PART 1 · CHAPTER 1

Type Theory

Alexander V. Gheorghiu
COMP0009 Logic · University College London

PROLOGUE

About this course

Before we begin.

PROLOGUE

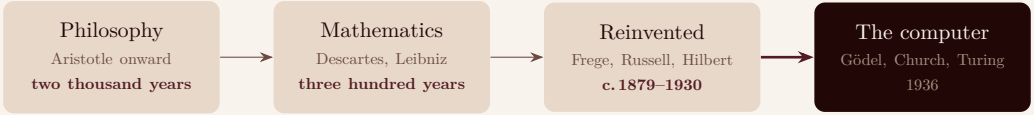
What logic is

*Logic is the **science of reasoning**.*

Not a branch of mathematics, not a branch of philosophy: the study of what it is to follow from something.

PROLOGUE

A short history



*Logic did not merely describe the computer. It **produced** it.*

PROLOGUE

Where this course starts

MANY COURSES

Stay with the story on the left of that picture.

Philosophy, mathematics, foundations: a fine story, and a long one.

THIS COURSE

Pick it up **from the computer**.

In the last century, logic found its biggest home in computer science. Let me show you why.

PROLOGUE

The question

How can we be sure?

The whole course asks it. Part 1 asks it of a program, with one hand tied: nothing here is about truth. Only strings, rules for rewriting them, and rules for checking them.

So let us begin where you have already met it: with a red line.

PART 1 · TYPE THEORY

A compiler error

- Not a crash. Not a warning.
- A red line under something that *would* have gone wrong.
- The function was never called. No input was ever tried.
- And yet the compiler was **certain**.

```
total = 3 + "hello"
            ^^^^^^^
```

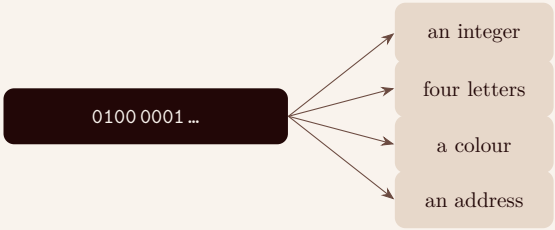
expected Int, got String

*Certainty that has cost nothing is either a **swindle** or a **theorem**.*

PART 1 · TYPE THEORY

Why have a type checker?

- Inside the machine, everything is bits.
- Thirty-two of them mean nothing by themselves.
- The processor neither knows nor cares: ask it to add a colour to an address, and it will.



Which one? Only the program's intentions decide.

PART 1 · TYPE THEORY

A common bug

A thing of one **kind** put where a thing of another kind was expected.

MARS CLIMATE ORBITER, 1999

- One program reported an impulse in **pound-force seconds**.
- The program reading it took the figure for **newton-seconds**.
- A spacecraft flew into the atmosphere of Mars.

NO TEST CAUGHT IT

A checker that knew the **kind** of each number would have refused to compile the hand-over.

PART 1 · TYPE THEORY

Type checkers are everywhere

BORN WITH ONE

Java
Rust

GREW ONE

JavaScript, as TypeScript, when the untyped web became unmanageable

ACQUIRED ONE

Python, with type hints, in 2015

*The checker's whole job is to say **no**, before anything runs.*

PART 1 · TYPE THEORY

What a type checker guarantees

A TEST

Checks the inputs *you thought of*.

Finitely many, and chosen by you.

THE CHECKER

Has run **none**, and vouches for **all**.

Never adds a string to a number; never calls a non-function.

*Whatever it is doing, it is not testing. So: what is it **looking at**? What is a **type**?*

PART 1 · TYPE THEORY

Monsters

- A few lines of code can, quite by accident, **run for ever**.
- No output, so no error to catch. It just goes on, like a loop with no exit.
- I call such programs monsters.

The red line was invented to keep these 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 · TYPE THEORY

Curry's proposal

Haskell Curry, 1920s, before there were computers: what is it to compute with functions?

EVERYTHING IS A FUNCTION

The only thing to do with one is to apply it to something.

COMPUTING IS REWRITING

A function with its arguments is replaced, by a fixed rule, with what it gives back.

TWO FUNCTIONS SUFFICE

One throws an argument away; one shares an argument out.

A language with no features at all, and it computes ***everything*** *any language can.*

Curry's own reasons were logical, not practical. The chapter ends with them.

CHAPTER 1

The Combinator Calculus

Why the compiler says no.

SECTION 1.1

Combinator algebra

Curry's outline, made exact.

1.1 · COMBINATOR ALGEBRA

Definition 1.1: terms

Fix two constants, K and S, and a stock of variables $x, y, z, f, g, \ldots$ The terms are defined inductively:

1. Each constant, K and 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.

IN ONE LINE

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

A term with no variables is a **combinator**.

$\mathsf{S}\mathsf{K}\mathsf{K}$ is one; $f\,(g\,x)$ is not.

1.1 · COMBINATOR ALGEBRA

Fewer brackets

Written out in full, a term is a thicket of brackets. Here is the term of Example 1.17, every bracket written:

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

*The convention: application groups to the **left**.*

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

$f\,(g\,x)$

f applied to $g\,x$: the brackets stay, because this grouping is not to the left.

$f\,g\,x$

$f\,g$ applied to x: that is, $((f\,g)\,x)$.

1.1 · COMBINATOR ALGEBRA

Rewriting

On their own, terms just sit there. We let them do **one** thing: rewrite.

- $M \rightsquigarrow N$: M steps to N, in one rewrite.
- There are infinitely many terms. We cannot say what each one does.
- Nor need we: say what K and S do, and how a step inside a term is a step of the whole.

Two rules to rewrite, and two to reach inside.

That is the whole of the next definition.

1.1 · COMBINATOR ALGEBRA

Definition 1.2: reduction

The relation $M \rightsquigarrow N$, "M steps to N", is the one given by these rules:

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

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

"K keeps the first of two arguments and drops the second; S hands its third argument to both the others; a step inside the function or the argument is a step of the application"

1.1 · COMBINATOR ALGEBRA

Reading a rule

Whenever what is above the bar holds, so does what is below.

NOTHING ABOVE

$$\frac{}{\mathsf{K}\,P\,Q \;\rightsquigarrow\; P}\;\mathsf{K}$$

Holds outright. (K) and (S) are the two ways to rewrite.

SOMETHING ABOVE

$$\frac{P \rightsquigarrow P'}{P\,Q \;\rightsquigarrow\; P'\,Q}\;\mathrm{app}_1$$

If P steps, so does P applied to Q. With (app_2) for the argument, a step can sit anywhere inside a term.

To show $M \rightsquigarrow N$ is to find rules that stack up to it.

1.1 · COMBINATOR ALGEBRA

K and S

K · DELETES

Keeps its first argument and **drops** the second.

$\mathsf{K}\,P\,Q \rightsquigarrow P$

S · COPIES

Hands its third argument to **both** the others and applies the first result to the second.

$\mathsf{S}\,P\,Q\,R \rightsquigarrow P\,R\,(Q\,R)$

Neither looks like anything you would want. Between them, they are everything you have ever wanted.

Terms built by application, with rules for rewriting them, make a *combinatory algebra*.

1.1 · COMBINATOR ALGEBRA

The brackets in the rules

The rules carry every bracket, because the brackets say which term is meant. With the convention, (K) reads $\mathsf{K}\,P\,Q \rightsquigarrow P$ and (S) reads $\mathsf{S}\,P\,Q\,R \rightsquigarrow P\,R\,(Q\,R)$.

WITH THE BRACKETS	AS WRITTEN	RULES
$((\mathsf{K}\,x)\,y) \rightsquigarrow x$	$\mathsf{K}\,x\,y \rightsquigarrow x$	(K)
$(((\mathsf{K}\,x)\,y)\,z) \rightsquigarrow (x\,z)$	$\mathsf{K}\,x\,y\,z \rightsquigarrow x\,z$	(K), then (app_1)
$(((\mathsf{S}\,x)\,y)\,z) \rightsquigarrow ((x\,z)\,(y\,z))$	$\mathsf{S}\,x\,y\,z \rightsquigarrow x\,z\,(y\,z)$	(S)
$(x\,((\mathsf{K}\,y)\,z)) \rightsquigarrow (x\,y)$	$x\,(\mathsf{K}\,y\,z) \rightsquigarrow x\,y$	(K), then (app_2)

Most of the mistakes you can make in this chapter are mistakes about where the hidden brackets go.

1.1 · COMBINATOR ALGEBRA

Example 1.3: a run, rule by rule

The first step of a run of $\mathsf{S}\,(\mathsf{K}\,x\,y)\,(\mathsf{K}\,u)\,w$, taking the inner step, with every rule drawn:

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

(K) does the rewriting; (app_2), then (app_1) twice, carry it out to the whole term, one pair of hidden brackets at a time.

1.1 · COMBINATOR ALGEBRA

Example 1.3: the other two steps

SECOND STEP: ONE RULE

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

The whole term has the shape of (S): x for P, $\mathsf{K}\,u$ for Q, w for R.

THIRD STEP: INTO THE ARGUMENT

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

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

1.1 · COMBINATOR ALGEBRA

Nobody draws the stack

A step is a step: write the run as a chain, with the rules on the right.

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

- A chain of steps is a run. $M \rightsquigarrow^* N$, "M runs to N": some run from M ends at N, zero steps included.
- Usually we name only the step at the top of the stack: "the K step".

1.1 · COMBINATOR ALGEBRA

A machine with one slot

- A term is a machine with a single slot: you put **one** thing in.
- Put f into K, and out comes a new machine, $\mathsf{K}\,f$, which ignores its next input and hands back f.
- So $\mathsf{K}\,f\,g$ is two machines used one after the other: $(\mathsf{K}\,f)\,g$. That is the bracketing the convention supplies.

Taking arguments one at a time is **currying**.

After Haskell Curry. Schönfinkel did it first, and Frege in passing; "schönfinkeling" never caught on.

1.1 · COMBINATOR ALGEBRA

A machine still waiting

A machine short of an input does nothing: it **waits**.

$$\mathsf{K}\,x \qquad \mathsf{S}\,x\,y \qquad \mathsf{K}$$

None of them is wrong. Each is a function waiting for another argument, and there is no step to take in it.

A CHOICE

Why not say in the grammar that K takes two arguments?

A tiresome bureaucracy, and it would hide that $\mathsf{K}\,f$, with one argument still to come, is a perfectly good term.

Example 1.4: the identity

Write I for $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$, a name for a term, not a new constant, and run it on any x:

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

Not one of the two constants, and yet **in the language**.

Whatever x is, $\mathsf{I}\,x \rightsquigarrow^{*} x$, in two steps.

1.1 · COMBINATOR ALGEBRA

Example 1.5: a puzzle about brackets

Run $\mathsf{K}\,\mathsf{I}\,f\,x$, counting the two steps of I as one. Two runs seem to be on offer:

$$\mathsf{K}\,\mathsf{I}\,f\,x \;\rightsquigarrow\; \mathsf{I}\,x \;\rightsquigarrow\; x \qquad \text{the } \mathsf{K} \text{ step, then the } \mathsf{I} \text{ step}$$

$$\mathsf{K}\,\mathsf{I}\,f\,x \;\rightsquigarrow\; \mathsf{K}\,f\,x \;\rightsquigarrow\; f \qquad \text{the } \mathsf{I} \text{ step, then the } \mathsf{K} \text{ step}$$

One term, two different results? One of these is ***not a run****.*

Example 1.5: put the brackets in

$$(((\mathsf{K}\,\mathsf{I})\,f)\,x)$$

- K is applied to I, the result to f, and that result to x.
- I and f stand side by side on the page, but I is one argument of K and f is another. Nowhere is I applied to f.
- So the I step of the second run has nothing to rewrite, and $\mathsf{K}\,\mathsf{I}\,f\,x \twoheadrightarrow^{*} x$.

The second run was about a different term, $\mathsf{K}\,(\mathsf{I}\,f)\,x$. Exercise 1.8 runs it.

SECTION 1.2

Confluence

Many runs, one answer.

1.2 · CONFLUENCE

Two runs of one term

(app_1) and (app_2) let a step happen anywhere inside a term. Take $E = \mathsf{K}\,x\,y\,(\mathsf{S}\,x\,y\,z)$: it can start at the front or in the argument.

$$
\begin{array}{lcll}
E & \rightsquigarrow & x\,(\mathsf{S}\,x\,y\,z) & (\mathsf{K}),\ (\mathrm{app}_1) \\
 & \rightsquigarrow & x\,(x\,z\,(y\,z)) & (\mathsf{S}),\ (\mathrm{app}_2) \\
E & \rightsquigarrow & \mathsf{K}\,x\,y\,(x\,z\,(y\,z)) & (\mathsf{S}),\ (\mathrm{app}_2) \\
 & \rightsquigarrow & x\,(x\,z\,(y\,z)) & (\mathsf{K}),\ (\mathrm{app}_1)
\end{array}
$$

Different routes, the same place: $x\,(x\,z\,(y\,z))$, where both runs stop.

1.2 · CONFLUENCE

Redex and reduct

A redex, short for *reducible expression*, is a term with the shape of the left of (K) or (S):

$$\mathsf{K}\,P\,Q \qquad\qquad \mathsf{S}\,P\,Q\,R$$

What the rule puts in its place is the reduct. A step rewrites exactly one redex.

In E: the redexes $\mathsf{K}\,x\,y$ and $\mathsf{S}\,x\,y\,z$, with reducts x and $x\,z\,(y\,z)$.

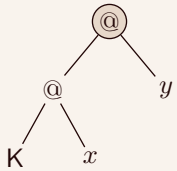
E itself is **not** a redex.

It is $\mathsf{K}\,x\,y$ applied to something more. It has two first steps because it contains two redexes.

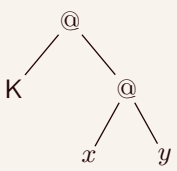
1.2 · CONFLUENCE

Redexes as trees

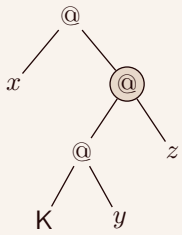
Draw each application as a node @: the function below it on the left, the argument on the right. Shaded: the top of a redex. The third tree has one inside it.



K x y



K (x y)



x (K y z)

A K*-redex: a* K *at the foot of* ***two*** *left branches. An* S*-redex:* ***three****.*

1.2 · CONFLUENCE

Finding the redexes

Read from the left. The first symbol is the *head*; the terms after it are its *arguments*. K needs two, S three. Then do the same inside every bracketed group.

TERM	ITS REDEXES
$\mathsf{K}\,x\,y$	the whole term; the step gives x
$\mathsf{K}\,(x\,y)$	none: 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 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: S has two arguments, each K one
$\mathsf{K}\,(\mathsf{K}\,x\,y)\,(\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,z)$	three: the whole term, $\mathsf{K}\,x\,y$, and $\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,z$

Rows 4 and 5 differ by one pair of brackets. Row 3: extra arguments are carried along.

1.2 · CONFLUENCE

Is that luck?

The two runs of E passed through different terms and arrived at the same one. ***Luck?***

IF IT WERE

"The result of running M" would mean nothing until you also said in what order, and the answer would depend on the implementation.

IT IS NOT

And the reason is a **theorem**.

1.2 · CONFLUENCE

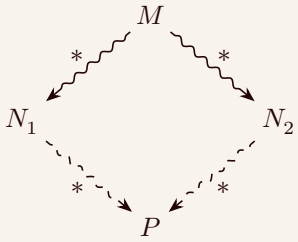
Theorem 1.6: Church–Rosser

THEOREM 1.6 · CHURCH–ROSSER · ON LOAN

If $M \twoheadrightarrow^* N_1$ and $M \twoheadrightarrow^* N_2$, then there is a term P with $N_1 \twoheadrightarrow^* P$ and $N_2 \twoheadrightarrow^* P$.

"however far two runs of the same term have gone apart, each can be continued so that they meet"

This property is *confluence*. Church and Rosser, 1936, for a close cousin of this calculus, the λ-calculus; it holds here too. On trust.



Given: the two runs at the top. Promised: the two dashed runs at the bottom.

1.2 · CONFLUENCE

Normal forms

A term with no redex is in normal form: no step can be taken, and a run that reaches one is over. The normal form is the run's result.

$$x \qquad \mathsf{K} \qquad \mathsf{K}\,x \qquad \mathsf{S}\,x\,y \qquad \mathsf{S}\mathsf{K}\mathsf{K}$$

What is normal about it? Nothing deep: it is the standard form a term settles into, if it settles at all.

$\mathsf{S}\mathsf{K}\mathsf{K}$ is a function **waiting for its argument**.

Give it one and a step appears. A normal form need not be a number, or short, or anything you would call an answer.

1.2 · CONFLUENCE

Lemma 1.7: the shape of a normal form

LEMMA 1.7 · THE SHAPE OF A NORMAL FORM

A term is in normal form exactly when it is a head that cannot fire, applied to normal forms $N_1, \dots, N_k$:

$$x\,N_1 \cdots N_k \qquad \mathsf{K} \quad \mathsf{K}\,N_1 \qquad \mathsf{S} \quad \mathsf{S}\,N_1 \quad \mathsf{S}\,N_1\,N_2$$

"a variable never fires; K fires on two arguments, S on three"

For combinators: K, $\mathsf{K}\,N$, S, $\mathsf{S}\,N$, $\mathsf{S}\,N\,N'$.

1.2 · CONFLUENCE

Lemma 1.7: why

Write the term as its head applied to its arguments, $h\,M_1 \cdots M_k$. Its redexes are of two kinds:

AT THE HEAD

h with its first two or three arguments, when h is K with $k \geq 2$ or S with $k \geq 3$.

INSIDE AN ARGUMENT

the redexes of $M_1, \ldots, M_k$.

No redex: a head that cannot fire, and arguments in normal form. SKKK is not one.

1.2 · CONFLUENCE

Corollary 1.8: one result at most

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 P with $N_1 \rightsquigarrow^* P$ and $N_2 \rightsquigarrow^* P$. No step can be taken in a normal form, so the only run from N_1 is the run of no steps: P is N_1. Likewise P is N_2.

That is what lets us speak of ***the*** *result of a term.*

1.2 · CONFLUENCE

Corollary 1.9: no step is a wrong turn

COROLLARY 1.9 · NO STEP IS A WRONG TURN

If $M \rightsquigarrow^* N$, where N is in normal form, and $M \rightsquigarrow^* M'$, then $M' \rightsquigarrow^* N$.

"however you have run M so far, its result is still within reach"

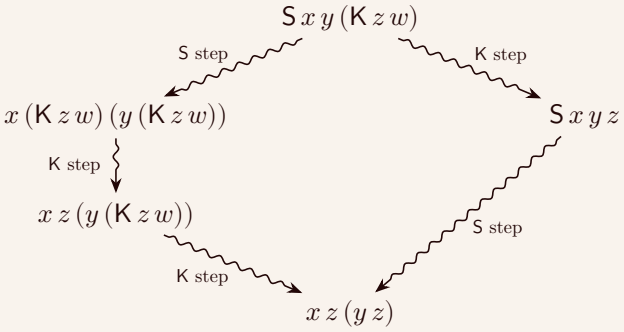
PROOF

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

A poor choice can put the result further off, never out of reach.

1.2 · CONFLUENCE

Example 1.10: a lopsided diamond



S first: the argument is handed to both, so two copies of $\mathsf{K}\,z\,w$ and two K steps.

K first: one S step finishes. Different lengths, the same normal form.

1.2 · CONFLUENCE

Example 1.10: K does the opposite

In $\mathsf{K}\,x\,(\mathsf{K}\,y\,z)$ the two redexes are the whole term and $\mathsf{K}\,y\,z$.

OUTER STEP FIRST

$$\mathsf{K}\,x\,(\mathsf{K}\,y\,z) \rightsquigarrow x$$

The inner redex is thrown away unrun.

INNER STEP FIRST

$$\mathsf{K}\,x\,(\mathsf{K}\,y\,z) \rightsquigarrow \mathsf{K}\,x\,y \rightsquigarrow x$$

One more step to reach the same x.

A step on one side of a diamond may need two steps on the other side, or ***none****.*

1.2 · CONFLUENCE

Why the theorem needs a proof

- S **copies** a redex; K **deletes** one. To delete and to copy is all a variable ever does.
- So the small diamonds are not all the same size.
- The obvious proof, tiling one-step diamonds, fails: the tiles do not fit.

Church and Rosser: keep track of a redex and *all its copies* at once.

The short proof of Tait and Martin-Löf rewrites all the copies in a single stride.

1.2 · CONFLUENCE

Exercise 1.7: redexes

Find every redex in each term, and say which are in normal form:

$\mathsf{K}\,(\mathsf{K}\,x)\,y \qquad \mathsf{K}\,(\mathsf{K}\,x\,y)$

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

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

The last has two redexes. Run it to normal form twice, taking each first, and **draw the diamond**.

EXERCISE 1.7

Three minutes. Count the arguments of every K and S.

SECTION 1.3

Evaluation strategies

Which redex first?

1.3 · EVALUATION STRATEGIES

Strict and lazy

An implementation must decide which redex to rewrite when there is more than one. The order it follows is its *evaluation strategy*.

STRICT, OR EAGER

Rewrite an **innermost** redex first, and work outwards: run the arguments before the call.

Python, Java, C, JavaScript, ML.

LAZY

Rewrite an **outermost** redex first, and work inwards: run an argument only when the function needs it.

Haskell.

*By Corollary 1.8 the choice cannot change the result, only the **work**.*

1.3 · EVALUATION STRATEGIES

The two diamonds, as strategies

$\mathsf{S}xy(\mathsf{K}zw)$: AN ARGUMENT USED TWICE

Strict: run $\mathsf{K}\,z\,w$, then the S step. Two steps.

Lazy: the S step copies the unrun argument, so it is run twice. Three steps.

$\mathsf{K}x(\mathsf{K}yz)$: AN ARGUMENT THROWN AWAY

Strict: a step on an argument that K then drops. Two steps.

Lazy: the K step at once. One step.

Each strategy wins in one case and loses in the other.

Lazy can repair its weakness by *sharing*: copy a pointer to the unrun argument, and run it once. That is *call by need*, Haskell's way.

1.3 · EVALUATION STRATEGIES

Each can imitate the other

STRICT, MADE LAZY BY HAND

Pass a function that computes the expression, and call it only when the value is wanted. In Python:

```
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))
```

Handed the branches as plain expressions, `my_if` would run both, and the recursion would never end.

LAZY, MADE STRICT BY HAND

Force a value before it is passed on: Haskell's `seq`.

What divides the strategies is the **default**.

1.3 · EVALUATION STRATEGIES

The default is a bet

1. **Predictable cost favours strict.** The work for an argument is paid at the call, so you can read the cost off the line. A lazy sum of a long list can pile up a million unrun additions.
2. **Effects favour strict.** Printing, reading, raising an error happen in the order written. Lazily, an effect happens when its result is demanded.
3. **Modularity favours lazy.** One part of a program describes an enormous list; another takes what it needs. Python's generators borrow the idea.

*Everything so far has assumed that the runs **end**.*

Confluence says two runs can be brought back together. It does not say that either one stops. Hold that thought until Section 1.5.

SECTION 1.4

Programming

What can so little compute?

1.4 · PROGRAMMING

What can it compute?

Running is plainly computing: a term goes in, a normal form comes out. So what can a language this small compute?

1. First, a way to write programs a person can read: **definitions**.
2. Then two surprises.

The promise so far: this tiny calculus captures what a computer can do.

Time to make good on it.

1.4 · PROGRAMMING

Definition 1.11: definitions

A definition is an equation

$$f\ x_1 \cdots x_n := M$$

where f is a new name, the x_i are distinct variables, and M is built from the constants, names defined earlier, and $x_1, \ldots, x_n$ alone. Then f is a term, and it steps by

$$\frac{}{f\,P_1 \cdots P_n \ \rightsquigarrow\ M \text{ with } P_i \text{ for } x_i}\ \text{def}$$

READ := AS

"is defined to be". It gives a name; it claims nothing.

That is all a function call is: the left side replaced by the right.

The I of Example 1.4 is now the definition $\mathsf{I} := \mathsf{S}\,\mathsf{K}\,\mathsf{K}$.

1.4 · PROGRAMMING

Is it a definition?

Check each against Definition 1.11: new name, distinct variables, body built from the constants, earlier names and the parameters.

EQUATION	A DEFINITION?
$twice\ f\ x := f\ (f\ x)$	**yes**
$swap\ f\ x\ y := f\ y\ x$	**yes**
$bad\ f\ x := f\ (g\ x)$	no: g is neither a parameter nor defined earlier
$bad\ f\ f := f$	no: the parameters are not distinct
$loop\ x := loop\ x$	no: a name cannot occur in its own definition

1.4 · PROGRAMMING

Example 1.12: three definitions

$$\mathit{twice}\ f\ x := f\ (f\ x)$$

$$\mathit{comp}\ f\ g\ x := f\ (g\ x)$$

$$\mathit{flip}\ f\ x\ y := f\ y\ x$$

- apply a function twice;
- chain two functions into one;
- swap two arguments.

A definition cannot reach outside itself: no numbers, no strings, no `if`*, no library, and* ***no recursion****.*

That looks crippling. The rest of this section is the evidence that it is not.

1.4 · PROGRAMMING

A debt

To watch definitions at work we want something visible to come out, and the language has no numbers.

So, for a while: the integers $0, 1, 2, \ldots$ and $+$ as **ready-made** constants, with the step

$$m + n \;\rightsquigarrow\; \text{the sum}$$

They are not terms of Definition 1.1. So this is a **debt**.

Example 1.14 pays it.

1.4 · PROGRAMMING

Example 1.14: numerals

A language with no numbers can make its own: a number n is "do it n times".

$$\overline{0}\, f\, x := x$$
$$\overline{1}\, f\, x := f\, x$$
$$\overline{2}\, f\, x := f\, (f\, x)$$
$$\overline{3}\, f\, x := f\, (f\, (f\, x))$$

- Add one: apply f once more.
- Add two numbers: apply f the first number of times to the result of applying it the second number of times.

$$succ\; n\, f\, x := f\, (n\, f\, x) \qquad add\; m\, n\, f\, x := m\, f\, (n\, f\, x)$$

1.4 · PROGRAMMING

Example 1.14: a printer

A numeral is a function, and a function cannot be read off the page until it is given something to work on. Give it the ready-made 0 and add-one:

$$inc\ y := y + 1$$

Then $\overline{n}\ inc\, 0$ adds one to 0, n times over, and out comes the digit n.

inc and 0 are a **printer** for numerals.

Like `print`: not part of the number, a way of seeing it.

1.4 · PROGRAMMING

Example 1.14: addition, printed

Run $add\,\overline{2}\,\overline{1}$ on inc and 0, one step per line.

$add\,\overline{2}\,\overline{1}\,inc\,0$	$\rightsquigarrow$	$\overline{2}\,inc\,(\overline{1}\,inc\,0)$	add
	$\rightsquigarrow$	$inc\,(inc\,(\overline{1}\,inc\,0))$	$\overline{2}$
	$\rightsquigarrow$	$inc\,(inc\,(inc\,0))$	$\overline{1}$
	$\rightsquigarrow$	$inc\,(inc\,(0+1))$	inc, innermost
	$\rightsquigarrow$	$inc\,(inc\,1) \rightsquigarrow inc\,(1+1)$	$0+1$; inc
	$\rightsquigarrow$	$inc\,2 \rightsquigarrow 2+1 \rightsquigarrow 3$	...

Nine steps, and the answer is 3*. I took the innermost* inc *at line 4; by Corollary 1.8 that cannot change the answer.*

1.4 · PROGRAMMING

Exercise 1.1: multiplication

Let $\mathit{mult}\, m\, n\, f := m\,(n\, f)$.

Run $\mathit{mult}\,\overline{2}\,\overline{3}\,\mathit{inc}\,0$, one step per line, naming each.

Then: what does *mult* compute in general? Why three parameters where *add* had four?

EXERCISE 1.1

Two minutes. Compare with your neighbour.

1.4 · PROGRAMMING

Exercise 1.1: answer

$$
\begin{array}{lll}
mult\,\overline{2}\,\overline{3}\,inc\,0 & & \\
\quad \rightsquigarrow & \overline{2}\,(\overline{3}\,inc)\,0 & mult \\
\quad \rightsquigarrow & \overline{3}\,inc\,(\overline{3}\,inc\,0) & \overline{2} \\
\quad \rightsquigarrow & \overline{3}\,inc\,(inc\,(inc\,(inc\,0))) & \overline{3} \\
\quad \rightsquigarrow & inc\,(inc\,(inc\,(inc\,(inc\,(inc\,0))))) & \overline{3}
\end{array}
$$

- Six unfoldings of inc give 6. In general: f applied n times, m times over, is mn times.
- Three parameters, because $mult\,m\,n\,f$ already names a function of x: namely $m\,(n\,f)$.

1.4 · PROGRAMMING

Example 1.15: booleans

$true := \mathsf{K}$

$false := \mathsf{K}\,\mathsf{I}$

`if b then M else N` is just $b\,M\,N$.

$$true\,M\,N \;\rightsquigarrow\; M$$
$$false\,M\,N \;\rightsquigarrow\; \mathsf{I}\,N \;\rightsquigarrow^*\; N$$

$false\,M\,N = \mathsf{K}\,\mathsf{I}\,M\,N$: K keeps I, drops M.

There is no `if` *in the language, and none is needed. The boolean* ***is*** *the conditional.*

1.4 · PROGRAMMING

Example 1.15: using a boolean

`if b then 10 else 20`, with $b = \mathit{false}$, is $\mathit{false}\ 10\ 20$:

$\mathit{false}\ 10\ 20$	$=$	$\mathsf{K}\,\mathsf{I}\,10\,20$	unfold false
	$\rightsquigarrow$	$\mathsf{I}\,20$	K step: keep I, drop 10
	$\rightsquigarrow^*$	20	Example 1.4

*The test chooses the branch by **what it does with its two arguments**.*

1.4 · PROGRAMMING

Theorem 1.16: two constants suffice

The first surprise. The defined names look as if they do the work, and K *and* S *are scaffolding. It is the* ***other way round****.*

THEOREM 1.16 · TWO CONSTANTS SUFFICE

Every definition by equations can be replaced by a combinator in K and S alone that runs the same way on every argument.

"anything you can write with parameters, you can write without them"

Schönfinkel, 1924. The proof comes with the Deduction Theorem, in the chapter on Hilbert proofs.

1.4 · PROGRAMMING

Example 1.17: composition

The *comp* of Example 1.12, from the constants alone: with $\mathsf{B} := \mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}$, we get $\mathsf{B}\,f\,g\,x \rightsquigarrow^* f\,(g\,x)$.

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

Four steps, and not a single parameter.

1.4 · PROGRAMMING

Example 1.18: twice, without parameters

twice $f\,x := f\,(f\,x)$ has two parameters and uses f twice. SBI has none:

$$\begin{aligned}
\mathsf{SBI}\,f\,x &\rightsquigarrow \mathsf{B}\,f\,(\mathsf{I}\,f)\,x && \mathsf{S}\text{ step: } P=\mathsf{B},\ Q=\mathsf{I},\ R=f\\
&\rightsquigarrow^{*} f\,(\mathsf{I}\,f\,x) && \mathsf{B}\text{, Example 1.17}\\
&\rightsquigarrow^{*} f\,(f\,x) && \mathsf{I}\text{, Example 1.4}
\end{aligned}$$

S *is there because* f *is used twice;* B *and* I *get each copy where it is wanted.*

1.4 · PROGRAMMING

Paying the debt

NUMBERS IN K AND S

$\overline{0} := \mathsf{K}\,\mathsf{I} \qquad succ := \mathsf{S}\,\mathsf{B}$

$\overline{1} := \mathsf{S}\,\mathsf{B}\,(\mathsf{K}\,\mathsf{I}),\ \ldots$

Check: $\mathsf{S}\,\mathsf{B}\,n\,f\,x \twoheadrightarrow^{*} f\,(n\,f\,x)$. And $\overline{0}$ is *false*.

AND THE READY-MADE PIECES

add is a combinator by Theorem 1.16. *inc* and 0 were only ever the printer.

$m + n \rightsquigarrow$ the sum is a shortcut, like a processor's `ADD`: fast, not necessary.

We keep the ready-made pieces, with a clear conscience.

1.4 · PROGRAMMING

Checking $\overline{1}$

Run $\overline{1} = \mathsf{S}\,\mathsf{B}\,(\mathsf{K}\,\mathsf{I})$ on any f and x:

$\mathsf{S}\,\mathsf{B}\,(\mathsf{K}\,\mathsf{I})\,f\,x$	$\rightsquigarrow$	$\mathsf{B}\,f\,(\mathsf{K}\,\mathsf{I}\,f)\,x$	S step
	$\rightsquigarrow^*$	$f\,(\mathsf{K}\,\mathsf{I}\,f\,x)$	B, Example 1.17
	$\rightsquigarrow$	$f\,(\mathsf{I}\,x)$	K step
	$\rightsquigarrow^*$	$f\,x$	I, Example 1.4

Exactly what $\overline{1}\,f\,x := f\,x$ asked for, with no parameter and no ready-made number.

1.4 · PROGRAMMING

Theorem 1.19: Turing completeness

The second surprise: what is left computes everything that can be computed.

THEOREM 1.19 · TURING COMPLETENESS · ON LOAN

The functions on numerals that combinators in K and S compute are exactly the functions that a Turing machine computes.

"anything any computer can compute, a combinator in K and S can compute"

Church, 1936, and his student Turing, 1937, for the closely related λ-calculus.

1.4 · PROGRAMMING

Example 1.20: factorial

ON THE WHITEBOARD

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

WHAT OUR LANGUAGE LACKS

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

And yet: watch what becomes of it.

1.4 · PROGRAMMING

Example 1.20: factorial as a combinator

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

where $A = \mathsf{S}\,(\mathsf{S}\,(\mathsf{K}\,isZero)\,\mathsf{I})\,(\mathsf{K}\,1)$, $C = \mathsf{S}\,(\mathsf{K}\,mult)\,\mathsf{I}$, $G = \mathsf{S}\,(\mathsf{K}\,pred)\,\mathsf{I}$

- No variable, parameter, `if` or recursive call: only K, S, ready-made constants and application.
- The conditional became $b\,M\,N$, the recursive call an extra first parameter r, and Theorem 1.16 compiled the parameters away.
- $F\,r\,n$ computes "if $n = 0$ then 1, else n times r applied to $n - 1$". Nobody has to write this: a compiler would.

1.4 · PROGRAMMING

Example 1.20: the loop

Missing: something to put in for r that behaves as F itself does. No name may occur in its own definition, but a function may be applied to itself.

$$\mathit{self}\ f\,x := f\,(x\,x) \qquad\qquad \mathsf{Y}\,f := \mathit{self}\ f\,(\mathit{self}\ f)$$

$\mathsf{Y}\,f$	$\rightsquigarrow$	$\mathit{self}\ f\,(\mathit{self}\ f)$	Y
	$\rightsquigarrow$	$f\,(\mathit{self}\ f\,(\mathit{self}\ f))$	self, with $\mathit{self}\ f$ for x

- Write Θ for $\mathit{self}\ f\,(\mathit{self}\ f)$: then $\Theta \rightsquigarrow f\,\Theta$. It hands f a copy of itself, as often as asked.
- With F for f, that copy is F's r, and $\mathsf{Y}\,F\,4$ runs to 24. (Run lazily. The venture fund Y Combinator is named after it.)

Recursion was in the language all along, hiding in ***self-application****.*

1.4 · PROGRAMMING

What is left of a program

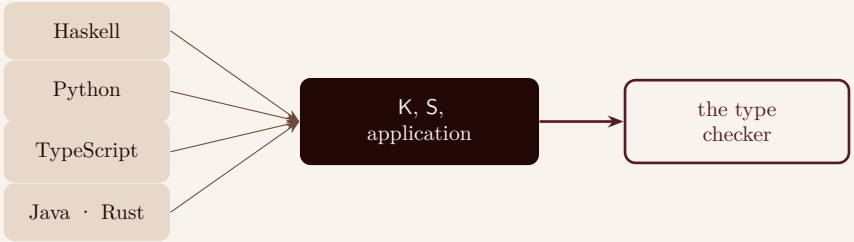
- A program is a finite list of definitions and a term to evaluate.
- A definition is an equation.
- To run the term is to replace left-hand sides by right-hand sides until none is left.
- No store, no stack, no instruction pointer. The order of steps cannot change the answer, though it can change whether you get one.

Not a small language chosen for convenience: **the general case**.

Your own languages, with everything a checker does not read deleted.

1.4 · PROGRAMMING

Why study this language?



*Understand the checker for **this** language once, and you understand type-checking for **all** of them.*

Hold on to that: something is about to go badly wrong here, and it goes wrong in yours too.

SECTION 1.5

The Monster

Programs that never stop.

1.5 · THE MONSTER

Self-application

Confluence says a term has at most one normal form. It does not say there is one. Nothing says what may be applied to what, so a function may be applied to **itself**:

$\omega\, x := x\, x \qquad \Omega := \omega\, \omega$ **the Monster**

Definition 1.11 allows it: the right-hand side uses only its one parameter.

One step is available. It returns the term it started from.

A perfectly legal construction that nobody who wrote the definitions had in mind.

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

1.5 · THE MONSTER

The cost of not stopping

- No output. No error. No crash.
- The machine is working, and will still be working tomorrow.
- A page that spins. A build that never finishes. A request holding its connection open, and then a thousand more behind it.

A wrong answer can at least be read, and argued with.

This returns **nothing**, and takes the machine with it.

1.5 · THE MONSTER

Compiling does not help

In the constants, ω is SII:

$$\begin{array}{lll} \mathsf{SII}x & \rightsquigarrow & \mathsf{I}x(\mathsf{I}x) \qquad\qquad \mathsf{S} \text{ step} \\ & \rightsquigarrow^* & x(\mathsf{I}x) \rightsquigarrow^* xx \qquad \mathsf{I}, \text{ twice} \end{array}$$

- So Ω is $\mathsf{SII}(\mathsf{SII})$: several steps to come back to itself instead of one. The loop is in the **shape**, not the notation.
- It is the loop of Y with nothing inside: *self* I does what ω does, and YI runs for ever.

1.5 · THE MONSTER

Another monster: Ω_3

There is not one monster but infinitely many. Apply to itself a function that uses its argument **three** times:

$$\omega_3\, x := x\, x\, x \qquad \Omega_3 := \omega_3\, \omega_3$$

$$\begin{aligned} \omega_3\,\omega_3 &\rightsquigarrow \omega_3\,\omega_3\,\omega_3 \\ &\rightsquigarrow \omega_3\,\omega_3\,\omega_3\,\omega_3 \\ &\rightsquigarrow \omega_3\,\omega_3\,\omega_3\,\omega_3\,\omega_3 \rightsquigarrow \cdots \end{aligned}$$

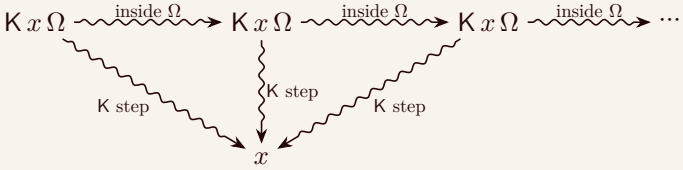
It never comes back to where it started: it **grows**.

So a loop need not repeat itself, and "watch for a repeat" cannot catch them all.

1.5 · THE MONSTER

When the order decides

$\mathsf{K}\,x\,\Omega$ has two redexes. The step inside Ω gives $\mathsf{K}\,x\,\Omega$ back; the K step gives x at once.



Here the order of steps decides not the cost of a run, but ***whether it ends****.*

1.5 · THE MONSTER

Strict and lazy on $\mathsf{K}\,x\,\Omega$

Exercise 1.4 asks you to watch both happen.

LAZY: THE OUTER STEP

$\mathsf{K}\,x\,\Omega \rightsquigarrow x$

A normal form; the run stops. In Haskell, `const 1 undefined` is `1`.

STRICT: THE INNER STEP

$\mathsf{K}\,x\,\Omega \rightsquigarrow \mathsf{K}\,x\,\Omega \rightsquigarrow \cdots$

Never stops. In Python, `K(1)(w(w))` runs until the stack gives out.

A fourth consideration, and it favours lazy: ***termination.***

Leftmost outermost first finds the normal form whenever there is one (Curry and Feys, 1958).

1.5 · THE MONSTER

What confluence does not promise

$\mathsf{K}\,x\,\Omega$ looks like a counterexample to Church–Rosser. It is not.

WHAT IT PROMISES

At most one normal form (Corollary 1.8). No step puts it out of reach (Corollary 1.9): the arrows down from the row.

Both are true of $\mathsf{K}\,x\,\Omega$.

WHAT IT DOES NOT

That *every* order of steps reaches the normal form, or that there is one to reach.

Both false: the inner-first run of $\mathsf{K}\,x\,\Omega$ never arrives, and Ω has nowhere to arrive at.

1.5 · THE MONSTER

Monsters come with the territory

- Theorem 1.19: the language computes everything a Turing machine computes, and a Turing machine can run for ever.
- Banish every monster, and the language **computes less**.
- Nor can a checker refuse *exactly* the programs that fail to stop, every one that loops and none that halts: the halting problem, proved for these terms in Chapter 8.

YOU WILL NOT SEE IT COMING

Ω is four symbols. Three library calls and a type nobody wrote down, and it announces nothing.

1.5 · THE MONSTER

What a checker can achieve

WHAT CANNOT BE HAD

A checker that refuses exactly the programs that fail to stop.

No such checker exists.

WHAT WE SHALL BUILD

A checker that refuses a class of programs we can **describe exactly**, with Ω inside it.

SECTION 1.6

Types

Write the bookkeeping down.

1.6 · TYPES

What went wrong

- ω takes an argument and returns a result.
- Nothing recorded *what kind* of either.
- So nothing could notice that ω had been handed something it was never built to take.

Functions and arguments, and **no bookkeeping whatever**.

1.6 · TYPES

The idea

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*.

1.6 · TYPES

Definition 1.21: types

Fix basic types, such as Int, $String$, $Bool$; write p, q, r for them. The types are defined inductively:

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

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

EXAMPLES

$Int \to Bool$: a test on integers

$(Int \to Int) \to Int$: takes a function

$(p \to q) \to p$

Read $A \to B$ as a **contract**: give me an A and I'll return a B.

1.6 · TYPES

Types in Java

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

JAVA	TYPE
`static int square(int n)`	$Int \to Int$
`Function<Integer, Boolean> isEven = n -> n % 2 == 0;`	$Int \to Bool$
`Function<Integer, Function<Integer, Integer>> add = m -> n -> m + n;`	$Int \to Int \to Int$

*Java nests the function type to the **right**, exactly as the next slide does.*

1.6 · TYPES

The arrow groups to the right

$A \to B \to C$ means $A \to (B \to C)$

A function of two arguments, taking them one at a time.

$(A \to B) \to C$ is **different**

A function that takes a function.

WHY THE TWO CONVENTIONS ARE OPPOSITE

If $f : A \to B \to C$, $x : A$ and $y : B$, then $f\,x : B \to C$ and $f\,x\,y : C$. The term $f\,x\,y$ is $(f\,x)\,y$, and its type $A \to (B \to C)$ gives up its arguments from the left in the same order.

*Each convention drops the brackets of the commonest shape, and the two shapes **match**.*

1.6 · TYPES

Definition 1.22: context and judgement

A context Γ is a finite list pairing each variable in scope with a type, none listed twice:

$$\Gamma \;=\; x_1 : A_1,\; \ldots,\; x_n : A_n$$

A judgement

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

"M has type A when its variables have the types Γ gives them." KS names the system. Empty context: a program typed outright.

YOU HAVE WRITTEN CONTEXTS

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

The declarations are Γ. Java checks the last line against them, which is the judgement

$f : Int \to Int,\; x : Int \;\vdash_{\mathrm{KS}}\; f\,(f\,x) : Int$

1.6 · TYPES

Definition 1.24: the type schemas

Each schema is just what its combinator does, written as a contract. A, B, C stand for any types at all.

K KEEPS

$\mathsf{K}\,P\,Q \rightsquigarrow P$

Give it an A, then a B. It keeps the A and drops the B:

(K) $\mathsf{K} : A \to B \to A$

S SHARES

$\mathsf{S}\,P\,Q\,R \rightsquigarrow P\,R\,(Q\,R)$

It shares R with both. If $R : A$, then $Q : A \to B$, and P takes an A and that B to a C:

(S) $\mathsf{S} : (A \to B \to C) \to (A \to B) \to A \to C$

Read the type off the rule: ***what goes in, what comes out.***

1.6 · TYPES

Definition 1.24: a typing

To show $\Gamma \vdash_{\mathrm{KS}} M : A$, write a typing: the context Γ above a line, and below it numbered lines $M_k : A_k$, each justified in one of four ways:

1. (K): M_k is K, and A_k is an instance of (K);
2. (S): M_k is S, and A_k is an instance of (S);
3. (var): M_k is a variable declared in Γ with type A_k;
4. (app): earlier lines i, j have $A_i = A_j \to A_k$ and $M_k = M_i\, M_j$.

WRITING THE JUSTIFICATIONS

"(app) i, j": line i is the function, line j the argument.

After (K) or (S): in brackets, the types put for A, B, C.

A typing whose last line is $M : A$ **witnesses** $\Gamma \vdash_{\mathrm{KS}} M : A$.

1.6 · TYPES

Instances of a schema

(K) is one line, but it gives K a type for every choice of A and B:

A	B	A TYPE OF K
Int	$Bool$	$Int \to Bool \to Int$
$Bool$	Int	$Bool \to Int \to Bool$
$Int \to Int$	$String$	$(Int \to Int) \to String \to (Int \to Int)$

A schema is what Java, Rust and TypeScript call a ***generic****.*

In Java, (K) is `static <A, B> Function<B, A> k(A a) { return b -> a; }`, and javac chooses `A` and `B` at each use.

1.6 · TYPES

Ready-made pieces

The integers, + and the strings are not terms of Definition 1.1, so a typing has no case for them, and needs none.

A program that uses them **says what they are**, in the context:

$$3 : Int \qquad \texttt{"hello"} : String$$

$$+ : Int \to Int \to Int$$

A line that uses one is justified by (var).

Only the pieces a term uses need be declared.

SKK uses none, so it is typed in the empty context.

1.6 · TYPES

Example 1.25: a first typing

`if true then 3 else 4` is $\mathsf{K}\,3\,4$. It has no variables, but it uses two ready-made numerals, so they are declared above the line: $3 : \mathit{Int}$, $4 : \mathit{Int}$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$\mathsf{K} : \mathit{Int} \to \mathit{Int} \to \mathit{Int}$	(K) $[\mathit{Int},\ \mathit{Int}]$
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

An Int, and we know it before anything runs.

With `"hello"` : String for 4 and $B := \mathit{String}$, $\mathsf{K}\,3$ `"hello"` : Int too: K does not care what it drops.

1.6 · TYPES

Example 1.26: a typing with parameters

The body of *twice* $f\,x := f\,(f\,x)$. Declare its parameters: $\Gamma = f : Int \to Int,\ x : Int$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$f : Int \to Int$	(var)
2.	$x : Int$	(var)
3.	$f\,x : Int$	(app) 1, 2
4.	$f\,(f\,x) : Int$	(app) 1, 3

Γ *is the parameter list.*

Line 1 is cited twice, which is allowed. The definition has no case for a *name* such as *twice* itself: a rule for names is still missing.

1.6 · TYPES

Example 1.27: the identity, typed

Type $\mathsf{I} = \mathsf{S}\mathsf{K}\mathsf{K}$ at $A \to A$, in the empty context. Write P for $A \to A$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$\mathsf{S} : (A \to P \to A) \to (A \to P) \to P$	(S) $[A,\ P,\ A]$
2.	$\mathsf{K} : A \to P \to A$	(K) $[A,\ P]$
3.	$\mathsf{S}\mathsf{K} : (A \to P) \to P$	(app) 1, 2
4.	$\mathsf{K} : A \to P$	(K) $[A,\ A]$
5.	$\mathsf{S}\mathsf{K}\mathsf{K} : A \to A$	(app) 3, 4

Line 1 is the one nobody writes unprompted: it is found backwards from the goal. The search takes a paragraph; the check takes none.

1.6 · TYPES

Example 1.27: finding line 1

Work backwards from the goal. Leave the letters of (S) unknown, and let the two K's decide them.

1. Goal: $\mathsf{SKK} : A \to A$.
2. (S) gives $\mathsf{S} : (X \to Y \to Z) \to (X \to Y) \to X \to Z$, so $\mathsf{SKK} : X \to Z$. Hence $X = A$ and $Z = A$.
3. The first K must have type $A \to Y \to A$: the shape $V \to W \to V$ of (K), whatever Y is.
4. The second K must have type $A \to Y$, and the shape forces $Y = W \to A$, for some W.
5. Any W will do. Choose $W = A$: then $Y = A \to A$, which is P. That is line 1.

Give each unknown a letter, read off the equations, solve them: ***unification****, what Haskell's and Rust's checkers do for you.*

1.6 · TYPES

Two schemas, one look-up, one match

That is the whole checker. Read the four cases as it reads your code.

COMBINATORS: A SCHEMA EACH

(K) $\mathsf{K} : A \to B \to A$ (S) $\mathsf{S} : (A \to B \to C) \to (A \to B) \to A \to C$

VARIABLES: LOOK IT UP

(var) x has the type it was declared with in Γ.

APPLICATIONS: MATCH

(app) $M : A \to B$ and $N : A$ give $M\,N : B$.

When the types do not match, this is the case that draws the red line.

1.6 · TYPES

Example 1.28: the red line

THE LINE YOU WALKED IN TO

```
total = 3 + "hello"
            ^^^^^^^

expected Int, got String
```

Without types nothing stops it: inside the machine 3 and `"hello"` are both just bits.

NOW THE CHECKER: TYPE $+3$ `"hello"`

$\Gamma = + : Int \to Int \to Int,\ 3 : Int,\ \texttt{"hello"} : String$

LINE	JUDGEMENT	JUSTIFICATION
1.	$+ : Int \to Int \to Int$	(var)
2.	$3 : Int$	(var)
3.	$+3 : Int \to Int$	(app) 1, 2
4.	$\texttt{"hello"} : String$	(var)
5.	(app) 3, 4? wants Int, gets $String$: **no match**	

No rule applies, so no typing exists: refused, before anything runs.

That is the whole content of the error in your editor, minus the decoration.

1.6 · TYPES

The same error in Java

Java's own + would quietly give `"3hello"`, so use an addition typed like ours, $Int \to Int \to Int$:

```
Function<Integer, Function<Integer, Integer>> plus = m -> n -> m + n;
int total = plus.apply(3).apply("hello");
```

JAVAC

```
error: incompatible types: String cannot be converted to Integer
int total = plus.apply(3).apply("hello");
                          ^
```

`plus.apply(3)` *is our line 3, an* $Int \to Int$*. The caret is our line 5:* ***no match.***

SECTION 1.7

Subject reduction

What the checker said stays said.

1.7 · SUBJECT REDUCTION

Types during a run

The checker gives its verdict *before* the program runs. But running **rewrites** the term:

$$\mathsf{K}\,3\,4 \rightsquigarrow 3$$

The checker typed $\mathsf{K}\,3\,4$, not 3. Is the term we end up with still an Int?

If one 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

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.

1.7 · SUBJECT REDUCTION

Theorem 1.29: subject reduction

THEOREM 1.29 · SUBJECT REDUCTION

If $\Gamma \vdash_{\mathrm{KS}} M : A$ and $M \twoheadrightarrow N$ by a K step, an S step or a ready-made step, then $\Gamma \vdash_{\mathrm{KS}} N : A$.

"the type the checker gave a program is still its type after any step of its run"

1.7 · SUBJECT REDUCTION

Proof: a K step

Suppose $\mathsf{K}\,P\,Q : A$ and $\mathsf{K}\,P\,Q \rightsquigarrow P$. Why is $P : A$? Read the typing backwards twice.

1. $\mathsf{K}\,P\,Q : A$ came by (app) from $\mathsf{K}\,P : B \to A$ and $Q : B$.
2. $\mathsf{K}\,P : B \to A$ came by (app) from $\mathsf{K} : C \to B \to A$ and $P : C$.
3. (K) gives K only types of the shape $V \to W \to V$, so $C = A$.

Hence $\Gamma \vdash_{\mathrm{KS}} P : A$.

Each kind of term has exactly one rule that could have typed it: that is what lets us read backwards.

1.7 · SUBJECT REDUCTION

Proof: an S step

Now M is $\mathsf{S}\,P\,Q\,R$ and N is $P\,R\,(Q\,R)$.

BACKWARDS, THREE TIMES

$R{:}D,\ \ Q{:}E,\ \ P{:}F,\ \ \mathsf{S}{:}F \to E \to D \to A.$

(S) forces $F = X \to Y \to Z,\ \ E = X \to Y,\ \ D = X,\ \ A = Z.$

THEN FORWARDS

(app): $P\,R : Y \to Z$

(app): $Q\,R : Y$

(app): $P\,R\,(Q\,R) : Z$, and Z is A.

1.7 · SUBJECT REDUCTION

Proof: the other cases

A READY-MADE STEP

$m + n \rightsquigarrow$ the sum. Γ declares $+$ and the integers, so both sides are Int.

INSIDE A LARGER TERM

Replace the lines that type the rewritten part by the lines just built, which end in the same type. Every later line stands: (app) looks only at the **types** of its two parts.

***What the checker said stays said**, at every step of the run.*

1.7 · SUBJECT REDUCTION

Theorem 1.30: normalisation

THEOREM 1.30 · NORMALISATION · ON LOAN

Every program with a typing has a normal form.

WHAT IT BANS

Ω has no normal form, so it has **no typing**; nor has any term with Ω inside it.

THE PRICE

Some programs that do stop are refused too: $\mathsf{K}\,x\,\Omega$ has a normal form and no typing.

*A typed program is safe twice over: no step goes wrong, and the run **ends**.*

1.7 · SUBJECT REDUCTION

Turing and Tait

TURING, EARLY 1940S

Some order of steps always finishes.

Always take the step whose function has the most complicated type: a measure of the term then goes down at every step, and cannot go down for ever.

Unpublished; Gandy printed it in 1980. Enough for Theorem 1.30.

TAIT, 1967

Every order of steps finishes.

No simple measure works for every order. Instead, prove something *stronger* than "finishes", by induction on the type.

The idea this course does not reach. Girard took it to generics in 1972.

After Tait: strict or lazy, a typed program finishes either way.

1.7 · SUBJECT REDUCTION

Recursion comes back

- Every real language gives the loop back: a `while`, a function that calls itself.
- Y cannot be typed: it is built from $x\,x$. So a typed language that wants recursion has to **add** it, as a new constant.
- Exercise 1.6 asks what that costs.

EXERCISE 1.6

Add fix with $\mathit{fix}\,f \rightsquigarrow f\,(\mathit{fix}\,f)$ and type $(A \to A) \to A$. Which theorem survives?

HISTORY

What Curry built it for

It was never meant to be a programming language.

HISTORY

Discovered twice

- The combinators were discovered twice in the 1920s, when there were no programs to write.
- Moses **Schönfinkel**, in Hilbert's Göttingen: a talk in 1920, printed 1924.
- Haskell **Curry**, a young American teaching mathematics at Harvard, around 1926.

Both were after something in **logic**.

To see what, we need the logic of the day.

HISTORY

A rulebook for mathematics

- Mathematicians wanted a rulebook for all of mathematics: a proof written down and checked step by step, by rule alone, with no judgement needed.
- The great attempt: Whitehead and Russell's *Principia Mathematica* (1910–13).
- It worked, at a price: hundreds of pages to reach $1 + 1 = 2$.

Checking by rule alone, without running anything, is what your type checker does to a program.

They wanted that for every proof in mathematics.

HISTORY

Schönfinkel

- He was after **economy**: logic from as few basic notions as possible. His title says so: *On the building blocks of mathematical logic.*
- The surprise: the variables of logic, the x in "for every x", can be done away with altogether. K and S do their work.
- A colleague, Heinrich Behmann, wrote the talk up for publication in 1924.

AFTERWARDS

He returned to Moscow, fell ill, and died in poverty in 1942. It is said his neighbours burned his papers for heat.

K and S are most of what survives of him.

HISTORY

Curry and substitution

- Detachment: from A and "if A then B", conclude B. Simple.
- Substitution: putting a formula in place of a variable throughout another. Far more complicated, and a rule that basic ought to be simple.
- He took it apart. A variable is only ever **kept, thrown away, copied or moved**; a few fixed operations that do those things can do all the work variables do.

K throws an argument away; S copies one: enough.

BEATEN TO IT

1927, Princeton: Curry finds Schönfinkel's paper, and is distressed.

HISTORY

Curry's plan

- If combinators do the work of variables, they can sit under the **whole of logic**: build the logic itself from constants and application.
- Functions can be everything. You have seen it: the booleans, the numbers and addition, all built from K and S.
- A few simple rules, easy to work with and easy to check. What could go wrong?

Within a few years the logic was shown **inconsistent**: Kleene and Rosser, 1935.

Church's logic of 1932, built on the same plan with λ, fell the same way. What broke them is the Monster in another guise.

HISTORY

Richard's paradox

Now, a puzzle. (Jules Richard, 1905.)

1. A description is an English phrase that pins down a function on numbers.

   "Double the number" describes
   $0 \mapsto 0,\ 1 \mapsto 2,\ 2 \mapsto 4, \ldots$
2. A description is a finite string of letters. So we can **list** them all: shortest first, and in dictionary order within each length.
3. So the functions they describe come in a list too:

   $f_0,\ f_1,\ f_2,\ f_3,\ \cdots$

   Phrases that describe no function are skipped.

THE START OF THE LIST, SAY

FUNCTION	DESCRIPTION
f_0	"add one"
f_1	"double the number"
f_2	"square the number"
f_3	…

HISTORY

Richard's paradox: the diagonal

EACH FUNCTION'S VALUES

VALUE AT	0	1	2	3
f_0	**1**	2	3	4
f_1	0	**2**	4	6
f_2	0	1	**4**	9
f_3	·	·	·	·
d	2	3	5	⋯

The plan: a function d that differs from *every* function on the list: from f_0 at 0, from f_1 at 1, from f_2 at 2, …

How: go down the diagonal and add one.

$$d\ n := f_n\ n + 1$$

In English: "add one to what the nth function on the list gives at n".

The paradox: that was a description. So d is on the list, say $d = f_k$. Then

$$d\ k\ =\ f_k\ k + 1\ =\ d\ k + 1.$$

A number equal to itself plus one.

Kleene and Rosser, 1935: a logic made of combinators can do this *inside* itself, and so proves $0 = 1$.

HISTORY

Richard's paradox and the Monster

- Going down the diagonal applies the nth function to **its own number**: $f_n\ n$. When d meets itself, at $n = k$, it too is applied to its own number.
- That is self-application, the move $\omega\, x := x\, x$ makes, dressed up as a list.
- Read Richard's equation as a rule for running, and it is a monster:

$$d\ k \ \rightsquigarrow\ d\ k{+}1 \ \rightsquigarrow\ (d\ k{+}1){+}1 \ \rightsquigarrow\ \cdots$$

A LOGIC

believes its equations, so a number equals itself plus one, and then anything at all.

A PROGRAM

only runs them, so it **never stops**.

Either way: a thing made to act on itself.

HISTORY

Through its types

Suppose ω could be typed, and give its parameter x some type A. Look at the body $x\,x$:

The first x is a function: type $A \to B$.

The second x is its argument: type A.

But it is the **same** x, with one type:

$$A \;=\; A \to B$$

- A type that contains itself as a proper part, as d had to be on the list it was built from. None does: $A \to B$ is longer than A.
- So ω has no type, and the checker refuses the Monster at the very step where it applies something to itself. Y falls the same way.

Unification calls this the *occurs check*. Ask Haskell for the type of `\x -> x x`: "infinite type".

HISTORY

The repair

- The repair that lasted was **types**: Curry began typing his combinators in 1934; Church built types into his logic in 1940, the simply typed λ-calculus.
- Types sit outside the language of terms. "$M : A$" 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 the diagonal never starts.

Your compiler is built on the repaired discipline. That is why the checker's whole job is to refuse.

Swindle or theorem? The compiler's certainty was the ***theorem****.*

HISTORY

Timeline

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.
1934	Curry works out the types of K and S.
1935	Kleene and Rosser: Curry's logic is inconsistent.
1936	Church and Turing: what can be computed. Church and Rosser: order 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.

A logic needed saving, and programmers inherited the repair.

SUMMARY

Chapter 1

- Under every language you use is a combinatory algebra: two constants and application, run by two rewrite rules, in which every definition you have written can be expressed. Unpoliced, it computes everything, including a program that never stops.
- A run may take its steps in any order. By Church–Rosser the order cannot change the result, though it can change the work and whether a result is reached.
- A type is a contract, a basic type or $A \to B$. Its certificate is a typing: the context above a line, numbered lines below, each justified by (K), (S), (var) or (app).
- Its *no* is (app) failing to match, as for $3 + \texttt{"hello"}$ and for $x\,x$. Its *yes* covers every input: by subject reduction it stays true at every step, and by normalisation the run ends.

READING

Sources

- Wadler, *Propositions as types.* CACM 2015. The story of Part 1, in a dozen pages.
- Schönfinkel (1924). Where K and S come from.
- Church (1936); Turing (1937). The calculus computes everything.
- Church and Rosser (1936). The order of steps does not change the answer.
- Curry (1930); Curry and Feys (1958). Curry's own start, and the book that came of it.
- Kleene and Rosser (1935); Curry (1942). Where his logic broke.
- Tait (1967). Normalisation.

Thank you.