

PART 1: TYPE THEORY · CHAPTER 3

Simply Typed Lambda Calculus

Alexander V. Gheorghiu

COMP0009 Logic · University College London

CHAPTER 3

Simply Typed Lambda Calculus

A program and a proof, side by side.

OPENING

A function you have written

HASKELL

```
compose = \f -> \g -> \x -> f (g x)  
-- (b -> c) -> (a -> b) -> a -> c
```

The compiler accepted it in a few microseconds.

What did it check?

By the end of this chapter the answer will be: a proof. And you will be able to say of what.

OPENING

Proving an implication

To prove “if A then B ”:

1. write *suppose* A ;
2. reason until you reach B ;
3. write *therefore* A *implies* B .

At that point the supposition is **spent**: it may not be used again, and the conclusion does not depend on it.

MOVE ONE

Suppose A , reach B , conclude $A \rightarrow B$, and drop the supposition.

OPENING

Using an implication, and using a supposition

MOVE TWO

We have $A \rightarrow B$, and we have A , so B .

MOVE THREE

A supposition in force may be written down wherever it is wanted.

Three moves. Every proof you have read in a mathematics lecture is built from them and their cousins for the other connectives.

OPENING

The three moves as rules

Write each move as a rule: premises above the bar, conclusion below. Γ is the list of suppositions in force.

$$\frac{}{\Gamma, A \vdash A}$$

LEFT RULE

A supposition in force may be written down.

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

MIDDLE RULE

From $A \rightarrow B$ and A , conclude B .

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

RIGHT RULE

If B follows on the supposition A , conclude $A \rightarrow B$ and drop the supposition.

OPENING

The same three moves

Take the rule for parameters that Chapter 2 licensed. Write it beside (var) and (app), and rub out the programs.

What is left is exactly the three moves.

The checker that reads your definition `f x = ...` is checking a proof written the way mathematicians write proofs.

This chapter is about the logic that makes that exact.

OPENING

Where we are going

SECTIONS

THE QUESTION

- | | |
|---------|--|
| 3.1–3.2 | What is a function with no name, and how does the checker read it? |
| 3.3 | What about pairs, tagged unions, the empty type? |
| 3.4–3.5 | Rub the programs out. Whose rules are left? |
| 3.6 | Normalisation: in the notes, not examined. |
| 3.7–3.8 | What does it cost? Curry's paradox, recursion, and factorial all the way down. |

SECTION 3.1

The λ -calculus

A function without a name.

3.1 · THE λ -CALCULUS

Why drop the name?

PYTHON: WRITE IT IN PLACE

```
map(lambda x: x + 1, xs)
```

PYTHON: NAME IT FIRST

```
def inc(x): return x + 1  
map(inc, xs)
```

Every time you hand a function to another function, you either name it first or write it in place.

Church's notation is “write it in place”, for **every** function: $\lambda x. M$.

3.1 · THE λ -CALCULUS

You have written the third form many times

HASKELL

 $\backslash x \rightarrow M$

PYTHON

`lambda x: M`

JAVASCRIPT

`x => M`

Every `def f(x): return M` is one with a **name** attached.

Every method in Java is one with a name and a class attached.

3.1 · THE λ -CALCULUS

Definition 3.1: λ -terms

Fix a stock of variables $x, y, z, f, \dots$. The terms are defined inductively:

1. every variable x is a term;
2. if M and N are terms, then $M N$ is a term:
the application of M to N ;
3. if x is a variable and M a term, then $\lambda x. M$ is
a term: the abstraction of M over x , the
function that takes x and returns M ;
4. nothing else is a term.

IN ONE LINE

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

3.1 · THE λ -CALCULUS

Reading terms: where do the brackets go?

- ▶ Application groups to the left, as before.
- ▶ A λ reaches as far to the right as it can: $\lambda x. M N$ is $\lambda x. (M N)$, not $(\lambda x. M) N$.
- ▶ The ready-made integers and $+$ stay as constants.

AS WRITTEN

WITH THE BRACKETS PUT IN

 $f x y$ $\implies$ $(f x) y$ $\lambda x. x y$ $\implies$ $\lambda x. (x y)$ $\lambda x. \lambda y. y x$ $\implies$ $\lambda x. (\lambda y. (y x))$ $(\lambda x. x) y$ $\implies$ *already bracketed: the identity applied to y* $\lambda f. \lambda x. f (f x)$ $\implies$ $\lambda f. (\lambda x. (f (f x)))$

A λ swallows everything to its right, up to the closing bracket.

3.1 · THE λ -CALCULUS

Your turn: bracket these

Put in every bracket the conventions leave out.

1. $\lambda x. f\ x\ x$
2. $(\lambda x. x\ x)\ \lambda x. x\ x$
3. $\lambda x. \lambda y. \lambda z. x\ z\ (y\ z)$

ONE MINUTE

The third one is S. Remember where its body ends.

3.1 · THE λ -CALCULUS

Your turn: the answer

ITEM AS WRITTEN

FULLY BRACKETED

- | | |
|---|--|
| 1. $\lambda x. f x x$ | $\lambda x. ((f x) x)$ |
| 2. $(\lambda x. x x) \lambda x. x x$ | $(\lambda x. (x x)) (\lambda x. (x x))$ |
| 3. $\lambda x. \lambda y. \lambda z. x z (y z)$ | $\lambda x. (\lambda y. (\lambda z. ((x z) (y z))))$ |

Item 2 is Ω from Chapter 1, now written without a name.

3.1 · THE λ -CALCULUS

Definitions are λ -terms

A definition $f\ x_1 \cdots x_n := M$ of Chapter 1 is the term $\lambda x_1. \cdots \lambda x_n. M$.

CHAPTER 1 DEFINITION	AS A λ -TERM
$twice\ f\ x := f\ (f\ x)$	$\lambda f. \lambda x. f\ (f\ x)$
$comp\ f\ g\ x := f\ (g\ x)$	$\lambda f. \lambda g. \lambda x. f\ (g\ x)$
$flip\ f\ x\ y := f\ y\ x$	$\lambda f. \lambda x. \lambda y. f\ y\ x$
$inc\ y := y + 1$	$\lambda y. y + 1$

3.1 · THE λ -CALCULUS

The two constants are terms

$$K := \lambda x. \lambda y. x$$

$$S := \lambda x. \lambda y. \lambda z. x z (y z)$$

That is why they need no longer be taken as given.

Chapter 1's whole language was this one, with the parameters compiled away. Theorem 3.3 will say that nothing is lost either way.

3.1 · THE λ -CALCULUS

Where the notation comes from (1)

- ▶ Alonzo Church introduced it at Princeton, in the paper we met in Chapter 1 (1932).
- ▶ The λ was there to write the functions of a logic meant to carry the **whole of mathematics**.
- ▶ The logic did not survive. The notation did.

1936: Church uses it to exhibit a question no procedure can settle. Turing proves, the same year, that it computes exactly what his machines compute.

1940: Church repairs it with **types**.

3.1 · THE λ -CALCULUS

Where the notation comes from (2)

SCHÖNFINKEL, 1924

Two combinators, published with the same object in view. Curry spent the 1930s on them.

CHURCH, 1932–40

The λ -calculus, then its typed repair, which your compiler inherited.

One language, two origins. That the calculus is also the heart of every programming language was noticed twenty years later.

3.1 · THE λ -CALCULUS

Bound and free: you already know the difference

```
def f(x): return x + y
```

x is the parameter: **bound**. y comes from outside: **free**.

Rename x to z : same function.

Rename y to w : a different function.

A bound name is a local label. A free name is a reference to the world.

3.1 · THE λ -CALCULUS

Free and bound variables

- ▶ The x in $\lambda x. M$ is the function's parameter: the λ binds it, and its scope is the body M .
- ▶ An occurrence of a variable is bound if it lies in the scope of a λ on that variable, and free otherwise.

EXAMPLE

$$(\lambda x. x y) x$$

The first two occurrences of x are bound. The last x , like y , is free.

3.1 · THE λ -CALCULUS

Your turn: free or bound?

For each term, mark every occurrence of each variable as free or bound.

1. $\lambda x. \lambda y. x z (y w)$
2. $(\lambda x. x) x y$
3. $\lambda x. (\lambda y. y) x y$

TWO MINUTES

Same name does not mean same variable.

3.1 · THE λ -CALCULUS

Your turn: the answer

ITEM	TERM	BOUND AND FREE
1.	$\lambda x. \lambda y. x z (y w)$	x, y bound; z, w free
2.	$(\lambda x. x) x y$	the first two x bound; the third x and y free
3.	$\lambda x. (\lambda y. y) x y$	both y in $(\lambda y. y)$ bound; x bound; the last y is <i>free</i>

In item 3 the λy closes at the bracket. The final y is outside its scope.

3.1 · THE λ -CALCULUS

Free variables, by recursion

The set $\text{FV}(M)$ of free variables is defined by recursion on the shape of M , one clause for each form.

1. M is a variable x : $\text{FV}(x) = \{x\}$.
2. M is an application $M_1 M_2$: $\text{FV}(M_1 M_2) = \text{FV}(M_1) \cup \text{FV}(M_2)$.
3. M is an abstraction $\lambda x. M_1$: $\text{FV}(\lambda x. M_1) = \text{FV}(M_1) \setminus \{x\}$.

The first two only collect. The third does something: only a λ removes a variable from the list, and only its own parameter.

3.1 · THE λ -CALCULUS

Worked: computing FV

Compute $FV((\lambda x. x y) x)$, innermost first.

EXPRESSION	VALUE	CLAUSE
$FV(x)$	$= \{x\}$	clause 1
$FV(y)$	$= \{y\}$	clause 1
$FV(x y)$	$= \{x, y\}$	clause 2
$FV(\lambda x. x y)$	$= \{x, y\} \setminus \{x\} = \{y\}$	clause 3
$FV((\lambda x. x y) x)$	$= \{y\} \cup \{x\} = \{x, y\}$	clause 2

3.1 · THE λ -CALCULUS

Closed terms, and renaming

CLOSED

A term with no free variables: a whole program rather than a fragment of one.

K , S and $\lambda f. \lambda g. \lambda x. f(gx)$ are closed.

RENAMING

$\lambda x. x$ and $\lambda y. y$ describe the same function, so we treat them as the same term.

A bound variable may be renamed whenever that is convenient.

Substitution

$M(N/x)$ replaces the **free** occurrences of x in M by N , and nothing else.

In Chapter 1 this was simply replacement, because nothing in that language bound a variable. Now it has to step around binders.

Defined by recursion on the shape of M , one clause per form, except that the variable and abstraction cases each split: is the variable met the very one being replaced?

3.1 · THE λ -CALCULUS

Substitution: the five clauses

1. M is the variable x itself: $x(N/x) = N$.
2. M is a variable y other than x : $y(N/x) = y$.
3. M is an application: $(M_1 M_2)(N/x) = M_1(N/x) M_2(N/x)$.
4. M is an abstraction on x itself: $(\lambda x. M_1)(N/x) = \lambda x. M_1$.
5. M is an abstraction on another variable y : $(\lambda y. M_1)(N/x) = \lambda y. M_1(N/x)$, provided $y \notin \text{FV}(N)$.

Clauses 1–3: replace when you meet it, otherwise pass the substitution down. Clause 4 stops: every x inside is bound. Clause 5 has a side condition; the next slides say why.

3.1 · THE λ -CALCULUS

Worked: substituting

Substitute f for x , and y for x , in three terms.

SUBSTITUTION	RESULT	CLAUSES USED
$(x\ y)$	$(f/x)\ y = f\ y$	clauses 3, 1, 2
$(\lambda x. x\ y)$	$(f/x)\ (\lambda x. x\ y) = \lambda x. x\ y$	clause 4: x is bound inside, nothing to replace
$(\lambda z. x\ z)$	$(y/x)\ (\lambda z. x\ z) = \lambda z. y\ z$	clause 5: $z \notin \text{FV}(y)$, so go under
$((\lambda x. x)\ x)$	$(f/x)\ ((\lambda x. x)\ x) = (\lambda x. x)\ f$	clause 3, then clause 4 on the left, clause 1 on the right

3.1 · THE λ -CALCULUS

A bug with a long history

```
#define SWAP(a, b) { int t = a; a = b; b = t; }  
  
SWAP(x, y) /* fine */  
  
SWAP(t, y) /* { int t = t; t = y; y = t; } */
```

The macro's own **t** **captured** the **t** you passed in.

A free variable of the argument became bound. The same happens when you substitute under a λ .

3.1 · THE λ -CALCULUS

The one subtlety: capture

Substitute y for x in $\lambda y. x y$. Clause 5 declines: $y \in \text{FV}(y)$.

BLINDLY

$\lambda y. y y$

The free y we brought in has been captured by the λy .

RENAME FIRST

$\lambda y. x y = \lambda z. x z$, so the result is $\lambda z. y z$.

The function that applies y .

3.1 · THE λ -CALCULUS

Rename first

Pushing N under a λy when y is free in N would turn that free y into a bound one and change what N means.

The remedy: rename the y of $\lambda y. M_1$ to a letter that occurs nowhere else, then substitute. Chapter 4 shows what goes wrong when a substitution skips this step, and the same bug on the logic side, where it has been known for a century.

3.1 · THE λ -CALCULUS

Running: one rule

To apply a function to an argument, substitute the argument for the parameter:

$$(\lambda x. M) N \rightsquigarrow M(N/x)$$

anywhere inside a term. The pair on the left is a redex; the step is called β .

That is what a function call does,
and it is all a function call does.

3.1 · THE λ -CALCULUS

Worked: running

Run $(\lambda x. x x) (\lambda y. y)$, and $(\lambda x. x + 1) 2$.

STEP	TERM	REASON
	$(\lambda x. x x) (\lambda y. y)$	
$\leadsto$	$(\lambda y. y) (\lambda y. y)$	β : $\lambda y. y$ for x
$\leadsto$	$\lambda y. y$	β : $\lambda y. y$ for y
	$(\lambda x. x + 1) 2$	
$\leadsto$	$2 + 1$	β : 2 for x
$\leadsto$	3	ready-made +

3.1 · THE λ -CALCULUSChapter 1's steps are all β

STEP	TERM	REASON
K	$M N = (\lambda x. \lambda y. x) M N$	
	$\rightsquigarrow (\lambda y. M) N$	β on x
	$\rightsquigarrow M$	β on y (y does not occur in M after renaming)
S	$M N P = (\lambda x. \lambda y. \lambda z. x z (y z)) M N P$	
	$\rightsquigarrow (\lambda y. \lambda z. M z (y z)) N P$	β on x
	$\rightsquigarrow (\lambda z. M z (N z)) P$	β on y
	$\rightsquigarrow M P (N P)$	β on z

*The **K** step is two β steps, the **S** step three, and a definition step is one for each parameter.*

3.1 · THE λ -CALCULUS**Example 3.2:** the same addition, parameters visible

Let

$$add := \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$$

$$\overline{2} := \lambda f. \lambda x. f (f x)$$

$$\overline{1} := \lambda f. \lambda x. f x$$

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

We will run $add \overline{2} \overline{1} inc 0$. The answer should be 3.

3.1 · THE λ -CALCULUSExample 3.2: four steps to unwrap *add*

STEP	TERM	REASON
	$add\bar{2}\bar{1}\text{inc}0$	
$\rightsquigarrow$	$(\lambda n. \lambda f. \lambda x. \bar{2} f (n f x)) \bar{1} \text{inc} 0$	$\bar{2}$ for m
$\rightsquigarrow$	$(\lambda f. \lambda x. \bar{2} f (\bar{1} f x)) \text{inc} 0$	$\bar{1}$ for n
$\rightsquigarrow$	$(\lambda x. \bar{2} \text{inc} (\bar{1} \text{inc} x)) 0$	inc for f
$\rightsquigarrow$	$\bar{2} \text{inc} (\bar{1} \text{inc} 0)$	0 for x

Chapter 1 did this in one step. Nothing has changed but the grain.

3.1 · THE λ -CALCULUS

Example 3.2: and the rest of the run

STEP	TERM	REASON
	$\bar{2} \text{ inc } (\bar{1} \text{ inc } 0)$	
$\rightsquigarrow$	$(\lambda x. \text{ inc } (\text{ inc } x)) (\bar{1} \text{ inc } 0)$	 inc for f in $\bar{2}$
$\rightsquigarrow$	$\text{ inc } (\text{ inc } (\bar{1} \text{ inc } 0))$	argument for x
$\rightsquigarrow$	$\text{ inc } (\text{ inc } ((\lambda x. \text{ inc } x) 0))$	 inc for f in $\bar{1}$
$\rightsquigarrow$	$\text{ inc } (\text{ inc } (\text{ inc } 0))$	0 for x
$\rightsquigarrow^*$	3	three uses of inc , each β then +

A λ is a parameter waiting for its argument, and β is the argument arriving.

3.1 · THE λ -CALCULUS

Your turn: run it

Run to a number:

$$(\lambda f. \lambda x. f (f x)) (\lambda y. y + 2) 1$$

Name each step.

TWO MINUTES

What does the first argument do twice?

3.1 · THE λ -CALCULUS

Your turn: the answer

STEP	TERM	REASON
	$(\lambda f. \lambda x. f(fx)) (\lambda y. y + 2) 1$	
$\rightsquigarrow$	$(\lambda x. (\lambda y. y + 2) ((\lambda y. y + 2) x)) 1$	β on f
$\rightsquigarrow$	$(\lambda y. y + 2) ((\lambda y. y + 2) 1)$	β on x
$\rightsquigarrow$	$(\lambda y. y + 2) (1 + 2)$	β on the inner y
$\rightsquigarrow$	$(\lambda y. y + 2) 3$	ready-made $+$
$\rightsquigarrow$	$3 + 2$	β on y
$\rightsquigarrow$	5	ready-made $+$

$\overline{2}$ applied to “add 2” and 1: add two, twice.

3.1 · THE λ -CALCULUS

Exercise 3.2: capture, and Ω again

Run $(\lambda x. x x) (\lambda x. x x)$ for three steps.

Then run $(\lambda x. \lambda y. x) y$, first blindly and then renaming the bound y first, and say which answer is the one meant.

TWO MINUTES

Which answer is the one you would get from $K y$?

3.1 · THE λ -CALCULUS

Exercise 3.2: the answer

STEP	TERM	REASON
	$(\lambda x. x x) (\lambda x. x x)$	
$\leadsto$	$(\lambda x. x x) (\lambda x. x x)$	each step gives the term back
	$(\lambda x. \lambda y. x) y$	
$\leadsto$	$\lambda y. y$	blindly: the free y is <i>captured</i> ; the identity
$=$	$(\lambda x. \lambda z. x) y \leadsto \lambda z. y$	renamed first: the constant function returning y

The second is meant: λy should be a function that ignores its argument and returns y .

SECTION 3.2

Typing

One new rule: (abs).

3.2 · TYPING

The vocabulary, recalled

TYPES (DEFINITION 1.9)

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

The arrow groups to the right.

CONTEXTS, JUDGEMENTS (1.10)

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

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

3.2 · TYPING

Two rules we already have

(VAR)

$$\frac{}{\Gamma \vdash x : A} \quad \text{if } x : A \text{ is in } \Gamma$$

Look the variable up.

(APP)

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

The argument has the type the function expects.

3.2 · TYPING

The third rule

The Deduction Theorem licensed it: if B follows from the hypothesis A , then $A \rightarrow B$ follows without it.

We may now adopt it outright, since adopting it lets nothing new be typed.

(ABS)

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$

3.2 · TYPING

The typing rules, together

$$\text{(var)} \quad \frac{}{\Gamma \vdash x : A} \text{ if } x : A \text{ is in } \Gamma$$

$$\text{(app)} \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

$$\text{(abs)} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$

3.2 · TYPING

Reading (abs) as the checker reads a definition

To check $\lambda x. M$:

1. add the parameter to the context with the type it is to have;
2. check the body;
3. the function has the arrow type.

If Γ already declares an x , rename the parameter first: a context lists each variable once.

(abs) is the first rule that **changes the context**.

Above the line there is one more variable in scope than below it.

3.2 · TYPING

The constants are gone

K and S are now terms, so they get their types by (abs).

A typing is a finite list of judgements as before, justified as before.

The one difference from Definition 1.11: the lines of a single typing no longer all carry the same context.

TO CHECK

That (abs) gives exactly (K) and (S) of Chapter 1: Exercise 3.1.
First, K together.

3.2 · TYPING

Worked: typing K

Type $\lambda x. \lambda y. x$. Read bottom-up what is needed: the body x , in a context with both parameters.

LINE	JUDGEMENT	JUSTIFICATION
1.	$x : A, y : B \vdash x : A$	(var)
2.	$x : A \vdash \lambda y. x : B \rightarrow A$	(abs) 1
3.	$\vdash \lambda x. \lambda y. x : A \rightarrow B \rightarrow A$	(abs) 2

Exactly (K). Watch the context shrink: two variables, one, none.

3.2 · TYPING

Exercise 3.1: typing S

Give a typing of $\lambda x. \lambda y. \lambda z. x z (y z)$ by the three rules, and confirm the type is (S).

Then run $(\lambda x. \lambda y. \lambda z. x z (y z)) M N P$ and count the β steps.

THREE MINUTES

Context $x : A \rightarrow B \rightarrow C, y : A \rightarrow B, z : A.$

3.2 · TYPING

Exercise 3.1: the answer

$\Gamma = x : A \rightarrow B \rightarrow C, y : A \rightarrow B, z : A.$

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash x : A \rightarrow B \rightarrow C$	(var)
2.	$\Gamma \vdash z : A$	(var)
3.	$\Gamma \vdash xz : B \rightarrow C$	(app) 1, 2
4.	$\Gamma \vdash y : A \rightarrow B$	(var)
5.	$\Gamma \vdash yz : B$	(app) 4, 2
6.	$\Gamma \vdash xz(yz) : C$	(app) 3, 5

Then three (abs) discharge z, y, x to give $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$. Running it: three β steps, one per parameter, give $MP(NP)$.

3.2 · TYPING

Theorem 3.3: λ -terms and combinator terms are one language

THEOREM 3.3

Every typed λ -term translates to a combinator term of the same type that runs the same way on every argument, and every typed combinator term is a typed λ -term of the same type.

Nothing is lost by changing notation.

3.2 · TYPING

Why Theorem 3.3 is true

COMBINATORS ARE λ -TERMS

Write $\lambda x. \lambda y. x$ for **K** and $\lambda x. \lambda y. \lambda z. x z (y z)$ for **S**. Their types come out as (K) and (S) by (abs): we checked both. (var) and (app) are shared.

 λ -TERMS ARE COMBINATORS

Take the innermost abstraction $\lambda x. M$. Run the Deduction Theorem's construction on the typing of M in the context $x:A$, with the programs left on. It writes **K**, **S** **K** **K**, **S**. No λ left in that piece: repeat.

in the notes, not examined here: *the full argument*.

3.2 · TYPING

Left to right, seen once

A λ -TERM $\lambda x. f x$

THE COMBINATOR TERM

 $S (K f) (S K K)$

The 11-line output of Chapter 2's Deduction Theorem run.

► f does not mention x : a K wraps it. x is x : the identity $S K K$. The application: an S .

Section 3.8 writes the recipe as three equations and runs it on a whole program.

What is gained

The checker of Chapter 1 and the checker of these three rules accept the same programs at the same types.

What is gained: the compiler no longer has to compile anything to decide whether to say yes. It reads the definition as written.

3.2 · TYPING

A worked typing: composition

The program from our first slide:

```
compose = \f -> \g -> \x -> f (g x)
```

In the context

$$\Gamma = f : B \rightarrow C, g : A \rightarrow B, x : A$$
PLAN

Lines 1–5 type the body with three look-ups and two (app). Lines 6–8: three (abs).

3.2 · TYPING

Composition: lines 1–5, the body

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash g : A \rightarrow B$	(var)
2.	$\Gamma \vdash x : A$	(var)
3.	$\Gamma \vdash gx : B$	(app) 1, 2
4.	$\Gamma \vdash f : B \rightarrow C$	(var)
5.	$\Gamma \vdash f(gx) : C$	(app) 4, 3

Three look-ups and two uses of (app), each matching an argument's type against the type a function expects.

3.2 · TYPING

Composition: lines 6–8, three (abs)

LINE JUDGEMENT

JUSTIFICATION

- | | |
|--|---------|
| 6. $f : B \rightarrow C, g : A \rightarrow B \vdash \lambda x. f(gx) : A \rightarrow C$ | (abs) 5 |
| 7. $f : B \rightarrow C \vdash \lambda g. \lambda x. f(gx) : (A \rightarrow B) \rightarrow A \rightarrow C$ | (abs) 6 |
| 8. $\vdash \lambda f. \lambda g. \lambda x. f(gx) : (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$ | (abs) 7 |

Each (abs) takes the last variable out of the context and puts a λ on the front. Watch the context shrink: at line 8 it is empty and the program is closed.

This is what the compiler did, in a few microseconds, when it accepted your composition function.

3.2 · TYPING

Rub out the programs

- ▶ Lines 1–5: a proof of C from the three hypotheses $B \rightarrow C$, $A \rightarrow B$ and A . Three hypothesis lines and two modus ponens.
- ▶ Lines 6–8: discharge A , then $A \rightarrow B$, then $B \rightarrow C$.
- ▶ The last line, $(B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$, is hypothetical syllogism as a theorem, with nothing assumed.

Chapter 2: seven lines, the first an instance of (II) with twenty letters.

Here: found and checked in three discharges.

The compiler read your program as that proof.

So the first slide's question has an answer: the compiler checked a proof of hypothetical syllogism.

3.2 · TYPING

Your turn: type *twice*

Give a typing of

$$\lambda f. \lambda x. f (f x)$$

and say what type it has.

THREE MINUTES

f is used twice. What does that force its type to be?

3.2 · TYPING

Your turn: the answer

 $\Gamma = f : A \rightarrow A, x : A.$

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash f : A \rightarrow A$	(var)
2.	$\Gamma \vdash x : A$	(var)
3.	$\Gamma \vdash f x : A$	(app) 1, 2
4.	$\Gamma \vdash f (f x) : A$	(app) 1, 3
5.	$f : A \rightarrow A \vdash \lambda x. f (f x) : A \rightarrow A$	(abs) 4
6.	$\vdash \lambda f. \lambda x. f (f x) : (A \rightarrow A) \rightarrow A \rightarrow A$	(abs) 5

f takes an A and returns an A , so its output can feed back in. The type is $(A \rightarrow A) \rightarrow A \rightarrow A$.

3.2 · TYPING

Where $\lambda x. x x$ gets stuck

Try to type $\lambda x. x x$.

To use (abs), we need $x : ? \vdash x x : B$.

(app) wants the first x to have an arrow type $A \rightarrow B$, and the second to have type A .

One variable would need type $A \rightarrow B$ and type A at once.

That is $A = A \rightarrow B$, which no type satisfies.

This is Proposition 1.15 again: Ω has no type.

SECTION 3.3

Conjunction, disjunction, and negation

Each is a type you have used.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

More than an arrow

The arrow is one connective. A logic needs *and*, *or* and *not* as well.

Each is a type you have used:

► a **pair**, a **tagged union**, the **empty type**, and each comes with its rules, the way $\rightarrow$ did.

TWO WORDS ORGANISE EVERYTHING

Introduction: the conclusion has the connective on the outside. On the program side it *builds* a value: a constructor.

Elimination: a premise has it on the outside. It *takes a value apart*: a pattern match.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

The arrow already fits

CONNECTIVE

INTRODUCTION (BUILDS)

ELIMINATION (TAKES APART)

 $A \rightarrow B$ $(\text{abs}): \lambda x. M$ $(\text{app}): M N$

The three rules of Section 3.2 were already of this shape: (abs) introduces, (app) eliminates, and (var) is the supposition both lean on.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Definition 3.4: conjunction

If A and B are formulas, so is $A \wedge B$. As a type it is the pair type, (A, B) in Haskell. The rules, with their terms:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \wedge B} \text{ } (\wedge I)$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \text{fst } M : A} \text{ } (\wedge E_1)$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \text{snd } M : B} \text{ } (\wedge E_2)$$

Conjunction, in words

TO ESTABLISH $A \wedge B$

Establish both. A proof of a conjunction is a **pair** of proofs.

TO USE IT

Take whichever half you need. The two eliminations are the two **projections**.

In Haskell: `(a, b)` builds the pair; `fst p` and `snd p` take it apart.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Worked: $A \wedge B \rightarrow B \wedge A$

The program $\lambda p. (snd\ p, fst\ p)$, in the context $\Gamma = p : A \wedge B$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash p : A \wedge B$	(var)
2.	$\Gamma \vdash snd\ p : B$	$(\wedge E_2)$ 1
3.	$\Gamma \vdash fst\ p : A$	$(\wedge E_1)$ 1
4.	$\Gamma \vdash (snd\ p, fst\ p) : B \wedge A$	$(\wedge I)$ 2, 3
5.	$\vdash \lambda p. (snd\ p, fst\ p) : A \wedge B \rightarrow B \wedge A$	(abs) 4

A function that swaps a pair.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Definition 3.5: disjunction

If A and B are formulas, so is $A \vee B$. As a type it is **Either** A B . The rules:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{Left } M : A \vee B} \text{ (}\vee\text{I}_1\text{)} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{Right } M : A \vee B} \text{ (}\vee\text{I}_2\text{)}$$

$$\frac{\Gamma \vdash M : A \vee B \quad \Gamma, x : A \vdash P : C \quad \Gamma, y : B \vdash Q : C}{\Gamma \vdash \text{case } M \text{ of Left } x \Rightarrow P \mid \text{Right } y \Rightarrow Q : C} \text{ (}\vee\text{E)}$$

Disjunction, in words

TO ESTABLISH $A \vee B$

Establish one of them and say which. A proof of a disjunction is a proof of one side with a **tag**.

TO USE IT

Argue by **cases**: if you can reach C supposing A , and reach C supposing B , you have C . Both suppositions are discharged, like the one in (abs).

The elimination is a **case** expression with one branch per tag.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Worked: $A \vee B \rightarrow B \vee A$

The program $\lambda x. \text{case } x \text{ of } Left\ a \Rightarrow Right\ a \mid Right\ b \Rightarrow Left\ b$. Write $\Gamma = x : A \vee B$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash x : A \vee B$	(var)
2.	$\Gamma, a : A \vdash a : A$	(var)
3.	$\Gamma, a : A \vdash Right\ a : B \vee A$	($\vee I_2$) 2
4.	$\Gamma, b : B \vdash b : B$	(var)
5.	$\Gamma, b : B \vdash Left\ b : B \vee A$	($\vee I_1$) 4
6.	$\Gamma \vdash \text{case } x \text{ of } Left\ a \Rightarrow Right\ a \mid Right\ b \Rightarrow Left\ b : B \vee A$	($\vee E$) 1, 3, 5
7.	$\vdash \lambda x. \text{case } \dots : A \vee B \rightarrow B \vee A$	(abs) 6

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Read it: one context per branch

- ▶ Lines 2–3 live in $\Gamma, a : A$. Lines 4–5 live in $\Gamma, b : B$.
- ▶ Line 6 returns to Γ : both branch variables are out of scope.
- ▶ a is in scope only in its own branch, and so is b . That is what “one supposition in each branch” means.

($\forall E$) is the one rule so far with two rules inside it changing the context.

Definition 3.6: falsity, and negation

$\perp$ is a formula, read *false*. As a type it is the type with no constructor, `data Void` in Haskell, of which there are no values.

It has no introduction rule and one elimination, for every formula C :

$$\frac{\Gamma \vdash M : \perp}{\Gamma \vdash \text{absurd } M : C} (\perp\text{E})$$

NEGATION

An abbreviation: $\neg A$ is $A \rightarrow \perp$.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Falsity, in words

- ▶ There is no way to establish $\perp$ outright, and that is the point of it.
- ▶ If you ever have $\perp$ in hand you may conclude anything: “from a contradiction, anything follows”. That is `absurd :: Void -> a`.
- ▶ $\neg A$ is the claim that from A you could reach $\perp$: a function from A to the empty type.
- ▶ Everything that looks like a negation is an implication into $\perp$, and the rules for $\neg$ are the rules for $\rightarrow$.

Whether $\perp$ can be established from some hypotheses is a different question. It is the question of Section 3.6.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

The whole language in one table

CONNECTIVE	TYPE	INTRODUCTION (CONSTRUCTOR)	ELIMINATION (PATTERN MATCH)
$A \rightarrow B$	$A \rightarrow B$	$\lambda x.M$	application $M N$
$A \wedge B$	(A, B)	(M, N)	fst, snd
$A \vee B$	Either $A B$	$Left M, Right M$	case with two branches
$\perp$	Void	—	$absurd$
$\neg A$	$A \rightarrow \text{Void}$	$\lambda x.M$	application

What to carry away

The connectives of logic are the data types of a programming language. The introduction rules are their constructors, and the elimination rules are the pattern matches that take them apart.

A proof is a program that builds and takes apart values of these types, and nothing else.

Example 3.7 (1): swap a pair

$$A \wedge B \rightarrow B \wedge A$$

The program is $\lambda p. (snd\ p, fst\ p)$. We just typed it.

A function that swaps a pair.

(Two more follow.)

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Example 3.7 (2): contraposition

$(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$. The program is $\lambda f. \lambda k. \lambda a. k(f a)$. Context $\Gamma = f : A \rightarrow B, k : B \rightarrow \perp, a : A$.

LINE JUDGEMENT

JUSTIFICATION

- | | |
|--|-------------------------|
| 1. $\Gamma \vdash f a : B$ | (app), with (var) twice |
| 2. $\Gamma \vdash k(f a) : \perp$ | (app), with (var) |
| 3. $f : A \rightarrow B, k : B \rightarrow \perp \vdash \lambda a. k(f a) : A \rightarrow \perp$ | (abs) 2 |
| 4. $f : A \rightarrow B \vdash \lambda k. \lambda a. k(f a) : (B \rightarrow \perp) \rightarrow A \rightarrow \perp$ | (abs) 3 |
| 5. $\vdash \lambda f. \lambda k. \lambda a. k(f a) : (A \rightarrow B) \rightarrow \neg B \rightarrow \neg A$ | (abs) 4 |

A function that composes a test with a function. Chapter 4 uses this one as a line in a proof.

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Example 3.7 (3): $A \rightarrow \neg\neg A$

The program is $\lambda a. \lambda k. k a$. Context $\Gamma = a : A, k : A \rightarrow \perp$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash k a : \perp$	(app), with (var) twice
2.	$a : A \vdash \lambda k. k a : (A \rightarrow \perp) \rightarrow \perp$	(abs) 1
3.	$\vdash \lambda a. \lambda k. k a : A \rightarrow (A \rightarrow \perp) \rightarrow \perp$	(abs) 2

A function that hands its first argument to its second.

In each case the program is one you could have written without knowing what it proved. The types were there all along.

Exercise 3.3: two programs

Find a closed program for each, and say which Haskell function it is.

1. $(A \rightarrow B \rightarrow C) \rightarrow B \rightarrow A \rightarrow C$
2. $(A \wedge B \rightarrow C) \rightarrow A \rightarrow B \rightarrow C$

FIVE MINUTES

Name the variables in the context, then three (abs).

Exercise 3.3: the answer

- From $f : A \rightarrow B \rightarrow C$, $b : B$, $a : A$: $f a : B \rightarrow C$, then $f a b : C$, then three discharges. The program is $\lambda f. \lambda b. \lambda a. f a b$: **flip**.
- From $f : A \wedge B \rightarrow C$, $a : A$, $b : B$: $(a, b) : A \wedge B$ by $(\wedge I)$, then $f (a, b) : C$, then three discharges. The program is $\lambda f. \lambda a. \lambda b. f (a, b)$: Haskell's **curry**.

Exercise 3.4: not-not-excluded-middle

Find a closed program of type

$$\neg\neg(A \vee \neg A)$$

and give its typing.

FIVE MINUTES

Start from the context $k : (A \vee \neg A) \rightarrow \perp$. How could you ever use k ?

3.3 · CONJUNCTION, DISJUNCTION, AND NEGATION

Exercise 3.4: the answer

The program is $\lambda k. k(\text{Right}(\lambda a. k(\text{Left } a)))$. Context $k : (A \vee \neg A) \rightarrow \perp$, $a : A$:

LINE	JUDGEMENT	JUSTIFICATION
1.	$\text{Left } a : A \vee \neg A$	($\vee I_1$)
2.	$k(\text{Left } a) : \perp$	(app)
3.	$\lambda a. k(\text{Left } a) : \neg A$	(abs) 2, discharging a
4.	$\text{Right}(\lambda a. k(\text{Left } a)) : A \vee \neg A$	($\vee I_2$) 3
5.	$k(\text{Right}(...)) : \perp$	(app) k , 4
6.	$\lambda k. k(\text{Right}(...)) : \neg\neg(A \vee \neg A)$	(abs) 5, discharging k

k is used twice. First on a *Left* built from an A we do not have yet: that is what the λa promises. Then on the *Right* that promise builds.

Exercise 3.5: a De Morgan law

Find a closed program of type

$$A \vee B \rightarrow \neg(\neg A \wedge \neg B)$$

FIVE MINUTES

Take apart the $\vee$. What does each branch have to produce?

Exercise 3.5: the answer

- ▶ $\lambda s. \lambda p. \text{ case } s \text{ of } Left\ a \Rightarrow fst\ p\ a \mid Right\ b \Rightarrow snd\ p\ b.$
- ▶ Here $p : \neg A \wedge \neg B$. In the first branch $fst\ p : A \rightarrow \perp$ is applied to $a : A$. In the second, $snd\ p : B \rightarrow \perp$ to $b : B$.
- ▶ Both branches have type $\perp$, as $(\vee E)$ requires.

SECTION 3.4

Natural deduction

Now rub the programs out.

3.4 · NATURAL DEDUCTION

Delete every term

Take the rules of Sections 3.2 and 3.3. Delete every term. Keep what is left: judgements $\Gamma \vdash A$, relating a set of formulas to a formula.

Ten rules survive.

BEFORE

$\Gamma \vdash M : A$

AFTER

$\Gamma \vdash A$

Definition 3.8: natural deduction (1)

A derivation of A from a set of formulas Γ is a finite list of judgements $\Delta \vdash C$, each justified by one of the rules below from judgements earlier in the list, ending in $\Gamma \vdash A$.

$$\overline{\Gamma \vdash A} \text{ (hyp) if } A \in \Gamma \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \text{ } (\rightarrow E) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \text{ } (\rightarrow I)$$

(hyp) was (var); $(\rightarrow E)$ was (app); $(\rightarrow I)$ was (abs).

Definition 3.8: natural deduction (2)

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} (\wedge E_1) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} (\wedge E_2) \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} (\wedge I)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} (\vee I_1) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} (\vee I_2)$$

Definition 3.8: natural deduction (3)

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \text{ (}\vee\text{E)}$$

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash C} \text{ (}\perp\text{E)}$$

I write $\Gamma \vdash_{\text{N}} A$ when a derivation exists, to keep it apart from the $\vdash$ of Chapter 2 until Theorem 3.10 shows the two agree.

3.4 · NATURAL DEDUCTION

Nothing has been added

RULE	WHAT IT WAS, WITH THE PROGRAM
$(\rightarrow I)$	a λ
$(\rightarrow E)$	an application
$(\wedge I)$	a pair
$(\vee E)$	a case
$(\perp E)$	the function absurd

What has changed is only what we are looking at.

3.4 · NATURAL DEDUCTION

Gentzen, 1934

- ▶ This is not a system invented for these notes, nor one that came out of computing at all.
- ▶ It is Gerhard Gentzen's, from his doctoral thesis: two years before Turing's machine, before there was a program in the world to type.
- ▶ He called it *natural* because he built it by *watching*.

He took proofs as mathematicians actually write them and wrote down the moves they actually make.

3.4 · NATURAL DEDUCTION

The design, in four sentences

- ▶ To prove an implication, suppose the antecedent. To use one, apply it.
- ▶ To prove a conjunction, prove both halves.
- ▶ To use a disjunction, argue by cases.
- ▶ Each connective gets rules that say how to prove a statement with that connective on the outside, and how to use one — and nothing else.

That is the whole design, and the ten rules are the record of it.

3.4 · NATURAL DEDUCTION

Example 3.9: the schema (I), derived

Natural deduction has no axioms. If it is to prove what the Hilbert system proves, it must derive (I) and (II).

LINE	JUDGEMENT	JUSTIFICATION
1.	$A, B \vdash A$	(hyp)
2.	$A \vdash B \rightarrow A$	($\rightarrow$ I) 1
3.	$\vdash A \rightarrow B \rightarrow A$	($\rightarrow$ I) 2

Three lines. The hypotheses A and B are discharged one at a time, last first.

3.4 · NATURAL DEDUCTION

Example 3.9: the schema (II), derived

With $\Gamma = \{A \rightarrow B \rightarrow C, A \rightarrow B, A\}$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$\Gamma \vdash A \rightarrow B \rightarrow C$	(hyp)
2.	$\Gamma \vdash A$	(hyp)
3.	$\Gamma \vdash B \rightarrow C$	($\rightarrow$ E) 1, 2
4.	$\Gamma \vdash A \rightarrow B$	(hyp)
5.	$\Gamma \vdash B$	($\rightarrow$ E) 4, 2
6.	$\Gamma \vdash C$	($\rightarrow$ E) 3, 5
7.	$A \rightarrow B \rightarrow C, A \rightarrow B \vdash A \rightarrow C$	($\rightarrow$ I) 6
8.	$A \rightarrow B \rightarrow C \vdash (A \rightarrow B) \rightarrow A \rightarrow C$	($\rightarrow$ I) 7
9.	$\vdash (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$	($\rightarrow$ I) 8

3.4 · NATURAL DEDUCTION

Put the programs back

DERIVATION OF (I)

Decorated, it is

$$\lambda x. \lambda y. x = K$$

DERIVATION OF (II)

Name the hypotheses x, y, z . Line 6 is $x z (y z)$; line 9 is

$$\lambda x. \lambda y. \lambda z. x z (y z) = S$$

The two constants of Chapter 1 are the two schemas, derived.

3.4 · NATURAL DEDUCTION

Worked: $p \rightarrow p$ costs two lines

Chapter 2 needed five lines, and a paragraph to find them.

LINE	JUDGEMENT	JUSTIFICATION
1.	$p \vdash p$	(hyp)
2.	$\vdash p \rightarrow p$	$(\rightarrow I)$ 1

Decorated: $\lambda x. x$. The Deduction Theorem is the rule $(\rightarrow I)$, proved admissible for the Hilbert system.

3.4 · NATURAL DEDUCTION

Worked: hypothetical syllogism, six lines

$\Gamma = \{p \rightarrow q, q \rightarrow r\}$ and $\Delta = \Gamma \cup \{p\}$. Chapter 2 needed seven lines of Hilbert proof.

LINE	JUDGEMENT	RULE	PROGRAM
1.	$\Delta \vdash p \rightarrow q$	(hyp)	f
2.	$\Delta \vdash p$	(hyp)	x
3.	$\Delta \vdash q$	$(\rightarrow E) 1, 2$	$f x$
4.	$\Delta \vdash q \rightarrow r$	(hyp)	g
5.	$\Delta \vdash r$	$(\rightarrow E) 4, 3$	$g(f x)$
6.	$\Gamma \vdash p \rightarrow r$	$(\rightarrow I) 5$	$\lambda x. g(f x)$

The right-hand column is the program: composition, again.

Theorem 3.10: two systems, one logic

THEOREM 3.10

Let Γ be a set of formulas and A a formula, all built from letters by $\rightarrow$ alone. Then $\Gamma \vdash_N A \iff \Gamma \vdash A$.

Natural deduction and the Hilbert system prove the same things. Two notations, one logic.

3.4 · NATURAL DEDUCTION

Why: the Deduction Theorem is $(\rightarrow I)$

HILBERT TO NATURAL DEDUCTION

An instance of (I) or (II): the derivation of Example 3.9. A hypothesis: (hyp). A modus ponens: $(\rightarrow E)$, the same rule under another name.

NATURAL DEDUCTION TO HILBERT

(hyp): a one-line proof. $(\rightarrow E)$: a modus ponens. $(\rightarrow I)$: the **Deduction Theorem**, which rewrites a proof from Γ, A into a proof of $A \rightarrow B$ from Γ .

The Deduction Theorem is the rule $(\rightarrow I)$, proved admissible for the Hilbert system. From here on I write $\vdash$ for both.

in the notes, not examined here: *the line-by-line proof.*

Theorem 3.11: the Curry–Howard correspondence

THEOREM 3.11 (CURRY–HOWARD)

Derivations of A from Γ in natural deduction and typings of λ -terms M with $\Gamma \vdash M : A$ are the **same lists**.

Erase the terms from a typing: (var), (app), (abs) become (hyp), ($\rightarrow$ E), ($\rightarrow$ I).

Decorate a derivation: a distinct variable for each hypothesis, an application at each ($\rightarrow$ E), a λ at each ($\rightarrow$ I). It becomes a typing.

Proofs are programs. Propositions are types.

3.4 · NATURAL DEDUCTION

The dictionary

LOGIC

formula

proof

 $A \rightarrow B$ $A \wedge B$ $A \vee B$ $\perp$

hypothesis

discharging a hypothesis ($\rightarrow$ I)modus ponens ($\rightarrow$ E)

PROGRAMMING

type

program

function from A to B

pair

Either A B **Void**

variable

binding a parameter, λ

calling a function

Curry–Howard, rule by rule

TYPING		DERIVATION
(var)	$\longleftrightarrow$	(hyp)
(app)	$\longleftrightarrow$	($\rightarrow$ E)
(abs)	$\longleftrightarrow$	($\rightarrow$ I)
pairs, projections	$\longleftrightarrow$	($\wedge$ I), ($\wedge$ E)
<i>Left, Right, case</i>	$\longleftrightarrow$	($\vee$ I), ($\vee$ E)
<i>absurd</i>	$\longleftrightarrow$	($\perp$ E)

Erasure and decoration are the relabelling of Theorem 2.4, with one more case.

3.4 · NATURAL DEDUCTION

Who saw it

CURRY, 1950S

The types of his combinators were the axioms of a Hilbert system.

HOWARD, 1969

The same held of the λ -calculus and natural deduction, rule for rule. And running a program corresponded to something Gentzen had already studied.

*The **Curry–Howard correspondence**. The next two sections are about the last part of it: what Gentzen had studied, and why he cared.*

SECTION 3.5

Derivations as trees

The shape of the argument.

3.5 · DERIVATIONS AS TREES

Lists are for checkers, trees are for readers

THE LIST

Right for a checker: each line can be checked on its own, because everything it depends on is written on it.

Wrong for a reader: the hypotheses are copied onto every line, and the shape of the argument has to be reconstructed from the justifications.

THE TREE

Gentzen's way, and every book since. Hypotheses sit at the leaves, written once each. The conclusion is at the root. The shape of the tree is the shape of the argument.

3.5 · DERIVATIONS AS TREES

How to read a tree

- ▶ Hypotheses at the **leaves**, at the top, written once each.
- ▶ Each rule is a horizontal bar: premises above, conclusion below, the rule used on the right.
- ▶ The conclusion of the whole derivation is the single formula at the **root**, at the bottom.
- ▶ You read a tree *downwards*.

3.5 · DERIVATIONS AS TREES

Discharge: what a tree must record

The rule ($\rightarrow$ I) proves $A \rightarrow B$ by using A as a hypothesis and then giving it up. A leaf is in one of two states:

- ▶ open: the conclusion rests on it;
- ▶ discharged by some rule below it: the conclusion does not.

A discharged leaf is written in **square brackets**: $[A]$.

Which hypotheses are open is not decoration: it is the whole content of the turnstile.

Definition 3.12: derivations (overview)

- ▶ The derivations are defined inductively.
- ▶ With each derivation $\mathcal{D}$ we define at the same time two things:
 - its conclusion, the formula at the root;
 - its set of open hypotheses $\text{OH}(\mathcal{D})$, a finite set of formulas.
- ▶ A formula in square brackets above a derivation is one the clause discharges.
- ▶ We write $\Gamma \vdash A$ when some derivation has conclusion A and $\text{OH} \subseteq \Gamma$.

3.5 · DERIVATIONS AS TREES

Definition 3.12, clauses 1–3

1. (*hypothesis*) For every formula A , the one-node tree labelled A is a derivation, with conclusion A and $\text{OH} = \{A\}$.
2. ($\rightarrow$ I) If $\mathcal{D}$ has conclusion B , then for every A

$$\frac{\begin{array}{c} [A] \\ \mathcal{D} \\ B \end{array}}{A \rightarrow B} \rightarrow\text{I} \quad \text{OH} = \text{OH}(\mathcal{D}) \setminus \{A\}$$

3. ($\rightarrow$ E) If $\mathcal{D}_1$ concludes $A \rightarrow B$ and $\mathcal{D}_2$ concludes A

$$\frac{\begin{array}{c} \mathcal{D}_1 \\ A \rightarrow B \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ A \end{array}}{B} \rightarrow\text{E} \quad \text{OH} = \text{OH}(\mathcal{D}_1) \cup \text{OH}(\mathcal{D}_2)$$

3.5 · DERIVATIONS AS TREES

Definition 3.12, clauses 4–5: conjunction

4. ($\wedge$ I) If $\mathcal{D}_1$ concludes A and $\mathcal{D}_2$ concludes B

$$\frac{\frac{\mathcal{D}_1}{A} \quad \frac{\mathcal{D}_2}{B}}{A \wedge B} \wedge\text{I} \quad \text{OH} = \text{OH}(\mathcal{D}_1) \cup \text{OH}(\mathcal{D}_2)$$

5. ($\wedge$ E) If $\mathcal{D}$ concludes $A \wedge B$, then each of

$$\frac{\frac{\mathcal{D}}{A \wedge B}}{A} \wedge\text{E}_1 \quad \text{and} \quad \frac{\frac{\mathcal{D}}{A \wedge B}}{B} \wedge\text{E}_2$$

is a derivation, with $\text{OH} = \text{OH}(\mathcal{D})$.

3.5 · DERIVATIONS AS TREES

Definition 3.12, clauses 6–7: disjunction

6. ($\vee$ I) If $\mathcal{D}$ concludes A , then for every B the tree with root $A \vee B$ and immediate subtree $\mathcal{D}$ is a derivation; symmetrically from B . In both cases $\text{OH} = \text{OH}(\mathcal{D})$.
7. ($\vee$ E) If $\mathcal{D}_0$ concludes $A \vee B$ and $\mathcal{D}_1, \mathcal{D}_2$ both conclude C

$$\frac{\begin{array}{ccc} \mathcal{D}_0 & [A] & [B] \\ A \vee B & \mathcal{D}_1 & \mathcal{D}_2 \\ & C & C \end{array}}{C} \vee\text{E}$$

with $\text{OH} = \text{OH}(\mathcal{D}_0) \cup (\text{OH}(\mathcal{D}_1) \setminus \{A\}) \cup (\text{OH}(\mathcal{D}_2) \setminus \{B\})$.

3.5 · DERIVATIONS AS TREES

Definition 3.12, clauses 8–9: falsity, and nothing else

8. ($\perp$ E) If $\mathcal{D}$ concludes $\perp$, then for every C the tree with root C and immediate subtree $\mathcal{D}$ is a derivation, with $\text{OH} = \text{OH}(\mathcal{D})$.
9. Nothing else is a derivation.

Nine clauses and a “nothing else”: exactly as Definition 1.1 built terms. It comes with a principle of proof, rule induction, which we meet shortly.

3.5 · DERIVATIONS AS TREES

Three remarks on discharge

EVERY OPEN LEAF AT ONCE

Clause 2 removes A from a *set*: every open leaf labelled A is discharged together. (Books that discharge a chosen subset prove nothing more.)

VACUOUS DISCHARGE

A need not occur in $\mathcal{D}$ at all. That is schema (I), and on the program side K : a function that ignores its argument.

The numbers $[A]^1$ and $\rightarrow I, 1$ are bookkeeping: they say which bracket belongs to which bar. They carry no information the definition lacks.

3.5 · DERIVATIONS AS TREES

Tree 1: hypothetical syllogism

Three bars. The two hypotheses are written once each, and they stay open.

$$\begin{array}{c}
 \frac{[p]^1 \quad p \rightarrow q}{q} \rightarrow E \quad q \rightarrow r \\
 \hline
 \frac{q \quad q \rightarrow r}{r} \rightarrow E \\
 \hline
 \frac{r}{p \rightarrow r} \rightarrow I, 1
 \end{array}$$

Read from the leaves down: suppose p ; with $p \rightarrow q$, get q ; with $q \rightarrow r$, get r ; discharge the supposition and conclude $p \rightarrow r$.

3.5 · DERIVATIONS AS TREES

Tree 1: computing the open hypotheses

NODE	OPEN HYPOTHESES	CLAUSE
leaf $[p]$	$\text{OH} = \{p\}$	clause 1
leaf $p \rightarrow q$	$\text{OH} = \{p \rightarrow q\}$	clause 1
q	$\text{OH} = \{p, p \rightarrow q\}$	clause 3
leaf $q \rightarrow r$	$\text{OH} = \{q \rightarrow r\}$	clause 1
r	$\text{OH} = \{p, p \rightarrow q, q \rightarrow r\}$	clause 3
$p \rightarrow r$	$\text{OH} = \{p \rightarrow q, q \rightarrow r\}$	clause 2: remove p

So the tree witnesses $\{p \rightarrow q, q \rightarrow r\} \vdash p \rightarrow r$.

3.5 · DERIVATIONS AS TREES

Tree 2: one hypothesis, discharged twice

The conclusion is $\vdash (A \wedge B) \rightarrow (B \wedge A)$, with no open hypotheses at all.

$$\frac{\frac{\frac{[A \wedge B]^1}{B} \wedge E_2 \quad \frac{[A \wedge B]^1}{A} \wedge E_1}{B \wedge A} \wedge I}{(A \wedge B) \rightarrow (B \wedge A)} \rightarrow I, 1$$

Clause 2, working on a set: both leaves labelled $A \wedge B$ close at once.

3.5 · DERIVATIONS AS TREES

Tree 2, on the program side

The program is $\lambda x. (snd\ x, fst\ x)$.

The two bracketed leaves are the two occurrences of the variable x in its body.

DICTIONARY

A variable used twice: a hypothesis discharged at two leaves.

A variable used once: at one.

A parameter never used: a vacuous discharge.

3.5 · DERIVATIONS AS TREES

Tree 3: two different hypotheses, one in each branch

Clause 7 discharges two *different* hypotheses. The conclusion is $\vdash (A \vee B) \rightarrow (B \vee A)$.

$$\frac{
 \frac{
 \frac{[A \vee B]^1}{B \vee A} \vee I_2 \quad \frac{[B]^3}{B \vee A} \vee I_1
 }{B \vee A} \vee E, 2, 3
 }{(A \vee B) \rightarrow (B \vee A)} \rightarrow I, 1$$

3.5 · DERIVATIONS AS TREES

Tree 3: argument by cases, written out

- ▶ We have $A \vee B$.
- ▶ In the case A we reach $B \vee A$. In the case B we reach $B \vee A$.
- ▶ So we have it either way.

$$\lambda x. \text{ \textbf{case } } x \text{ \textbf{ of } } Left\ a \Rightarrow Right\ a \mid Right\ b \Rightarrow Left\ b$$

The discharged leaves $[A]^2$ and $[B]^3$ are the branch variables a and b , each in scope only in its own branch.

3.5 · DERIVATIONS AS TREES

Your turn: draw the tree for (I)

Draw a tree with conclusion

$$A \rightarrow B \rightarrow A$$

and no open hypotheses. Where is discharge
vacuous?

TWO MINUTES

You need only one leaf.

Your turn: the answer

$$\frac{\frac{[A]^1}{B \rightarrow A} \rightarrow \text{I}, 2}{A \rightarrow B \rightarrow A} \rightarrow \text{I}, 1$$

- One leaf, $[A]^1$. The first bar, labelled 2, discharges B : but no leaf is labelled B . That is the vacuous discharge.
- The last bar discharges A . $\text{OH} = \emptyset$.

On the program side: $\lambda x. \lambda y. x = \text{K}$.

Your turn: contraposition as a tree

Draw a tree for

$$(A \rightarrow B) \rightarrow (B \rightarrow \perp) \rightarrow (A \rightarrow \perp)$$

with no open hypotheses. Remember $\neg X$ is $X \rightarrow \perp$.

THREE MINUTES

You need three leaves, discharged by three bars. The program was $\lambda f. \lambda k. \lambda a. k (f a)$.

3.5 · DERIVATIONS AS TREES

Your turn: the answer

$$\begin{array}{c}
 \frac{\frac{[B \rightarrow \perp]^2 \quad \frac{[A \rightarrow B]^1 \quad [A]^3}{B} \rightarrow E}{\perp} \rightarrow E}{A \rightarrow \perp} \rightarrow I, 3 \\
 \frac{A \rightarrow \perp}{(B \rightarrow \perp) \rightarrow (A \rightarrow \perp)} \rightarrow I, 2 \\
 \frac{(B \rightarrow \perp) \rightarrow (A \rightarrow \perp)}{(A \rightarrow B) \rightarrow (B \rightarrow \perp) \rightarrow (A \rightarrow \perp)} \rightarrow I, 1
 \end{array}$$

Open hypotheses: none. Each leaf is one variable of the program $\lambda f. \lambda k. \lambda a. k(f a)$: $[A \rightarrow B]^1 = f$, $[B \rightarrow \perp]^2 = k$, $[A]^3 = a$.

3.5 · DERIVATIONS AS TREES

Theorem 3.13: lists and trees agree

THEOREM 3.13

For every set of formulas Γ and every formula A : some derivation in the sense of Definition 3.12 has conclusion A and open hypotheses inside Γ if and only if some list in the sense of Definition 3.8 derives $\Gamma \vdash A$.

So we may write lists, because the checker does, and draw trees when the shape is the point.

in the notes, not examined here: *the proof, by induction in each direction.*

3.5 · DERIVATIONS AS TREES

Rule induction

To show that every derivation has a property P , it is enough to show that

- ▶ each one-node derivation has P , and
- ▶ for each of clauses 2–8, if the subderivations the clause uses have P , so does the derivation the clause builds.

The same move as induction on the natural numbers, with the clauses in place of zero and successor.

When you meet “by induction on the derivation” in any book on proof theory, this is what is meant.

SECTION 3.6

Normalisation

What Gentzen had studied.

3.6 · NORMALISATION

In the notes, not examined

- ▶ A proof can take a **detour**: an introduction met by its own elimination. In a program that is a redex, and removing it is a step of computation.
- ▶ Running a program preserves its type (Theorem 3.15), and every typed program stops (Theorem 3.16, on loan).
- ▶ So the logic is **consistent**: no closed proof of $\perp$ or of a bare letter (Theorem 3.19), and a proof of “or” says which side (Corollary 3.20).
- ▶ Exercises 3.6 to 3.8 go with it.

WHERE TO FIND IT

Section 3.6 of the notes. Read it for the story of why Gentzen cared. We will not lecture it, and it will not be examined.

SECTION 3.7

Curry's paradox

A sentence that proves everything.

3.7 · CURRY'S PARADOX

Recall: the sentence 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. We met it at the end of Chapter 1.

3.7 · CURRY'S PARADOX

An intuitive version

Consider the sentence S : “*If this sentence is true, then Santa Claus exists.*”

1. Suppose S is true.
2. Then what it says holds: if S is true, Santa exists.
We supposed S true, so Santa exists.
3. So we have shown: *if S is true, then Santa exists*,
with nothing supposed.
4. That is exactly what S says. So S is true.
5. So Santa exists.

Replace “Santa exists” by anything
you like: $2 + 2 = 5$, or $\perp$.

Each step looks like ordinary reasoning.
The trouble is that S talks about itself.

3.7 · CURRY'S PARADOX

An older ancestor: 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.

THE COMMON SHAPE

A statement or a description that refers to the very system it lives in.

Kleene and Rosser built Richard's puzzle inside the calculus in 1935. Curry then found the one ingredient doing the damage.

3.7 · CURRY'S PARADOX

Two disasters, one shape

A PROGRAM

$\omega\omega$ never stops

A PROOF

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

Chapter 1 promised these were one object. With the programs on, here they are.

3.7 · CURRY'S PARADOX

The argument, written out exactly

The argument we just recalled, now with the programs on.

It needs one thing the rules do not allow. Seeing where the rules refuse is the point.

THE ASSUMPTION

Suppose there were a type X with

$$X = X \rightarrow B$$

for some formula B chosen in advance.

3.7 · CURRY'S PARADOX

Example 3.21: Curry's paradox, with its program

LINE	JUDGEMENT	JUSTIFICATION
1.	$x : X \vdash x : X$	(var)
2.	$x : X \vdash x : X \rightarrow B$	(var), reading X as $X \rightarrow B$
3.	$x : X \vdash x x : B$	(app) 2, 1
4.	$\vdash \lambda x. x x : X \rightarrow B$	(abs) 3
5.	$\vdash \lambda x. x x : X$	line 4, reading $X \rightarrow B$ as X
6.	$\vdash (\lambda x. x x) (\lambda x. x x) : B$	(app) 4, 5

3.7 · CURRY'S PARADOX

Rub out the programs

- ▶ Suppose X ; then $X \rightarrow B$, since that is what X says; so B .
- ▶ Discharge the supposition and conclude $X \rightarrow B$; which is X ; so B .

It is the argument of Chapter 1 line for line, with $(\rightarrow I)$ for “discharge”. Since B was arbitrary, it proves everything.

And the program on line 6 is Ω . The looping program of Chapter 1 and the proof of everything are one term.

3.7 · CURRY'S PARADOX

Now look at where it fails

- ▶ Lines 2 and 5 are justified by **no rule**. They rest on the equation $X = X \rightarrow B$.
- ▶ Proposition 1.15 said why no type satisfies it: a type is a finite tree, and $X \rightarrow B$ is strictly larger than X .
- ▶ Take the equation away: line 2 cannot be written, so line 3 cannot, so $\lambda x. x x$ has no type, and Ω has none.

The one fact that refused the looping program is the one fact that refuses the paradox.

Remark 3.22: what recursion costs (1)

Every language you use has recursion. Exercise 1.6 modelled it by a constant *fix* with

- ▶ type $(A \rightarrow A) \rightarrow A$ for every A ,
- ▶ step $\text{fix } f \rightsquigarrow f(\text{fix } f)$.

That constant is exactly what Example 3.21 needed: it lets a program **feed on itself** without an equation between types.

Remark 3.22: what recursion costs (2)

- ▶ $\text{fix}(\lambda x. x)$ has every type A there is: $\perp$ and every letter included. Consistency (Section 3.6) fails.
- ▶ It runs for ever. Normalisation fails with it.
- ▶ The two go together, as they came together.

Read as a logic, a language with general recursion proves everything. A type in such a language promises only what the program does if it returns.

3.7 · CURRY'S PARADOX

Safe is not correct

That is not a reason to give up recursion. It is the reason the checker's *yes*, in the languages you use, means **safe** and never **correct**.

Nothing we care about — a sorted list, a transitive order, a discount on every day of the sale — is proved from *no* hypotheses. By the consistency theorem of Section 3.6, nothing with a bare claim at the top ever is.

Such things are proved from a **specification**: a set of sentences about lists, orders or sales taken as hypotheses.

A specification says *for all*, which no formula of this chapter can say.

SECTION 3.8

From pseudocode to combinators

Watch something ordinary travel the whole distance.

Three languages, one language

This part has three languages in it:

- ▶ the two constants of Chapter 1;
- ▶ the λ -calculus of this chapter;
- ▶ the proofs that both turned out to be.

They are one language. The way to believe that is to watch something ordinary travel the whole distance.

Here is factorial, as you would write it on a whiteboard.

Factorial on the whiteboard

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

FOUR THINGS NOT IN CHAPTER 1'S
LANGUAGE

a parameter
a name that calls itself
a conditional
arithmetic

Take them away one at a time.

Step 1: the parameter becomes a λ

A definition with a parameter is an abstraction. Use a boolean as its own conditional: “if b then M else N ” is $b M N$.

$$fact := \lambda n. isZeron\ 1\ (mult\ n\ (fact\ (pred\ n)))$$

Nothing has really happened yet: $fact$ still occurs on the right-hand side, and a term may not refer to itself.

Reminder: booleans are their own conditional

EXAMPLE 1.5

$$\begin{aligned} \text{true} &= \mathbf{K} \\ \text{false} &= \mathbf{K}\mathbf{I} \end{aligned}$$

SO

$$\begin{aligned} \mathbf{K} M N &\rightsquigarrow M \\ \mathbf{K}\mathbf{I} M N &\rightsquigarrow N \end{aligned}$$

isZeron returns $\mathbf{K}$ if $n = 0$ and $\mathbf{K}\mathbf{I}$ otherwise, so *isZeron* $1\ E$ is 1 or E .

Step 2: the self-reference becomes a parameter

Abstract over the recursive call as well, and the self-reference disappears.

$$F := \lambda r. \lambda n. \text{isZero } n \ 1 \ (\text{mult } n \ (r \ (\text{pred } n)))$$

F mentions nothing but its own two parameters: give it the function to call and a number, and it does one layer of the work.

Step 3: the fixed point

We want a term *fact* with $fact = F fact$: a fixed point of F .

That is exactly what the constant *fix* delivers, since $fix F \rightsquigarrow F (fix F)$. So

$$fact := fix F$$

One step shows why: $fix F \bar{3} \rightsquigarrow F (fix F) \bar{3}$, whose r is $fix F$ again.

The recursive call is the fixed point unfolding once more, on demand and no sooner.

Worked: running $fix\ F$ on 2

STEP TERM

REASON

 $fix\ F\ 2$ $\rightsquigarrow F\ (fix\ F)\ 2$

fix

 $\rightsquigarrow^* isZero\ 2\ 1\ (mult\ 2\ (fix\ F\ (pred\ 2)))$ two β $\rightsquigarrow^* mult\ 2\ (fix\ F\ 1)$ $isZero\ 2$ is K!; $pred\ 2 = 1$ $\rightsquigarrow^* mult\ 2\ (mult\ 1\ (fix\ F\ 0))$ fix, β s; $isZero\ 1$ false; $pred\ 1 = 0$ $\rightsquigarrow^* mult\ 2\ (mult\ 1\ 1)$ fix, β s; $isZero\ 0 = K$: base case 1 $\rightsquigarrow^* 2$ $mult\ 1\ 1 = 1$, then $mult\ 2\ 1 = 2$

Three unfoldings of fix, one per layer of recursion. The numbers, test and product are ready-made: only the loop is new.

Step 4: the arithmetic was combinators already

- ▶ The three operations and the numerals are the ready-made constants of Chapter 1.
- ▶ Theorem 1.8 says each is a term in K and S : Example 1.6 builds the numerals and addition, Exercise 1.1 multiplication. The predecessor is a famous enough puzzle to have its own anecdote.

Nothing is being smuggled in.

We keep them as names only to keep the term on the page.

Step 5: the λ s become K and S

The proof of the Deduction Theorem, run with the programs on, takes a term with a parameter and returns one without (Theorem 3.3).

As a recipe: eliminate a single λ from a term M containing no other, producing $[x]M$ in which x does not occur.

THE GUARANTEE

$([x]M) N$ runs to M with N in place of x , for every N .

The recipe: three equations

$$[x]M = K M \qquad \text{if } x \text{ does not occur in } M$$

$$[x]x = S K K$$

$$[x](M N) = S ([x]M) ([x]N) \quad \text{otherwise}$$

The three cases of the Deduction Theorem's proof, wearing programs: a line that does not use the hypothesis; the hypothesis itself ($S K K$ proves $A \rightarrow A$); a modus ponens.

Worked: eliminating a parameter

Compile $\lambda x. f(fx)$, that is $[x](f(fx))$.

EXPRESSION EQUALS

$$[x](f(fx)) = S([x]f)([x](fx))$$

$$[x]f = Kf$$

$$[x](fx) = S([x]f)([x]x) = S(Kf)I$$

$$\text{result } S(Kf)(S(Kf)I)$$

RULE

third equation

first: x not in f

third, first, second

Here $I = SKK$.

Worked: check it does what $\lambda x. f(f x)$ does

Apply the result to an argument a .

STEP	TERM	REASON
	$S(K f)(S(K f)I) a$	
$\leadsto$	$K f a (S(K f)I a)$	S
$\leadsto$	$f(S(K f)I a)$	K
$\leadsto$	$f(K f a (I a))$	S
$\leadsto$	$f(f(I a))$	K
$\leadsto^*$	$f(f a)$	I (three steps in SKK form, one if I is a constant)

Exactly $(\lambda x. f(f x)) a$. The parameter is gone and the behaviour is not.

Your turn: eliminate the parameter

Compute

$$[x] (g x x)$$

that is, $[x]((g x) x)$. Then apply the result to a and check you get $g a a$.

THREE MINUTES

Which equation applies first?

Which pieces contain x ?

Your turn: the answer

EXPRESSION EQUALS

RULE

 $[x]((gx)x) = S([x](gx))([x]x)$

third equation

 $[x](gx) = S(Kg)I$

as before

 $[x]x = I$

second

result $S(S(Kg)I)I$

STEP TERM

REASON

 $S(S(Kg)I)Ia$
 $\leadsto S(Kg)Ia(Ia)$

S

 $\leadsto^* ga a$

S, K, then I twice

Apply it to F : the innermost parameter first

Write B for the body, so $F = \lambda r. \lambda n. B$, and read B as the application

$(isZero\ n\ 1)\ (mult\ n\ (r\ (pred\ n)))$

The third equation splits it; the first and second finish each half.

Eliminate n first, then r .

With I written for SKK .

After eliminating n

$$[n]B = S \underbrace{(S(S(K\textit{isZero})I)(K1))}_{\text{the test and the base case}} \underbrace{(S(S(K\textit{mult})I)(S(Kr)(S(K\textit{pred})I))))}_{\text{the recursive case}}$$

Twenty-six symbols, with r still free in it.

After eliminating r : every variable is gone

$$F = S (K (S A)) (S (K (S C)) (S (S (K S) (S (K K) I)) (K E)))$$

$$A = S (S (K \text{isZero}) I) (K 1)$$

$$C = S (K \text{mult}) I$$

$$E = S (K \text{pred}) I$$

Thirty-eight symbols, counting I as its three.

Factorial is this combinator

- ▶ There is not a variable, a parameter, a conditional or a recursive call anywhere in F .
- ▶ Only K , S , three arithmetic constants and juxtaposition.
- ▶ $fix\ F$ applied to $\bar{4}$ runs, by the two rules of Definition 1.3, to 24.

The promise of Chapter 1, kept: the whiteboard algorithm, as a combinator.

(Checked: the term above is what the three equations produce, and a small evaluator runs $fix\ F$ on 0 to 5 to 1, 1, 2, 6, 24, 120.)

The whole part in one object

- ▶ A program you could have written in your first week is a closed term in two constants (Chapter 1).
- ▶ It has a type, $Int \rightarrow Int$, and the typing that proves so is a proof (Chapter 2).
- ▶ The proof is the one a mathematician would write, in Gentzen's rules, with the terms rubbed out (Section 3.4).

Every well-typed thing you compile sits on K and S in exactly this way.

With one exception

The two constants did not supply the loop: *fix* had to be added, and Remark 3.22 has said what adding it costs.

Normalisation goes (Section 3.6), so *fix* F is a typed term with no normal form. Read as a proof, *fix* proves everything, $\perp$ included.

The price of factorial is the **consistency** of the logic.

Every language you use pays it, knowingly: the loop is worth more to a programmer than consistency. A programmer who wants both must say in the type where the recursion may go.

Chapter 3

- ▶ $\lambda x. M$ is a function without a name, run by β (substitution, with care for capture) and typed by (var), (app), (abs). λ -terms and combinator terms are one language.
- ▶ Pairs, tagged unions and the empty type are $\wedge, \vee, \perp$: constructors are introduction rules, pattern matches are eliminations. Negation is an arrow into $\perp$.
- ▶ Rub the programs out and what is left is Gentzen's **natural deduction**. **Curry–Howard**: a typing is a derivation, a program is a proof, a type is a proposition.
- ▶ Curry's paradox is Ω ; recursion costs consistency; and factorial, whiteboard to combinator, is Part 1 in one object.

Next: for all, and proving from a specification.

READING

Sources

- ▶ Girard, Lafont and Taylor, *Proofs and Types*, chapters 2–4. Natural deduction, proofs as programs, detours, and normalisation for $\rightarrow$ and $\wedge$. Short and free online.
- ▶ Gentzen (1934). The original.
- ▶ Prawitz (1965). The book that made normal proofs a subject.
- ▶ Howard (written 1969, published 1980). The correspondence as Howard wrote it.
- ▶ Wadler, *Propositions as types*. CACM 2015. The story for programmers.

Thank you.