\mathsf{S}$ and $j = 3$. There is such a part exactly when $h$ is $\mathsf{K}$ and $k \geq 2$, or $h$ is $\mathsf{S}$ and $k \geq 3$.
- *Inside an argument.* These are the redexes of $M_1, \ldots, M_k$.

The term has no redex, then, exactly when $h$ is a variable, or $\mathsf{K}$ with at most one argument, or $\mathsf{S}$ with at most two, and each $M_i$ is in normal form.

[5] Church and Rosser proved it in 1936 (Church and Rosser, 1936), for a close cousin of this calculus, the $\lambda$-calculus, and it holds here too.

Those are the three shapes. A combinator has no variables, so the first shape cannot occur in it, and the shapes that remain are the five listed. □

So $x\,(y\,z)$ and $\mathsf{S}\,(\mathsf{K}\,x)\,(\mathsf{K}\,y)$ are normal forms, and $\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,\mathsf{K}$ is not.

Confluence has two consequences for results, and both are short enough to prove on the spot.

**Corollary 1.8** (one result at most)**.** *If $M \rightsquigarrow^* N_1$ and $M \rightsquigarrow^* N_2$, and $N_1$ and $N_2$ are both in normal form, then $N_1$ and $N_2$ are the same term.*

*Proof.* Theorem 1.6 gives a $P$ with $N_1 \rightsquigarrow^* P$ and $N_2 \rightsquigarrow^* P$. But no step can be taken in a normal form, so the only run from $N_1$ is the run of no steps, and $P$ is $N_1$. In the same way $P$ is $N_2$, and so $N_1$ and $N_2$ are the same term. □

This is what lets us speak of *the* result of a term: whichever redex a run takes first, it can arrive at only one place.

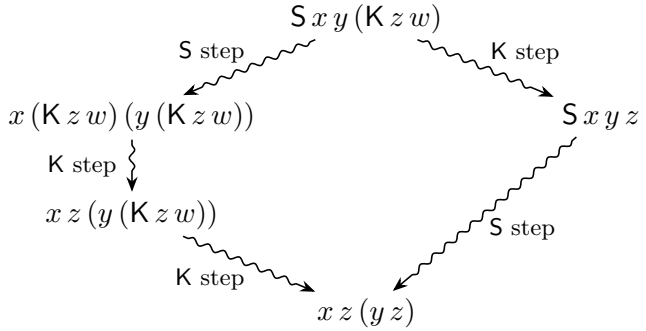
**Corollary 1.9** (no step is a wrong turn)**.** *If $M \rightsquigarrow^* N$, where $N$ is in normal form, and $M \rightsquigarrow^* M'$, then $M' \rightsquigarrow^* N$.*

*Proof.* Theorem 1.6 gives a $P$ with $M' \rightsquigarrow^* P$ and $N \rightsquigarrow^* P$, and since $N$ is in normal form, $P$ is $N$. □

In words: however you have run $M$ so far, and in whatever order, its result is still within reach. A poor choice can put the result further off; no choice can put it out of reach, although a run that keeps on choosing poorly need never arrive.

The two runs of $E$ took two steps each before they met, but the two sides of a diamond need not be the same length. An $\mathsf{S}$ step copies its third argument, and with it every redex inside; a $\mathsf{K}$ step throws its second argument away, and every redex inside with it. So a single step on one side of a diamond may need two steps on the other side, or none. That is why Theorem 1.6 is not obvious: the small diamonds can't be fitted together like tiles, and the proof has to keep track of a redex and all its copies at once.

**Example 1.10** (two lopsided diamonds)**.** Take the term $\mathsf{S}\,x\,y\,(\mathsf{K}\,z\,w)$. It has two redexes: the whole term, which is $\mathsf{S}$ with three arguments, and $\mathsf{K}\,z\,w$, inside the third of them. The two runs below take each first.




Take the $\mathsf{S}$ step first, on the left, and the third argument is handed to both of the others: there are now two copies of the redex $\mathsf{K}\,z\,w$, and it takes two $\mathsf{K}$ steps to finish. Take the $\mathsf{K}$ step first, on the right, and a single $\mathsf{S}$ step

finishes. So the two sides take different numbers of steps, and they meet all the same, at $x\,z\,(y\,z)$.

The rule for $\mathsf{K}$ does the opposite. In $\mathsf{K}\,x\,(\mathsf{K}\,y\,z)$ the outer step throws the inner redex away and gives $x$ at once, while the inner step gives $\mathsf{K}\,x\,y$, and one more step is needed to reach the same $x$.

1.3. **Evaluation Strategies.** An implementation of the combinator calculus has to decide which redex to rewrite when a term has more than one. By Corollary 1.8 the choice can't change the result, but it can change how much work the result takes. The order an implementation follows is its *evaluation strategy*. Two strategies matter, and between them they divide the languages you use.

- *Strict, or eager.* Rewrite an innermost redex first, one with no redex inside it, and work outwards. In programming terms: run the arguments before the call.[6]
- *Lazy.* Rewrite an outermost redex first, one that sits inside no other, and work inwards. In programming terms: run an argument only when the function needs it.[7]

The two strategies part company on the terms of the diamonds above. In $\mathsf{S}\,x\,y\,(\mathsf{K}\,z\,w)$ a strict run takes the inner $\mathsf{K}$ step first, down the right of the diamond, and finishes in two steps; a lazy run takes the $\mathsf{S}$ step first, copies the unrun argument, and needs three. In $\mathsf{K}\,x\,(\mathsf{K}\,y\,z)$ it is the other way round: the strict run spends a step on an argument that $\mathsf{K}$ then throws away, two steps in all, and the lazy run needs one.

Why do different languages make different choices? Because each strategy wins in one case and loses in the other. Strict does well when an argument is used more than once, since it runs the argument once and copies the result, and badly when an argument is thrown away, since the work on it was for nothing. Lazy is the reverse.[8]

Nor does either strategy give you anything the other can't. A strict language can be made lazy by hand: instead of passing an expression, pass a function that computes it, and call that function only when the value is wanted. A lazy language can be made strict by hand too, by forcing a value before it is passed on. So what divides the strategies is the *default*, and what the default costs you each time you want the other.

---

For computer scientists. Here is laziness made by hand in Python, with a conditional written as an ordinary function. Handed its two branches as expressions, as in `my_if(n == 0, 1, n * fact(n - 1))`, it would run both before choosing, and the recursive call in the branch not taken would never end. Handed them as functions, it runs only the one it picks:

[6] Python, Java, C, JavaScript and ML work this way.

[7] Haskell works this way.

[8] A lazy implementation can repair half of its weakness by *sharing*: it copies a pointer to the unrun argument instead of the argument itself, so that the two copies of $\mathsf{K}\,z\,w$ above are one object, run once, when it is first needed. This is *call by need*, and it is what Haskell does; the price is the bookkeeping for each suspended argument, called a *thunk*. The other two names you'll meet are *call by value* (strict) and *call by name* (lazy, without sharing).

```
def my_if(b, m, n): return m() if b else n()
def fact(n): return my_if(n == 0, lambda: 1, lambda: n * fact(n - 1))
fact(5)    # 120
```

The `lambda:`s are the laziness. Going the other way, Haskell's `seq` forces a value before it is passed on.

---

The default is a bet on which behaviour you will want more often. Three considerations bear on the bet.

1. *Predictable cost favours strict.* In a strict language the work for an argument is paid at the call, and you can read the cost off the line. In a lazy language the cost of an expression depends on who demands its value and when, and unrun work can pile up unseen: adding up a long list with a lazy accumulator can build a chain of a million unrun additions before it performs one, and the usual remedy is to force the accumulator by hand.
2. *Effects favour strict.* Theorem 1.6 licenses any order of steps only where nothing happens on the side. Printing, writing a file, reading input and raising an error are all things that happen on the side. In a strict language they happen in the order written: arguments left to right, then the call. In a lazy language the moment of an effect depends on when its result is demanded, which depends on the rest of the program.[9]
3. *Modularity favours lazy.* A lazy language lets one part of a program describe an enormous or infinite list while another part consumes as much of it as it needs. Strict languages borrow the idea for exactly that reason.[10]

Everything in this section has assumed that the runs end. Confluence says that two runs of a term can always be brought back together; it doesn't say that either of them ever stops. Hold that thought until §1.5.

1.4. **Programming.** The promise so far has been that this tiny calculus captures what a computer can do, and it is time to make good on it. Running a term is plainly a form of computation: a term goes in and a normal form comes out. So we may ask what a language as small as this one can compute. The answer comes in two parts, and both are surprising. First, though, we need a way to write programs in it that a person can read.

A term such as $\mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}$ is hard on the eye, and the same patterns turn up again and again. So we'll give a pattern of terms a name and write the name in place of the pattern. That's all a definition will be: an abbreviation for a term, with blanks in it to be filled.

**Definition 1.11** (definitions)**.** A *definition* is an equation

$$f\ x_1 \cdots x_n \ \coloneqq\ M,$$

where:

- $f$ is a new name;

[9] Haskell's answer is to keep effects out of ordinary expressions and to mark them in the type, which is workable and costs something in every program that prints.

[10] In Haskell `take 3 [1..]` is `[1,2,3]`. Python's generators and Java's streams are lazy sequences inside strict languages.

- $x_1, \ldots, x_n$ are distinct variables;
- $M$ is a term built from constants, names defined earlier, and the variables $x_1, \ldots, x_n$ alone.

Once defined, the name $f$ may be used as a term, and it steps by the rule

$$\frac{}{((\cdots((f\,P_1)\,P_2)\cdots)\,P_n) \;\rightsquigarrow\; M \text{ with } P_i \text{ for each } x_i}\ \text{def}$$

alongside the rules of Definition 1.2.

The symbol $\coloneqq$ is read "is defined to be". It gives something a name, which is a different job from the $=$ of arithmetic.[11] The rule (def) says that wherever $f$ stands with its $n$ arguments, it may be replaced by the right-hand side, with the arguments written in for the parameters; with ($\text{app}_1$) and ($\text{app}_2$), that can happen anywhere in a term. That is all a function call is. The $\mathsf{I}$ above is now official: it is the definition $\mathsf{I} \coloneqq \mathsf{S}\,\mathsf{K}\,\mathsf{K}$, with no parameters. Three more will be useful:

$$\mathit{twice}\; f\, x \coloneqq f\,(f\,x),$$
$$\mathit{comp}\; f\, g\, x \coloneqq f\,(g\,x),$$
$$\mathit{flip}\; f\, x\, y \coloneqq f\, y\, x.$$

The first applies a function twice, the second chains two functions into one, and the third swaps the order of two arguments.

**Example 1.12** (definitions)**.** Read the definition of *twice* against the clauses of Definition 1.11:

- *twice* is a new name;
- $f$ and $x$ are distinct variables;
- the body $f\,(f\,x)$ is built from those two variables by application alone, which the clause for application in Definition 1.1 allows.

A definition can't reach outside itself. The right-hand side may use the two constants, the parameters and names defined earlier, and nothing else: no numbers, no strings, no `if`, nothing from a library. A name can't occur in its own definition either, so there's no recursion. That looks crippling. It turns out not to be, and the rest of this section is the evidence.

To watch definitions at work we want something visible to come out, and the language has no numbers. So for a page or two we allow ourselves the integers $0, 1, 2, \ldots$ and the operator $+$ as ready-made constants, with the obvious step $m + n \rightsquigarrow$ the sum, and we let definitions use them. They aren't terms of Definition 1.1, so this is a debt, and the numerals pay it: the numbers need not be ready-made, since a language with no numbers can make its own, by making each number a function. A number $n$ is "do it $n$ times": a function that takes an $f$ and an $x$ and applies $f$ to $x$ that many times.

[11] In C, `=` means assignment, `==` means equality, and `===` means JavaScript. Our $\coloneqq$ means "I am naming this", and claims nothing.

**Definition 1.13** (numerals)**.** For each natural number $n$, the *numeral* $\overline{n}$ is the name given by the definition

$$\overline{n}\,f\,x \;\coloneqq\; \underbrace{f\,(f\,(\cdots(f}_{n}\;x)\cdots)).$$

So $\overline{0}\,f\,x \coloneqq x$, $\overline{1}\,f\,x \coloneqq f\,x$ and $\overline{2}\,f\,x \coloneqq f\,(f\,x)$.

Arithmetic is definitions too. To add one to a number, apply $f$ once more; to add two numbers, apply $f$ the first number of times to the result of applying it the second number of times:

$$\mathit{succ}\;n\,f\,x \coloneqq f\,(n\,f\,x) \qquad \text{and} \qquad \mathit{add}\;m\,n\,f\,x \coloneqq m\,f\,(n\,f\,x).$$

A numeral is a function, and a function can't be read off the page until it is given something to work on. So give it the ready-made 0 and the ready-made add-one, $\mathit{inc}\;y \coloneqq y+1$. Then $\overline{n}\;\mathit{inc}\,0$ adds one to 0, $n$ times over, and out comes the digit $n$: the pair *inc*, 0 is a printer for numerals.

**Example 1.14** (addition, printed)**.** Here is $\mathit{add}\,\overline{2}\,\overline{1}$ printed, one step per line, with the definition unfolded at each named on the right:

$$\begin{aligned}
\mathit{add}\,\overline{2}\,\overline{1}\;\mathit{inc}\,0 &\rightsquigarrow \overline{2}\;\mathit{inc}\,(\overline{1}\;\mathit{inc}\,0) && \mathit{add}\\
&\rightsquigarrow \mathit{inc}\,(\mathit{inc}\,(\overline{1}\;\mathit{inc}\,0)) && \overline{2}\\
&\rightsquigarrow \mathit{inc}\,(\mathit{inc}\,(\mathit{inc}\,0)) && \overline{1}\\
&\rightsquigarrow \mathit{inc}\,(\mathit{inc}\,(0+1)) && \mathit{inc}\text{, innermost}\\
&\rightsquigarrow \mathit{inc}\,(\mathit{inc}\,1)\\
&\rightsquigarrow \mathit{inc}\,(1+1) && \mathit{inc}\text{, innermost}\\
&\rightsquigarrow \mathit{inc}\,2\\
&\rightsquigarrow 2+1\\
&\rightsquigarrow 3.
\end{aligned}$$

Nine steps, and the answer is 3: applied to *inc* and 0, the program $\mathit{add}\,\overline{2}\,\overline{1}$ computes exactly what $\overline{3}$ computes. At the fourth line any of the three occurrences of *inc* could have been unfolded, and by Corollary 1.8 the choice can't change the answer. Exercise 1.1 does the same for multiplication.

A boolean is a choice between two things, so let *true* be the function that keeps the first of two arguments and *false* the one that keeps the second:

$$\mathit{true} \;\coloneqq\; \mathsf{K} \qquad \text{and} \qquad \mathit{false} \;\coloneqq\; \mathsf{K}\,\mathsf{I}.$$

With that choice, `if b then M else N` is simply $b\,M\,N$: the boolean applied to the two branches, with $\mathit{true}\,M\,N \rightsquigarrow^* M$ and $\mathit{false}\,M\,N \rightsquigarrow^* N$. There is no `if` in the language, and none is needed: the boolean *is* the conditional.

**Example 1.15** (booleans)**.** Here are the two cases run:

$$\begin{aligned}
\mathit{true}\,M\,N \;=\; \mathsf{K}\,M\,N &\rightsquigarrow M && \text{keep the first}\\
\mathit{false}\,M\,N \;=\; \mathsf{K}\,\mathsf{I}\,M\,N &\rightsquigarrow \mathsf{I}\,N && \mathsf{K}\text{ keeps }\mathsf{I}\text{, drops }M\\
&\rightsquigarrow^* N && \text{the run of }\mathsf{I}.
\end{aligned}$$

So `if false then 10 else 20` is $\mathit{false}\;10\;20 = \mathsf{K}\,\mathsf{I}\,10\,20 \rightsquigarrow \mathsf{I}\,20 \rightsquigarrow^* 20$.

So much for writing programs. Now for the two surprises. The first concerns the defined names. It may look as though they are where the work gets done, and the two constants are scaffolding. It is the other way round: definitions are a convenience, and we can do without them.

**Theorem 1.16** (two constants suffice)**.** *Let $f\,x_1 \cdots x_n \coloneqq M$ be a definition whose right-hand side $M$ is built from $\mathsf{K}$, $\mathsf{S}$ and the parameters alone. Then there is a combinator $F$ in $\mathsf{K}$ and $\mathsf{S}$ alone such that, for all terms $P_1, \ldots, P_n$, the term $F\,P_1 \cdots P_n$ runs to $M$ with $P_i$ for each $x_i$.*

In words: anything you can write with parameters, you can write without them. A definition that uses names defined earlier is handled by replacing those names by their combinators first. For instance, $\mathsf{B} \coloneqq \mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}$ does the work of *comp*, since $\mathsf{B}\,f\,g\,x \rightsquigarrow^* f\,(g\,x)$, and $\mathsf{S}\,\mathsf{B}\,\mathsf{I}$ does the work of *twice*, since $\mathsf{S}\,\mathsf{B}\,\mathsf{I}\,f\,x \rightsquigarrow^* f\,(f\,x)$.

**Example 1.17** (composition)**.** Here is $\mathsf{B}$ run on three arguments:

$$\begin{aligned}
\mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}\,f\,g\,x &\rightsquigarrow \mathsf{K}\,\mathsf{S}\,f\,(\mathsf{K}\,f)\,g\,x && \text{the } \mathsf{S} \text{ step}\\
&\rightsquigarrow \mathsf{S}\,(\mathsf{K}\,f)\,g\,x && \text{the } \mathsf{K} \text{ step}\\
&\rightsquigarrow \mathsf{K}\,f\,x\,(g\,x) && \text{the } \mathsf{S} \text{ step}\\
&\rightsquigarrow f\,(g\,x) && \text{the } \mathsf{K} \text{ step.}
\end{aligned}$$

Four steps, and $\mathsf{B}\,f\,g\,x \rightsquigarrow^* f\,(g\,x)$, without a single parameter.

**Example 1.18** (a definition without parameters)**.** The definition of *twice* has two parameters and uses $f$ twice; the combinator $\mathsf{S}\,\mathsf{B}\,\mathsf{I}$ has none, and does the same job:

$$\begin{aligned}
\mathsf{S}\,\mathsf{B}\,\mathsf{I}\,f\,x &\rightsquigarrow \mathsf{B}\,f\,(\mathsf{I}\,f)\,x && \text{the } \mathsf{S} \text{ step, with } P = \mathsf{B},\ Q = \mathsf{I} \text{ and } R = f\\
&\rightsquigarrow^* f\,(\mathsf{I}\,f\,x) && \text{the run of } \mathsf{B}\\
&\rightsquigarrow^* f\,(f\,x) && \text{the run of } \mathsf{I}.
\end{aligned}$$

The $\mathsf{S}$ is there because $f$ is used twice; the work of $\mathsf{B}$ and $\mathsf{I}$ is to get each copy to where it is wanted.

The idea behind the theorem is that the variables in a definition are bookkeeping, and that two fixed combinations do all the bookkeeping there is.[12] We won't prove it here; the proof comes with the Deduction Theorem, in the chapter on Hilbert proofs, and it's a surprise.

Now we can pay the debt of the ready-made numbers. The numerals need nothing but $\mathsf{K}$ and $\mathsf{S}$: take $\overline{0} \coloneqq \mathsf{K}\,\mathsf{I}$ and $\mathit{succ} \coloneqq \mathsf{S}\,\mathsf{B}$, so that $\overline{1}$ is $\mathsf{S}\,\mathsf{B}\,(\mathsf{K}\,\mathsf{I})$, and so on. (Check that $\mathsf{S}\,\mathsf{B}\,n\,f\,x \rightsquigarrow^* f\,(n\,f\,x)$.) Then $\overline{0}$ and *false* are the same combinator.[13] And *add*, a definition with parameters, is a combinator by Theorem 1.16. The ready-made *inc* and 0 were only ever the printer. The step $m + n \rightsquigarrow$ the sum is a shortcut, like a processor's `ADD` instruction; real languages build numbers in because it's fast, and nothing forces them to. We keep the ready-made pieces with a clear conscience.

[12] The theorem is Moses Schönfinkel's (Schönfinkel, 1924).

[13] C programmers have suspected this for years.

The second surprise is that what is left computes everything that can be computed. A combinator $F$ *computes* a function $\phi$ on the natural numbers if, for every $n$, the term $F\,\overline{n}\,f\,x$ runs to $f$ applied $\phi(n)$ times to $x$.

**Theorem 1.19** (Turing completeness)**.** *The functions on the natural numbers that combinators in* $\mathsf{K}$ *and* $\mathsf{S}$ *compute are exactly the functions that Turing machines compute.*

Between them, this theorem and the last say that this small language is no toy: anything any computer can compute, a combinator in $\mathsf{K}$ and $\mathsf{S}$ can compute, and anything you can write with parameters you can write without them. This theorem we won't prove at all.[14]

Recursion is the one thing the definitions seem to lack. No name may occur in its own definition, but a function may be applied to itself, and that is enough. Define

$$\mathit{self}\ f\,x \coloneqq f\,(x\,x) \qquad \text{and} \qquad \mathsf{Y}\,f \coloneqq \mathit{self}\ f\,(\mathit{self}\ f),$$

neither of which mentions itself, and run $\mathsf{Y}$ on any $f$:

$$\begin{aligned} \mathsf{Y}\,f &\rightsquigarrow \mathit{self}\ f\,(\mathit{self}\ f) && \mathsf{Y} \\ &\rightsquigarrow f\,(\mathit{self}\ f\,(\mathit{self}\ f)) && \mathit{self}\text{, with } \mathit{self}\ f \text{ for } x. \end{aligned}$$

Write $\Theta$ for $\mathit{self}\ f\,(\mathit{self}\ f)$. The second line says that $\Theta \rightsquigarrow f\,\Theta$: $\Theta$ hands $f$ a copy of itself, and that copy, when it is run, hands $f$ another, as many times as the run asks for.[15] A function that is to call itself takes the copy as an extra first parameter, and $\mathsf{Y}$ supplies it.

**Example 1.20** (factorial)**.** A larger example is factorial, as you'd write it on a whiteboard:

```
fact(n):
    if n = 0 then return 1
    else return n * fact(n - 1)
```

Four things in it aren't in our language:

- a parameter;
- a name that calls itself;
- a conditional;
- arithmetic.

And yet it becomes

$$F = \mathsf{S}\,(\mathsf{K}\,(\mathsf{S}\,A))\,(\mathsf{S}\,(\mathsf{K}\,(\mathsf{S}\,C))\,(\mathsf{S}\,(\mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,(\mathsf{S}\,(\mathsf{K}\,\mathsf{K})\,\mathsf{I}))\,(\mathsf{K}\,G))),$$

$$\text{where} \quad \begin{cases} A = \mathsf{S}\,(\mathsf{S}\,(\mathsf{K}\,\mathit{isZero})\,\mathsf{I})\,(\mathsf{K}\,1), \\ C = \mathsf{S}\,(\mathsf{K}\,\mathit{mult})\,\mathsf{I}, \\ G = \mathsf{S}\,(\mathsf{K}\,\mathit{pred})\,\mathsf{I}, \end{cases}$$

[14] It is due to Alonzo Church (Church, 1936) and his student Alan Turing (Turing, 1937), for the closely related $\lambda$-calculus, and it carries over to combinators.

[15] Run it lazily. A strict run unfolds $\Theta$ inside the argument for ever, and strict languages use a variant of $\mathsf{Y}$ that waits for its argument. The most famous combinator in Silicon Valley is a venture fund: Paul Graham named Y Combinator after this one. A type checker forbids it (§1.7).

with no variable, no parameter, no `if` and no recursive call: only $\mathsf{K}$, $\mathsf{S}$, a few ready-made arithmetic constants, and application.[16] Getting from the whiteboard to this is mechanical: the conditional becomes an application $b\,M\,N$, the recursive call becomes an extra first parameter $r$, and Theorem 1.16 compiles the parameters away. So $F\,r\,n$ computes "if $n = 0$ then 1, else $n$ times $r$ applied to $n - 1$".

What is still missing is the loop: something to put in for $r$ that behaves as $F$ itself does. That is $\mathsf{Y}\,F$. The copy that $\Theta$ hands to $F$ is what $F$ receives as its $r$, so every recursive call unfolds into $F$ once more, and $\mathsf{Y}\,F\,4$ runs to 24. Recursion was in the language all along, hiding in self-application; §1.5 shows what else hides there.

We've taken a great deal away from your idea of a program and added nothing. What's left is this:

- A program is a finite list of definitions together with a term to evaluate.
- A definition is an equation.
- To run the term is to replace, over and over, a left-hand side by its right-hand side, until no replacement is left to make.

There's no store, no stack, no instruction pointer, and no order of evaluation to settle: which step you take first can't change the answer you get (Corollary 1.8), though it can change whether you get one at all.

This language is the general case, reached by deleting from your own languages everything a checker doesn't read. If you understand the checker for this one, you understand type-checking for all of them. Hold on to that, because something is about to go badly wrong here, and what goes wrong is a thing that goes wrong in yours too. Theorem 1.19 promises as much: this language can do everything a computer can do, and one of the things a computer can do is fail to stop.

1.5. **The Monster.** Confluence promises that a term has at most one normal form. It doesn't promise that there is one, and here the connection with computing is the whole point: not every program stops. Nothing in Definition 1.1 says what may be applied to what, and so a function may be applied to itself. Consider the definition

$$\omega\,x \coloneqq x\,x.$$

It is the function that applies its argument to itself. That's an odd thing to want, perhaps, but Definition 1.11 allows it: the right-hand side uses nothing but the one parameter.

Now apply $\omega$ to itself,

$$\Omega \coloneqq \omega\,\omega,$$

and run it:

$$\omega\,\omega \;\rightsquigarrow\; \omega\,\omega \;\rightsquigarrow\; \omega\,\omega \;\rightsquigarrow\; \cdots.$$

The step, (def) for $\omega$ with $\omega$ written in for $x$, returns the term it started from; so there is a second step, and it returns it again, and no number of steps ever finishes. This is the Monster: a perfectly legal construction that

[16] Nobody would ever write this. Nobody has to: a compiler would.

nobody who wrote the definitions had in mind. It is the loop of $\mathsf{Y}$ with nothing inside it: *self* $\mathsf{I}$ does what $\omega$ does, and $\mathsf{Y}\,\mathsf{I}$ runs for ever.

So some programs never stop. Why does that matter? Run $\Omega$ and there is no output, no error and no crash. The machine is working. It will still be working tomorrow. You've met this: a page that spins; a build that never finishes; a request whose reply never comes, holding its connection open, and then a thousand more behind it, until a server that was answering everybody is answering nobody. A wrong answer can at least be read, and argued with. Here the program returns nothing and takes the machine with it.

Changing the notation doesn't remove the loop. Compiled into the constants, $\omega$ is $\mathsf{S}\,\mathsf{I}\,\mathsf{I}$:

$$\begin{aligned} \mathsf{S}\,\mathsf{I}\,\mathsf{I}\,x &\rightsquigarrow \mathsf{I}\,x\,(\mathsf{I}\,x) && \text{the } \mathsf{S} \text{ step} \\ &\rightsquigarrow^{*} x\,(\mathsf{I}\,x) \rightsquigarrow^{*} x\,x && \mathsf{I}\text{, twice,} \end{aligned}$$

so $\Omega$ is $\mathsf{S}\,\mathsf{I}\,\mathsf{I}\,(\mathsf{S}\,\mathsf{I}\,\mathsf{I})$, which takes several steps to come back to itself instead of one. Compiling gains nothing: the loop is in the shape of the term.

Nor is there only one monster. There are infinitely many, and here is another, built from a function that uses its argument three times:

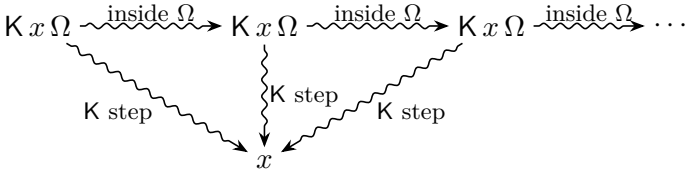
$$\omega_3\,x \coloneqq x\,x\,x \qquad \text{and} \qquad \Omega_3 \coloneqq \omega_3\,\omega_3.$$

Running it gives

$$\omega_3\,\omega_3 \rightsquigarrow \omega_3\,\omega_3\,\omega_3 \rightsquigarrow \omega_3\,\omega_3\,\omega_3\,\omega_3 \rightsquigarrow \cdots .$$

The term grows at every step and never repeats. So a loop need not repeat itself, and a checker that watches for a repeat can't catch them all.

Whether a run stops can even depend on the order of its steps. In §1.3 the order changed only the cost of a run; here it changes whether the run ends. The term $\mathsf{K}\,x\,\Omega$ has two redexes. The outer one, the $\mathsf{K}$ step, gives $x$ at once, a normal form, and the run stops. The inner one, inside $\Omega$, gives $\mathsf{K}\,x\,\Omega$ back, and a run that takes it for ever never stops. In the picture the inner steps go right and the $\mathsf{K}$ steps go down:




The two strategies of §1.3 fall on opposite sides of this term. A strict run takes the inner step first, so it never reaches the $\mathsf{K}$ step at all; a lazy run takes the outer step and finishes at once.[17]

So there is a fourth consideration to add to the three of §1.3, and it favours lazy: *termination*. For terms built from $\mathsf{K}$ and $\mathsf{S}$, always taking the leftmost of the outermost redexes reaches the normal form whenever there is one.[18] A strict run can loop on a term that has a normal form, as it does on $\mathsf{K}\,x\,\Omega$;

[17] Real languages behave the same way. In Haskell, which is lazy, `const 1 undefined` is `1`. In Python, which is strict, with `K = lambda a: lambda b: a` and `w = lambda x: x(x)`, the call `K(1)(w(w))` never returns `1`, and Python gives up only when its stack runs out. Exercise 1.4 asks you to watch both happen.

[18] This normalisation theorem for combinators is Curry and Feys' (Curry and Feys, 1958).

a lazy run that takes the leftmost outermost redex cannot. If what you most want is an answer whenever one exists, lazy is the safe choice, and the price is paid in the cost and the effects of §1.3. But choosing the order, though it can rescue $\mathsf{K}\,x\,\Omega$, is no help to $\Omega$ itself: $\omega\,\omega$ has one redex, so every strategy takes the same step, and none of them gets anywhere.

Be clear about what confluence has and has not promised, because $\mathsf{K}\,x\,\Omega$ looks at first like a counterexample to it. It isn't. Corollary 1.8 says that the term has at most one normal form, and Corollary 1.9 says that no step puts it out of reach: after any number of inner steps the $\mathsf{K}$ step is still there to be taken, which is what the arrows down from the row show. Both are true of $\mathsf{K}\,x\,\Omega$. What neither says is that every order of steps reaches the normal form, nor that there is a normal form to reach. Those are different claims, and they are false: the inner-first run of $\mathsf{K}\,x\,\Omega$ never arrives, and $\Omega$ has nowhere to arrive at.

The monsters can't be banished without giving something up. Theorem 1.19 says that our language computes everything a Turing machine computes, and a Turing machine can run for ever; a language in which every program stops computes less. Nor can we keep the whole language and refuse exactly the monsters. No checker refuses exactly the programs that fail to stop, every one that loops and none that halts: that is the undecidability of the halting problem, and Chapter 8 proves it for these very terms. So we'll ask for something weaker and get something more useful: a checker that refuses a class of programs we can describe exactly, and that has $\Omega$ inside that class.

1.6. **Types.** What went wrong isn't hard to name. The function $\omega$ takes an argument and returns a result, and nothing anywhere recorded what kind of argument it takes or what kind of result it gives back. So nothing was in a position to notice that when $\omega$ was handed $\omega$, it had been handed something it was never built to take. The language has functions and arguments, and no bookkeeping whatever.

The idea is simple, and the rest of this chapter is its execution. Write the bookkeeping down.

1. Label every term with what it takes and what it returns.
2. Allow an application only when the argument's label is the one the function asks for.
3. Check, before anything runs, that the labels fit together throughout.

The labels are called *types*.

**Definition 1.21** (types)**.** Fix a stock of *basic types*, such as *Int*, *String* and *Bool*; I write $p, q, r, \ldots$ for basic types when it does not matter which. The *types* are defined inductively:

1. Every basic type $p$ is a type.
2. If $A$ and $B$ are types, then $A \to B$ is a type: the *function type*, the type of functions that take an $A$ and return a $B$.
3. Nothing else is a type.

In one line,

$$A, B ::= p \mid (A \to B).$$

A program of type $A \to B$ is a function that takes an input of type $A$ and produces an output of type $B$. Read $A \to B$ as a contract: "give me an $A$ and I'll return a $B$".[19] So

- $Int \to Bool$ is the type of a test on integers;
- $(Int \to Int) \to Int$ is the type of something that takes a function;
- $(p \to q) \to p$ is a type too, whatever the basic types $p$ and $q$.

As with terms, writing out every bracket of a type is cumbersome, and a convention lets us drop most of them. For terms we grouped to the left; for types we make the opposite choice and group to the *right*: $A \to B \to C$ is $A \to (B \to C)$, a function that takes an $A$ and returns a function from $B$ to $C$. The type $(A \to B) \to C$ is different: it is the type of a function that takes a function. The conventions are opposite so that terms and types fit together. A function of two arguments takes them one at a time: if $f : A \to B \to C$ and $x : A$, then $f\,x : B \to C$, and if also $y : B$, then $f\,x\,y : C$. Each convention leaves out the brackets of the commonest shape, and the two shapes match: $f\,x\,y$ is $(f\,x)\,y$, and its type, $A \to (B \to C)$, gives up its arguments from the left in the same order.

We have the types; now we need to say how to work out the type of a term, written $\vdash_{\mathrm{KS}} M : A$ and read "$M$ has type $A$". But a term such as $f\,(f\,x)$ has variables in it, and its type depends on theirs, so the claim has to carry a list declaring them, written $\Gamma$.

**Definition 1.22** (context)**.** A *context* $\Gamma$ is a finite list $x_1 : A_1,\ \ldots,\ x_n : A_n$ pairing each variable in scope with a type, no variable listed twice. The empty list counts, and I write $\Gamma, x : A$ for $\Gamma$ with the pair $x : A$ added at the end.

A context is what a programmer would call the declarations in scope. It may also declare ready-made constants, as the text after Definition 1.24 explains. The checker's basic claim is made in a context.

**Definition 1.23** (judgement)**.** A *judgement* is a claim

$$\Gamma \vdash_{\mathrm{KS}} M : A,$$

read "$M$ has type $A$ when its variables have the types $\Gamma$ gives them". When $\Gamma$ is empty it is written $\vdash_{\mathrm{KS}} M : A$.

When $\Gamma$ is empty, $M$ is a program with type $A$ outright. The subscript names the system, the typed combinators, KS after the two constants $\mathsf{K}$ and $\mathsf{S}$; every system in these notes carries a subscript of its own, so that a turnstile always says which system it belongs to.

---

[19] In Java, `Function<A, B>` is the type of functions that take an `A` and return a `B`, which is $A \to B$. More:

- A method `static int square(int n)` has type $Int \to Int$;
- the test `n -> n % 2 == 0` has type `Function<Integer, Boolean>`, which is $Int \to Bool$;
- the addition `m -> n -> m + n`, which takes its arguments one at a time, has type `Function<Integer, Function<Integer, Integer>`, which is $Int \to Int \to Int$, nested to the right just as our arrow is.

FOR COMPUTER SCIENTISTS. You've written contexts. In Java,

```
Function<Integer, Integer> f = ...;
int x = 3;
... f.apply(f.apply(x)) ...
```

declares $f$ and $x$ and then uses them. The declarations are $\Gamma$, and what Java checks about the last line is the judgement $f : \mathit{Int} \to \mathit{Int},\ x : \mathit{Int} \vdash_{\mathrm{KS}} f\,(f\,x) : \mathit{Int}$.

---

The question for us is this: given a context $\Gamma$ and a term $M$, which type, if any, may $M$ have? Suppose $\Gamma$ declares $f : \mathit{Int} \to \mathit{Int}$. Then $f\,x$ should be well typed only when $x$ is an integer, and $f\,x$ is then an integer itself. All a checker does is confirm, before anything runs, that your code keeps to this discipline throughout. By Definition 1.1 there are four kinds of piece to check, one for each way of building a term (counting the two constants separately). The constants have fixed types, a variable has the type the context gives it, and an application is typed from the types of its two parts; it is there that a checker says no.

The certificate the checker produces is called a *typing*, and it is laid out as a proof is laid out on paper: what is given at the top, a line under it, and below the line an argument in which every step is numbered and carries its reason.

**Definition 1.24** (typing)**.** The two constants have the type schemas

$$\text{(K)}\quad \mathsf{K} : A \to B \to A \tag{1.1}$$

$$\text{(S)}\quad \mathsf{S} : (A \to B \to C) \to (A \to B) \to A \to C \tag{1.2}$$

where $A$, $B$ and $C$ stand for arbitrary types: each line is a *schema*, and gives one type for each choice of letters.

Let $\Gamma$ be a context, $M$ a term and $A$ a type. A *typing* of $\Gamma \vdash_{\mathrm{KS}} M : A$ is a finite sequence of judgements

$$M_1 : A_1,\ M_2 : A_2,\ \ldots,\ M_n : A_n$$

whose last judgement $M_n : A_n$ is $M : A$, and in which each judgement $M_k : A_k$, read "$M_k$ has type $A_k$ in the context $\Gamma$", is justified in one of four ways:

1. (*K*) $M_k$ is $\mathsf{K}$, and $A_k$ is an instance of (1.1);
2. (*S*) $M_k$ is $\mathsf{S}$, and $A_k$ is an instance of (1.2);
3. (*var*) $M_k$ is a variable $x$ (or a ready-made constant, see below), and $x : A_k$ is declared in $\Gamma$;
4. (*app*) there are two earlier judgements $M_i : A_i$ and $M_j : A_j$, with $i, j < k$, such that $A_i$ is the type $A_j \to A_k$ and $M_k$ is the term $M_i\,M_j$.

If such a typing exists, we write $\Gamma \vdash_{\mathrm{KS}} M : A$, and say that the typing *witnesses* $\Gamma \vdash_{\mathrm{KS}} M : A$.

We set a typing out as a box, with the context $\Gamma$ above the horizontal line and the judgements below it, one to a numbered line, each with its justification on the right:

| | $\Gamma$ | |
|---|---|---|
| 1 | $M_1 : A_1$ | *reason* |
| 2 | $M_2 : A_2$ | *reason* |
| | $\vdots$ | |
| $n$ | $M_n : A_n$ | *reason* |

Each line is really the judgement $\Gamma \vdash_{\mathrm{KS}} M_k : A_k$, with the $\Gamma \vdash_{\mathrm{KS}}$ left off because it is the same $\Gamma$ all the way down; the vertical line is there to say so. A declaration in $\Gamma$ is one that stands above the line. The justifications are written like this:

- "(app) $i$, $j$" names line $i$ as the function and line $j$ as the argument;
- after (K) or (S) I give in square brackets the types put for the schema's letters.

If the word *schema* is new, the idea is not. Schema (1.1) is one line, but it gives $\mathsf{K}$ a type for every choice of $A$ and $B$: $\mathit{Int} \to \mathit{Bool} \to \mathit{Int}$, and $\mathit{Bool} \to \mathit{Int} \to \mathit{Bool}$, and $(\mathit{Int} \to \mathit{Int}) \to \mathit{String} \to (\mathit{Int} \to \mathit{Int})$, each of them a typing of $\mathsf{K}$.[20]

The integers, the operator + and the strings aren't terms of Definition 1.1, so a typing has no case for them, and needs none. A program that uses them has to say what they are, and it says so in the context. The declarations $3 : \mathit{Int}$, `"hello"` $: \mathit{String}$ and $+ : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ (so that $y + 1$ is + applied to $y$ and then to 1) are declarations like any other, and a line that uses one is justified by (var). Only the pieces a term actually uses need be declared. A term that uses none, such as $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$, is typed in the empty context.

**Example 1.25** (a first typing)**.** Recall that `if b then M else N` is just $b\,M\,N$ and that $\mathit{true} \coloneqq \mathsf{K}$, so that `if true then 3 else 4` is $\mathsf{K}\,3\,4$. Let us type it. It has no variables, but it uses two ready-made numerals, and a checker has to be told what they are, so they are declared above the line.

| | $3 : \mathit{Int},\ 4 : \mathit{Int}$ | |
|---|---|---|
| 1 | $\mathsf{K} : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ | (K) [*Int*, *Int*] |
| 2 | $3 : \mathit{Int}$ | (var) |
| 3 | $\mathsf{K}\,3 : \mathit{Int} \to \mathit{Int}$ | (app) 1, 2 |
| 4 | $4 : \mathit{Int}$ | (var) |
| 5 | $\mathsf{K}\,3\,4 : \mathit{Int}$ | (app) 3, 4 |

The last line is $\mathsf{K}\,3\,4 : \mathit{Int}$, so the box witnesses $3 : \mathit{Int},\ 4 : \mathit{Int} \vdash_{\mathrm{KS}} \mathsf{K}\,3\,4 : \mathit{Int}$: an *Int*, and we know it before anything runs. Declare `"hello"` $: \mathit{String}$ in place of 4 and choose $B$ to be *String* in line 1, and $\mathsf{K}\,3$ `"hello"` $: \mathit{Int}$ too: $\mathsf{K}$ doesn't care what it drops.

**Example 1.26** (a typing with parameters)**.** Now a term with parameters: the body of *twice* $f\,x \coloneqq f\,(f\,x)$. To type it, declare its parameters, $\Gamma = f : \mathit{Int} \to \mathit{Int},\ x : \mathit{Int}$.

[20] A schema is what Java, Rust and TypeScript call a *generic*: in Java, $\mathsf{K}$ is `static <A, B> Function<B, A> k(A a) { return b -> a; }`, and the types the compiler chooses for `A` and `B` at each use of $\mathsf{K}$ are the types written in square brackets after (K) in a typing.

| | $f : \mathit{Int} \to \mathit{Int},\ x : \mathit{Int}$ | |
|---|---|---|
| 1 | $f : \mathit{Int} \to \mathit{Int}$ | (var) |
| 2 | $x : \mathit{Int}$ | (var) |
| 3 | $f\,x : \mathit{Int}$ | (app) 1, 2 |
| 4 | $f\,(f\,x) : \mathit{Int}$ | (app) 1, 3 |

Line 1 is cited twice, which is allowed. The context $\Gamma$ is the parameter list.[21] The last line is $f\,(f\,x) : \mathit{Int}$, so the box witnesses $f : \mathit{Int} \to \mathit{Int},\ x : \mathit{Int} \vdash_{\mathrm{KS}} f\,(f\,x) : \mathit{Int}$.

The definition has no case for a *name*. It types constants, variables and applications, and a name introduced by a definition is none of the three: to type *twice* itself, or $\mathit{add}\,\overline{2}\,\overline{1}\,\mathit{inc}\,0$, the checker would need the type of the name, and nothing above gives it one. A rule for names is still missing.

A typing is easy to check, and finding one can take search. A checker that has to find types instead of being told them[22]

1. gives each unknown type a letter;
2. reads off from the rules the equations the letters must satisfy;
3. solves them.

The procedure is called *unification*, and it works backwards from the goal, not forwards from the constants.

**Example 1.27** (the identity, typed)**.** The typing of $\mathsf{I} \coloneqq \mathsf{S}\,\mathsf{K}\,\mathsf{K}$ at the type $A \to A$ is below. The square brackets name the types substituted for the schema's letters, in order.

| | *empty context* | |
|---|---|---|
| 1 | $\mathsf{S} : (A \to (A \to A) \to A) \to (A \to (A \to A)) \to A \to A$ | (S) $[A,\ A \to A,\ A]$ |
| 2 | $\mathsf{K} : A \to (A \to A) \to A$ | (K) $[A,\ A \to A]$ |
| 3 | $\mathsf{S}\,\mathsf{K} : (A \to (A \to A)) \to A \to A$ | (app) 1, 2 |
| 4 | $\mathsf{K} : A \to (A \to A)$ | (K) $[A,\ A]$ |
| 5 | $\mathsf{S}\,\mathsf{K}\,\mathsf{K} : A \to A$ | (app) 3, 4 |

Five lines, and the last is $\mathsf{S}\,\mathsf{K}\,\mathsf{K} : A \to A$, so the box witnesses $\vdash_{\mathrm{KS}} \mathsf{S}\,\mathsf{K}\,\mathsf{K} : A \to A$. Nothing stands above the line: there are no variables and nothing ready-made, so the context is empty.

To find line 1, leave the letters of (S) unknown, and let the two $\mathsf{K}$s decide them.

1. The goal is $\mathsf{S}\,\mathsf{K}\,\mathsf{K} : A \to A$.
2. (S) gives $\mathsf{S} : (X \to Y \to Z) \to (X \to Y) \to X \to Z$, so $\mathsf{S}\,\mathsf{K}\,\mathsf{K} : X \to Z$; hence $X = A$ and $Z = A$.
3. The first $\mathsf{K}$ must have type $A \to Y \to A$, which has the shape $V \to W \to V$ of (K) whatever $Y$ is.

[21] In Java, `static int twice(Function<Integer, Integer> f, int x)`.

[22] Haskell's, OCaml's and Rust's, for instance.

4. The second K must have type $A \to Y$, and the shape $V \to W \to V$ forces $Y = W \to A$ for some $W$.
5. Any $W$ will do. Choose $W = A$: then $Y = A \to A$, and that is line 1.

The search took a paragraph; the check takes none. Line 1 is the line nobody would write unprompted, and unification finds it.

Two schemas, one look-up, one match: that is the whole checker. Read the four cases as the checker reads your code; each schema is just what its combinator does, written as a contract.

- *(K).* K *keeps*: it takes an $A$, then a $B$, and returns the $A$, and (1.1) says so for every choice of $A$ and $B$.
- *(S).* S *shares*: it takes a function $M$ of two arguments, a function $N$ of one, and an argument $P$, and returns $M\,P\,(N\,P)$. For that to make sense $P$ must be what both $M$ and $N$ expect: $P : A$, $N : A \to B$ and $M : A \to B \to C$, and the result $M\,P\,(N\,P)$ is then a $C$. Schema (1.2) is that sentence written down.
- *(var).* A variable has the type it was declared with, and the checker has only to look $x$ up in $\Gamma$.
- *(app).* An application $M\,N$ is well typed only if $M$ is a function and $N$ is an argument of exactly the type $M$ expects, and then the result has $M$'s result type. When the types don't match, this is the case that draws the red line.

**Example 1.28** (the red line)**.** Back to the line from the start of the chapter: `total = 3 + "hello"`, underlined in red, with the message *expected Int, got String*. Without types nothing stops it: inside the machine 3 and `"hello"` are both just bits, and the processor would add them. Now add the checker, and try to type $+\,3$ `"hello"`.

| | | | |
|---|---|---|---|
| | $+ : \mathit{Int} \to \mathit{Int} \to \mathit{Int},\ 3 : \mathit{Int},$ `"hello"` $: \mathit{String}$ | | |
| 1 | $+ : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ | (var) | |
| 2 | $3 : \mathit{Int}$ | (var) | |
| 3 | $+\,3 : \mathit{Int} \to \mathit{Int}$ | (app) 1, 2 | |
| 4 | `"hello"` $: \mathit{String}$ | (var) | |
| 5 | $+\,3$ `"hello"` $: ?$ | (app) 3, 4 | *refused* |

Line 5 should be (app) applied to lines 3 and 4. But (app) needs the argument's type to be the $A$ of the function's $A \to B$, and here the function wants *Int* and is offered *String*. Nor is there another way round: since + is declared at *Int* alone, line 3 gives the only type the function can have.

No rule applies, there is no typing, and the checker refuses, before anything runs. That refusal, *expected Int, got String*, is the whole content of the error your editor shows, minus the decoration a real language adds.

1.7. **Subject Reduction.** The checker gives its verdict *before* the program runs. But running rewrites the term: $\mathsf{K}\,3\,4 \rightsquigarrow 3$, and the checker typed $\mathsf{K}\,3\,4$, not 3. Is the term we end up with still an *Int*? If a single step could turn an

*Int* into a *String*, the checker's *yes* would expire half-way through the run, and every guarantee with it. You rely on this not happening: a variable declared `int` never ends up holding a `String` while the program runs. In our language the reason is a theorem, and one we can prove.

**Theorem 1.29** (subject reduction)**.** *If* $\Gamma \vdash_{\mathrm{KS}} M : A$ *and* $M \rightsquigarrow N$ *by a* $\mathsf{K}$ *step, an* $\mathsf{S}$ *step or a ready-made step, then* $\Gamma \vdash_{\mathrm{KS}} N : A$*.*

*Proof.* First the cases in which the whole of $M$ is the part rewritten, and then the case in which that part sits inside something larger.

- *A* $\mathsf{K}$ *step.* Then $M$ is $\mathsf{K}\,P\,Q$ and $N$ is $P$. Read the typing of $M$ backwards twice. The line $\mathsf{K}\,P\,Q : A$ came by (app) from $\mathsf{K}\,P : B \to A$ and $Q : B$, and the line $\mathsf{K}\,P : B \to A$ came by (app) from $\mathsf{K} : C \to B \to A$ and $P : C$. Schema (1.1) gives $\mathsf{K}$ no types but those of the shape $V \to W \to V$, so $C = A$. Hence $\Gamma \vdash_{\mathrm{KS}} P : A$, which is what we want.
- *An* $\mathsf{S}$ *step.* Then $M$ is $\mathsf{S}\,P\,Q\,R$ and $N$ is $P\,R\,(Q\,R)$. Reading backwards three times gives $R : D$, $Q : E$, $P : F$ and $\mathsf{S} : F \to E \to D \to A$, and schema (1.2) forces $F = X \to Y \to Z$, $E = X \to Y$, $D = X$ and $A = Z$. Now build the typing of $N$ forwards:
  1. (app) on $P : X \to Y \to Z$ and $R : X$ gives $P\,R : Y \to Z$;
  2. (app) on $Q : X \to Y$ and $R : X$ gives $Q\,R : Y$;
  3. (app) on those two gives $P\,R\,(Q\,R) : Z$, and $Z$ is $A$.
- *A ready-made step.* $m + n \rightsquigarrow$ the sum: $\Gamma$ declares $+$ and the integers, as the context of any term that uses them does, so both sides have type *Int*.
- *Inside a larger term.* If the part rewritten is a proper subterm of $M$, the lines of the typing that type that subterm are replaced by the lines just constructed, which end in the same type; every later line of the typing is unchanged, because (app) looks only at the types of its two parts and not at what they are.

□

In words: the verdict the checker gave before the program ran stands as the program runs. At every step the term in hand still has the type the checker assigned, so an argument of the wrong kind can never arise part-way through, however long the run. *What the checker said stays said.* Without this theorem, a type system would be pointless.

Subject reduction says that no step goes wrong. It doesn't say that the steps come to an end, and the Monster was a program whose steps never do. The next theorem says that no typed program behaves like that.

**Theorem 1.30** (normalisation; on loan)**.** *If* $\Gamma \vdash_{\mathrm{KS}} M : A$*, then* $M$ *has a normal form.*

This bans the Monster and all its kin. The term $\Omega$ has no normal form, so by the theorem it has no typing; and since (app) types an application only when both of its parts are typed, no term with $\Omega$ inside it has a typing either. A typed program, then, is safe in both senses: by subject reduction no step goes wrong, and by normalisation the run ends. That answers the

question the chapter began with. A checker that looks only at kinds can keep out every program that never stops. The price is that it keeps out some that do stop, and $\mathsf{K}\,x\,\Omega$, which has a normal form and no typing, is one of them. We take the theorem on trust.

The theorem comes in two strengths. The weaker says that *some* order of steps always finishes: always take the step whose function has the most complicated type, and a measure of the term goes down at every step, and cannot go down for ever. That is enough for Theorem 1.30 as stated. The stronger says that *every* order of steps finishes. No simple measure works for every order; the proof instead shows something stronger than "finishes", by induction on the type, and that is a technique this course doesn't reach.[23]

Given the stronger form, the choice between strict and lazy that mattered for $\mathsf{K}\,x\,\Omega$ makes no difference to a typed program: it finishes either way. So the termination consideration of §1.5 disappears for typed programs, and what is left to choose between is the cost and the effects of §1.3. Every real language gives the loop back by adding recursion under another name (a loop, or a function that calls itself), and Exercise 1.6 asks what that costs.

**History.** The combinators were discovered twice in the 1920s, when there were no programs to write: by Moses Schönfinkel, working in Hilbert's Göttingen, and by Haskell Curry, a young American teaching mathematics at Harvard. Both were after something in logic, and to see what, we need the logic of the day.

In the 1920s mathematicians wanted a *rulebook* for the whole of mathematics: a system in which a proof could be written down and then checked step by step, by rule alone, with no judgement needed. Checking by rule alone, without running anything, is exactly what your type checker does to a program; they wanted it for every proof in mathematics. The great attempt was Whitehead and Russell's *Principia Mathematica* (1910–13), and it worked, at a price: it takes hundreds of pages to reach $1 + 1 = 2$.[24]

Schönfinkel was after economy. He wanted to build logic from as few basic notions as possible, and the title of his paper, *On the building blocks of mathematical logic*, says as much. Its surprise was that the variables of logic, the $x$ in "for every $x$", could be done away with altogether, and that the two operations we have called $\mathsf{K}$ and $\mathsf{S}$ could do the variables' work. He introduced them in a talk at Göttingen in 1920, and a colleague, Heinrich Behmann, wrote the talk up for publication in 1924. Schönfinkel returned to Moscow, fell ill, and died in poverty in 1942; it is said that his neighbours burned his papers for heat. The two letters $\mathsf{K}$ and $\mathsf{S}$ are most of what survives of him.

Curry, reading the first chapter of *Principia*, was bothered by something small. One of its rules, *detachment*, is as simple as a rule can be: from $A$ and "if $A$ then $B$", conclude $B$. Another, *substitution* (putting one formula

[23] The weaker form is Turing's, from the early 1940s. He never published the proof; Robin Gandy printed it in 1980, long after Turing's death, in a volume of essays for Haskell Curry. The stronger form is Tait's, from 1967 (Tait, 1967), and Girard took his technique in 1972 to polymorphism, the theory behind generics.

[24] *Principia* finishes the proof of $1 + 1 = 2$ only in its second volume (1912), at proposition $*110.643$, and adds the remark: "The above proposition is occasionally useful."

in place of a variable throughout another), was far more complicated. A rule that basic, Curry thought, ought to be simple too.

So he set out to take substitution apart, and found that it came apart into very little. Whatever a variable is doing in a formula, it is being kept, thrown away, copied or moved; so a few fixed operations that keep, throw away, copy and move their arguments can do all the work that variables do. Those operations are the combinators: K throws an argument away and S copies one, and that is why the two of them turned out to be enough.[25]

Curry's plan was bigger than substitution. If combinators could do the work of variables, they could sit underneath the whole of logic: one would build the logic itself from constants and application, and the rulebook for mathematics would rest on a handful of constants and the rules for running them. You've seen in this chapter why that might work. Functions can be everything: the booleans, the numbers and addition were all built from K and S. And a calculus of functions has only a few simple rules, easy to work with and easy to check.

So what could go wrong? A great deal, as it turned out. Within a few years the logic was shown to be inconsistent (Kleene and Rosser, 1935).[26] Curry's logic is not worth developing here. What broke it is worth seeing, though, because it is the Monster in another guise.

The first break came in 1935, from Stephen Kleene and Barkley Rosser, and it used an old puzzle of Jules Richard's (1905). The phrases of English that describe functions on numbers ("add one", "double the number") can be listed, shortest first, so the functions they describe come in a list too: $f_0, f_1, f_2, \ldots$. Now go down the diagonal and add one, $d\ n \coloneqq f_n\ n + 1$, so that $d$ differs from $f_0$ at 0, from $f_1$ at 1, and from every function on the list somewhere.

But "add one to what the $n$th function on the list gives at $n$" is itself an English description. So $d$ is on the list, say $d = f_k$, and then $d\ k = f_k\ k{+}1 = d\ k + 1$: a number equal to itself plus one.

In English the blame falls on "the $n$th function on the list", which isn't a precise description, since nobody ever pinned down which phrases count as descriptions. A logic made of combinators *can* pin it down, inside itself: Kleene and Rosser (1935) showed that it can list its own definitions and carry the construction out, and so it proves $d\,k = d\,k{+}1$, and hence $0 = 1$.[27]

Richard's $d$ and the Monster are the same construction. Going down the diagonal means applying the $n$th function to its own number, $f_n\ n$; and when $d$ meets itself, at $n = k$, it is applied to its own number too. That

[25] In 1927, at Princeton, Curry came across Schönfinkel's paper of 1924 and found that he had been beaten to the idea; he was, for a while, distressed. He went to Göttingen all the same, where Hilbert examined his thesis on combinatory logic in 1929 (Curry, 1930), and he spent the next fifty years on the subject. A programming language and a verb (*currying*) are named after him; nothing is named after Schönfinkel.

[26] Curry wasn't alone. Alonzo Church, at Princeton, had built a logic on the same plan in 1932, writing his functions with $\lambda$, in what became the $\lambda$-calculus, instead of combinators, and the same paradox broke both.

[27] Kleene and Rosser were Church's doctoral students, and the same argument broke Church's logic too. Every supervisor's nightmare: your students break your system. It worked out; both became giants of the subject.

is self-application, the move $\omega\, x \coloneqq x\, x$ makes, dressed up as a list. Read Richard's equation as a rule for running instead of as a claim, and it is a monster:

$$d\ k \ \rightsquigarrow\ d\ k + 1 \ \rightsquigarrow\ (d\ k + 1) + 1 \ \rightsquigarrow\ \cdots,$$

a run that grows at every step and never ends, as the run of $\Omega_3$ did. A logic and a program fail differently on the same construction. The logic believes its equations, so it concludes that a number is equal to itself plus one, and from there anything at all.[28] The program only runs them, and so it never stops. Either way the trouble is a thing made to act on itself.

Look at the Monster through its types and the self-reference is plain to see. Suppose $\omega$ could be typed, and give its parameter $x$ some type $A$. In the body $x\, x$, the first $x$ is a function, so its type must be $A \to B$ for some $B$, and the second $x$ is its argument, of type $A$. But it is the same $x$, with one type, so

$$A \ = \ A \to B.$$

The type of $x$ would have to contain itself as a proper part, as Richard's $d$ had to be on the list it was defined from. No type does: $A \to B$ is longer than $A$, and a type can't be longer than itself. So $\omega$ has no type, and the checker refuses the Monster at the very step where it applies something to itself.[29] The $\mathsf{Y}$ of §1.4 is refused for the same reason, since it is built from the same $x\, x$.

That is the repair, and it is the one that lasted. Curry had begun giving his combinators types in 1934, and Church built them into his logic in 1940.[30] Types sit outside the language of terms: they are built from basic types by arrows, and "$M : A$" is a judgement the checker makes by four rules, not a term that could be applied to itself. A thing that must take itself as an argument has no type, so it can't be written, and the diagonal never gets started. Your compiler is built on the repaired discipline, and that is why the checker's whole job is to refuse. So of the two options we started with, the compiler's certainty was the theorem.

A logic needed saving, and programmers inherited the repair: Figure 1.1 gives the dates.

*Summary.* Let me gather it up. Under every language you use is a combinatory algebra: two constants and application, run by two rewrite rules, in which every definition you've ever written can be expressed; and unpoliced it computes everything, including a program that never stops. A run may take its steps in any order: by the theorem of Church and Rosser the order cannot change the result, a normal form, though it can change whether a result is reached.

A *type* is a contract, a basic type or $A \to B$; the checker's job is to allow an application only when the contract is met. Its certificate is a *typing*: the context $\Gamma$ above a line, and below it a list of numbered lines $M_k : A_k$, each

[28] Curry (1942) later found a shorter version, with no numbers and no list: a statement that says of itself "if I hold, then $B$", for any $B$ you like.

[29] In unification this failure has a name, the *occurs check*: a letter may not be set equal to a type in which it occurs. Ask Haskell for the type of `\x -> x x` and it complains of an *infinite type*, which is $A = A \to B$ in its own words.

[30] Church's repaired logic is the simply typed $\lambda$-calculus (Church, 1940).

| Year | Event |
|---|---|
| 1920 | Schönfinkel introduces K and S in a Göttingen talk (printed 1924). |
| 1920s | Curry, puzzled by substitution in *Principia*, rediscovers the combinators; thesis 1929, printed 1930. |
| 1934 | Curry works out the types of K and S, and recognises them as the first two of Frege's axioms for implication. |
| 1935 | Kleene and Rosser: Curry's logic, built from combinators, is inconsistent. |
| 1936 | Church and Turing: what can be computed. Church and Rosser: the order of steps does not change the answer. |
| 1940s | Turing: some order of steps always finishes (unpublished). |
| 1942 | Curry's paradox: the short version. |
| 1967 | Tait: every order of steps finishes. |
| 1972 | Girard extends Tait's idea to polymorphism, the theory of generics. |
| 1980 | Gandy publishes Turing's proof. |

FIGURE 1.1. Timeline.

justified in one of the four ways (K), (S), (var) and (app), and it witnesses the judgement its last line states. Its *no* is case (app) failing to match, as it does for $3 + \texttt{"hello"}$ and for the self-application $x\,x$ inside the Monster. Its *yes* mentions no input and so covers every input: by subject reduction what it says of a program stays said at every step of the program's run, and by normalisation the run ends.

**Exercises.**

**Exercise 1.1.** Let $\mathit{mult}\, m\, n\, f \coloneqq m\,(n\, f)$. Run $\mathit{mult}\,\overline{2}\,\overline{3}\,\mathit{inc}\,0$ by the rules of Definitions 1.2 and 1.11, one step per line, naming each. Then say what *mult* computes in general, and why the definition has three parameters where *add* had four.

**Exercise 1.2.** Example 1.17 wrote *comp* with no parameters. Do the same for $\mathit{flip}\, f\, x\, y \coloneqq f\, y\, x$: run $\mathsf{S}\,(\mathsf{B}\,\mathsf{B}\,\mathsf{S})\,(\mathsf{K}\,\mathsf{K})\, f\, x\, y$ by the rules of Definitions 1.2 and 1.11, using the runs of $\mathsf{B}$ and $\mathsf{I}$ as single steps where they help, and confirm that it reaches $f\, y\, x$. Which of the three things a parameter can do (be used once, several times, or not at all) does each of $\mathsf{K}$ and $\mathsf{S}$ take care of here?

**Exercise 1.3.** Using Definition 1.24:

(1) In the context $f : B \to C,\ g : A \to B$, give a typing of $\mathsf{S}\,(\mathsf{K}\, f)\, g$, the term that $\mathsf{B}\, f\, g$ runs to after two steps.

(2) Give a typing of the program $\mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}$ itself, with the empty context.

Keep both typings; the chapter on Hilbert proofs uses them again.

**Exercise 1.4.** §1.5 said that Python takes the inner step of $\mathsf{K}\, x\, \Omega$ and Haskell the outer one. Watch it happen: in Python define `K = lambda a: lambda b: a` and `w = lambda x: x(x)` and evaluate `K(1)(w(w))`; in Haskell evaluate `const 1 undefined`. Then do the same exercise on paper for *false* $\Omega\, x$: find a run that stops and a run that never does, and say which of them each language takes. Does either contradict Theorem 1.6?

**Exercise 1.5.**

1. Give typings of $\mathsf{K}\,\mathsf{I}$ and of $\mathsf{S}\,\mathsf{K}$.
2. Then run $\mathsf{S}\,\mathsf{K}\, M\, x$ for an arbitrary $M$, and say what $\mathsf{S}\,\mathsf{K}\, M$ is.
3. Finally, find one type that $\mathsf{K}$ and $\mathsf{K}\,\mathsf{I}$ both have, and say why it is the right type for a boolean.

**Exercise 1.6.** Every real language has recursion, but the $\mathsf{Y}$ of §1.4 has no type, since it is built from $x\, x$ (§1.7). So a typed language that wants recursion has to add it. Model it by a new constant *fix* with the step $\mathit{fix}\, f \rightsquigarrow f\,(\mathit{fix}\, f)$ and the rule $\Gamma \vdash_{\mathrm{KS}} \mathit{fix} : (A \to A) \to A$, for every type $A$.

(1) Write a program with a typing that never stops.

(2) Which of Theorems 1.29 and 1.30 survives the addition, and which does not?

(3) Read the type of *fix* as a formula. What does it say, and what does your answer to (1) say about every type $A$?

**Exercise 1.7.** This exercise goes with §1.2.

(1) Find every redex in each of these terms, and say which of the terms are in normal form:

$$\mathsf{K}\,(\mathsf{K}\, x)\, y, \qquad \mathsf{K}\,(\mathsf{K}\, x\, y), \qquad x\,(\mathsf{S}\, y)\,(\mathsf{K}\, z\, w), \qquad \mathsf{S}\,\mathsf{K}\,(\mathsf{K}\, x), \qquad \mathsf{S}\,\mathsf{K}\,\mathsf{K}\,(\mathsf{K}\, x\, y).$$

(2) The last of these has two redexes. Run it to normal form twice, once taking each redex first, and draw the diamond that Theorem 1.6 promises.

(3) In the diamond for $\mathsf{S}\,x\,y\,(\mathsf{K}\,z\,w)$ (§1.2), the run that took the outer step first was the longer. Find a term in which it is the shorter. Which order does a strict language take, and which a lazy one (§1.3)? Is either order always the quicker?

**Exercise 1.8.** This exercise goes with the bracket convention of §1.1 and with Example 1.5.

(1) Put in all the brackets in

$$\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,x, \qquad \mathsf{K}\,(\mathsf{K}\,x)\,y\,z, \qquad x\,y\,(z\,w)\,v.$$

Then take out every bracket that the convention makes unnecessary from

$$((\mathsf{K}\,x)\,(y\,z))\,w, \qquad (\mathsf{S}\,(\mathsf{K}\,x))\,y, \qquad \mathsf{K}\,((x\,y)\,z).$$

(2) Find every redex in $\mathsf{K}\,\mathsf{I}\,f\,x$ and in $\mathsf{K}\,(\mathsf{I}\,f)\,x$, and run each to its normal form. Which of the two reaches $x$? Which has two redexes, and do both orders agree?

(3) Python's `def k2(a, b): return a` takes a pair. Write `curry` and `uncurry`, one line each, such that `curry(k2)(1)(2)` gives `1` and so does `uncurry(K)(1, 2)`, where `K = lambda a: lambda b: a`. What happens on `k2(1)`, and what on `K(1)`? Which of the two is a term of our language?

**Exercise 1.9.** This exercise goes with §1.3. Let $A$ be the term $\mathsf{K}\,(\mathsf{K}\,z\,w)\,v$, which runs to $z$ in two steps.

(1) Run $\mathsf{S}\,x\,y\,A$ three ways:

- strictly;
- lazily, running each copy of $A$ separately;
- lazily with sharing.

Count the steps in each.

(2) Do the same for $\mathsf{K}\,x\,A$, strict and lazy only.

(3) In Python, `my_if` written to take its branches as plain expressions, `def my_if(b, m, n): return m if b else n`, together with `def fact(n): return my_if(n == 0, 1, n * fact(n - 1))`, makes `fact(5)` raise a `RecursionError`. Without looking back at the text:

- rewrite `my_if` so that its second and third arguments are functions to be called when wanted;
- change `fact` to match;
- check that `fact(5)` is `120`.

In one sentence, say what the `lambda:`s are doing.

(4) When an argument is used twice, and when it is thrown away, which strategy does the less work, and what does sharing add?

## References.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

Haskell B. Curry. Grundlagen der kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.

Haskell B. Curry. The inconsistency of certain formal logics. *Journal of Symbolic Logic*, 7(3):115–117, 1942.

Haskell B. Curry and Robert Feys. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, 1958.

Stephen C. Kleene and J. Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

Mars Climate Orbiter Mishap Investigation Board. Phase I report. Technical report, NASA, Washington, DC, November 1999.

Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

Alan M. Turing. Computability and $\lambda$-definability. *Journal of Symbolic Logic*, 2(4): 153–163, 1937.