

PART 1: TYPE THEORY · CHAPTER 4

Polymorphism

Alexander V. Gheorghiu

COMP0009 Logic · University College London

CHAPTER 4

Polymorphism

From “safe to run” to “correct”.

OPENING

Chapter 3 ended one step short

Everything we proved in Chapter 3 was proved from **no hypotheses at all**.

Nothing worth proving about a program is like that.

Here are two things you would actually want to know.

A bug report.

A sorting routine.

Neither is a formula of Chapter 3.

OPENING

A bug report

“Discount not applied on the last day of the sale.” A model proposes this fix, a reviewer approves it, and the tests pass.

```
- return start <= today && today <= end;  
+ return start < today && today < end;
```

Is the patch correct?

OPENING

It makes the failing test pass

- ▶ It removes the discount on the *first* day of the sale. No test covers that.
- ▶ It is perfectly well typed. No compiler will ever object.
- ▶ By everything in Chapters 1–3 it is **safe to run**.

Every *yes* you could give rests on something nobody has checked.

What a sale *is*: the discount applies on *every* day from the start to the end, those two included.

OPENING

What a sorting routine promises

In the list it hands back, **every** element is at most the next one.

- ▶ Both claims say *for all*.
- ▶ Chapter 3 had no way to say *for all*.
- ▶ Worse: no way even to *name* the things quantified over, the days, the positions in a list. Its letters p , q , r stood for whole claims and had nothing inside.

WHAT IS MISSING

Names for things.

Claims about them.

Every and *some*.

OPENING

Two things are missing

1. A LANGUAGE

Terms that name things, predicates that say something about them, quantifiers that range over them.

Section 4.1. It is the language the rest of the course is written in.

2. THE RULES

Here you are further ahead than you think. You have been writing $\forall$ for years.

Sections 4.2 and 4.3.

OPENING

The $\forall$ you already write

The type of `length` is `[a] -> Int`. With the quantifier it silently carries:

$$\forall a ([a] \rightarrow Int)$$

Your compiler has rules for that $\forall$ and applies them hundreds of times a second.

They range over *types*, not over days. A different quantifier.

The same rules, side conditions included.

OPENING

Where we are going

SECTION

QUESTION

- | | |
|-----|---|
| 4.1 | What is the language of specifications? |
| 4.2 | What are $\forall$ and $\exists$ in the type checker you use? |
| 4.3 | The same four rules, written as logic. |
| 4.4 | Derivations as trees; the one condition that needs care. |
| 4.5 | A sorting contract, proved. |
| 4.6 | Checking is easy. Finding is not. |

SECTION 4.1

First-order syntax

A language with names, claims, and every.

A type cannot say which integer

A type says what *kind* of thing a value is: an integer, a list of strings.

It cannot say *which* integer. A specification is about which.

To write *the discount applies on every day of the sale* we need:

- ▶ symbols for days, for the start and the end;
- ▶ a relation: *is on or after*;
- ▶ a way to say *every*.

The language that supplies them is the one Frege arrived at in 1879.

Nobody has had to replace it.

The sale, in that language

Symbols: s and e for the start and end days, $\text{Disc}(d)$ for “the discount applies on day d ”, and $\leq$.

$$\forall d \left((s \leq d \wedge d \leq e) \rightarrow \text{Disc}(d) \right)$$

IN WORDS

For every day d : if d is on or after the start and on or before the end, the discount applies.

The first and last days are the two $\leq$ s.

The patch turned them into $<$. Now the bug has a place to live.

Definition 4.1: first-order formulas (1) signature and terms

1. A signature fixes some constant symbols $c, d, \dots$, some function symbols $f, g, \dots$ and some predicate symbols $P, Q, \dots$, each function and predicate symbol with an arity: the number of arguments it takes. We always include the binary predicate $=$.
2. There is a stock of variables $x, y, z, \dots$
3. The terms are the variables, the constant symbols, and $f(t_1, \dots, t_n)$ for an n -ary function symbol f and terms t_i .

4.1 · FIRST-ORDER SYNTAX

Definition 4.1: first-order formulas (2) formulas

The formulas are defined inductively:

1. $P(t_1, \dots, t_n)$ is a formula, for an n -ary predicate symbol P and terms t_i . (An atomic formula.)
2. $\perp$ is a formula.
3. If A and B are formulas, so are $A \rightarrow B$, $A \wedge B$ and $A \vee B$.
4. If A is a formula and x a variable, $\forall x A$ and $\exists x A$ are formulas.
5. Nothing else is.

As before $\neg A$ abbreviates $A \rightarrow \perp$, and binary symbols such as $<$, $=$, $+$ are written between their arguments.

Definition 4.1 in one line

$$A, B ::= P(t_1, \dots, t_n) \mid \perp \mid A \rightarrow B \mid A \wedge B \mid A \vee B \mid \forall x A \mid \exists x A$$

$$t ::= x \mid c \mid f(t_1, \dots, t_n)$$

Chapter 3's grammar, with two new rows: the quantifiers. And atomic formulas now have terms inside, which is what the letters p, q, r lacked.

4.1 · FIRST-ORDER SYNTAX

A term names a thing; a formula makes a claim

Keep the two apart, as a parser would, by giving them different types.

In $x + 1 < y$:

- ▶ 1 is a constant symbol, + a function symbol, < a predicate symbol;
- ▶ x and y are variables;
- ▶ $x + 1$ is a **term**; $x + 1 < y$ is a **formula**.

It is a *type error* to write either where the other belongs.

$(x + 1) \wedge y$ and $\forall(x + 1) y < z$ are not formulas.

Your turn: term, formula, or neither?

Signature: constants c, d ; a unary function f ; a binary function $+$; unary predicate P ; binary predicate $<$.

1. $f(c)$
2. $P(f(c))$
3. $c \wedge d$
4. $\forall x (x < f(x))$
5. $f(P(c))$
6. $P(x) \rightarrow \exists y (x < y + c)$

4.1 · FIRST-ORDER SYNTAX

Your turn: the answer

ITEM EXPRESSION

VERDICT

- | | |
|---|--|
| 1. $f(c)$ | a term |
| 2. $P(f(c))$ | a formula (atomic) |
| 3. $c \wedge d$ | neither: $\wedge$ joins formulas, and c, d are terms |
| 4. $\forall x (x < f(x))$ | a formula, in fact a sentence |
| 5. $f(P(c))$ | neither: f wants a term, and $P(c)$ is a formula |
| 6. $P(x) \rightarrow \exists y (x < y + c)$ | a formula, not a sentence: x is free |

A signature for access control

The rules that decide who may read what.

- ▶ *Constants*: users and resources, `alice` and `cfg`.
- ▶ *Predicates*: `adm( $x$ )`, `own( $x, y$ )`, `read( $x, y$ )`.

Written *Admin*, *Owns*, *CanRead* in the notes; I abbreviate from here on.

READ AS

`adm( $x$ )`: x is an administrator.

`own( $x, y$ )`: x owns y .

`read( $x, y$ )`: x may read y .

A policy is a set of sentences

POLICY IN ENGLISH

AS A SENTENCE

Administrators may read everything $\forall x \forall y (\text{adm}(x) \rightarrow \text{read}(x, y))$ *Owners may read what they own* $\forall x \forall y (\text{own}(x, y) \rightarrow \text{read}(x, y))$

Sentences a security team writes in English and a system enforces in code.

The question “can Alice read the config file?” is the question whether $\text{read}(\text{alice}, \text{cfg})$ follows from them.

By the end of this chapter that question has a certificate.

Your turn: from English to formula

Same signature. Write each as a formula.

1. Alice is an administrator.
2. Alice owns something.
3. Everyone who owns the config file is an administrator.
4. Somebody can read the config file.

Your turn: the answer

ITEM	SENTENCE	KIND
1.	$\text{adm}(\text{alice})$	atomic
2.	$\exists y \text{ own}(\text{alice}, y)$	$\exists$: <i>some</i> y
3.	$\forall x (\text{own}(x, \text{cfg}) \rightarrow \text{adm}(x))$	$\forall$ with $\rightarrow$ inside
4.	$\exists x \text{ read}(x, \text{cfg})$	$\exists$ with a constant

A for all claim is usually a conditional: for every x , if this, then that.

Definition 4.2: free, bound, substitution (1) scope

- ▶ In $\forall x A$ and $\exists x A$ the formula A is the scope of the quantifier, which binds x there, exactly as λ bound its parameter in Chapter 3.
- ▶ An occurrence of a variable is bound if it lies in the scope of a quantifier on that variable, and free otherwise.
- ▶ A sentence is a formula with no free variables.

4.1 · FIRST-ORDER SYNTAX

Definition 4.2: free, bound, substitution (2) substitution

- ▶ $A(t/x)$ is the result of replacing every *free* occurrence of x in A by the term t , bound occurrences left alone.
- ▶ A term t is substitutable for x in A if no variable of t lands in the scope of a quantifier on that variable when t is put in for the free occurrences of x . In words: t arrives in A with its variables still free.
- ▶ A closed term is always substitutable, and so is a variable on which A has no quantifier.

When the condition fails: rename the bound variable to a letter that occurs nowhere else, then substitute. The remedy Chapter 3 gave for λ .

Worked: free or bound?

Take $\forall x (x < y) \rightarrow \exists y (x = y)$. Mark each occurrence.

OCCURRENCE	FREE OR BOUND	BOUND BY
x in $\forall x (x < y)$:	bound	by $\forall x$
y in $\forall x (x < y)$:	free	no $\forall y$ around it
x in $\exists y (x = y)$:	free	the $\forall x$ ended before
y in $\exists y (x = y)$:	bound	by $\exists y$

A formula, but not a sentence.

The first quantifier's scope is only $(x < y)$.

Worked: is y substitutable for x ?

In $A = \exists y (x = y)$. Put y for the free x and look where it lands.

$t := y$: CAPTURED

$\exists y (y = y)$

The new y lands under $\exists y$ and is bound.
The claim has changed.

$t := z$: FINE

$\exists y (z = y)$

z arrives free. Still says: z equals something.

y is not substitutable for x here. z is.

Your turn: substitute

Compute $A(t/x)$, or say t is not substitutable and rename first.

1. $A = \forall y (x < y), \quad t = c$
2. $A = \forall y (x < y), \quad t = y$
3. $A = P(x) \wedge \forall x Q(x), \quad t = f(c)$
4. $A = \exists z (x < z), \quad t = z + 1$

Your turn: the answer

ITEM RESULT

WHY

- | | |
|------------------------------------|---|
| 1. $\forall y (c < y)$ | closed term: always substitutable |
| 2. not substitutable | y would land under $\forall y$. Rename: $\forall w (x < w)$, then $\forall w (y < w)$ |
| 3. $P(f(c)) \wedge \forall x Q(x)$ | the bound x is left alone |
| 4. not substitutable | z would be captured. Rename: $\exists w (x < w)$, then $\exists w (z + 1 < w)$ |

SECTION 4.2

Polymorphic type theory

The type system you use, with its quantifier.

What type should `length` have?

It takes a list and returns a number. But a list *of what*?

Of anything.

Without a way to say *of anything* you have three options, and all are bad.

Built the way Chapters 1 and 3 built theirs: types, terms, judgments, rules, typings.

Added: a quantifier over types, and two rules for it.

Three bad options

COPY IT

`lengthOfIntList,`
`lengthOfStringList,`
and one more for every
element type.

Copied until one copy is
fixed and the others are not.

CAST

`[Object] -> Int`, casting
in and out.

Java, until 2004. The casts
are unchecked: a compile
error became an exception
in production.

TURN THE CHECKER OFF

Dynamic languages start
here.

It is where the Mars Cli-
mate Orbiter ended.

What the quantifier buys

One piece of code. One proof of safety. Every element type at once. Nothing unchecked anywhere.

That is why `map`, `filter`, `sort`, every container and every interface worth the name have it in their types.

Definition 4.3: polymorphic types (1)

- Fix a stock of type variables $a, b, c, \dots$ alongside the basic types of Definition 1.9.
- The types are extended by two clauses: every type variable a is a type; and if A is a type and a a type variable then $\forall a A$ is a type.

In one line, with p a basic type:

$$A, B ::= p \mid a \mid A \rightarrow B \mid A \wedge B \mid A \vee B \mid \perp \mid \forall a A$$

Definition 4.3: polymorphic terms and contexts (2)

The terms of Definition 3.1 are extended by two clauses:

$$M, N ::= \dots \mid \Lambda a. M \mid M[A]$$

- ▶ $\Lambda a. M$ is a type abstraction: the program M made to work for every a .
- ▶ $M[A]$ is a type application: that program used at the type A .

A context Γ is now a finite list of two kinds of entry, type variables a and typings $x : A$, no name listed twice, and every type variable free in a typing in Γ must be declared earlier in Γ .

A type variable in the context

A type variable in the context is a type nobody has chosen yet.

That is the only new idea. Both new rules turn on it.

A CONTEXT NOW LOOKS LIKE

$a, b, f : a \rightarrow b, x : a$

Two types nobody has chosen, and two variables that use them.

The two rules for $\forall$

$$(\forall\text{I}) \frac{\Gamma, a \vdash M : A}{\Gamma \vdash \lambda a. M : \forall a A} \quad \text{if } a \text{ is free in no type in } \Gamma$$

$$(\forall\text{E}) \frac{\Gamma \vdash M : \forall a A}{\Gamma \vdash M [B] : A(B/a)}$$

Rules (4.1) in the notes. $A(B/a)$ is A with B put in for the type variable a .

$(\forall E)$ is a call

$$\frac{\Gamma \vdash M : \forall a A}{\Gamma \vdash M [B] : A(B/a)}$$

A program that works for every a may be used at any particular B . Using it is substituting B for a throughout its type.

It is the one you use every time you call a library function.

length at strings: $a := \text{String}$.

$(\forall I)$ is a finished definition

$$\frac{\Gamma, a \vdash M : A}{\Gamma \vdash \Lambda a. M : \forall a A} \quad a \text{ free in no type in } \Gamma$$

If you typed the body with a in the context, that is with a standing for no type in particular, then the program works for every a , and you may say so.

**Proved for an arbitrary a ,
hence for all.**

The side condition is what *arbitrary* means. We will meet it in a compiler error.

Example 4.4: the identity, polymorphically

$\text{id } x = x$ should work at every type, and does. Context $a, x : a$: (var), then (abs) discharges x , then ($\forall$ I) discharges a .

LINE	JUDGEMENT	JUSTIFICATION
1.	$a, x : a \vdash x : a$	(var)
2.	$a \vdash \lambda x. x : a \rightarrow a$	(abs) 1
3.	$\vdash \Lambda a. \lambda x. x : \forall a (a \rightarrow a)$	($\forall$ I) 2

Line 3 is legal because by then the context is empty, so a is free in nothing.

Example 4.4: using the identity

Using it at *Int* is one application of ($\forall$ E).

LINE	JUDGEMENT	JUSTIFICATION
3.	$\vdash \Lambda a. \lambda x. x : \forall a (a \rightarrow a)$	from before
4.	$\vdash (\Lambda a. \lambda x. x) [Int] : Int \rightarrow Int$	($\forall$ E) 3, $B := Int$

IN HASKELL

`id 3`

You write neither the Λ nor the $[Int]$.

The compiler inserts both.

Which is why the quantifier is easy to use for years without seeing it.

Your turn: type K polymorphically

Type the program $\lambda x. \lambda y. x$ so that it works for every a and every b .

Start from the context $a, b, x : a, y : b$. Work down, then discharge in the right order.

AIM

$$\forall a \forall b (a \rightarrow b \rightarrow a)$$

4.2 · POLYMORPHIC TYPE THEORY

Your turn: the answer

LINE JUDGEMENT

JUSTIFICATION

- | | |
|---|------------------|
| 1. $a, b, x : a, y : b \vdash x : a$ | (var) |
| 2. $a, b, x : a \vdash \lambda y. x : b \rightarrow a$ | (abs) 1 |
| 3. $a, b \vdash \lambda x. \lambda y. x : a \rightarrow b \rightarrow a$ | (abs) 2 |
| 4. $a \vdash \Lambda b. \lambda x. \lambda y. x : \forall b (a \rightarrow b \rightarrow a)$ | ($\forall$ I) 3 |
| 5. $\vdash \Lambda a. \Lambda b. \lambda x. \lambda y. x : \forall a \forall b (a \rightarrow b \rightarrow a)$ | ($\forall$ I) 4 |

Line 4: b is free in no type in the context a . Line 5: a is free in nothing. The context shrinks one entry at a time.

Example 4.5: `twice`, and where the type is forced

Let $twice := \lambda f. \lambda x. f(fx)$. Context: $a, f : a \rightarrow a, x : a$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$a, f : a \rightarrow a, x : a \vdash f x : a$	(app)
2.	$a, f : a \rightarrow a, x : a \vdash f(fx) : a$	(app) again
3.	$a \vdash twice : (a \rightarrow a) \rightarrow a \rightarrow a$	(abs) twice
4.	$\vdash \Lambda a. twice : \forall a ((a \rightarrow a) \rightarrow a \rightarrow a)$	($\forall$ I)

Example 4.5: what the type rules out

The type could *not* have been

$$\forall a \forall b \left((a \rightarrow b) \rightarrow a \rightarrow b \right)$$

There is no way to type $f(f\ x)$ unless f returns what it takes. With $f : a \rightarrow b$ the second application has nothing to feed on.

The type is not a label we chose.

It is what the two uses of (app) forced.

Your turn: use **twice**

Use $\Lambda a. \textit{twice}$ at $\textit{Int}$. What is the type? Then apply it to $(+1)$ and 0 .

One rule: $(\forall E)$. Then (app) , twice.

EXPECT

a number of type $\textit{Int}$, after two steps of $(+1)$.

Your turn: the answer

LINE JUDGEMENT

1. $\vdash \Lambda a. \text{twice} : \forall a ((a \rightarrow a) \rightarrow a \rightarrow a)$
2. $\vdash (\Lambda a. \text{twice}) [Int] : (Int \rightarrow Int) \rightarrow Int \rightarrow Int$
3. $\vdash (\Lambda a. \text{twice}) [Int] (+1) : Int \rightarrow Int$
4. $\vdash (\Lambda a. \text{twice}) [Int] (+1) 0 : Int$

It runs to 2: $(+1)((+1)0)$.

JUSTIFICATION

Example 4.5

($\forall E$) 1

(app) 2, $(+1) : Int \rightarrow Int$

(app) 3, $0 : Int$

The side condition, which you have met as an error message

Rule ($\forall$ I) carries a proviso. It is the only interesting one in this chapter.

Generalising over a is allowed only when a is genuinely arbitrary: **fixed by no typing still in the context**.

Line 3 of Example 4.4 was legal because the context was empty by then. Inside the body it would not have been.

With $x : a$ still in the context we may not conclude $x : \forall a a$.

That would make x a value of every type at once.

Try it

```
f :: a -> b  
f x = x
```

GHC answers: *couldn't match expected type `b` with actual type `a`*, adding that `a` is a *rigid type variable* bound by the signature.

The signature promised a program that works for every *a* and every *b* *separately*.

The body only works when they are the same.

4.2 · POLYMORPHIC TYPE THEORY

The compiler is enforcing the proviso

Try to type $\lambda x. x$ at $\forall a \forall b (a \rightarrow b)$. The body has $x : a$ and needs $x : b$.

LINE	JUDGEMENT	JUSTIFICATION
1.	$a, b, x : a \vdash x : a$	(var)
2.	$a, b, x : a \vdash x : b$	no rule gives this

a and b are two different types nobody has chosen. The only way to get b from a is to choose them equal, and the caller chooses.

Section 4.3 writes the same proviso as a side condition on a rule of logic, where it does the same job and catches the same mistake.

The other quantifier: a type can hide something

Suppose you want a counter: a starting state, a way to step it, a way to read off a number.

You do not want to say what the state is. That is the implementation's business.

In Haskell, with the `ExistentialQuantification` extension on:

A *data* declaration with a `forall`. The next slide.

The counter, in code

```
data Counter = forall a. MkCounter a (a -> a) (a -> Int)

intCounter = MkCounter (0 :: Int) (+1) id
listCounter = MkCounter ([] :: [()]) (()::) length

tick3 :: Counter -> Int
tick3 (MkCounter s step out) = out (step (step (step s)))
```

Both counters are values of the one type `Counter`, although one carries an `Int` and the other a list. `tick3` returns 3 for each without being told which.

The type of Counter

$$\exists a \left(a \wedge (a \rightarrow a) \wedge (a \rightarrow \text{Int}) \right)$$

READ AS

There is a type a with a starting state, a step, and a way to read out an `Int`.

TWO RULES

Building one: `MkCounter`.

Taking one apart: the pattern match in `tick3`.

The two rules for $\exists$

$$(\exists\text{I}) \frac{\Gamma \vdash M : A(B/a)}{\Gamma \vdash \mathbf{pack} \langle B, M \rangle : \exists a A}$$

$$(\exists\text{E}) \frac{\Gamma \vdash M : \exists a A \quad \Gamma, a, x : A \vdash N : C}{\Gamma \vdash \mathbf{open} M \mathbf{as} \langle a, x \rangle \mathbf{in} N : C}$$

($\exists\text{E}$) provided a is free in no type in Γ and not free in C . Rules (4.2) in the notes.

$(\exists I)$ builds a module; $(\exists E)$ opens one

$(\exists I)$: THE CONSTRUCTOR

Supply the hidden type B and a value of the interface at that type. Out comes a value that mentions neither.

$(\exists E)$: THE PATTERN MATCH

You may use the package to produce anything you could have produced from a *fresh* type variable a and a value of the interface at a .

Worked: `intCounter` by $(\exists I)$

Hide *Int*. Let *M* be the triple $(0, (+1), id)$. The hypothesis of $(\exists I)$ is the interface *A* with $B := Int$ put in for *a*.

LINE JUDGEMENT

1. $\vdash M : Int \wedge (Int \rightarrow Int) \wedge (Int \rightarrow Int)$

2. $\vdash \mathbf{pack} \langle Int, M \rangle : \exists a (a \wedge (a \rightarrow a) \wedge (a \rightarrow Int))$

JUSTIFICATION

the interface at *Int*

$(\exists I)$ 1, $B := Int$

Line 2 mentions *Int* only as the first half of the package. The type of the whole mentions it nowhere.

The abstraction barrier

The side condition on $(\exists E)$ is worth seeing at work.

- ▶ `tick3` returns an `Int`. That does not mention a . It type-checks.
- ▶ A function that tried to return the counter's *state* would have result type a . Then a is free in the result type, and $(\exists E)$ refuses.

GHC says the rigid type variable a *would escape its scope*.

The same words, almost, as before.

A module boundary is a side condition

The implementation type may be used inside the match and may not leave it.

One side condition on one rule. Not a convention anybody has to remember.

Mitchell and Plotkin (1988) were the first to put it this way.

An honesty about the correspondence

This is the polymorphic λ -calculus, found twice: by Jean-Yves Girard in 1971 and by John Reynolds in 1974. It is the core of every language with generics.

Its $\forall$ and $\exists$ range over *types*. The quantifiers of a specification range over the things a program handles: days, users, positions in a list.

Different quantifiers over different domains.

The logic whose proofs are polymorphic programs is a different and stronger one than this chapter builds.

4.2 · POLYMORPHIC TYPE THEORY

What is exactly the same: four rules

RULE	WHAT IT DOES, AND ITS PROVISIO
$(\forall E)$	instantiate
$(\forall I)$	generalise, but not a variable the context has fixed
$(\exists I)$	package a witness
$(\exists E)$	unpack one, and do not let the witness escape

Having seen them as type rules, you will recognise them in Section 4.3 on sight.

SECTION 4.3

The quantifier rules

The four moves of Section 4.2, written as logic.

4.3 · THE QUANTIFIER RULES

The same treatment as the connectives

Chapter 3 gave each connective a rule that *introduces* it and a rule that *eliminates* it, and nothing else.

The quantifiers get the same treatment: two rules each.

The four rules are the four moves of Section 4.2.

Plus two for $=$.

Added to the ten rules of Definition 3.8.

4.3 · THE QUANTIFIER RULES

Definition 4.6: natural deduction, with quantifiers (1) $\forall$

Add to the rules of Definition 3.8 the following, where t is a term, y a variable, and $A(t/x)$ is substitution as in Definition 4.2.

$$(\forall E) \frac{\Gamma \vdash \forall x A}{\Gamma \vdash A(t/x)} \quad \text{if } t \text{ is substitutable for } x \text{ in } A$$

$$(\forall I) \frac{\Gamma \vdash A(y/x)}{\Gamma \vdash \forall x A} \quad \text{if } y \text{ is free in no formula of } \Gamma$$

Definition 4.6: natural deduction, with quantifiers (2) $\exists$

$$(\exists\text{I}) \quad \frac{\Gamma \vdash A(t/x)}{\Gamma \vdash \exists x A} \quad \text{if } t \text{ is substitutable for } x \text{ in } A$$

$$(\exists\text{E}) \quad \frac{\Gamma \vdash \exists x A \quad \Gamma, A(y/x) \vdash C}{\Gamma \vdash C} \quad \text{if } y \text{ is free in none of } \Gamma, C, \exists x A$$

4.3 · THE QUANTIFIER RULES

Definition 4.6: natural deduction, with quantifiers (3) equality

$$\text{(=I)} \frac{\text{—————}}{\Gamma \vdash t = t}$$

$$\text{(=E)} \frac{\Gamma \vdash s = t \quad \Gamma \vdash A(s/x)}{\Gamma \vdash A(t/x)}$$

A **derivation** is a finite list of judgements as in Definition 3.8, each justified by one of these rules or one of the ten there. I write $\Gamma \vdash A$, as before.

4.3 · THE QUANTIFIER RULES

Read the four against the compiler (1): $\forall$

($\forall E$) IS INSTANTIATION: A CALL

You have something that holds for every x ; you use it at the t in hand.

The side condition stops t being captured on the way in.

($\forall I$) IS GENERALISATION: A DEFINITION

You proved A of a y about which you assumed nothing, so you have proved it of everything.

Assumed nothing: y may not be free in any hypothesis still in use. This is *rigid type variable*.

4.3 · THE QUANTIFIER RULES

Read the four against the compiler (2): $\exists$ $(\exists I)$ IS BUILDING A MODULE

Supply the witness t and the evidence $A(t/x)$, and what comes out mentions neither.

A proof of $\exists x A$ *is* the pair.

 $(\exists E)$ IS USING ONE

From $\exists x A$ you may go on to anything you could have derived from $A(y/x)$ for a fresh y .

You may use the thing, but you may not learn which thing it is.

4.3 · THE QUANTIFIER RULES

The dictionary, again

LOGIC

 $(\forall E)$ $(\forall I)$ $(\exists I)$ $(\exists E)$

COMPILER

call a polymorphic
function

finish a definition

build a module

pattern match on one

THE PROVISOR

 t substitutable: not captured y not free in a hypothesis in use

none

 y fresh, and it may not escape

4.3 · THE QUANTIFIER RULES

Equality: Leibniz's law

- ▶ $(=I)$: everything equals itself.
- ▶ $(=E)$: equals may be substituted for equals in any claim.
- ▶ Symmetry and transitivity are **not rules**, because they follow from these two.

Exercise 4.1 shows it.

Symmetry first: suppose $x = y$ and apply $(=E)$ to the instance $x = x$ of $(=I)$, substituting in the left-hand slot.

Exercise 4.1: symmetry and transitivity

Derive, from (=I) and (=E) alone,

► symmetry: $\vdash \forall x \forall y (x = y \rightarrow y = x)$;

► transitivity:

$\vdash \forall x \forall y \forall z (x = y \wedge y = z \rightarrow x = z)$.

Why are symmetry and transitivity therefore not rules?

HINT FOR SYMMETRY

The claim to substitute into is $v = x$, in the variable v .

4.3 · THE QUANTIFIER RULES

Exercise 4.1: the answer, symmetry

LINE	JUDGEMENT	JUSTIFICATION
1.	$x = y \vdash x = y$	(hyp)
2.	$x = y \vdash x = x$	(=I)
3.	$x = y \vdash y = x$	(=E) 1, 2: $s := x, t := y$, claim $v = x$
4.	$\vdash x = y \rightarrow y = x$	($\rightarrow$ I) 3
5.	$\vdash \forall y (x = y \rightarrow y = x)$	($\forall$ I) 4
6.	$\vdash \forall x \forall y (x = y \rightarrow y = x)$	($\forall$ I) 5

Lines 5 and 6: after line 4 there are no hypotheses, so x and y are free in none.

4.3 · THE QUANTIFIER RULES

Exercise 4.1: the answer, transitivity

LINE	JUDGEMENT	JUSTIFICATION
1.	$x = y \wedge y = z \vdash x = y \wedge y = z$	(hyp)
2.	$\dots \vdash x = y$	($\wedge$ E) 1
3.	$\dots \vdash y = z$	($\wedge$ E) 1
4.	$\dots \vdash x = z$	(=E) 3, 2: $s := y, t := z$, claim $x = v$
5.	$\vdash x = y \wedge y = z \rightarrow x = z$	($\rightarrow$ I) 4
6.	$\vdash \forall x \forall y \forall z (\dots)$	($\forall$ I) three times

Substitution of equals does the work of both. That is why Leibniz's law is the only elimination rule equality needs.

4.3 · THE QUANTIFIER RULES

Notice what is not here

Chapter 2 had a **Deduction Theorem**. There is nothing left for it to say.

Its content, that anything provable on the supposition A yields $A \rightarrow B$ without the supposition, is the rule ($\rightarrow$ I), which this system simply has.

Chapter 2 had to prove that supposing was safe, and paid for every supposition with instances of (II).

Natural deduction supposes and discharges as a matter of course. That is what makes the proofs below short enough to read.

Where the eigenvariable condition bites

The proviso on $(\forall I)$ is worth one example of its own. Getting it wrong is a real class of bug.

The policy is being checked for a particular user u .
The hypothesis in use is $\text{adm}(u)$: *this* user is an administrator.

THE TEMPTATION

$$\text{adm}(u)$$

$$\forall x \text{adm}(x)$$

One application of $(\forall I)$.

4.3 · THE QUANTIFIER RULES

What that bug would prove

$\forall x \text{ adm}(x)$: *everyone* is an administrator. With the policy, *everyone may read everything*.

The rule forbids it, and the condition is exactly the right one: u is free in the hypothesis $\text{adm}(u)$, which is still in use. So u is not arbitrary.

Discharge the hypothesis first, by $(\rightarrow\text{I})$, and generalisation becomes legal.

But what you generalised is $\text{adm}(u) \rightarrow \dots$: the harmless and true thing.

4.3 · THE QUANTIFIER RULES

Exercise 4.5: if $(\forall I)$ had no side condition

1. From the hypothesis $u = c$, for a constant c , derive $\forall x (x = c)$ in one step, then $\vdash u = c \rightarrow \forall x (x = c)$, and then $\vdash \forall x (x = c)$.
2. Conclude $\perp$ from the extra hypothesis $\neg \forall x (x = c)$, which says only that c is not the only thing there is.
3. Say which of the two steps in (1) the real rule forbids, and state the corresponding compiler error.

4.3 · THE QUANTIFIER RULES

Exercise 4.5: the answer

LINE JUDGEMENT

JUSTIFICATION

- | | |
|---|---|
| 1. $u = c \vdash u = c$ | (hyp) |
| 2. $u = c \vdash \forall x (x = c)$ | the <i>illegal</i> ($\forall I$) 1 |
| 3. $\vdash u = c \rightarrow \forall x (x = c)$ | ($\rightarrow I$) 2 |
| 4. $\vdash \forall x (x = c)$ | generalise, instantiate at c ; ($=I$) gives $c = c$ |

(2) With $\neg \forall x (x = c)$ one ($\rightarrow E$) gives $\perp$.

(3) Line 2: u is free in $u = c$, still in use. The compiler says *a is a rigid type variable*.

4.3 · THE QUANTIFIER RULES

Substitutability: the same bug twice

($\forall$ E) lets you go from $\forall x A$ to $A(t/x)$: what holds of everything holds of t , *if t is substitutable for x in A* .

Drop the condition and the system proves $\perp$ from a specification any programmer would write.

The one hypothesis:

$$\forall x \neg \forall y (y = x)$$

“No element is the only element”: what you say of any collection with two things in it.

4.3 · THE QUANTIFIER RULES

Setting up the cheat

Take A to be $\neg\forall y (y = x)$ and t to be the variable y .

Is y substitutable for x in A ?

No: the free x sits under $\forall y$, so the y we put in would land under that quantifier.

Pretend $(\forall E)$ did not care.

WHAT WE SUBSTITUTE

$A(y/x)$ is $\neg\forall y (y = y)$.

The new y is captured.

4.3 · THE QUANTIFIER RULES

Five lines to $\perp$

LINE	FORMULA	JUSTIFICATION
1.	$\forall x \neg \forall y (y = x)$	hypothesis
2.	$\neg \forall y (y = y)$	the illegal ($\forall E$) on 1, $t := y$
3.	$y = y$	(=I)
4.	$\forall y (y = y)$	($\forall I$) on 3; y is free in no hypothesis
5.	$\perp$	($\rightarrow E$) 2, 4, since line 2 is $\forall y (y = y) \rightarrow \perp$

Exactly the disaster Theorem 3.19 ruled out. The only thing that let it happen was line 2.

4.3 · THE QUANTIFIER RULES

What went wrong, and the fix

WHAT WENT WRONG

The y we substituted meant *some given element*. On landing under $\forall y$ it came to mean *whichever y the quantifier picks*.

THE FIX: RENAME FIRST

The hypothesis is equally $\forall x \neg \forall z (z = x)$. Its legal instance at y is $\neg \forall z (z = y)$.

Nothing follows from that but what should.

4.3 · THE QUANTIFIER RULES

You have met this bug

It is **variable capture**: the bug that clause 5 of substitution in Chapter 3 exists to prevent.

Running a program was substitution,
 $(\lambda x. M) N \rightsquigarrow M(N/x)$, and it needs the same condition.

Take $\lambda x. \lambda y. x$, which is K , applied to a free variable y .

Run it blindly. Next slide.

Capture in a program

BLINDLY

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

The free y landed under the inner λy and was captured. The identity returns whatever it is given. The meaning has changed.

RENAME FIRST

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

The constant function we meant.

Capture swaps one theorem for another

With the free y of type B and the parameter of type A :

- ▶ the program before the step is a proof of $A \rightarrow B$ from the hypothesis B ;
- ▶ the blind result $\lambda y. y$ is a proof of $A \rightarrow A$ from nothing.

One bug, one fix, on both sides of the correspondence.

An implementation never substitutes text; it carries the value in an environment. So you have met the bug in macros and refactoring tools, which do.

SECTION 4.4

Derivations as trees

Four more clauses, and the one condition that needs care.

Recall: trees, and open hypotheses

Definition 3.12 built derivations as trees for the connectives. For each derivation $\mathcal{D}$ it tracked the set $\text{OH}(\mathcal{D})$ of hypotheses it still rests on.

Four clauses extend it to the quantifiers.

The side conditions become **easier** to state in trees, not harder.

That is the reason for doing it.

Definition 4.7: derivations, with quantifiers (clause 10)

Add to the clauses of Definition 3.12 the following four.

10. ($\forall E$) If $\mathcal{D}$ concludes $\forall x A$ and t is a term substitutable for x in A , then

$$\frac{\mathcal{D}}{\forall x A} \quad \forall E \quad \text{is a derivation,} \quad \text{OH} = \text{OH}(\mathcal{D})$$
$$A(t/x)$$

Definition 4.7: derivations, with quantifiers (clause 11)

11. ($\forall$ I) If $\mathcal{D}$ concludes $A(y/x)$ and the variable y is free in no formula of $\text{OH}(\mathcal{D})$, then

$$\frac{\mathcal{D} \quad A(y/x)}{\forall x A} \forall\text{I} \quad \text{is a derivation,} \quad \text{OH} = \text{OH}(\mathcal{D})$$

The condition is on the *open hypotheses*, not on a context.

Definition 4.7: derivations, with quantifiers (clause 12)

12. ($\exists$ I) If $\mathcal{D}$ concludes $A(t/x)$ and t is substitutable for x in A , then

$$\frac{\mathcal{D} \quad A(t/x)}{\exists x A} \exists\text{I} \quad \text{is a derivation,} \quad \text{OH} = \text{OH}(\mathcal{D})$$

4.4 · DERIVATIONS AS TREES

Definition 4.7: derivations, with quantifiers (clause 13)

13. ($\exists E$) If $\mathcal{D}_0$ concludes $\exists x A$ and $\mathcal{D}_1$ concludes C , and the variable y is free neither in C , nor in $\exists x A$, nor in any formula of $\text{OH}(\mathcal{D}_1) \setminus \{A(y/x)\}$, then

$$\frac{\begin{array}{c} \mathcal{D}_0 \\ \exists x A \end{array} \quad \begin{array}{c} [A(y/x)] \\ \mathcal{D}_1 \\ C \end{array}}{C} \quad \exists E$$

is a derivation, with $\text{OH} = \text{OH}(\mathcal{D}_0) \cup (\text{OH}(\mathcal{D}_1) \setminus \{A(y/x)\})$.

The eigenvariable

The variable y in clauses 11 and 13 is the eigenvariable of the step.

It is the only thing in this chapter that needs care.

Clause 11: y in no open hypothesis.

Clause 13: three conditions.

Read each one.

4.4 · DERIVATIONS AS TREES

What the condition says, and why the tree says it better

Clause 11: y free in no *open hypothesis* of the subderivation.

That is the exact sense of *y was arbitrary*: nothing the derivation still leans on says anything about y in particular, so whatever was shown of y was shown of anything.

LIST FORM: COARSER

$(\forall I)$ of Definition 4.6 asks that y be free in no formula of Γ , whatever set the line happens to carry. It may include formulas the derivation never used.

A tree carries exactly the hypotheses it needs.

Where the tree is sharper

This is the one place where the tree notation is not merely nicer to read but says something sharper.

It is why proof theory is written in trees.

Clause 13: three ways of cheating, ruled out

- (a) **y not free in C** : you may not let the witness escape in the answer.
- (b) **y not free in $\exists x A$** : the fresh name must really be fresh, not a variable the existential claim already mentions.
- (c) **y not free in the other open hypotheses**: nothing else in use may already say something about y .

Cheat (a): the witness escapes

*Some user can read the file. Call them y .
Therefore y can read the file.*

That is not a valid step. y was a name you made up.

The conclusion $C = \text{read}(y, \text{cfg})$ has y free: clause 13 refuses.

IN THE COMPILER

Return the counter's state. Result type a , with a free: *would escape its scope*.

The type-theoretic condition, word for word

The three conditions are the condition of $(\exists E)$ in Section 4.2, translated with y for the type variable a :

The implementation type may be used inside the match and may not leave it.

A derivation that breaks one of them is a program GHC rejects.

With: *rigid type variable would escape its scope.*

The policy, as trees

$$\Gamma_{\text{pol}} = \{ \forall x \forall y (\text{adm}(x) \rightarrow \text{read}(x, y)), \quad \forall x \forall y (\text{own}(x, y) \rightarrow \text{adm}(x)) \}$$

Administrators may read anything, and owners are administrators. We are also told $\text{adm}(\text{alice})$.

Can Alice read the config file?

Two instantiations and one application.

The moves of Section 4.3.

Can Alice read the config file?

$$\begin{array}{c}
 \forall x \forall y (\text{adm}(x) \rightarrow \text{read}(x, y)) \\
 \hline
 \forall y (\text{adm}(\text{alice}) \rightarrow \text{read}(\text{alice}, y)) \quad \forall E, t := \text{alice} \\
 \hline
 \text{adm}(\text{alice}) \rightarrow \text{read}(\text{alice}, \text{cfg}) \quad \forall E, t := \text{cfg} \quad \text{adm}(\text{alice}) \\
 \hline
 \text{read}(\text{alice}, \text{cfg}) \quad \rightarrow E
 \end{array}$$

Three bars. The open hypotheses are the one policy line and the one fact.

The tree is a certificate

The tree witnesses

$$\Gamma_{\text{pol}} \cup \{\text{adm}(\text{alice})\} \vdash \text{read}(\text{alice}, \text{cfg})$$

An auditor who disbelieves the answer need not rerun the system, and need not understand it either.

They check three bars.

Each is a pattern match against a rule, side conditions included.

Exercise 4.2: the policy

Take Γ_{pol} and the extra fact $\text{own}(\text{bob}, \text{notes})$.

1. Derive $\text{read}(\text{bob}, \text{notes})$, as a list and as a tree.
2. Write out in full the derivation $\mathcal{D}$ that the second tree of this section leaves as a name, and then the whole tree, checking the side condition at each ($\forall$ I).
3. Add the policy line “nobody may read the audit log except administrators” and say which of $\forall$, $\exists$, $\neg$ you needed and why the obvious first attempt says something weaker.

Exercise 4.2 (1): the answer as a tree

Write b for bob and n for notes.

$$\begin{array}{c}
 \frac{\frac{\frac{\forall x \forall y (\text{adm}(x) \rightarrow \text{read}(x, y))}{\text{adm}(b) \rightarrow \text{read}(b, n)} \quad \forall E \times 2}{\text{adm}(b) \rightarrow \text{read}(b, n)} \quad \forall E \times 2 \quad \frac{\frac{\frac{\forall x \forall y (\text{own}(x, y) \rightarrow \text{adm}(x))}{\text{own}(b, n) \rightarrow \text{adm}(b)} \quad \forall E \times 2 \quad \text{own}(b, n)}{\text{adm}(b)} \quad \forall E \times 2}{\text{read}(b, n)} \quad \forall E \times 2
 \end{array}$$

Owners are administrators; administrators may read. Four instantiations, two applications. As a list: the same steps, one per line, hypotheses first.

Something no policy engine can evaluate

A claim about *all* users and *all* resources at once:

$$\Gamma_{\text{pol}} \vdash \forall x \forall y (\text{own}(x, y) \rightarrow \text{read}(x, y))$$

Every owner may read what they own. You cannot test it by running the system on every user.

Let $\mathcal{D}$ put the two policy lines together on a supposition $\text{own}(u, v)$.

Four instantiations and two applications. It ends in $\text{read}(u, v)$.

Every owner may read what they own

$$\begin{array}{c}
 \mathcal{D} \\
 \hline
 \text{read}(u, v) \quad \rightarrow\text{I}, 1 \\
 \text{own}(u, v) \rightarrow \text{read}(u, v) \\
 \hline
 \forall\text{I} \\
 \forall y (\text{own}(u, y) \rightarrow \text{read}(u, y)) \\
 \hline
 \forall\text{I} \\
 \forall x \forall y (\text{own}(x, y) \rightarrow \text{read}(x, y))
 \end{array}$$

The supposition $[\text{own}(u, v)]^1$ sits at a leaf of $\mathcal{D}$ and is discharged at the first bar.

Check it against clause 11

After the discharge, OH of the subderivation is just the two policy lines.

Both are sentences, so neither v nor u is free in either. Both $(\forall I)$ steps are legal.

Now you can *check* that, not take it on trust.

IN THE WRONG ORDER

Generalise before discharging, and $\text{own}(u, v)$ is still open. v is free in it. Clause 11 does not apply.

That is the complaint that we claimed the policy for all users on the strength of one.

A proof of *there is* names the thing

From the same policy and `adm(alice)`, one more bar on the first tree:

$$\frac{\text{read}(\text{alice}, \text{cfg})}{\exists x \text{ read}(x, \text{cfg})} \exists\text{I}, t := \text{alice}$$

Somebody can read the config file.

Read as a program, the step packs the witness `alice` with the evidence that she qualifies.

In this logic that is the only way an $\exists$ can be introduced.

4.4 · DERIVATIONS AS TREES

Remark 4.8: a proof of *there is* hands you a witness

REMARK 4.8 · THE EXISTENCE PROPERTY · ON LOAN

If $\vdash \exists x A$, then there is a term t with $\vdash A(t/x)$.

A proof that something satisfies A does not merely rule out that nothing does. It produces a thing that does.

Why it is believable

1. A closed proof of $\exists x A$ is, read as a program, a closed program of that type.
2. Run it. Chapter 3's shape lemma says what it runs to must be an introduction form.
3. The only way to introduce an $\exists$ is $(\exists I)$, which packages a term t with a proof of $A(t/x)$.
4. The witness is the first half of that package: the very t you supply when you build a module.

On loan, as normalisation was: the step needs a normalisation theorem for the quantifier rules, which we have not proved.

That is exactly what an audit wants

Is this file readable by anyone outside the team? is an existence question.

An answer of *yes* that cannot say *who* is no use to a security engineer.

In this logic every *yes* comes with the name attached.

4.4 · DERIVATIONS AS TREES

It is also strong enough to be worth distrusting

- ▶ *Either the program halts or it does not:* you would like to assert this without producing a halting run or a proof there is none.
- ▶ *Some input breaks this function:* you would like to assert this without producing the input.

In the logic of these four chapters you may assert neither.

Whether that is a virtue or a restriction we cannot yet ask.

Asking needs the word Part 2 supplies.

Exercise 4.4: existentials

1. Derive $\vdash \exists x (x = x)$.
2. Derive $\vdash \exists x A \rightarrow \neg \forall x \neg A$, using $(\exists E)$, and say what program it is.
3. Try to derive the converse, $\neg \forall x \neg A \rightarrow \exists x A$, and say where you get stuck. What would Remark 4.8 have to give up for it to go through?

Exercise 4.4: the answer to (1) and (2)

(1)

$(=I)$ gives $x = x$. $(\exists I)$ with $t := x$ gives $\exists x (x = x)$.

(2)

Suppose $\exists x A$ and $\forall x \neg A$. By $(\exists E)$ take a fresh y with $A(y/x)$. $(\forall E)$ on the second gives $\neg A(y/x)$. $(\rightarrow E)$ gives $\perp$, which does not mention y , so $(\exists E)$ is legal. Discharge twice.

As a program: unpack the module, apply the function from the second hypothesis to the witness and its evidence.

Exercise 4.4: the answer to (3)

You get stuck at once. No rule produces an $\exists$ except $(\exists I)$, and $(\exists I)$ demands a witness, which the hypothesis $\neg\forall x\neg A$ does not give you.

The converse would have to be bought by giving up the existence property.

Which is what Part 2 does, and names.

SECTION 4.5

Sorting, and what its type cannot say

A contract, stated and proved.

4.5 · SORTING

Insertion sort, with the comparison as a parameter

```
sort(lt, xs):  
    out = []  
    for x in xs:  
        i = 0  
        while i < length(out) and lt(out[i], x):  
            i = i + 1  
        insert x into out at position i  
    return out
```

This is how every standard library writes it: the library cannot know what you are sorting.

4.5 · SORTING

Everything in this part applies to it

It has a type, and the type is polymorphic in the sense of Section 4.2:

$$\textit{sort} : \forall a \left((a \rightarrow a \rightarrow \textit{Bool}) \rightarrow [a] \rightarrow [a] \right)$$

One program for every element type,
instantiated by ($\forall$ E) at each call.

The checker proves that type, and by
Chapter 3 the proof *is* the program.

What the compiler has certified

- ▶ This code will not add a string to a number.
- ▶ It will not call a non-function.
- ▶ It will hand back a list of the type it was given.

It has certified nothing whatever about sorting.

4.5 · SORTING

The gap is the parameter

Give `sort` a comparison that returns `True` when the two arguments are *equal*: `<=` where `<` was meant.

The single commonest mistake in the business.

The type is unchanged, because `<=` and `<` have the same type.

It compiles. It runs. It returns a list.

The list is sometimes not sorted. On a longer input the routine may do something stranger.

The exception Java really throws

```
java.lang.IllegalArgumentException:  
Comparison method violates its general contract!
```

Read it carefully, because it is unusual and exact.

4.5 · SORTING

What the sort actually noticed

The sort has *not* checked the contract. Checking it would need every pair of elements, which costs more than the sort.

What it noticed is that its own bookkeeping has become impossible: it merged two runs it had recorded as ordered and found them not ordered. That cannot happen if the comparison behaves.

It detected the failure of a **consequence** of the contract.

To say what that means we must be able to state the contract and derive the consequence. Neither is possible in a type.

4.5 · SORTING

The contract

What `sort` needs of `lt` is that it be a *strict total order*. Write $x < y$ for `lt(x,y)`.

WHAT THE LIBRARY NEEDS

AS A SENTENCE

any two elements compare one way, the other, or are equal

$$\forall x \forall y (x < y \vee y < x \vee x = y)$$

nothing is less than itself

$$\forall x \neg(x < x)$$

if $x < y$ and $y < z$ then $x < z$

$$\forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z)$$

Call those three Γ

- ▶ They say *for all*, so no type can hold them.
- ▶ They are about the elements, so Chapter 3 could not write them.
- ▶ They are what the library's documentation is trying to say in English.

Three sentences in the language of Section 4.1: a **specification**.

4.5 · SORTING

The consequence the exception was reporting

A comparator built from `<=` violates *asymmetry*: for a pair of equal elements it reports $a < b$ and $b < a$.

No comparator obeying the contract can.

$$\Gamma \vdash \forall x \forall y \neg(x < y \wedge y < x)$$

That is not one of the three sentences. It has to be **derived**.

4.5 · SORTING

The informal argument, in one line

If $u < v$ and $v < u$, then by transitivity $u < u$, which irreflexivity forbids.

The derivation makes each of those words a step. Write Δ for Γ together with the supposition

$$u < v \wedge v < u.$$

4.5 · SORTING

The derivation (1): strip transitivity, fire it

LINE JUDGEMENT

JUSTIFICATION

- | | |
|---|-----------------------------|
| 1. $\Delta \vdash u < v \wedge v < u$ | (hyp), the supposition |
| 2. $\Delta \vdash \forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z)$ | (hyp), transitivity |
| 3. $\Delta \vdash \forall y \forall z (u < y \wedge y < z \rightarrow u < z)$ | ($\forall E$) 2, $t := u$ |
| 4. $\Delta \vdash \forall z (u < v \wedge v < z \rightarrow u < z)$ | ($\forall E$) 3, $t := v$ |
| 5. $\Delta \vdash u < v \wedge v < u \rightarrow u < u$ | ($\forall E$) 4, $t := u$ |
| 6. $\Delta \vdash u < u$ | ($\rightarrow E$) 5, 1 |

Lines 2–5 strip the three quantifiers. Line 5 is the step that says “then $u < u$ ”. Line 6 fires it on the supposition.

4.5 · SORTING

The derivation (2): irreflexivity, to $\perp$

LINE	JUDGEMENT	JUSTIFICATION
7.	$\Delta \vdash \forall x \neg(x < x)$	(hyp), irreflexivity
8.	$\Delta \vdash \neg(u < u)$	($\forall$ E) 7, $t := u$
9.	$\Delta \vdash \perp$	($\rightarrow$ E) 8, 6

Line 9: the supposition leads to absurdity.

4.5 · SORTING

The derivation (3): discharge, and generalise

LINE JUDGEMENT

JUSTIFICATION

- | | |
|--|--|
| 10. $\Gamma \vdash \neg(u < v \wedge v < u)$ | $(\rightarrow I)$ 9, discharging the supposition |
| 11. $\Gamma \vdash \forall y \neg(u < y \wedge y < u)$ | $(\forall I)$ on 10 |
| 12. $\Gamma \vdash \forall x \forall y \neg(x < y \wedge y < x)$ | $(\forall I)$ on 11 |

Line 10 turns the supposition leads to absurdity *into* the supposition is false.

4.5 · SORTING

Check the side conditions at lines 11 and 12

After line 10 the hypotheses in use are the three contract sentences.

A sentence has no free variables. So neither u nor v is free in any of them.

Both generalisations satisfy the side condition, and now you can say exactly why.

Had we generalised at line 9, with $u < v \wedge v < u$ still open, u and v would be free in a hypothesis in use.

Clause 11 would refuse.

Read it in four blocks

LINES WHAT THEY DO

Lines 2–5 strip the three quantifiers off transitivity, choosing $t := u$ last

Line 6 fire it on the supposition

Lines 7–9 strip irreflexivity and reach $\perp$

Lines 10–12 discharge, then generalise twice

RULES

that is the step that says

$(\rightarrow E)$

$(\forall E), (\rightarrow E)$

$(\rightarrow I), (\forall I), (\forall I)$

Two things about the shape of that derivation

1. THE SUPPOSITION DID THE WORK

No axiom in it anywhere, and no theorem imported from elsewhere. $(\rightarrow I)$ lets the derivation suppose, and then pay for the supposition.

Chapter 2 needed contraposition proved separately, and the Deduction Theorem to license the supposition at all.

2. EVERY LINE IS CHECKABLE

By pattern matching against the rules of Definition 4.6, side conditions included.

So the twelve lines are a certificate, in the sense this part has used throughout.

4.5 · SORTING

A theorem the type checker could not state

A comparator obeying the contract can never report both $a < b$ and $b < a$.

When the sort catches a comparator doing exactly that, the exception is reporting a **theorem** being violated.

A theorem the type checker could not state, let alone prove.

A verifier, a type checker whose types may quantify, could.

Safe and correct

That is the whole distance between safe and correct, measured on a program you have called a hundred times.

4.5 · SORTING

Exercise 4.3: break the contract

Write a comparator, in any language, that violates exactly one of the three lines of the contract.

- ▶ Which consequence of the contract does it break?
- ▶ Which violation would a sort routine detect, and how?

START FROM

`<=` where `<` was meant.

Which of the three lines does it break?

SECTION 4.6

Checking is easy; finding is not

The ambition of the next part.

We can now ask the question

Does A follow from the specification Γ ?

When the answer is yes, a derivation exists.

It is tempting to conclude that we have an algorithm: to decide the question, *just* check whether a derivation exists.

PROBLEM 4.9

Why is that conclusion wrong?

Think before the next slide.

Problem 4.9: why it is wrong

You do not know how long a derivation of A , if there is one, might be.

Suppose you knew a bound: whenever A follows from Γ , it does so by a derivation of at most a thousand symbols. Then enumerate the finitely many strings of at most that length and check each.

A BOUND ON LINES WOULD NOT DO

Each line can be as long as you like.

A checker and no finder

No such bound is known. A search that has not found a derivation yet cannot tell whether to give up.

What we have is a *checker* for certificates, and no *finder*.

CHECKING

Pattern matching. Your compiler does it.

FINDING

So far, cleverness.

The ambition of the next part

If a proof can be checked mechanically, why can it not be *found* mechanically?

Mathematics would be a thing one could automate: state the specification, state the claim, let the machine report.

Not a fantasy, and not finished.

It goes further than you would guess
and stops sooner than Hilbert hoped.

Your compiler is in our position

It checks the proof your program *is*. It does not find one.

You now have tools that hand you a program of a type you name, or a patch for a bug you describe. What they hand you is a **candidate**.

The compiler's check is what makes a candidate more than a guess.

When the type can say *for all*, the check is a verification against the specification.

Finding is cheap; checking is what is left to trust

Checking is what is left to trust, and it is the subject of this course.

4.6 · CHECKING AND FINDING

A specification is a set of sentences

Three lines of a comparator contract. Two lines of an access-control policy. Whatever a sale is.

A follows from the specification is a precise, checkable claim.

- ▶ The language: first-order logic, Definition 4.1.
- ▶ The rules: Gentzen's, four for the quantifiers, Definition 4.6.
- ▶ A proof of an $\exists$ hands back its witness: Remark 4.8.

The four rules are the compiler's two moves for a polymorphic type and its two ways with a module, side conditions and all.

A type checker and a verifier

A TYPE CHECKER

certifies that a program is **safe to run**.

A VERIFIER

a type checker whose types may say *for all*. Certifies that it is **correct**: the postcondition is provable from the precondition and the specification.

The discount patch is correct or not according to three or four sentences about sales. Once they are written down, the question has an answer a checker can verify.

Three questions about any proof system

QUESTION

1. Could it prove something false?
2. Does it prove everything it should?
3. Can the *search* be made mechanical?

ANSWER

We showed it does not prove *everything*, Theorem 3.19. The
We cannot even ask.
Not yet.

A puzzle to take into next week

Peirce's law is a formula built from letters and arrows like any other:

$$((p \rightarrow q) \rightarrow p) \rightarrow p$$

Try to write a program of that type before next week.

You will not manage. Nothing in this part can tell you whether that is your fault or the formula's.

Where Part 2 begins

Part 2 begins by giving the formulas a meaning, and the first thing it finds out is what Peirce's law is.

Exercise 4.6: types with their quantifiers

1. Write the types of `length` and `map` with their quantifiers explicit.
2. Type the use `map show [1, 2]`, naming the instantiation at each quantifier.
3. Declare `f :: a -> b` and define `f x = x`. Show the exact line of the typing at which the checker is asked to generalise over a variable the context has fixed, and write the error in the words of Section 4.2.

Chapter 4

- ▶ A type cannot say *which*. First-order formulas have terms inside and quantifiers over them: Definition 4.1. Bound, free, substitutable: Definition 4.2.
- ▶ The type system you use has $\forall$ and $\exists$ over *types*: Definition 4.3, Examples 4.4 and 4.5. Their provisos are the compiler's *rigid type variable* and *would escape its scope*.
- ▶ Logic has the same four rules, with the same provisos: Definition 4.6. Capture is one bug on both sides.
- ▶ Trees: the eigenvariable condition is on the *open hypotheses*: Definition 4.7. A proof of $\exists$ hands you the witness: Remark 4.8.
- ▶ A sorting contract was stated and a consequence of it derived. Safe is not correct; a verifier checks *correct*. Checking is easy; finding is not.

Next: Part 2. What the formulas mean.

Sources

- ▶ Girard, Lafont and Taylor, *Proofs and Types*, chapters 2–4. Natural deduction with quantifiers, and the normalisation theorem Remark 4.8 borrows.
- ▶ Open Logic Project, chapter 13. The same rules at more leisure, and the Hilbert-style presentation this chapter did not need.
- ▶ Mitchell and Plotkin (1988). Abstract types have existential type: modules as existentials.
- ▶ Girard (1971); Reynolds (1974). The polymorphic λ -calculus, found twice.

Thank you.