

PART 1: TYPE THEORY · CHAPTER 1

# Combinators

---

Alexander V. Gheorghiu

COMP0009 Logic · University College London

PROLOGUE

# A confession

---

*Before we begin: where this course stands.*

# What logic is

---

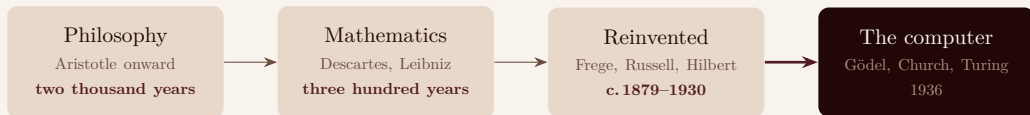
*Logic is the **science of reasoning**.*

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

PROLOGUE

# A short history, told fast

---



*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.

## The only 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.*

CHAPTER 1

# Combinators

---

*Why the compiler says no.*

# Your compiler refused

---

- ▶ 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"
```

```
~~~~~
```

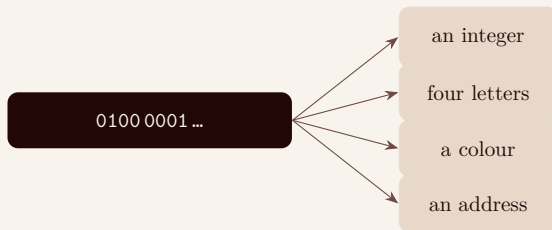
```
expected Int, got String
```

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

## Why is there a checker at all?

---

- ▶ 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.

## The commonest 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.

# Every language you use has one

---

## 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.*

## What does *yes* mean?

---

### 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**?*

SECTION 1.1

# Terms and reduction

---

*Keep only what every checker reads.*

## What does a checker attend to?

---

Hundreds of languages, hundreds of checkers, and no two alike on the page. The mathematician's move: **abstract**.

- ▶ Not what a function computes, only what it takes and returns.
- ▶ Not the value of an expression, only how it is put together.
- ▶ Not the keywords or libraries, only *which thing is applied to which*.

Strip everything else away, and very little is left:

things that are applied, things they are applied to, and the act of applying.

## Definition 1.1: terms

---

Fix two constants,  $K$  and  $S$ , and a stock of variables  $x, y, z, f, g, \dots$ . 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

$$M, N ::= K \mid S \mid x \mid M N$$

Application groups left:  $f x y$  means  $(f x) y$ . Brackets only for grouping to the right, as in  $f (g x)$ .

IN HASKELL

```
data Term = K | S | Var String | App
  Term Term
```

## Definition 1.1: definitions and programs

---

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  $x_1, \dots, x_n$  alone.

READ := AS

“is defined to be”. It introduces a name; it does not claim two things are equal.

A **program** is a *closed* term: no variable left.

In a closed term there is nothing left to supply.

## 1.1 · TERMS AND REDUCTION

## Program or fragment?

---

A term with variables in it is a fragment: it says what to do *once* they are given.

| TERM    | PROGRAM OR FRAGMENT?         |
|---------|------------------------------|
| $SKK$   | <b>program</b>               |
| $K(SK)$ | <b>program</b>               |
| $f(gx)$ | fragment: $f, g, x$ are open |
| $SxK$   | fragment: $x$ is open        |

*The checker will need to be told the kinds of the variables a fragment leaves open. Keep that in mind for Section 1.3.*

## Example 1.2: definitions you have already written

---

$twice\ f\ x := f\ (f\ x)$

$comp\ f\ g\ x := f\ (g\ x)$

$flip\ f\ x\ y := f\ y\ x$

`twice f x = f (f x)`

`comp f g x = f (g x)`

`flip f x y = f y x`

In Python, three defs.

*You have written all three. What a definition may **not** do: reach outside itself. No numbers, no strings, no `if`, nothing from a library.*

## 1.1 · TERMS AND REDUCTION

## Is it a definition?

---

Check each against Definition 1.1: new name, distinct variables, body built from the constants, earlier names and the parameters.

| CANDIDATE DEFINITION                | IS IT A DEFINITION?                                |
|-------------------------------------|----------------------------------------------------|
| $\textit{twice } f\ x := f\ (f\ x)$ | <b>yes</b>                                         |
| $\textit{swap } f\ x\ y := f\ y\ x$ | <b>yes</b>                                         |
| $\textit{bad } f\ x := f\ (g\ x)$   | no: $g$ is neither a parameter nor defined earlier |
| $\textit{bad } f\ f := f$           | no: the parameters are not distinct                |

## Definition 1.3: running

---

A step rewrites a term in one of three ways, anywhere inside a larger term:

$$\mathbf{K} M N \rightsquigarrow M$$

$$\mathbf{S} M N P \rightsquigarrow M P (N P)$$

$$f M_1 \cdots M_n \rightsquigarrow M \text{ with } M_i \text{ for each } x_i$$

the third whenever  $f x_1 \cdots x_n := M$  is a definition.

- A term with no step available is in normal form.
- To run a term is to take steps until normal form.  $M \rightsquigarrow^* N$ :  $N$  is reachable from  $M$ , zero steps included.

## What the two constants do

---

K

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

$$K M N \rightsquigarrow M$$

S

Hands its third argument to **both** the others and applies the results.

$$S M N P \rightsquigarrow M P (N P)$$

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

Terms plus rewrite rules make a *combinatory algebra*; the constants are the *combinators*. These two are Schönfinkel's.

## Example 1.4: the identity

---

Define  $I := S K K$  and run it on any  $x$ :

$$\begin{array}{ll} S K K x & \rightsquigarrow K x (K x) & \text{S step: } M = N = K, P = x \\ & \rightsquigarrow x & \text{K step: keep } x, \text{ drop } K x \end{array}$$

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

Whatever  $x$  is,  $I x$  runs to  $x$ .

## Example 1.4: composition, from the constants alone

---

Claim: with  $B := S(KS)K$ , we get  $B f g x \rightsquigarrow^* f(g x)$ .

$$\begin{array}{lll} S(KS)K f g x & \rightsquigarrow & KS f (K f) g x & \text{S step} \\ & \rightsquigarrow & S(K f) g x & \text{K step} \\ & \rightsquigarrow & K f x (g x) & \text{S step} \\ & \rightsquigarrow & f(g x) & \text{K step} \end{array}$$

*Four steps. The comp you wrote in Haskell, without a single parameter.*

## Running a definition

---

Let  $\text{inc } y := y + 1$ . The integers and  $+$  are ready-made:  $m + n \rightsquigarrow$  the sum.

|                       |                    |                             |                                  |
|-----------------------|--------------------|-----------------------------|----------------------------------|
| $\text{twice inc } 0$ | $\rightsquigarrow$ | $\text{inc}(\text{inc } 0)$ | definition of <i>twice</i>       |
|                       | $\rightsquigarrow$ | $\text{inc}(0 + 1)$         | definition of <i>inc</i> , inner |
|                       | $\rightsquigarrow$ | $\text{inc } 1$             | $0 + 1$                          |
|                       | $\rightsquigarrow$ | $1 + 1$                     | definition of <i>inc</i>         |
|                       | $\rightsquigarrow$ | $2$                         | $1 + 1$                          |

*That is all a function call does: replace the left-hand side by the right.*

## Example 1.5: a boolean is a choice

---

$true := K$

$false := KI$

if  $b$  then  $M$  else  $N$  is just  $b M N$ .

$true M N \rightsquigarrow M$

$false M N \rightsquigarrow I N \rightsquigarrow^* N$

$false M N = KI M N$ :  $K$  keeps  $I$ , drops  $M$ .

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

## Example 1.5: using a boolean

---

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

|                         |              |                |                         |
|-------------------------|--------------|----------------|-------------------------|
| $\text{false } 10 \ 20$ | $=$          | $KI \ 10 \ 20$ | unfold <i>false</i>     |
|                         | $\leadsto$   | $I \ 20$       | K step: keep I, drop 10 |
|                         | $\leadsto^*$ | $20$           | Example 1.4             |

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

## Example 1.6: a number is “do it $n$ times”

---

$$\bar{0} f x := x$$

$$\bar{1} f x := f x$$

$$\bar{2} f x := f (f x)$$

$$\bar{3} f x := f (f (f x))$$

- 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.

$$\text{succ } n f x := f (n f x)$$

$$\text{add } m n f x := m f (n f x)$$

## Example 1.6: watch it add

---

Run  $add\bar{2}\bar{1}$  on  $inc$  and 0.

|                         |                                                                 |                   |
|-------------------------|-----------------------------------------------------------------|-------------------|
| $add\bar{2}\bar{1}inc0$ | $\rightsquigarrow \bar{2}inc(\bar{1}inc0)$                      | $add$             |
|                         | $\rightsquigarrow inc(inc(\bar{1}inc0))$                        | $\bar{2}$         |
|                         | $\rightsquigarrow inc(inc(inc0))$                               | $\bar{1}$         |
|                         | $\rightsquigarrow inc(inc(0+1))$                                | $inc$ , innermost |
|                         | $\rightsquigarrow inc(inc1) \rightsquigarrow inc(1+1)$          | $0+1$ ; $inc$     |
|                         | $\rightsquigarrow inc2 \rightsquigarrow 2+1 \rightsquigarrow 3$ | $\dots$           |

*Nine steps. The answer is 3: exactly what  $\bar{3}$  computes.*

## Your turn: multiplication

---

Let  $mult\ m\ n\ f := m\ (n\ f)$ .

Run  $mult\ \bar{2}\ \bar{3}\ inc0$ , 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.

## Your turn: the answer

---

$$\begin{array}{ll}
 mult \bar{2} \bar{3} inc 0 & \rightsquigarrow \bar{2} (\bar{3} inc) 0 & mult \\
 & \rightsquigarrow \bar{3} inc (\bar{3} inc 0) & \bar{2} \\
 & \rightsquigarrow \bar{3} inc (inc (inc (inc 0))) & \bar{3} \\
 & \rightsquigarrow inc (inc (inc (inc (inc (inc 0)))) & \bar{3}
 \end{array}$$

- ▶ 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)*.

## Remark 1.7: the order of steps does not matter

---

In the addition run, at one line there were *two* steps available: *inc* could have been unfolded at the outside instead of the inside. I chose the inside.

**Church–Rosser** (1936). If a term runs to a normal form, it runs to only one.

It lets us speak of **the** result of a program.

On loan: we take it on trust.

## Theorem 1.8: two constants suffice

---

### THEOREM 1.8 · TWO CONSTANTS SUFFICE · ON LOAN

Every definition by equations can be replaced by a program in  $K$  and  $S$  alone that runs the same way on every argument; and the functions on numerals that such programs compute are exactly the functions a Turing machine computes.

First half: Schönfinkel, 1924. Second half: Church 1936 and Turing 1937, the year he reached Princeton to work under Church.

*On loan. The compiling is Chapter 2's surprise.*

## Not a toy: every language you have used

---

HASKELL

```
twice f x =  
  f (f x)
```

PYTHON

```
def twice(f, x):  
    return f(f(x))
```

JAVASCRIPT

```
const twice =  
  f => x => f(f(x))
```

*One equation, three spellings:  $\text{twice } f\ x := f(f\ x)$ .*

A name, some parameters, a body. Java methods and C functions too.

## Not a toy: any algorithm, such as 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.*

## Not a toy: factorial, as a combinator

$$F = S(K(SA)) \\ (S(K(SC))(S(S(KS)(S(KK)I))(KE)))$$

where  $A = S(S(K isZero)I)(K1)$ ,  $C = S(K mult)I$ ,  $E = S(K pred)I$

- ▶ No variable. No parameter. No **if**. No recursive call.
- ▶ Only **K**, **S**, three arithmetic constants, and application.
- ▶  $fix\ F\ \bar{4}$  runs to 24 (*fix*: one extra constant, for the loop).

### A PROMISE

How to get from the whiteboard to this: Section 3.8. I shall show you then.

## Not a toy: so why study it?

---



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

## Take stock

---

- ▶ A program is a finite list of equations and a term to evaluate.
- ▶ To run it is to replace left-hand sides by right-hand sides until none is left.
- ▶ No store. No stack. No instruction pointer. No order of evaluation to settle.
- ▶ The conditional, the numbers, addition: built in front of you, out of nothing.

This is not a small language chosen for convenience. It is **the general case**: your own languages, with everything a checker does not read deleted.

*Something is about to go badly wrong in it. It will not be a quirk of a toy. It will be a thing that goes wrong in **yours**.*

SECTION 1.2

# Non-termination

---

*Four symbols, and no way out.*

## Apply a function to itself

---

Definition 1.1 never says what may be applied to what. A function may be applied to **itself**.

$$\omega x := x x$$
$$\Omega := \omega \omega$$

A legal definition: its right-hand side uses only its one parameter.

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

$$\omega \omega \rightsquigarrow \omega \omega \rightsquigarrow \omega \omega \rightsquigarrow \dots$$

## What it costs

---

- ▶ 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.

## In the constants: $\omega = SII$

---

Check that  $SIIx \rightsquigarrow^* xx$ :

$$\begin{array}{lll} SIIx & \rightsquigarrow & Ix(Ix) & \text{S step} \\ & \rightsquigarrow & x(Ix) & Ix \rightsquigarrow x \\ & \rightsquigarrow & xx & Ix \rightsquigarrow x \end{array}$$

- So  $\Omega$  is  $SII(SII)$ : three steps to come back to itself, instead of one.
- Compiling changes nothing. The loop is in the **shape**, not the notation.

## Nobody did anything wrong

---

- ▶  $\Omega$  is four symbols: two sensible decisions in a row.
- ▶ The grammar did not object:  $\omega\omega$  is a perfectly good term.
- ▶ The rule did not object: it had a step, and it took it.

YOU WILL NOT SEE IT COMING

Three library calls and a type nobody wrote down, and it announces nothing.

## Does the order of steps matter after all?

---

Exercise 1.4. Consider  $Kx\Omega$ . There are two steps available.

OUTER STEP

$$Kx\Omega \rightsquigarrow x$$

A normal form. The run stops.

INNER STEP

$$Kx\Omega \rightsquigarrow Kx\Omega \rightsquigarrow \dots$$

Never stops.

*No contradiction with Church–Rosser: it says at most one normal form, not that every order of steps finds it.*

## So ask for less

---

### WHAT CANNOT BE HAD

A checker that catches *every* program that fails to stop.

No such checker exists. We cannot yet prove this.

### WHAT WE SHALL BUILD

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

SECTION 1.3

# Types and typings

---

*Write the bookkeeping down.*

## What went wrong, named

---

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

The idea: label every term with what it takes and returns; allow an application only when the labels fit; check before anything runs.

READ  $A \rightarrow B$  AS A CONTRACT

Give me an  $A$ , and I shall return a  $B$ .

It can be checked without being honoured.

## Definition 1.9: 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,  $A \rightarrow B$  is a type: the function type, of functions taking an  $A$  and returning a  $B$ .
3. Nothing else is a type.

$$A, B ::= p \mid A \rightarrow B$$

### EXAMPLES

$Int \rightarrow Bool$ : a test on integers

$(Int \rightarrow Int) \rightarrow Int$

$(p \rightarrow q) \rightarrow p$  (look at its shape)

## The arrow groups to the right

---

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

A function of two arguments, taking them one at a time. As application grouped left.

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

A function that takes a function.

*Every real checker has more: records, lists, generics. And every one of them has this.*

## Definition 1.10: context and judgement

---

A context is a finite list

$$\Gamma = x_1 : A_1, \dots, x_n : A_n$$

pairing each variable in scope with a type, none listed twice. Write  $\Gamma, x : A$  to add a pair at the end.

The checker's basic claim is a judgement  $\Gamma \vdash M : A$ :

“ $M$  has type  $A$  when its variables have the types  $\Gamma$  gives them.”

### EMPTY CONTEXT

Write  $\vdash M : A$ :  $M$  is a program with type  $A$  outright.

## Definition 1.11: the four cases

---

(K)

 $K : A \rightarrow B \rightarrow A$ 

(S)

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

(VAR)

 $x : A$  whenever  $x : A$  is in  $\Gamma$ 

(APP)

 $M : A \rightarrow B$  and  $N : A$  give  $M N : B$ 

$A, B, C$  are arbitrary types: each of (K), (S) is a schema, one type for each choice of letters.

## Definition 1.11: a typing

---

A typing of  $\Gamma \vdash M : A$  is a finite sequence of judgements

$$\Gamma \vdash M_1 : A_1, \dots, \Gamma \vdash M_n : A_n$$

with the same  $\Gamma$ , ending in  $\Gamma \vdash M : A$ , each line justified by (K), (S), (var), or (app) from *earlier* lines. Nothing else is allowed.

### HOW TO WRITE JUSTIFICATIONS

“(app)  $i, j$ ”: line  $i$  is the function, line  $j$  the argument.

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

### READY-MADE PIECES

$n : \text{Int} \quad + : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$   
`"hello" : String`

## Worked: (var) and (app)

---

In the context  $\Gamma = f : Int \rightarrow Bool, x : Int$ , type  $fx$ .

| LINE | JUDGEMENT                                | JUSTIFICATION |
|------|------------------------------------------|---------------|
| 1.   | $\Gamma \vdash f : Int \rightarrow Bool$ | (var)         |
| 2.   | $\Gamma \vdash x : Int$                  | (var)         |
| 3.   | $\Gamma \vdash fx : Bool$                | (app) 1, 2    |

*The function wants an Int and gets an Int: the contract is met.*

There is no case for a *name* such as *twice*: the four cases type constants, variables and applications only.

## Worked: using the schema (K)

---

Type  $K\ 3\ \text{"hello"}$ . Instantiate (K) with  $A := Int$  and  $B := String$ .

| LINE | JUDGEMENT                                           | JUSTIFICATION            |
|------|-----------------------------------------------------|--------------------------|
| 1.   | $\vdash K : Int \rightarrow String \rightarrow Int$ | (K) [ $Int$ , $String$ ] |
| 2.   | $\vdash 3 : Int$                                    | ready-made               |
| 3.   | $\vdash K\ 3 : String \rightarrow Int$              | (app) 1, 2               |
| 4.   | $\vdash \text{"hello"} : String$                    | ready-made               |
| 5.   | $\vdash K\ 3\ \text{"hello"} : Int$                 | (app) 3, 4               |

*It runs to 3, and it is typed  $Int$ : the type says what comes out before anything runs.*

## 1.3 · TYPES AND TYPINGS

## Example 1.12: the identity, typed

---

Type  $I = SKK$  at  $A \rightarrow A$ . Write  $P$  for  $A \rightarrow A$ .

| LINE | JUDGEMENT                                                                                | JUSTIFICATION   |
|------|------------------------------------------------------------------------------------------|-----------------|
| 1.   | $\vdash S : (A \rightarrow P \rightarrow A) \rightarrow (A \rightarrow P) \rightarrow P$ | (S) $[A, P, A]$ |
| 2.   | $\vdash K : A \rightarrow P \rightarrow A$                                               | (K) $[A, P]$    |
| 3.   | $\vdash SK : (A \rightarrow P) \rightarrow P$                                            | (app) 1, 2      |
| 4.   | $\vdash K : A \rightarrow P$                                                             | (K) $[A, A]$    |
| 5.   | $\vdash SKK : A \rightarrow A$                                                           | (app) 3, 4      |

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

## Example 1.13: the red line, at last

---

### LINE JUDGEMENT

1.  $+:Int \rightarrow Int \rightarrow Int$
2.  $3 : Int$
3.  $+ 3 : Int \rightarrow Int$  (app) 1, 2
4. `"hello" : String`
5. (app) 3, 4 ... no match

The function wants *Int* and is offered *String*.

No rule applies. No typing exists.

*Expected Int, got String.*

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

## Your turn: type K!

---

Give a typing of K!

Hint: you already have a typing of I at  $B \rightarrow B$ .

What must K's first letter be, so that its argument is I?

### EXERCISE 1.5

Then explain why this is the type of *false*.

## Your turn: the answer

---

LINE JUDGEMENT

JUSTIFICATION

1.  $\vdash I : B \rightarrow B$ Example 1.12 with  $B$  for  $A$ 2.  $\vdash K : (B \rightarrow B) \rightarrow A \rightarrow (B \rightarrow B)$ (K)  $[B \rightarrow B, A]$ 3.  $\vdash KI : A \rightarrow B \rightarrow B$ 

(app) 2, 1

*KI takes an  $A$ , drops it, and returns the identity. That is false: it keeps the second argument.*

## Lemma 1.14: the shape of $\mathbf{S K K}$

LEMMA 1.14

If  $\Gamma \vdash \mathbf{S K K} : T$  then  $T$  is  $U \rightarrow U$  for some type  $U$ .

Proof: read the typing backwards.

The last line came by (app), so four earlier lines exist:

$\mathbf{S} : A'' \rightarrow A' \rightarrow T$ ,  $\mathbf{K} : A''$ ,  $\mathbf{K} : A'$ , and  $\mathbf{S K} : A' \rightarrow T$ .

1. (S):  $A'' = X \rightarrow Y \rightarrow Z$ ,  $A' = X \rightarrow Y$ ,  $T = X \rightarrow Z$ .
2. (K) has shape  $V \rightarrow W \rightarrow V$ ;  $A''$  has it, so  $Z = X$ .
3.  $A' = X \rightarrow Y$  has it too, so  $Y = W \rightarrow X$ .

Hence  $T = X \rightarrow Z = X \rightarrow X$ .

The backwards reading is the shape of every proof in Chapter 2.

## Proposition 1.15: $\Omega$ has no type

### PROPOSITION 1.15 · $\Omega$ HAS NO TYPE

There is no context  $\Gamma$  and no type  $T$  with  $\Gamma \vdash SII : T$ . Consequently  $\Omega$  has no typing.

- ▶  $\Omega$  contains  $SII$ . Read its typing backwards.
- ▶ (S) wants the first  $l : X \rightarrow Y \rightarrow Z$  and the second  $l : X \rightarrow Y$ . Lemma 1.14: each has shape  $U \rightarrow U$ . So  $X = Y \rightarrow Z$  and  $X = Y$ .

$$Y = Y \rightarrow Z$$

$Y \rightarrow Z$  is strictly longer than  $Y$ , and equal types have equal length. No type is a proper part of itself.

The parameter was the problem, and compiling it away did not remove it.

## Theorem 1.16: subject reduction

---



### THEOREM 1.16 · SUBJECT REDUCTION

If  $\Gamma \vdash M : A$  and  $M \leadsto N$  by a **K** step, an **S** step or a ready-made step, then  $\Gamma \vdash N : A$ .

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

## Theorem 1.16: the K case

---

Suppose  $K P Q : A$  and  $K P Q \rightsquigarrow P$ . Why is  $P : A$ ?

Read the typing backwards twice.

1.  $K P Q : A$  came by (app) from  $K P : B \rightarrow A$  and  $Q : B$ .
2.  $K P : B \rightarrow A$  came by (app) from  $K : C \rightarrow B \rightarrow A$  and  $P : C$ .
3. (K) gives K only types  $V \rightarrow W \rightarrow V$ , so  $C = A$ .

Hence  $P : A$ . ✓

The S case is the same walk, three times backwards, then forwards.

## Theorem 1.17: normalisation

---

THEOREM 1.17 · NORMALISATION · ON LOAN

Every program with a typing runs to a normal form.

$\Omega$  is refused. **No** typed program loops.

Tait, 1967. We take it on trust, and lean on it in Chapter 3.

*Real languages give the loop back, as recursion: **fix**, **while**, a method that calls itself. What that costs is Exercise 1.6.*

CLOSING SECTION

# What Church built it for

---

*It was never meant to be a programming language.*

WHAT CHURCH BUILT IT FOR

## 1932: a logic, not a language

---

- ▶ There were no programs to write.
- ▶ Church offered the calculus as a logic: statements, and functions from statements to statements.
- ▶ The plan: all of mathematics, done in it.

Within three years it was **broken**. What broke it was the self-application you just met.

WHAT CHURCH BUILT IT FOR

## The statement that feeds on itself

---

Let  $B$  be any statement. Let  $X$  say: “*if  $X$  holds, then  $B$  holds.*”

1. Suppose  $X$ .
2.  $X$  says  $X \rightarrow B$ , so on that supposition,  $X \rightarrow B$ .
3. From  $X$  and  $X \rightarrow B$ :  $B$ .
4. Discharge the supposition:  $X \rightarrow B$ , with nothing supposed.
5. But that is what  $X$  says. So  $X$ , and so  $B$ .

Nothing was assumed about  $B$ .

The logic proves **everything**, and so proves nothing.

Curry's paradox, 1942.

WHAT CHURCH BUILT IT FOR

## Where it first broke: Richard's puzzle

---

- ▶ A finite English phrase can name a real number. The phrases can be listed: a first, a second, a third...
- ▶ Name this number: *the number whose  $n$ th digit differs from the  $n$ th digit of the  $n$ th number on the list.*
- ▶ That phrase is finite English, so it is on the list, at place  $k$ . But it differs from the  $k$ th number at the  $k$ th digit.

KLEENE AND ROSSER, 1935

Built this inside the calculus, since it can name all of its own names. Curry then saw the one ingredient that did the damage.

The diagonal returns in Chapter 11, put to work.

WHAT CHURCH BUILT IT FOR

## Two disasters, one shape

---

A PROGRAM

$\omega\omega$  never stops

A PROOF

$X = (X \rightarrow B)$  proves everything

*Built the same way. A program and a logical disaster turn out to be **one object**.*

## The repair is the checker

---

- ▶ Church's repair, 1940: attach a type to every term, and forbid an application unless the types fit.
- ▶ That is rule (app), and Proposition 1.15 is the repair *working*.
- ▶ The self-feeding statement can no longer be formed.

Your compiler is built around the **repaired** calculus. That is why its whole job is to refuse.

WHAT CHURCH BUILT IT FOR

## A word you have not defined

---

IN CURRY'S PARAGRAPH

*statement, suppose, so, discharge, conclude, proves*

Every one is a word about **proofs**.

IN THE TYPES OF K AND S

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

Letters and arrows. Formulas.

*We have a checker, and no idea yet what makes its four rules rules of **reasoning**. That is Chapter 2.*

## What this chapter said

---

- ▶ Under every language you use is a combinatory algebra: two constants and application, run by two rewrite rules. Unpoliced, it computes everything, including a program that never stops.
- ▶ A type is a contract, a basic type or  $A \rightarrow B$ . The checker allows an application only when the contract is met.
- ▶ Its certificate is a typing: a finite list of judgements  $\Gamma \vdash M : A$ , each justified by (K), (S), (var), or (app).
- ▶ Its *no* is (app) failing to match. Its *yes* names no input, so covers them all, and by subject reduction stays true as the program runs.

*Next: what a **proof** is, and why the compiler's rules are its rules.*

## Sources

---

- ▶ Wadler, *Propositions as types*. CACM 2015. The story of Chapters 1 to 3, in a dozen pages.
- ▶ Schönfinkel, *Über die Bausteine der mathematischen Logik*. 1924. Where K and S come from.
- ▶ Church (1936); Turing (1937). The calculus computes everything.
- ▶ Church, *A formulation of the simple theory of types*. 1940. Where it gets its types.
- ▶ Kleene and Rosser (1935); Curry (1942). Why it needed them.
- ▶ Tait (1967). Normalisation.

Thank you.