COMP0009 Logic

Lecture notes · 2026–27

Part 1: Type Theory

Chapter 5

Second-Order Logic (Optional)

Alexander V. Gheorghiu

Department of Computer Science, University College London

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

Contents

5. Second-Order Logic (Optional) 1
5.1. Quantifying over propositions 1
5.2. The connectives, defined 3
5.3. Generics 4
5.4. The polymorphic λ-calculus 7
5.5. Normalisation and parametricity 9
History 10
Exercises 13

5. Second-Order Logic (Optional)

Suppose you review code for a bank. A pull request from another team changes a helper that runs whenever the bank's two offices hold different copies of the same customer's record and one of them has to win. The helper is four hundred lines long, and its type is

```
pick :: a -> a -> a
```

The question on the review ticket is the one a bank cares about: can this change corrupt a record, by mixing the fields of two records, say, or blanking a card number? You can approve it after reading one line, the type.

Here is the reasoning. The type says that `pick` works for every type `a`: whatever a record is, it takes two of them and returns one. It was written without knowing what an `a` is, so it can't look inside one, can't change a field, and can't build a new record, because nothing it was given says how. The only `a`s it has are the two it was handed, so what it returns is one of those, as it arrived. Whatever the four hundred lines do, they can't corrupt a record.[1] The one thing left to review is which of the two it keeps, and a single test settles that.

That's a theorem about a program, read off its type, and the type is a quantifier over types: `pick` has the type $\Pi a\,(a \to a \to a)$. The chapter begins with the logic that quantifier belongs to.

The quantifiers of first-order logic range over things. Let one range over propositions instead and you get *second-order logic*, in which a schema becomes a single theorem and the connectives $\wedge$, $\vee$ and $\bot$ can be defined from $\to$ and $\forall$. The quantifier is of a kind Russell banned, so when the logic is on the page its consistency is an open question.

Then the programs go back on. What comes out is something you already use: generics, the `a` in `length :: [a] -> Int`, with the side conditions as two compiler error messages. Written out in full it is the polymorphic λ-calculus, also called System F. Girard's proof that its programs stop answers the logic's question, and turns the reviewer's argument into a theorem. Pushed one step further, to a type of all types, the system collapses, as Church's logic did before types repaired it.

5.1. **Quantifying over propositions.** The first-order quantifiers range over things: kinds, drivers, days. Look back at the propositional theorems, though, and you'll find a quantifier of another kind that nobody wrote down. The theorem $p \to p$ was never about p in particular. Its proof works with every formula in the place of p, and that is why the Hilbert system states its axioms as schemas, with Greek letters standing for every formula.

A schema is a claim about all formulas made one instance at a time: the same thing written forty times, which is the complaint the first-order quantifier answered. The cure is the same. Let a quantifier range over propositions, and write the claim once: $\forall X\,(X \to X)$, "whatever proposition X is, X implies X".

[1] The argument needs a language that keeps its promises: no casts, no reflection, no escape hatches that let code inspect a value of a type it was told nothing about. In Java every object has a `toString` and can be cast, and the argument fails. Haskell's escape hatches are named `unsafe` for this reason, and a review checklist can ban them.

The variable X needs a symbol of its own. The propositional letters p, q, r are *constants*: each stands for one fixed claim, and nothing can bind it. A variable for propositions is as different from a letter as the variable x is from the constant cam, so we write propositional variables differently, as X, Y, Z.

Definition 5.1 (second-order propositional logic)**.** Fix the propositional letters $p, q, r, \ldots$ and, separately, a stock of *propositional variables* $X, Y, Z, \ldots$. The formulas are

$$A, B \;::=\; p \;\mid\; X \;\mid\; A \to B \;\mid\; \forall X\, A,$$

where $\forall X\, A$ binds X in A. Free propositional variables, substitutability, and the substitution $A[B/X]$ of a formula B for the free occurrences of X in A are defined by the clauses for first-order formulas.

The derivations are built by the clauses for hypotheses, ($\to$I) and ($\to$E), and two more:

$$\frac{\begin{array}{c}\mathcal{D}\\ A\end{array}}{\forall X\, A}\ \forall\mathrm{I} \quad \text{if } X \text{ is free in no formula of } \mathrm{OH}(\mathcal{D}),$$

$$\frac{\begin{array}{c}\mathcal{D}\\ \forall X\, A\end{array}}{A[B/X]}\ \forall\mathrm{E} \quad \text{for every formula } B \text{ substitutable for } X \text{ in } A,$$

each with the open hypotheses of $\mathcal{D}$. We write $\Gamma \vdash_{\mathrm{NJ}^2} A$ for derivability in this system, second-order NJ.

The two quantifier rules are the first-order ones with propositions in the place of things: ($\forall$I) generalises over a proposition nothing open says anything about, and ($\forall$E) uses the result at a particular one.

Example 5.2 (two schemas, as theorems)**.** Two schemas become single theorems: the identity, and Frege's first axiom, schema (I) of the Hilbert system.

$$\frac{\dfrac{[X]^1}{X \to X}\ {\to}\mathrm{I},\,1}{\forall X\,(X \to X)}\ \forall\mathrm{I} \qquad\qquad \frac{\dfrac{\dfrac{\dfrac{[X]^1}{Y \to X}\ {\to}\mathrm{I}}{X \to Y \to X}\ {\to}\mathrm{I},\,1}{\forall Y\,(X \to Y \to X)}\ \forall\mathrm{I}}{\forall X\,\forall Y\,(X \to Y \to X)}\ \forall\mathrm{I}$$

In each tree the ($\forall$I) bars come after every hypothesis has been discharged, so nothing open mentions X or Y and the bars are legal. Both trees have $\mathrm{OH} = \varnothing$, so they witness $\vdash_{\mathrm{NJ}^2} \forall X\,(X \to X)$ and $\vdash_{\mathrm{NJ}^2} \forall X\,\forall Y\,(X \to Y \to X)$. An instance of the schema is now one step: ($\forall$E) at p turns $\forall X\,(X \to X)$ into $p \to p$.

Remark (the capture condition). The condition on ($\forall$E), that B be substitutable for X in A, is the first-order one, and it earns its place in the same way. The formula $\forall X\,(\forall Y\,(Y \to X) \to X)$ is a theorem: from the hypothesis $\forall Y\,(Y \to X)$, ($\forall$E) at $Y := \forall Y\,(Y \to X)$ gives $\forall Y\,(Y \to X) \to X$, then ($\to$E) on the hypothesis gives X, ($\to$I) discharges the hypothesis, and ($\forall$I) is legal because nothing is open. Now instantiate it at $B := Y$ without renaming, so that the Y put in is caught by the inner $\forall Y$: the result is $\forall Y\,(Y \to Y) \to Y$. Its premiss $\forall Y\,(Y \to Y)$ is a theorem, by ($\to$I)

and (∀I), so (→E) gives Y from no hypotheses at all, and (∀I) gives $\forall Y\, Y$, from which (∀E) gives every formula. Rename the bound variable first, to $\forall Z\,(Z \to Y) \to Y$, and nothing goes wrong.

5.2. **The connectives, defined.** The logic has → and ∀ and nothing else, and needs nothing else, because ∧, ∨ and ⊥ can be written with those two. Say what you can *do* with a conjunction: from $A \wedge B$ you can conclude every claim that follows from A and B together. Then let that be the definition.

Proposition 5.3 (the connectives in second-order logic)**.** *Define, with X a propositional variable not free in A or B,*

$$\begin{aligned} \bot &:= \forall X\, X, \\ A \wedge B &:= \forall X\, \big((A \to B \to X) \to X\big), \\ A \vee B &:= \forall X\, \big((A \to X) \to (B \to X) \to X\big). \end{aligned}$$

With these definitions every rule of NJ for ⊥, ∧ and ∨ is derivable: for each rule, its conclusion is derivable from its premisses, discharging what the rule discharges.

Proof. Write Δ for the formulas the premisses are derived from, and C for the conclusion wanted. The variable X is chosen free in no formula of Δ, so every (∀I) below is legal.

- *(⊥E).* From $\Delta \vdash_{\mathrm{NJ}^2} \forall X\, X$, (∀E) at C gives $\Delta \vdash_{\mathrm{NJ}^2} C$.
- *(∧I).* From $\Delta \vdash_{\mathrm{NJ}^2} A$ and $\Delta \vdash_{\mathrm{NJ}^2} B$:

$\Delta,\ A \to B \to X \vdash_{\mathrm{NJ}^2} X$	two (→E)s
$\Delta \vdash_{\mathrm{NJ}^2} (A \to B \to X) \to X$	(→I)
$\Delta \vdash_{\mathrm{NJ}^2} A \wedge B$	(∀I).

- *($\wedge E_1$).* From $\Delta \vdash_{\mathrm{NJ}^2} A \wedge B$:

$\Delta \vdash_{\mathrm{NJ}^2} (A \to B \to A) \to A$	(∀E) at A
$\vdash_{\mathrm{NJ}^2} A \to B \to A$	two (→I)s, the inner one discharging nothing
$\Delta \vdash_{\mathrm{NJ}^2} A$	(→E).

  ($\wedge \mathrm{E}_2$) is the same with B and $A \to B \to B$.
- *($\vee I_1$).* From $\Delta \vdash_{\mathrm{NJ}^2} A$, (→E) gives $\Delta,\ A \to X,\ B \to X \vdash_{\mathrm{NJ}^2} X$; two (→I)s and (∀I) give $\Delta \vdash_{\mathrm{NJ}^2} A \vee B$. ($\vee \mathrm{I}_2$) is the same from B.
- *(∨E).* From $\Delta \vdash_{\mathrm{NJ}^2} A \vee B$, and the two branches $\Delta,\ A \vdash_{\mathrm{NJ}^2} C$ and $\Delta,\ B \vdash_{\mathrm{NJ}^2} C$:

$\Delta \vdash_{\mathrm{NJ}^2} (A \to C) \to (B \to C) \to C$	(∀E) at C
$\Delta \vdash_{\mathrm{NJ}^2} A \to C$ and $\Delta \vdash_{\mathrm{NJ}^2} B \to C$	(→I) on each branch
$\Delta \vdash_{\mathrm{NJ}^2} C$	two (→E)s.

□

Each connective is defined by its elimination rule:

- the conjunction $A \wedge B$ says "whatever follows from A and B together, follows";

- the disjunction $A \vee B$ says "whatever follows from A and also from B, follows", which is proof by cases;
- absurdity, $\bot$, says "everything follows".

So propositional NJ sits inside this one, built from $\to$ and $\forall$ alone.

Remark 5.4 (higher-order logic). Second-order propositional logic quantifies over propositions. *Higher-order logic* goes the whole way: over properties of things, relations between them, properties of properties, and functions of every type. Its language is the simply typed λ-calculus itself, with a type o of propositions and a type ι of things, so that a property of things is a term of type $\iota \to o$, and a variable P of that type can be bound by $\forall P$. This is Church's *simple theory of types* (Church, 1940), the logic the simply typed λ-calculus was made to write. In it, induction on the natural numbers is one sentence,

$$\forall P\,\big(P(0) \wedge \forall n\,(P(n) \to P(n+1)) \to \forall n\,P(n)\big),$$

where first-order logic needs a schema, one axiom for each formula.

The power has a price. Once the quantifiers are read as ranging over *all* properties, no proof system whose proofs can be checked mechanically captures that reading, even for second-order logic; Part 2 ends with the reason.

Impredicativity. Look at ($\forall$E) once more. The formula B put in for X may be any formula at all, including $\forall X\,A$ itself: $\forall X\,(X \to X)$ may be used at $\forall X\,(X \to X)$, and $\bot$, which is $\forall X\,X$, at $\bot$. A quantifier over all propositions ranges over a collection that includes the proposition it is helping to form. Definitions of that kind are called *impredicative*, and the first theory of types was built to ban them, for fear of the self-application that produced the paradoxes; the History tells that story.

So we have to ask whether this logic is consistent. For first-order logic the answer came from programs:

1. decorate the derivations;
2. run them;
3. look at what they stop at.

Do the same here.

5.3. **Generics.** Put the programs on. A second-order derivation is decorated like a first-order one:

- the bars for $\to$ become λ and application, as before;
- a ($\forall$I) bar over X becomes a *type abstraction* $\Lambda a.\,M$, a program that takes a type and returns a program;
- a ($\forall$E) bar at B becomes an *instantiation* $M\,[B]$, which hands it one.

The formulas become types, with the usual renaming: a propositional variable X becomes a *type variable* a, a type nobody has chosen yet, and $\forall X$ becomes Πa, the Π of dependent types with types in the place of things. The derivations of $\forall X\,(X \to X)$ and $\forall X\,\forall Y\,(X \to Y \to X)$, each a ($\to$I) or two followed by ($\forall$I)s, decorate to

$$\Lambda a.\,\lambda h.\,h \;:\; \Pi a\,(a \to a) \qquad \text{and} \qquad \Lambda a.\,\Lambda b.\,\lambda h.\,\lambda k.\,h \;:\; \Pi a\,\Pi b\,(a \to b \to a).$$

What has come out is something you've been writing for years, and the type of the helper `pick`. The type $\Pi a\, A$ is that of a program that works for every type a, and the rules are (ΠI) and (ΠE) of dependent types with types in the place of things,

$$(\Pi\text{I})\ \frac{\Gamma,\ a \vdash_{\mathrm{F}} M : A}{\Gamma \vdash_{\mathrm{F}} \Lambda a.\, M : \Pi a\, A} \qquad (\Pi\text{E})\ \frac{\Gamma \vdash_{\mathrm{F}} M : \Pi a\, A}{\Gamma \vdash_{\mathrm{F}} M\,[B] : A[B/a]},$$

where B may mention only type variables declared in Γ. The context is the same stack, and running is one more step, $(\Lambda a.\, M)\,[B] \rightsquigarrow M[B/a]$.

Remark 5.5 (finding a program from its type). Let the type decide, from the outside in. A Πa asks for a Λa, an arrow asks for a λ, and what is left has to be built from the variables declared so far. A type variable is neither a function nor a pair, so a variable of type a can only be handed on. This is a rule of thumb: it says where to look for a program, not that there is nothing else to find. Whether a type such as $\Pi a\,(a \to a)$ admits any other program is the question of Section 5.5.

Example 5.6 (the identity)**.** The first of the two programs above is the function `id x = x`, and its typing is the first tree of Example 5.2 with the programs on:

1. declare a,
2. declare $h : a$,
3. pop h with (abs),
4. pop a with (ΠI).

Using it at `Int` is one (ΠE), $(\Lambda a.\, \lambda h.\, h)\,[\mathit{Int}] \rightsquigarrow \lambda h.\, h : \mathit{Int} \to \mathit{Int}$, and Haskell inserts that instantiation for you at every call.

The type of `length` is `[a] -> Int`, and the `a` in it is a type variable. Haskell leaves the quantifier out, and writes it `forall` only when asked. Written in, it reads $\mathit{length} : \Pi a\,([a] \to \mathit{Int})$, so `length` is a function that works for every element type at once.

Without a way to say *for every* you'd have three options.

1. The first is one copy per element type,

   `lengthOfIntList`, `lengthOfStringList`, `lengthOfBoolList`, ...

   which is a schema written out by hand, one instance per type.
2. The second is to give it the type `[Object] -> Int` and cast on the way in and out. Java made people do that until 2004, and a wrong cast is caught only when the program runs.
3. The third is to turn the checker off for that part of the program.

With the quantifier you write the code once and check it once, for every type, and that's why `map`, `filter`, `sort` and every container you've used have one in their types. The logic got there first: a generic function is a proof of a schema, written once.

Remark (looking back: schemas). Chapter 1 typed `K` by a schema, $A \to B \to A$, and recorded in square brackets after (K) the types chosen for A and B at each use. The brackets are instantiations: `K` is the program

$\Lambda a.\,\Lambda b.\,\lambda h.\,\lambda k.\,h : \Pi a\,\Pi b\,(a \to b \to a)$, and the typing "(K) [*Int*, *Int*]" is the term $\mathsf{K}\,[\mathit{Int}]\,[\mathit{Int}]$. Java's `static <A, B>` declares the two type variables in the same place.

The side condition on (∀I), the one whose absence runs to ⊥, turns up as an error message. Try

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

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

Here's what happened in the stack. The signature declared a and b and promised a program that works for both separately. The body can only be typed if a and b are the same, which means a hypothesis about a is still in force when the checker reaches the (ΠI) that would pop it. *Rigid* is the compiler's word for *declared and not last*, a variable the context has fixed and you may not generalise over.

A Σ over types is an *existential type*, and it is how a typed language writes a module: a hidden type with operations on it. The rule that opens one, (ΣE), carries an escape condition: the hidden type may be used inside the match and may not leave it. A module boundary is a boundary because of that one side condition on one rule, not because of a convention anybody has to remember.[2]

Example 5.7 (a counter)**.** Suppose you want a counter: a starting state, a way to step it, and a way to read off a number, with the state type left to the implementation.

```
{-# LANGUAGE ExistentialQuantification #-}
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`, though one carries an `Int` and the other a list, and `tick3` returns 3 for each without being told which. The type `Counter` is $\Sigma a\,(a \times (a \to a) \times (a \to \mathit{Int}))$, *there is* a type with these three operations. `MkCounter` packs a witness, the state type, as (ΣI) packed a thing, and the pattern match in `tick3` opens the pair, as (ΣE) did; its result type `Int` doesn't mention the state type.

A function that tried to return the counter's *state* would have result type a, and GHC says that the rigid type variable `a` *would escape its scope*: the condition written on (ΣE), in the compiler's words.

A `Counter` is Ivanova's kernel handle from Chapter 4, written as a type: a hidden state, three operations on it, and a client that may run them and may not see the state. Had the camera driver's record been one of these,

[2] Mitchell and Plotkin (1988) were the first to put it this way; the title of their paper is "Abstract types have existential type". Girard found the polymorphic λ-calculus in 1971 and called it System F (Girard, 1972); Reynolds found it independently in 1974 (Reynolds, 1974). Its quantifier is the one behind generics in most typed languages.

the escape condition would have stopped the microphone's bytes being read as a camera's.[3]

The generic quantifier, side conditions and all, has been at work every time the compiler has read a definition with a type variable in it. It is second-order logic with its programs on, and the next section writes the calculus down in full.

5.4. **The polymorphic λ-calculus.** Here is the calculus the section on generics used informally.

Definition 5.8 (polymorphic λ-calculus)**.** Fix a stock of *type variables* $a, b, c, \ldots$. The *polymorphic types* and the *terms* are

$$A, B \;::=\; a \;\mid\; A \to B \;\mid\; \Pi a\, A,$$
$$M, N \;::=\; h \;\mid\; M\, N \;\mid\; \lambda h.\, M \;\mid\; \Lambda a.\, M \;\mid\; M\,[B],$$

where h is an evidence variable. In $\Pi a\, A$ and $\Lambda a.\, M$ the type variable a is bound, and $A[B/a]$ substitutes the type B for the free occurrences of a in A, by recursion on the shape of A:

$$A[B/a] \coloneqq \begin{cases} B & \text{if } A \text{ is } a, \\ A & \text{if } A \text{ is a type variable } b \neq a, \\ A_1[B/a] \to A_2[B/a] & \text{if } A \text{ is } A_1 \to A_2, \\ \Pi a\, A_1 & \text{if } A \text{ is } \Pi a\, A_1, \\ \Pi b\, A_1[B/a] & \text{if } A \text{ is } \Pi b\, A_1,\ b \neq a,\ b \notin \mathrm{FV}(B). \end{cases}$$

A *context* is a finite list of declarations of type variables a and of typings $h : A$, in which:

- no variable is declared twice;
- every type variable free in a type in the context is declared earlier than the typing that mentions it.

The judgement $\Gamma \vdash_{\mathrm{F}} M : A$, with F for System F, is defined by the rules of Figure 5.1, and β-reduction has one rule for each kind of abstraction,

$$\frac{}{(\lambda h.\, P)\, Q \;\rightsquigarrow\; P[Q/h]}\ \beta \qquad\qquad \frac{}{(\Lambda a.\, P)\,[B] \;\rightsquigarrow\; P[B/a]}\ \beta$$

together with a rule for each place a subterm can sit, as (app_1), (app_2) and (abs), and two more of the same shape for the inside of $\Lambda a.\, M$ and of $M\,[B]$.

The rules for Π are those of dependent types with the domain changed from things to types, and the context is the same stack. There are no pairs, no unions and no $\bot$, for the reason the logic has already given: they can be defined.

Curry–Howard goes through as before: erasure and decoration undo each other, so a formula without letters is derivable from no hypotheses exactly when a closed program of the corresponding type exists. (The letters are the logic's constants, and the calculus has no counterpart for them, just as it has no types like `Int` built in.)

[3]This is the argument for writing operating-system drivers in a language that enforces the condition. Rust, which does, was accepted into the Linux kernel for exactly this kind of boundary from version 6.1 in 2022.

$$\frac{}{\Gamma \vdash_{\mathrm{F}} h : A}\ \text{var} \quad (\text{if } h : A \text{ is in } \Gamma) \qquad \frac{\Gamma \vdash_{\mathrm{F}} M : A \to B \qquad \Gamma \vdash_{\mathrm{F}} N : A}{\Gamma \vdash_{\mathrm{F}} M\,N : B}\ \text{app}$$

$$\frac{\Gamma,\ h : A \vdash_{\mathrm{F}} M : B}{\Gamma \vdash_{\mathrm{F}} \lambda h.\, M : A \to B}\ \text{abs}$$

. . . . . .

$$\frac{\Gamma,\ a \vdash_{\mathrm{F}} M : A}{\Gamma \vdash_{\mathrm{F}} \Lambda a.\, M : \Pi a\, A}\ \Pi\text{I} \qquad \frac{\Gamma \vdash_{\mathrm{F}} M : \Pi a\, A}{\Gamma \vdash_{\mathrm{F}} M\,[B] : A[B/a]}\ \Pi\text{E}$$

(if B's type variables are declared in Γ)

FIGURE 5.1. The typing rules of the polymorphic λ-calculus.

Data types, defined. Decorate the derivations in the proof of the proposition on the connectives, and the definitions of $\wedge$, $\vee$ and $\bot$ become programs, with a for X and C for the result type:

$$\begin{aligned}
(M, N) &\coloneqq \Lambda a.\, \lambda k.\, k\,M\,N && (\wedge\text{I}), \text{ with } k : A \to B \to a,\\
\mathit{fst}\,P &\coloneqq P\,[A]\,(\lambda h.\, \lambda k.\, h) && (\wedge\text{E}_1),\\
\mathit{snd}\,P &\coloneqq P\,[B]\,(\lambda h.\, \lambda k.\, k) && (\wedge\text{E}_2),\\
\mathit{Left}\,M &\coloneqq \Lambda a.\, \lambda f.\, \lambda g.\, f\,M && (\vee\text{I}_1);\ \mathit{Right}\,M \text{ has } g\,M,\\
\textbf{case}\ P\ \textbf{of}\ \{\ldots\} &\coloneqq P\,[C]\,(\lambda h.\, Q)\,(\lambda k.\, R) && (\vee\text{E}),\\
\mathit{absurd}\,P &\coloneqq P\,[C] && (\bot\text{E}).
\end{aligned}$$

In the line for ($\vee$E) the braces hold the usual two branches, $\mathit{Left}\,h \Rightarrow Q$ and $\mathit{Right}\,k \Rightarrow R$. Each branch becomes a function of its branch variable, $\lambda h.\, Q$ and $\lambda k.\, R$, and the union itself chooses which of the two to call.

They also run as they should:

$$\begin{aligned}
\mathit{fst}\,(M, N) &= (\Lambda a.\, \lambda k.\, k\,M\,N)\,[A]\,(\lambda h.\, \lambda k.\, h)\\
&\rightsquigarrow (\lambda k.\, k\,M\,N)\,(\lambda h.\, \lambda k.\, h)\\
&\rightsquigarrow (\lambda h.\, \lambda k.\, h)\,M\,N\\
&\rightsquigarrow^{*} M,
\end{aligned}$$

and a **case** on $\mathit{Left}\,M$ runs to $Q[M/h]$ in the same way. So pairs, unions and the empty type come free with functions and Π.

The same trick defines the data types you know. The booleans and the natural numbers are

$$\begin{aligned}
\mathit{Bool} &\coloneqq \Pi a\,(a \to a \to a),\\
\mathit{Nat} &\coloneqq \Pi a\,\big((a \to a) \to a \to a\big),
\end{aligned}$$

with

$$\begin{aligned}
\mathit{true} &\coloneqq \Lambda a.\, \lambda h.\, \lambda k.\, h,\\
\mathit{false} &\coloneqq \Lambda a.\, \lambda h.\, \lambda k.\, k,\\
\overline{n} &\coloneqq \Lambda a.\, \lambda f.\, \lambda h.\, f\,(\cdots(f\,h)) \qquad \text{the numeral, with } n \text{ copies of } f.
\end{aligned}$$

Addition is

$$add \;\coloneqq\; \lambda m.\, \lambda n.\, \Lambda a.\, \lambda f.\, \lambda h.\;\; m\,[a]\,f\,(n\,[a]\,f\,h) \;:\; Nat \to Nat \to Nat :$$

do n steps, then m more.

Remark (looking back: booleans and numerals)*.* These are the booleans and numerals of Chapter 1, where

- the boolean *true* was K,
- the boolean *false* was KI,
- the numeral $\overline{n}$, given f and x, applied f to x n times.

That chapter never gave the numerals a type. In the simply typed calculus each of them has the type $(A \to A) \to A \to A$ for one A chosen in advance, and then exponentiation, which uses one numeral at the type of another, has no type that takes two numerals of the same type to a third. The polymorphic type lets every use choose its own:

$$exp \;\coloneqq\; \lambda m.\, \lambda n.\, \Lambda a.\; n\,[a \to a]\,(m\,[a]) \;:\; Nat \to Nat \to Nat$$

computes m^n.

And the two projections that define $\wedge$, $\lambda h.\, \lambda k.\, h$ and $\lambda h.\, \lambda k.\, k$, are *true* and *false* again.

Existential types, defined. In the polymorphic calculus Σ over types is defined like $\vee$, by what you can do with it:

$$\Sigma a\, A \;\coloneqq\; \Pi b\,\big(\Pi a\,(A \to b) \to b\big),$$

"whatever follows from A, whatever a is, follows".

To open a package at a result type C, instantiate b at C and hand the package $\Lambda a.\, \lambda h.\, N$, where $N : C$ uses the hidden type a and the operations h. That substitution puts C under the Πa, so it is legal only when C doesn't mention a, which is the escape condition once more. (In the counter of Example 5.7, C is `Int`.)

5.5. **Normalisation and parametricity.** Now the question that impredicativity left open: is second-order logic consistent? Every program of the calculus is a decorated second-order derivation, impredicative instantiations and all, so if the programs all stop, the old argument goes through.

Theorem 5.9 (normalisation for the polymorphic λ-calculus; on loan)**.** *Every typed term of the polymorphic λ-calculus runs to a normal form.*

In words: every program of the calculus stops, whatever it does with its types. The theorem is on loan because its proof is much harder than the normalisation proofs it extends.[4] The shape of closed normal forms is as before, and so is the conclusion: there is no closed term of type $\Pi a\, a$, and second-order logic doesn't prove $\bot$. Impredicativity is safe here.

The same argument says more.

[4]Girard proved it in 1971, by extending Tait's method of computable terms. The method is *reducibility candidates*: a property of terms is defined for every type, and for $\Pi a\, A$ the definition quantifies over all candidate properties, impredicatively, as the type does. The proof is in Girard et al. (1989, chapter 14).

1. A closed normal term of type $\Pi a\,(a \to a)$ is $\Lambda a.\,\lambda h.\,M$, with $M : a$ in the context $a, h : a$.
2. The type a has no introduction form, so M is neutral (a variable with eliminations applied to it), with head h.
3. Nothing can be eliminated from $h : a$, so M is h.

The only program of type $\Pi a\,(a \to a)$ is the identity. The type says what the program does, without your having to read the program: reading a program off its type, from the outside in, is no longer a rule of thumb but a theorem.

This is *parametricity*, and its slogan is *theorems for free*.[5] The same reasoning one type over shows that every closed program of type $\Pi a\,(a \to a \to a)$ runs to $\Lambda a.\,\lambda h.\,\lambda k.\,h$ or to $\Lambda a.\,\lambda h.\,\lambda k.\,k$, and to nothing else (Exercise 5.3).

Example 5.10 (Okafor: the helper `pick`)**.** Okafor reviews code for the bank of the opening, and the helper on her ticket has the type `a -> a -> a`, which is $\Pi a\,(a \to a \to a)$. So, whatever its four hundred lines do, `pick` runs to *true* or to *false*: it hands back the first record or the second, untouched. She approves the change after reading the type, and one test tells her which record it keeps.

The language is also strong. Every function it can type from *Nat* to *Nat* stops on every input, so it has no *fix*; but the functions it can type are exactly those that second-order arithmetic can prove total, which is far more than any programmer has needed.[6]

There is a limit. It is the old story of a collection that contains itself. In the polymorphic λ-calculus a type quantifies over types, but the collection it ranges over is not itself a type, so no program can be handed the collection of all types as a value.

The first type theory to take that step, in 1971, had a type Type of all types, with $\mathsf{Type} : \mathsf{Type}$, and it proves everything. There is a closed term of every type, and it never stops, built, like the Monster and Russell's paradox before it, from a collection that contains itself.[7] The repair, in the type theories that followed, is a hierarchy of universes, $\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \cdots$, none of them a member of itself.

History. System F was found twice, like the quantifier and like natural deduction. Girard found it from the logic's side in 1971, as the programs of second-order logic, and used it to study second-order arithmetic. Reynolds found it from the programmer's side in 1974, as the language a generic function is written in, and later gave the reason types like that of `pick` say so much. Neither knew of the other.

The fear behind it is older. Russell banned impredicative definitions in 1908 after the paradoxes, and *Principia* built its types to keep them out; Church's simple theory of types of 1940 is that idea made small enough to use. Girard's normalisation theorem showed that the ban was more than

[5] Reynolds named it in 1983, and the slogan is Wadler's (Wadler, 1989).

[6] Girard's *representation theorem* (Girard, 1972); Girard et al. (1989, chapter 15). Ackermann's function, which no program built from bounded loops computes, is easy to write in System F.

[7] The theory is Martin-Löf's; the term is *Girard's paradox* (Girard, 1972), a typed cousin of Russell's.

Who	When	What
Russell	1908	impredicative definitions banned; the ramified theory of types
Church	1940	higher-order logic: the simple theory of types
Girard	1971	System F; second-order logic decorated; normalisation
Martin-Löf	1971	a type of all types, $\mathsf{Type} : \mathsf{Type}$
Girard	1972	$\mathsf{Type} : \mathsf{Type}$ proves everything
Reynolds	1974	the polymorphic λ-calculus, independently
Reynolds	1983	parametricity
Wadler	1989	parametricity as theorems for free

FIGURE 5.2. Second-order logic and polymorphism: who did what.

this logic needs, and his paradox of 1972 showed where the real limit lies. Figure 5.2 collects the dates.

Summary. A quantifier can range over propositions as well as things. Then a schema, a claim about every formula made one instance at a time, becomes a single theorem such as $\forall X\,(X \to X)$, and $\wedge$, $\vee$ and $\bot$ needn't be primitive: each is defined by what you can do with it, from $\to$ and $\forall$ alone.

Decorated, the derivations of this second-order logic are the programs of the polymorphic λ-calculus. A ($\forall$I) bar is a type abstraction and a ($\forall$E) bar an instantiation, and pairs, unions, booleans and numerals are all definable.

The quantifier is impredicative, since it ranges over a collection that contains the propositions it helps to form, so consistency needed a new argument. Girard's normalisation theorem supplies it, and with it parametricity: a type such as $\Pi a\,(a \to a \to a)$ says what its programs do without anyone reading them. Go one step further, to a type of all types, and the system proves everything.

That is the end of Part 1, and three questions to ask of every proof system.

1. Could it prove something false? We've shown that these logics don't prove *everything*, which is all that "false" can mean until the word has a meaning.
2. Does it prove everything it should? We can't even ask.
3. Can the *search* for a proof be made mechanical, an algorithm rather than a genius? Not yet.

And a puzzle to take into next week. Peirce's law,

$$((p \to q) \to p) \to p,$$

is a formula built from letters and arrows like any other. With type variables in place of the letters it is the type $\Pi a\,\Pi b\,(((a \to b) \to a) \to a)$, in Haskell `((a -> b) -> a) -> a`. Try to write a program of that type, in Haskell if you like, without recursion and without `undefined`. You won't manage,

and nothing so far can tell you whether that's your fault or the formula's. Part 2 begins by giving the formulas a *meaning*, and the first thing it finds out is what Peirce's law is.

Exercises.

Exercise 5.1.

(a) Write the types of `length` and `map` with their quantifiers explicit.

(b) Type the use `map show [1, 2]`, naming the instantiation at each quantifier.

(c) Declare `f :: a -> b` and define `f x = x`: show the exact judgement at which the checker is asked to generalise over a variable that is not on top of the stack, and say what GHC means by a *rigid* type variable.

Exercise 5.2. With $A \wedge B$ and $A \vee B$ defined in second-order logic, type the programs (M, N), *fst* and **case** that decorate the proof of the proposition defining the connectives, and run

$$\mathbf{case}\ (\mathit{Right}\, M)\ \mathbf{of}\ \{\, \mathit{Left}\, h \Rightarrow Q \mid \mathit{Right}\, k \Rightarrow R \,\}$$

to normal form.

Exercise 5.3. Show that the closed normal terms of type $\Pi a\,(a \to a \to a)$ are exactly *true* and *false*. What are the closed normal terms of type $\Pi a\, a$, and what does that say about $\bot$?

Exercise 5.4.

(a) Type *add* and *exp* on *Nat*.

(b) Give a program $\mathit{mult} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$.

(c) Run $\mathit{add}\,\overline{2}\,\overline{1}\,[a]\,f\,h$ to normal form.

References.

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

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat d'état, Université Paris VII, 1972.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. Free online.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Paris, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359. ACM, 1989.