

COMP0009 LOGIC

Lecture notes · 2026–27

Part 1: Type Theory

Chapter 4

QUANTIFIERS

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

4. Quantifiers	1
4.1. Predicates and quantifiers	2
4.2. Dependent types	7
4.3. First-order natural deduction	13
4.4. Side conditions	17
4.5. Generics	19
4.6. Specifications	21
4.7. Second-order logic and System F (optional)	24
4.8. History	29
Exercises	31
Reading	32

4. QUANTIFIERS

Six months on, Hughes’s release bot is still running the same code. On a Monday morning Hughes finds that version 2.4 of the library went out over the weekend at the request of a contributor called Reyes, who has no access of any kind. The log says what happened. None of it is a bug in the usual sense. Remember the arrangement: a request to publish carries its own ticket, a value of type `MayPublish`, and all the bot does is check the ticket’s type before it acts. I praised this as how access control ought to be done, and I still think so. The declarations were

```
smith           :: Maintainer
maintainersWrite :: Maintainer -> WriteAccess
writeMayPublish :: WriteAccess -> MayPublish
```

so the only way to make a `MayPublish` was to apply the two policy functions to evidence that somebody is a maintainer. The ticket for Smith’s release of 2.3 was

```
writeMayPublish (maintainersWrite smith)
```

and the bot’s job log, which is public, printed it. Reyes copied the ticket out of the log, attached it to a request of his own, and sent it. The bot checked the ticket. It was a `MayPublish`. It published.

What did the type say? `MayPublish` is a claim about *somebody*: they may publish. The evidence inside the ticket was about Smith, and it was good evidence. But the type the bot checked doesn’t say *whom* it’s about, so to the checker a ticket made for Smith is just a ticket. The checker did refuse Jones, when he wanted to rotate the signing key, because the only evidence in scope for him was a `WriteAccess` and no function in the policy turns that into a `Maintainer`. Reyes has no evidence in scope at all. He doesn’t need any. Hand him somebody else’s `MayPublish` and the compiler has nothing to say. Security people have a name for this, a *confused deputy*.¹ The bot has authority, the credential it was given is genuine, and it used that credential on behalf of the wrong person, because nothing in the credential says whose it is.

Hughes’s first thought is the right one, and I want to watch it fail. The propositional letters stood for whole claims: p was “Smith is a maintainer”, r was “Smith may publish”. If the trouble is that r doesn’t say *Smith*, write a letter that does. Let p_{smith} be “Smith is a maintainer”, q_{smith} “Smith has write access” and r_{smith} “Smith may publish”. Do the same for Brown and everyone else, and write the policy once per person:

$$\begin{array}{lll} p_{\text{smith}} \rightarrow q_{\text{smith}}, & q_{\text{smith}} \rightarrow r_{\text{smith}}, \\ p_{\text{brown}} \rightarrow q_{\text{brown}}, & q_{\text{brown}} \rightarrow r_{\text{brown}}, & \dots \end{array}$$

¹The name is Norm Hardy’s, from a one-page paper of 1988 (Hardy, 1988). The deputy in his story was a compiler on a time-sharing system, which had permission to write into the system directory so that it could keep usage statistics there, and which also wrote a debugging listing to whatever file the user named. A user named a file in the system directory, the billing file, and the compiler, acting with its own authority on the user’s behalf, overwrote it. Hardy’s diagnosis was the one above: the compiler had been handed a name, when it should have been handed a credential that said what it was for.

Now a ticket proves r_{smith} , which isn't r_{brown} , and Reyes's replay fails. So far so good. But look at what she's had to write. MAINTAINERS.md lists thirty-seven people, so the policy is now a hundred and eleven letters and seventy-four implications, every one of them a copy of the same two sentences with a different subscript. The same thing written thirty-seven times is wrong in one of them. Worse, the file changes. Somebody joins on Tuesday. The policy now has seventy-six implications, and the proof Hughes checked on Monday was checked against the wrong one. And there's a third problem, the one the chapter is about. The clerk who checks a proof by comparing symbols can't see the subscript. To him p_{smith} and r_{smith} are two letters, as unrelated as p and r were. Nothing in his list of shapes says that the "smith" in the one is the "smith" in the other, and $p_{\text{smith}} \rightarrow r_{\text{brown}}$ is as well formed as any formula on the page. Hughes has written the name inside the letter, and inside the letter is where the clerk can't look.

What MAINTAINERS.md actually says is one sentence: *every maintainer has write access*. Every maintainer, including the one who joins on Tuesday: nothing so far can say that. It keeps coming up. Baker's runbook said what happens on *every* payment, and she wrote it for the one on her ticket. A sorting routine promises that *every* element of its output is at most the next. A sale gives its discount on *every* day from the start to the end. A bug report said "discount not applied on the last day of the sale"; a patch changed $\leq$ to $<$ at both ends; the tests passed; and the discount quietly disappeared from the first day. None of these is a finite list, and the one we tried to write as a finite list has just failed in three ways.

So: how do you say *every* and *some*, in a type and in a proof, and what does saying it cost? The language is Frege's, and in the types the quantifiers turn out to be two things you've programmed with for years. Rubbing out the programs, as before, leaves the proof rules, and by then you'll know the trick well enough to guess them before I give them; I'd like you to try. Pointed at types instead of people, the same quantifier gives generics. The chapter ends where Hughes began, with sentences about things, her policy among them, written down as specifications and proved from. A last section, which you may skip, runs the story the other way round, from a logic to its programs.

4.1. Predicates and quantifiers. The subscript was the right instinct. A letter with a name inside it is two things, a claim with a gap in it and a name, and Hughes's only mistake was to glue them together where the clerk couldn't see the join. Pull them apart:

p_{smith} becomes $\text{Maint}(\text{smith})$,

"Smith is a maintainer", where Maint is a *predicate*, a claim with a hole in it, and smith is a name that fills the hole. Now $\text{Maint}(\text{smith})$ and $\text{Pub}(\text{smith})$ visibly share a part, and the clerk, who compares symbols, can see that they do. The sentence $\text{Maint}(\text{smith}) \rightarrow \text{Pub}(\text{reyes})$ is still well formed, but it's now visibly a different sentence from $\text{Maint}(\text{smith}) \rightarrow \text{Pub}(\text{smith})$, and a policy that said the first would be seen to say it.

This is Frege's move, and it's the one his notation of 1879 was built around. Logic before him analysed a sentence into a subject and a predicate, "Smith"

and “is a maintainer”, which is the grammar of Latin; Frege analysed it instead into a *function and an argument*, $\text{Maint}(\)$ and smith , which is the grammar of mathematics. He thought it the most important thing in the book.² It matters here because a function can take a *variable* as its argument. The formula $\text{Maint}(x) \rightarrow \text{Write}(x)$ is the policy sentence with the name left out, a claim about whoever x is, and it’s one formula, not thirty-seven. On the type side the same move turns **Maintainer** from a type into a function from people to types, **Maintainer smith**, **Maintainer x**, a type with a hole for a name.

Universal and existential quantifiers. A formula with a hole in it isn’t a claim yet, any more than $x + 1$ is a number. There are two ways to close the hole, and they’re the two things Hughes was trying to write. The first is to say that the claim holds *whoever* fills it. For the thirty-seven that would be the conjunction

$$(\text{Maint}(\text{smith}) \rightarrow \text{Write}(\text{smith})) \wedge (\text{Maint}(\text{brown}) \rightarrow \text{Write}(\text{brown})) \wedge \dots$$

with one conjunct per person, except that the list won’t be complete on Tuesday and the clerk, as we saw, can’t tell that the conjuncts are alike. So write the conjunct once, hole and all, with a mark in front that says “for every way of filling it”:

$$\forall x (\text{Maint}(x) \rightarrow \text{Write}(x)).$$

That’s *for all*, and it’s an *and* with one case for every thing there is, written once with a parameter. The second way to close the hole is to say that *something* fills it: the review rule, “some maintainer approved this pull request”, is the disjunction

$$(\text{Maint}(\text{smith}) \wedge \text{Appr}(\text{smith}, \text{pr})) \vee (\text{Maint}(\text{brown}) \wedge \text{Appr}(\text{brown}, \text{pr})) \vee \dots$$

with one disjunct per person, and we write it once:

$$\exists x (\text{Maint}(x) \wedge \text{Appr}(x, \text{pr})).$$

That’s *there is*, an *or* with one case for every thing, and the thing is the tag. Logicians call it the *witness*. The quantifiers are $\wedge$ and $\vee$ with the list of cases made unbounded, and I’ll say so twice more before the chapter is out. Everything else about them (the programs, the proof rules, the side conditions) is the bill for that.

You’ve met both ideas as a programmer. A test suite checks finitely many inputs: it is the long *and* with its cases written out, Hughes’s thirty-seven letters again, and it says nothing about input thirty-eight. A function says something about every input, because its body is written once, for a parameter, and has to work for whatever arrives: that is *for all*, in code. And a single failing test settles “some input breaks this” by exhibiting the input: a witness.

²“I believe that the replacement of the concepts *subject* and *predicate* by *argument* and *function*, respectively, will stand the test of time” (Frege, 1879, preface), in the translation in van Heijenoort. It has. Frege’s own symbol for *every* was a dip in the horizontal line of a formula with a Gothic letter sitting in it. His formulas were two-dimensional, which is one reason almost nobody read the book. The notation lost; the analysis won.

Frege's language, in the notation everyone now uses, comes in three layers: the symbols; the *terms* built from them, which name things; and the *formulas*, which make claims.

Definition 4.1 (first-order language). The symbols of a first-order language are of two kinds.

1. The *logical symbols*, which are the same in every language: the connectives $\rightarrow$, $\wedge$ and $\vee$; the constant $\perp$; the quantifiers $\forall$ and $\exists$; and a stock of *variables* $x, y, z, \dots$
2. The *non-logical symbols*, which make up the *signature* and change from one language to the next: some *constant symbols* $c, d, \dots$; some *function symbols* $f, g, \dots$; and some *predicate symbols* $P, Q, \dots$. Each function symbol and each predicate symbol has an *arity*, the number of arguments it takes.

A predicate symbol of arity 0 is a *propositional letter*, and is written $p, q, r, \dots$.

Definition 4.2 (terms). The *terms* are defined inductively.

1. Every variable is a term.
2. Every constant symbol is a term.
3. If f is a function symbol of arity n and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.
4. Nothing else is a term.

A term with no variables in it is *closed*.

Definition 4.3 (formulas). The *formulas* are defined inductively.

1. $\perp$ is a formula.
2. If P is a predicate symbol of arity n and $t_1, \dots, t_n$ are terms, then $P(t_1, \dots, t_n)$ is a formula. (When $n = 0$ it is a propositional letter p , written without brackets.)
3. If A and B are formulas, then $(A \rightarrow B)$, $(A \wedge B)$ and $(A \vee B)$ are formulas.
4. If A is a formula and x is a variable, then $\forall x A$ and $\exists x A$ are formulas.
5. Nothing else is a formula.

The formulas given by clauses 1 and 2 are *atomic*.

Three conventions make formulas readable. First, $\neg A$ abbreviates $A \rightarrow \perp$. Second, familiar symbols such as $<$ and $+$ are written between their arguments, so that $x + 1 < y$ is the atomic formula $< (+ (x, 1), y)$. Third, brackets are left out where nothing is lost: the outermost pair goes; $\wedge$ and $\vee$ bind more tightly than $\rightarrow$; the arrow groups to the right; and $\neg$ and the quantifiers bind most tightly of all. So $\forall x A \rightarrow B$ is $(\forall x A) \rightarrow B$, and a quantifier whose scope is meant to be larger is followed by brackets, as in $\forall x (A \rightarrow B)$. With the conventions in force, the two definitions fit on two lines:

$$\begin{aligned} t &::= x \mid c \mid f(t_1, \dots, t_n), \\ A, B &::= \perp \mid P(t_1, \dots, t_n) \mid A \rightarrow B \mid A \wedge B \\ &\quad \mid A \vee B \mid \forall x A \mid \exists x A. \end{aligned}$$

Remark (looking back: letters). Nothing in the definition replaces anything. A letter of the last two chapters is a predicate symbol of arity 0, so every formula of those chapters is still a formula, and, once derivations have been defined, every derivation will still be a derivation.

A term names a thing and a formula makes a claim. Keep the two apart, as a parser would by giving them different types. In $x + 1 < y$ the symbols 1 and $+$ are a constant and a function symbol, $<$ is a predicate symbol, and x and y are variables; $x + 1$ is a term and $x + 1 < y$ a formula. Writing either where the other belongs would be a type error.

Hughes’s signature, which the chapter uses throughout, has constants for the people and the pull requests, **smith**, **reyes**, **brown**, **pr**; a function symbol **author**(p), the person who opened pull request p ; and five predicates:

Predicate	Reads
Maint (x)	x is a maintainer
Write (x)	x has write access
Pub (x)	x may publish a release
Appr (x, p)	x approved pull request p
Rel (p)	pull request p may be released

People and pull requests are things of one kind as far as the language goes. A variable ranges over both; the predicates say which is which. So **Maint**(**pr**) is a formula, though not one anybody would assert. (Logics with several sorts of thing, each with its own variables, avoid that at the cost of more bookkeeping.) With these, MAINTAINERS.md is three sentences, which I’ll call Γ_{pol} :

What the file says	As a sentence
every maintainer has write access	$\forall x (\text{Maint}(x) \rightarrow \text{Write}(x))$
anyone with write access may publish	$\forall x (\text{Write}(x) \rightarrow \text{Pub}(x))$
a maintainer’s approval releases a pull request	$\forall x \forall p (\text{Maint}(x) \wedge \text{Appr}(x, p) \rightarrow \text{Rel}(p))$

The merge rule as the file states it, “a pull request needs a maintainer’s approval, and its author needs write access”, is $\exists x (\text{Maint}(x) \wedge \text{Appr}(x, p)) \wedge \text{Write}(\text{author}(p))$. The term **author**(p) names a person without saying who; that is what a function symbol is for. The sentence is the kind a security team writes in English and would rather a machine checked.

Reading quantified sentences. The order of the quantifiers matters, and so does the connective under them. $\forall x \exists p \text{Appr}(x, p)$ says that everybody approved some pull request, perhaps each a different one; $\exists p \forall x \text{Appr}(x, p)$ says that one pull request was approved by everybody. The symbols are the same and the claims are very different. The file’s first sentence is $\forall x (\text{Maint}(x) \rightarrow \text{Write}(x))$. Written with $\wedge$, as $\forall x (\text{Maint}(x) \wedge \text{Write}(x))$, it would say that *everyone* is a maintainer with write access, which no file says. Under $\exists$ the trap runs the other way. *Some maintainer approved* is $\exists x (\text{Maint}(x) \wedge \text{Appr}(x, \text{pr}))$, with $\wedge$; written with $\rightarrow$, as $\exists x (\text{Maint}(x) \rightarrow \text{Appr}(x, \text{pr}))$, it could be established by anybody known not to be a maintainer, whether or not they approved anything.

Definition 4.4 (free variables). In $\forall x A$ and $\exists x A$ the formula A is the *scope* of the quantifier. An occurrence of a variable x in a formula is *bound* if it is the variable of a quantifier $\forall x$ or $\exists x$, or lies in the scope of one, and *free* otherwise. The set $\text{FV}(A)$ of *free variables* of a formula A is defined by recursion on A .

1. $\text{FV}(\perp) = \emptyset$; $\text{FV}(P(t_1, \dots, t_n))$ is the set of variables that occur in $t_1, \dots, t_n$.
2. $\text{FV}(A \rightarrow B) = \text{FV}(A \wedge B) = \text{FV}(A \vee B) = \text{FV}(A) \cup \text{FV}(B)$.
3. $\text{FV}(\forall x A) = \text{FV}(\exists x A) = \text{FV}(A) \setminus \{x\}$.

A formula with no free variables is a *sentence*.

In $\forall x \exists y (x < y)$, “every number has a larger one”, every occurrence of a variable is bound, and the formula is a sentence. Free and bound occurrences can also mix, even of one variable. In $\forall x (x < y) \rightarrow \exists y (x < y)$ the first two occurrences of x are bound and the third is free, and the first y is free and the other two are bound. So this is a formula and not a sentence. Everything in Γ_{pol} is a sentence, which is what makes it a policy rather than a remark about somebody. The merge rule is not: its p is free, and it becomes a claim about a particular pull request only when a term is put in for p . That operation needs care.

Definition 4.5 (substitution). Let t be a term and x a variable.

1. For a term s , the term $s(t/x)$ is s with every occurrence of x replaced by t .
2. The term t is *substitutable* for x in a formula A if no free occurrence of x in A lies in the scope of a quantifier on a variable that occurs in t .
3. If t is substitutable for x in A , the formula $A(t/x)$ is defined by recursion on A , with $\circ$ standing for any of $\rightarrow$, $\wedge$ and $\vee$, and Q for either quantifier:
 - (a) $\perp(t/x) = \perp$, $P(t_1, \dots, t_n)(t/x) = P(t_1(t/x), \dots, t_n(t/x))$;
 - (b) $(A \circ B)(t/x) = A(t/x) \circ B(t/x)$;
 - (c) $(Qx A)(t/x) = Qx A$;
 - (d) $(Qy A)(t/x) = Qy A(t/x)$, for a variable y other than x .

So $A(t/x)$ puts t in for the *free* occurrences of x , and only those: clause (c) stops at a quantifier on x , because every x in its scope is bound. Substitutability is what makes clause (d) safe. If y occurred in t and x occurred free under the Qy , pushing t under the quantifier would turn that y into a bound one and change what t means. The term y is not substitutable for x in $\exists y (x < y)$, for this reason; the terms z and **smith** are. When a term isn’t substitutable, rename the bound variable first, to one that occurs nowhere else. Two formulas that differ only in the names of their bound variables are treated as the same formula, so after renaming every term is substitutable. Ignoring the condition produces a bug that the section on side conditions runs all the way to $\perp$.

Remark (looking back: substitution in the λ -calculus). The clauses are those for substitution in λ -terms in Chapter 3, with a quantifier in the place of a λ , and the third clause for FV is the one for an abstraction, $\text{FV}(\lambda x. M) = \text{FV}(M) \setminus \{x\}$. A quantifier binds its variable exactly as a λ

binds its parameter, and renaming before substituting is the same remedy for the same danger.

4.2. Dependent types. What type should `maintainersWrite` have? In the bot's old code it was an arrow from `Maintainer` to `WriteAccess`, and we've seen what that cost. The policy function has to do two things the old type couldn't say. It has to work for *every* person, Tuesday's included. And it has to say, in its result, *which* person it was given, so that the ticket it makes for Smith is a ticket for Smith. Frege's move gives the types. With `Maintainer x` and `WriteAccess x` as types with a hole in them, Hughes writes

```
smith          : Person
smithMaint     : Maintainer smith      -- MAINTAINERS.md
maintainersWrite : (x : Person) -> Maintainer x -> WriteAccess x
writeMayPublish : (x : Person) -> WriteAccess x -> MayPublish x
publish        : (x : Person) -> MayPublish x -> IO ()
```

Read the third line slowly, because it's the only new thing here. The function `maintainersWrite` takes a person x , then evidence that x is a maintainer, and returns evidence that x has write access. The result type mentions the argument. In the old code the name and the evidence were one value, `smith :: Maintainer`, and that was the bug. Here they're two: the person, and a separate value `smithMaint` that's about him, whose type has the person in it. So the type of the function is an arrow whose right-hand side depends on what went in on the left. None of the type systems so far can write that.³

On the program side, the long *and* is a record with one field per maintainer: thirty-seven tickets, one for each person. A function from maintainers to tickets holds the same information (hand it Smith and it returns Smith's field), and when you don't know how many fields there will be, a function is the only way to write the record down. So a proof of *for all* is a function from things. The long *or* is a thirty-seven-way **Either**: a value with a tag saying which case it is. Here the tags are people, so the value is a pair of a person and evidence about that person. So a proof of *there is* is a pair of a thing and evidence about it.

One change of notation comes with the types. Logic writes $\forall, \exists, \wedge$ and $\vee$; type theory writes $\Pi, \Sigma, \times$ and $+$ for the same four things. This section uses the type theorist's letters, the next goes back to the logician's, and the table at the end of the section lines them up.

Definition 4.6 (dependent types, terms and contexts). Fix a signature. Its terms t name things, and its variables $x, y, z, u, v, \dots$ stand for things. A second stock of variables $h, k, f, g, \dots$ stands for evidence. The *types* are the formulas over the signature, written as a type theorist writes them:

$$A, B, C ::= P(t_1, \dots, t_n) \mid \perp \mid A \rightarrow B \mid A \times B \mid A + B \\ \mid \Pi x A \mid \Sigma x A,$$

³The code is Haskell with one change: an argument that the type mentions is named on the arrow, as in `(x : Person) -> Maintainer x`, which is how languages whose types may mention values write such a type. Haskell doesn't accept the five lines by default, because its types are not allowed to mention values. That is a design decision, not a limitation of the idea.

where $A \times B$ is the pair type, $A + B$ the sum type, $\Pi x A$ the *dependent function type* and $\Sigma x A$ the *dependent pair type*, with x ranging over the one kind of thing the signature talks about; negation $\neg A$ is $A \rightarrow \perp$. The *terms* are

$$\begin{aligned} M, N, P, Q ::= & h \mid M N \mid \lambda h. M \mid (M, N) \mid \text{fst } M \mid \text{snd } M \\ & \mid \text{Left } M \mid \text{Right } M \mid \text{absurd } M \\ & \mid \text{case } M \text{ of } \text{Left } h \Rightarrow P \mid \text{Right } k \Rightarrow Q \\ & \mid \lambda x. M \mid M t \mid \langle t, M \rangle \mid \text{let } \langle x, h \rangle = M \text{ in } N, \end{aligned}$$

where h and k are evidence variables, x is a variable for things and t is a term of the signature. The first three lines are the propositional terms; the fourth has the four new forms, an abstraction over a thing, an application to a thing, a pair of a thing and evidence, and the match that opens such a pair. A *context* Γ is a finite list of entries of two kinds: a variable x on its own, which *declares* a thing, and a typing $h : A$, which declares evidence. No variable is declared twice, and every variable free in a type A in Γ is declared earlier in Γ than the entry $h : A$ that mentions it. A term t of the signature is *in scope* in Γ if every variable in t is declared in Γ .

A declared variable x is a thing nobody has chosen yet, a parameter, and the four new rules all turn on it. Here is the whole system. The rules for $\rightarrow$, $\times$, $+$ and $\perp$ are the propositional ones, except that a context may now contain declarations.

Definition 4.7 (typing rules). The judgement $\Gamma \vdash M : A$ is defined by the rules below. In every rule, each context written must be a context in the sense just defined; in (abs), for example, every variable free in A must be

declared in Γ .

$$\begin{array}{l}
(\text{var}) \frac{}{\Gamma \vdash h : A} \text{ if } h : A \text{ is in } \Gamma \quad (\text{abs}) \frac{\Gamma, h : A \vdash M : B}{\Gamma \vdash \lambda h. M : A \rightarrow B} \\
(\text{app}) \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \quad (\times I) \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \\
(\times E_1) \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{fst } M : A} \quad (\times E_2) \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{snd } M : B} \quad (\perp E) \frac{\Gamma \vdash M : \perp}{\Gamma \vdash \text{absurd } M : C} \\
(+I_1) \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{Left } M : A + B} \quad (+I_2) \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{Right } M : A + B} \\
(+E) \frac{\Gamma \vdash M : A + B \quad \Gamma, h : A \vdash P : C \quad \Gamma, k : B \vdash Q : C}{\Gamma \vdash \text{case } M \text{ of Left } h \Rightarrow P \mid \text{Right } k \Rightarrow Q : C} \\
(\text{III}) \frac{\Gamma, x \vdash M : A}{\Gamma \vdash \lambda x. M : \Pi x A} \\
(\text{IIE}) \frac{\Gamma \vdash M : \Pi x A}{\Gamma \vdash M t : A(t/x)} \text{ if } t \text{ is in scope in } \Gamma \\
(\Sigma I) \frac{\Gamma \vdash M : A(t/x)}{\Gamma \vdash \langle t, M \rangle : \Sigma x A} \text{ if } t \text{ is in scope in } \Gamma \\
(\Sigma E) \frac{\Gamma \vdash M : \Sigma x A \quad \Gamma, x, h : A \vdash N : C}{\Gamma \vdash \text{let } \langle x, h \rangle = M \text{ in } N : C} \text{ if } x \notin \text{FV}(C)
\end{array}$$

Read (IIE) first, because it's a function call. You have a program that works for every x and you call it on the t in hand. The type of the result is the body with t put in for x : substitution. And t has to be in scope, because a program can mention only declared names. (III) is what happens when a definition is finished. You typed the body with x declared and nothing said about it, so the program works for every x , and you may say so. Proved for an arbitrary x , hence for all. (ΣI) is the constructor: supply a thing and evidence about it, and out comes a value whose type mentions neither. (ΣE) is the pattern match. Open the pair, giving names x and h to the thing and the evidence, and go on to produce something. The something may use x and h , but its *type* may not mention x , because x was a name you made up for the duration of the match. Compare the rules for $\times$ and $+$, which are these four with the list of cases cut down to two: a pair with two fixed fields instead of a function from things, a tag from $\{\text{Left}, \text{Right}\}$ instead of a thing. That's why the letters are Π and Σ : a function from things is a product with one factor per thing, and a pair with a tag is a member of a sum with one summand per thing, just as $\sum_{r=1}^n a_r$ is a sum with one term for each r .

Side conditions and the context. Logic textbooks give the rule for *for all* a proviso: x must not occur free in any hypothesis. The proviso is needed;

without it the rules prove $\perp$, as an exercise at the end of the chapter shows. Yet (III) above carries no proviso, because the context already enforces it. In Γ, x the variable x is the *last* thing declared. By the definition of a context, an entry $h : A$ may mention only variables declared before it, so no $h : A$ in Γ can say anything about x . If you want to generalise over x while some hypothesis about x is still in force, the rule doesn't apply, because x isn't last. You have to discharge every hypothesis that mentions x first, with a λh each, and only then does x come to the top. A context is a stack, and you can only pop x when everything pushed after it has gone.⁴

The one condition the stack can't enforce is the one written on (ΣE). The type C of the result isn't in the context, so nothing stops it mentioning x unless the rule says so. That is why (ΣE) is the only rule with a freshness condition; the conditions on (ΠE) and (ΣI) only ask that t use names already declared. It is also the condition a *module system* enforces: the part of a language that lets a library hide how it is built.

Now some typings, as sequent-style trees, in which every node is a whole judgement with its context. Hughes's declarations become one context. The people are constants of the signature, which are always in scope, so they need no declaring, and what is left is the evidence: the three policy functions (**mw** and **wp** for the two above, and **rel** for the review rule), **publish**, and **sm** for **smithMaint**. With the types written in the predicates of Hughes's signature, that context is

$$\begin{aligned} \Gamma = & \text{mw} : \Pi x (\text{Maint}(x) \rightarrow \text{Write}(x)), \quad \text{wp} : \Pi x (\text{Write}(x) \rightarrow \text{Pub}(x)), \\ & \text{rel} : \Pi x \Pi p (\text{Maint}(x) \times \text{Appr}(x, p) \rightarrow \text{Rel}(p)), \\ & \text{publish} : \Pi x (\text{Pub}(x) \rightarrow \text{IO}()), \quad \text{sm} : \text{Maint}(\text{smith}). \end{aligned}$$

It stays the same in every tree below; whatever a rule adds to it is written out in full.

Example 4.8 (Smith's ticket). Smith's ticket is built in two stages. First the write-access ticket, from the first policy function and the file's evidence:

$$\frac{\frac{\Gamma \vdash \text{mw} : \Pi x (\text{Maint}(x) \rightarrow \text{Write}(x))}{\Gamma \vdash \text{mw smith} : \text{Maint}(\text{smith}) \rightarrow \text{Write}(\text{smith})} \text{var}}{\Gamma \vdash \text{mw smith sm} : \text{Write}(\text{smith})} \Pi E$$

and then the publishing ticket, from the second policy function and the tree just drawn, which I write as dots:

$$\frac{\frac{\Gamma \vdash \text{wp} : \Pi x (\text{Write}(x) \rightarrow \text{Pub}(x))}{\Gamma \vdash \text{wp smith} : \text{Write}(\text{smith}) \rightarrow \text{Pub}(\text{smith})} \text{var}}{\Gamma \vdash \text{wp smith (mw smith sm)} : \text{Pub}(\text{smith})} \Pi E$$

Each (ΠE) bar picks the person the policy sentence is being used for. The program at the root is the ticket, and its type now has Smith's name in it.

⁴Two entries may be swapped when neither depends on the other, so the stack is a stack only as far as the dependencies go. A hypothesis that doesn't mention x can always be moved below it, and generalising over x with such a hypothesis in force is fine, as it should be.

Example 4.9 (Reyes’s replay). Reyes’s request handler passes the ticket from the log to `publish reyes`,⁵ and the checker gets as far as the ticket:

$$\frac{\frac{\Gamma \vdash \text{publish} : \Pi x (\text{Pub}(x) \rightarrow IO())}{\Gamma \vdash \text{publish reyes} : \text{Pub}(\text{reyes}) \rightarrow IO()} \Pi E \quad \frac{\vdots}{\Gamma \vdash \text{ticket} : \text{Pub}(\text{smith})}}{(\text{app}) \text{ fails: expected Pub(reyes), got Pub(smith)}}$$

Here *ticket* is the term at the root of Smith’s tree, `wp smith (mw smith sm)`, copied from the log. The confused deputy is a type error, and the error says whose ticket it was. Nothing in the bot’s code had to be clever.

Remark (looking back: Jones). The tree for Smith’s ticket is the typing of the same ticket in Chapter 2, with the two (ΠE) bars added. And Reyes is refused for the reason Jones was refused there, when he asked to rotate the signing key: a missing piece of evidence. What has changed is that the type can now say whom the evidence is about.

Example 4.10 (every maintainer may publish). The policy composed and then generalised is the typing that answers Hughes’s original question: may every maintainer publish? In the context $\Gamma, x, h : \text{Maint}(x)$, with x declared and then a hypothesis about it, two (ΠE)s and two (app)s give `wp x (mw x h) : Pub(x)`, by the trees of Example 4.8 with x for `smith` and h for `sm`. Then two bars close it, and the order they come in is the side condition:

$$\frac{\frac{\Gamma, x, h : \text{Maint}(x) \vdash \text{wp } x (\text{mw } x h) : \text{Pub}(x)}{\Gamma, x \vdash \lambda h. \text{wp } x (\text{mw } x h) : \text{Maint}(x) \rightarrow \text{Pub}(x)} \text{abs}}{\Gamma \vdash \lambda x. \lambda h. \text{wp } x (\text{mw } x h) : \Pi x (\text{Maint}(x) \rightarrow \text{Pub}(x))} \text{III}$$

The (abs) bar pops h , the hypothesis about x . Then x is on top and (III) pops it. Try them the other way round and the (III) bar has nothing to apply to, because x isn’t the last entry. The program is function composition with a person threaded through it, and its type, read as a sentence, is *every maintainer may publish*, which isn’t in the policy but follows from it.

Example 4.11 (the review rule). Brown approves the pull request, and the bot records

$$\langle \text{brown}, (\text{bm}, \text{ba}) \rangle : \Sigma x (\text{Maint}(x) \times \text{Appr}(x, \text{pr})),$$

by (ΣI) from `bm : Maint(brown)` and `ba : Appr(brown, pr)`: the approver and the evidence, packed together, in a value whose type says only that *somebody* approved. To release, the bot needs `Rel(pr)`, and the third policy sentence `rel :  $\Pi x \Pi p (\text{Maint}(x) \times \text{Appr}(x, p) \rightarrow \text{Rel}(p))$` gives it, once the pair is opened. The bot holds the record as a , so its context is $\Gamma, a : \Sigma x A(x)$, where $A(x)$ is short for the type $\text{Maint}(x) \times \text{Appr}(x, \text{pr})$. Opening the pair adds two entries, a person y and the evidence k about them, and in that context the policy applies. Two (ΠE)s and an (app) give

$$\Gamma, a : \Sigma x A(x), y, k : A(y) \vdash \text{rel } y \text{ pr } k : \text{Rel}(\text{pr}),$$

which I write as dots in the tree that opens the pair:

$$\frac{\frac{\frac{\vdots}{\Gamma, a : \Sigma x A(x), y, k : A(y) \vdash \text{rel } y \text{ pr } k : \text{Rel}(\text{pr})} \vdots}{\Gamma, a : \Sigma x A(x) \vdash \text{let } \langle y, k \rangle = a \text{ in rel } y \text{ pr } k : \text{Rel}(\text{pr})} \Sigma E}{\Gamma, a : \Sigma x A(x) \vdash a : \Sigma x A(x)} \text{var}$$

⁵ $IO()$ is the type of an action that returns nothing of interest.

The right-hand branch calls the policy at y and $\mathbf{pr}$ and applies it to k , where y is a name for whoever it was. The side condition holds because $\mathbf{Rel}(\mathbf{pr})$ doesn't mention y . Suppose instead that the handler tried to conclude, from the approval record, that the approver is a maintainer:

let $\langle y, k \rangle = a$ **in** $\mathbf{fst} k : \mathbf{Maint}(y)$ — refused: y is free in $\mathbf{Maint}(y)$.

Inside the match, $\mathbf{fst} k : \mathbf{Maint}(y)$ is perfectly good. Outside it there's no y . It was a name the match made up, and a type that mentions it is a type that mentions nothing. "Some maintainer approved; call them y ; so y is a maintainer" isn't a step the rules allow, and the rule says why in four words.

Reduction. Running a program means rewriting an introduction met by its own elimination, and the quantifiers add two such steps, with the shapes of β and of a **case** on a tag.

Definition 4.12 (reduction). A step rewrites, anywhere inside a term, an introduction met by its elimination:

$$\begin{aligned} (\lambda h. M) N &\rightsquigarrow M(N/h), \\ \mathbf{fst} (M, N) &\rightsquigarrow M, \\ \mathbf{snd} (M, N) &\rightsquigarrow N, \\ \mathbf{case} (\mathbf{Left} M) \mathbf{of} \mathbf{Left} h \Rightarrow P \mid \mathbf{Right} k \Rightarrow Q &\rightsquigarrow P(M/h), \\ \mathbf{case} (\mathbf{Right} M) \mathbf{of} \mathbf{Left} h \Rightarrow P \mid \mathbf{Right} k \Rightarrow Q &\rightsquigarrow Q(M/k), \\ (\lambda x. M) t &\rightsquigarrow M(t/x), \\ \mathbf{let} \langle x, h \rangle = \langle t, M \rangle \mathbf{in} N &\rightsquigarrow N(t/x)(M/h). \end{aligned}$$

There is no step for *absurd*, because $\perp$ has no introduction form for it to meet. A term in which no step can be taken is in *normal form*.

The last two are the new ones. Call a function on a thing and the thing goes in for the parameter. Open a pair that was packed and the thing and the evidence go in for the two names. The table of connectives and types gains two rows:

Connective	Type	Introduction (constructor)	Elimination (pattern match)
$A \rightarrow B$	$A \rightarrow B$	$\lambda h. M$	application $M N$
$A \wedge B$	$A \times B$	(M, N)	$\mathbf{fst}, \mathbf{snd}$
$A \vee B$	$A + B$	$\mathbf{Left} M, \mathbf{Right} M$	case with two branches
$\perp$	$\perp$	—	<i>absurd</i>
$\neg A$	$A \rightarrow \perp$	$\lambda h. M$	application
$\forall x A$	$\Pi x A$	$\lambda x. M$	application $M t$
$\exists x A$	$\Sigma x A$	$\langle t, M \rangle$	let with two names

A function from things, and a pair of a thing and evidence: that's what Hughes's bot checks. The first two columns are a dictionary between two notations: $\times$ for $\wedge$, $+$ for $\vee$, Π for $\forall$ and Σ for $\exists$, with $\rightarrow$ and $\perp$ the same on both sides.

Types that mention values weren't invented for Hughes. In 1967 Nicolaas de Bruijn, a mathematician at Eindhoven, set out to have a machine check a whole mathematics textbook, and found that a checker couldn't do it unless

the types were allowed to mention the things the theorems were about. The language he built for the job, Automath, has the rules above, and in 1977 one of his students fed it the whole of Landau's *Foundations of Analysis*, every lemma, and it checked.⁶ Most proof assistants built on dependent types count it as an ancestor.

Remark (looking back: general recursion). The remark on general recursion at the end of Chapter 3 said that a programmer who wants both recursion and a consistent logic must say in the type where the recursion is allowed to go, and that this takes a logic with more to say than the propositional one. Types that may quantify over things are that logic.

4.3. First-order natural deduction. Twice now a typing has had its programs rubbed out to leave a logic: Hilbert's proofs under the combinators, and Gentzen's under λ , pairs and tags. The Curry–Howard correspondence says this always works. Beside every type system there's a logic, and the logic is what the typing rules say once the terms are deleted. We've just built a type system, so there's a logic standing beside it.

Before you look for it, one thing changes on the page, and it's the letters. Peirce, who found the quantifiers at about the same time as Frege and without knowing of him, wrote them as Π and Σ , for the reason given in the first section: a long *and* is a product and a long *or* a sum (Peirce, 1885). Logic traded his letters for $\forall$ and $\exists$, and nearly every logic book since has used those. Type theory took them back: ninety years after Peirce, Martin-Löf wrote the two dependent types as Π and Σ , for the reason given with the typing rules above.⁷ So rubbing out the programs means renaming as well as deleting: Π becomes $\forall$ and Σ becomes $\exists$, along with $\times$ to $\wedge$ and $+$ to $\vee$. The logic is written in the logic's letters from here on, and before I write it down I'd like you to find it.

Problem 4.13. Take the four rules for Π and Σ , delete every program, and rename the two types, leaving the contexts and the formulas. What are the four rules of proof that remain, and where have you heard each of them said aloud?

Here's what's left, with the declarations kept, since a declaration is a name and not evidence:

$$\frac{\Gamma, x \vdash A}{\Gamma \vdash \forall x A} \quad \frac{\Gamma \vdash \forall x A}{\Gamma \vdash A(t/x)} \quad \frac{\Gamma \vdash A(t/x)}{\Gamma \vdash \exists x A} \quad \frac{\Gamma \vdash \exists x A \quad \Gamma, x, A \vdash C}{\Gamma \vdash C}$$

with the side conditions carried over unchanged (t in scope in the second and third, x not free in C in the fourth), and here's where you've heard

⁶de Bruijn (1970); the thesis is Jutting's (1977). De Bruijn's name sits beside Howard's in the history of the correspondence, and this is why: he found the correspondence on his own, from the checker's end, and built the checker. Howard's notes of 1969 already had these two types, in their second half, as the constructions that go with the quantifiers of arithmetic. Martin-Löf made them the basis of a whole foundation for mathematics from 1972, after a first attempt that proved everything.

⁷Martin-Löf's Π and Σ may range over any type, including types that are themselves propositions, and in his system $\forall$ and $\exists$ are simply what Π and Σ are called when the body is a proposition. The chapter's Π and Σ range over the signature's one domain, which is the first-order case, and the one whose logic is Gentzen's.

them. *Let x be arbitrary*: the first. *In particular, for t* : the second. *Take x to be t* : the third. *Let y be such a thing; then*: the fourth. You’ve used all four as a programmer. A function body, written for its parameter without knowing which value will arrive, is *let x be arbitrary*. Calling the function on the value in hand is *in particular, for t* . A failing test is *take x to be t* : one input, exhibited, settles “some input breaks this”. And code that receives “some maintainer approved” and works with whoever it was, without being told who, is *let y be such a thing*.

Gentzen had the four rules in 1934. Natural deduction was first-order from the day it was invented, and the propositional rules are the half of his system without quantifiers.⁸ Here is the whole system as trees.

Definition 4.14 (derivations, with quantifiers). The *derivations* are defined inductively, and with each derivation $\mathcal{D}$ we define at the same time its *conclusion*, the formula at the root, and its set of *open hypotheses* $\text{OH}(\mathcal{D})$, a finite set of formulas. In the clauses below a derivation is drawn as a name with its conclusion written underneath it, and a formula in square brackets above a derivation is one that the clause discharges. Throughout, t is a term, y is a variable, and substitution is as defined earlier in the chapter.

1. (*hypothesis*) For every formula A , the one-node tree whose only node is labelled A is a derivation, with conclusion A and $\text{OH} = \{A\}$.

2. ($\rightarrow$ I) If $\mathcal{D}$ has conclusion B , then for every formula A ,

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

3. ($\rightarrow$ E) If $\mathcal{D}_1$ has conclusion $A \rightarrow B$ and $\mathcal{D}_2$ has conclusion A , then

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

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

$$\frac{\begin{array}{c} \mathcal{D}_1 \\ A \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ B \end{array}}{A \wedge B} \wedge\text{I} \quad \text{is a derivation,} \quad \text{OH} = \text{OH}(\mathcal{D}_1) \cup \text{OH}(\mathcal{D}_2).$$

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

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

are derivations, each with $\text{OH} = \text{OH}(\mathcal{D})$.

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

⁸Gentzen (1934); the *Allzeichen* $\forall$ first appears there, Peano’s $\exists$ dates from 1897. The side conditions, as Gentzen stated them for trees, are the ones in the definition below, and he called the variable they restrict the *Eigenvariable*, the variable proper to that step.

7. ($\vee E$) If $\mathcal{D}_0$ concludes $A \vee B$, and $\mathcal{D}_1$ and $\mathcal{D}_2$ both conclude C , then

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

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

8. ($\perp E$) If $\mathcal{D}$ concludes $\perp$, then for every formula C the tree with root C and immediate subtree $\mathcal{D}$ is a derivation, with $\text{OH} = \text{OH}(\mathcal{D})$.
9. ($\forall E$) If $\mathcal{D}$ concludes $\forall x A$ and t is substitutable for x in A , then

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

10. ($\forall I$) If $\mathcal{D}$ concludes $A(y/x)$, where y is substitutable for x in A , and y is free neither in $\forall x A$ nor in any formula of $\text{OH}(\mathcal{D})$, then

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

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

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

12. ($\exists E$) If $\mathcal{D}_0$ concludes $\exists x A$ and $\mathcal{D}_1$ concludes C , where y is substitutable for x in A , and 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

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

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

13. Nothing else is a derivation.

We write $\Gamma \vdash A$, and say that A is *derivable from* Γ , when some derivation has conclusion A and $\text{OH} \subseteq \Gamma$; such a derivation *witnesses* $\Gamma \vdash A$. The variable y of ($\forall I$) and ($\exists E$) is the *eigenvariable* of the step.

A tree has no declarations. It records what a context records by computing OH , and it records the stack in the same way. Where the type system said *x is the last thing declared*, the ($\forall I$) clause says two things: y is free in no open hypothesis, which is the same fact read off the leaves; and y is not free in $\forall x A$, which a fresh declaration guarantees. The ($\exists E$) clause states the one condition the type system had to write, y not free in C , and two that the context gave for free: y is fresh for $\exists x A$ and for the other open hypotheses.

Set the two clauses beside the turnstile, writing Δ for the open hypotheses of the tree above the bar, and the correspondence is visible bar for bar. On the left is the tree; on the right, what it witnesses:

$$\frac{\mathcal{D} \quad \frac{A(y/x)}{\forall x A} \forall I}{\Delta \vdash A(y/x)} \quad \frac{\Delta \vdash A(y/x)}{\Delta \vdash \forall x A}$$

with y free in no formula of Δ , nor in $\forall x A$; and

$$\frac{\frac{\mathcal{D}_0 \quad \exists x A}{C} \quad \frac{\mathcal{D}_1 \quad [A(y/x)] \quad C}{\Delta_1, A(y/x) \vdash C} \exists E}{\Delta_0, \Delta_1 \vdash C} \quad \frac{\Delta_0 \vdash \exists x A \quad \Delta_1, A(y/x) \vdash C}{\Delta_0, \Delta_1 \vdash C}$$

with y free in none of C , $\exists x A$ and the formulas of Δ_1 . Read on the right, the conditions are the erased rules with the declaration turned into a test: where the context declared y last, the turnstile asks that nothing to the left of it mention y .

Theorem 4.15 (Curry–Howard, with quantifiers). *The correspondence of Theorem 3.9 extends to the quantifiers. Erasing the terms from a typing, and writing $\wedge$, $\vee$, $\forall$ and $\exists$ for $\times$, $+$, Π and Σ , turns it into a derivation, with (ΠI) , (ΠE) , (ΣI) and (ΣE) becoming $(\forall I)$, $(\forall E)$, $(\exists I)$ and $(\exists E)$; and decorating a derivation with terms turns it into a typing. Writing $|\Gamma|$ for the set of types of the evidence entries of Γ , read as formulas,*

$$\Gamma \vdash M : A \text{ for some term } M \quad \text{if and only if} \quad |\Gamma| \vdash A,$$

for every context Γ that declares the variables free in A .

Erasure treats the two kinds of variable differently, and that is the difference between quantifiers and connectives. Erasing $\lambda h. M$ deletes h , since a variable standing for evidence has nothing to say once the evidence is gone. Erasing $\lambda x. M$ doesn't delete x , though, because a declared thing isn't evidence but a name, and the formula $\forall x A$ still needs it. So a logic keeps its variables when it loses its programs, and derivations with quantifiers have variables in them, where propositional derivations had none.

Normalisation and consistency. The theorems about running programs carry over, each with two more cases. A lemma proved for an arbitrary x and then used at a particular t is a detour, $(\forall I)$ met by $(\forall E)$, and it is removed by the step $(\lambda x. M)t \rightsquigarrow M(t/x)$. A witness packed and at once unpacked is $(\exists I)$ met by $(\exists E)$, and it is removed by the step for **let**.

Theorem 4.16 (subject reduction). *If $\Gamma \vdash M : A$ and $M \rightsquigarrow N$, then $\Gamma \vdash N : A$.*

In words: running a program never changes its type.

Theorem 4.17 (normalisation). *Every typed term runs to a normal form. Equivalently, every derivation can be rewritten into one without detours (Prawitz, 1965).*

Proof for terms without $+$ and Σ . The counting proof of Theorem 3.12 goes through with one more kind of redex. Give an atomic type degree 1 and $\Pi x A$

degree one more than A . Putting a term in for x changes no type's shape, so $A(t/x)$ has the degree of A . The observation still holds for the new step $(\lambda x. M) t \rightsquigarrow M(t/x)$. What it substitutes is a term of the signature, which names a thing and is never an introduction form, so the substitution creates no redex. And if M lands under an elimination and is an introduction form, the new redex has the degree of $A(t/x)$, less than that of $\Pi x A$, just as the body of a β step does. The rest of the count is word for word. $\square$

Σ breaks the count as $+$ does, and for the same reason: $\exists$ is an $\vee$ whose tags are things. A step on a **let** puts its body N where the **let** stood, and N has the result type C , which can have any degree at all. If the **let** stood under an elimination, the new redex can have a larger degree than the one removed. Prawitz's proof covers every connective and both quantifiers, with the permutative conversions added; Tait's method of computable terms extends to the quantifiers too. For terms with $+$ or Σ we take the theorem on trust.

Proposition 4.18 (closed normal forms). *A closed typed term in normal form is an introduction form: $\lambda h. M$, (M, N) , *Left* M , *Right* M , $\lambda x. M$ or $\langle t, M \rangle$.*

Theorem 4.19 (consistency). *In first-order natural deduction, no atomic formula is derivable from no hypotheses: $\not\vdash \perp$, and $\not\vdash P(t_1, \dots, t_n)$.*

Corollary 4.20 (disjunction property). *If $\vdash A \vee B$, then $\vdash A$ or $\vdash B$.*

Corollary 4.21 (existence property). *If $\vdash \exists x A$, then there is a term t with $\vdash A(t/x)$.*

The last three follow from the first three as Theorem 3.15 did. No introduction form has type $\perp$ or an atomic type, so there is no closed normal term of either. A closed normal term of type $A + B$ is *Left* M or *Right* M , and its tag says which disjunct holds. A closed normal term of type $\Sigma x A$ is $\langle t, M \rangle$ with $\vdash M : A(t/x)$, and its first half is the witness.⁹

4.4. Side conditions. Three things can go wrong with a quantifier rule, and each of them is a bug you've met in a program. The examples of the section on dependent types refused two of them already; here all three are set side by side, in both notations. You remember a side condition once you've watched its absence prove everything, so the third bug is run all the way to $\perp$ below; one of the exercises does the same for the first.

Generalising over a fixed variable. Suppose Hughes's bot is handling a request from some user u , and the hypothesis in force is **Maint**(u): *this* requester is a maintainer. One application of ($\forall$ I) would give $\forall x \text{ Maint}(x)$, everyone is a maintainer, from which, with the policy, everyone may publish.

⁹All three are about theorems, with no hypotheses. From a hypothesis $\exists x A$ you can't name a witness, since the hypothesis is a variable and a variable names nothing, which is why Baker's log could say "declined or approved" without saying which. And with no constant in the signature, $\vdash \exists x (\text{Maint}(x) \rightarrow \text{Maint}(x))$ fails, since there is no term to be the witness: the logic assumes nothing it isn't told, where most logic textbooks assume that something exists.

Both notations refuse it, and they refuse it for one reason. In the type system, u was declared before $h : \text{Maint}(u)$, so it isn't on top of the stack and (III) doesn't apply. In a tree, u is free in the open hypothesis $\text{Maint}(u)$, and the $(\forall I)$ clause says no. Discharge the hypothesis first, by $(\rightarrow I)$, and generalising becomes legal. But what you've generalised then is $\text{Maint}(u) \rightarrow \dots$, and that is harmless.

Of the two, the tree says it more sharply. The context asks that y be free in no hypothesis *available*; the tree asks that y be free in no hypothesis *used*. A hypothesis about y lying in the context untouched blocks (III), but it doesn't block $(\forall I)$, since it isn't open. Nothing provable changes, by the Curry–Howard theorem. But the tree states the condition that matters and nothing more, and that is one reason proof theory is written in trees.

Escaping witnesses. “Some maintainer approved the pull request; call them y ; so y is a maintainer.” We saw the type system refuse this in Example 4.11, with the result type $\text{Maint}(y)$, and the tree refuses it by the first condition of the $(\exists E)$ clause, that y not be free in the conclusion. The other two conditions rule out two more ways of cheating. You might choose as the fresh name a variable that $\exists x A$ already mentions, or one that some other open hypothesis already says something about. Either way “call them y ” would be naming a thing that was already in play. Together the three say that you may use the thing and may not learn which thing it is, which is what a module boundary says.

Variable capture. The third bug is the reason $(\forall E)$ asks that t be substitutable. Drop that condition and the system proves $\perp$ from two hypotheses any programmer would accept of the natural numbers:

$$\forall x \exists y (x < y) \quad \text{and} \quad \forall x \neg(x < x),$$

“every number has a larger one” and “no number is less than itself”. Take t to be the variable y , which is *not* substitutable for x in $\exists y (x < y)$, since it would land under the $\exists y$, and pretend $(\forall E)$ didn't care. The left-hand bar below is that instance, $t := y$, and it's the illegal one.

$$\frac{\frac{\forall x \exists y (x < y)}{\exists y (y < y)} \forall E \quad \frac{\frac{\forall x \neg(x < x)}{\neg(z < z)} \forall E \quad [z < z]^1}{\perp} \rightarrow E}{\perp} \exists E, 1$$

Four bars, and OH is the two harmless hypotheses, so the tree would witness $\forall x \exists y (x < y), \forall x \neg(x < x) \vdash \perp$. The system has proved $\perp$ from two facts about the natural numbers. The only thing that let it happen is the left-hand bar. (The $(\exists E)$ at the root is legal: z is free in none of $\perp$, $\exists y (y < y)$ and $\forall x \neg(x < x)$.) You can see what went wrong in the formulas. The y we substituted meant “some given number”. On landing under $\exists y$ it came to mean “whichever y the quantifier picks”. So $\exists y (y < y)$ says that some number is less than itself, which the hypothesis never said. With the condition in force the left-hand bar isn't available. The fix is to rename first: the hypothesis is equally well written $\forall x \exists z (x < z)$, its legal instance at y is $\exists z (y < z)$, and nothing follows from that but what should.

Remark (looking back: capture in the λ -calculus). Chapter 3 promised this bug on the programming side, where substitution renames a bound variable before passing under a λ that would capture. Running a program is substitution, $(\lambda h. M) N \rightsquigarrow M(N/h)$, and it needs the same condition. Take $\lambda x. \lambda y. x$, the combinator **K** that keeps its first argument (written in that chapter’s letters), apply it to a free variable y , and run it blindly:

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

The free y has landed under the inner λy and been captured. The result is the identity, which returns whatever it’s given. The program’s meaning has changed. Rename first, $(\lambda x. \lambda z. x) y \rightsquigarrow \lambda z. y$, and the constant function we meant comes out. On the proof side, 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 , and the blind result $\lambda y. y$ is a proof of $A \rightarrow A$ from nothing. Capture has swapped one theorem for another, which is what it did on the logic side too. One bug, one fix, on both sides of the correspondence.

A compiler or interpreter doesn’t substitute text when it calls a function; it carries the argument’s value in an environment. Macros and refactoring tools do substitute text, and that is where you’ve met the bug. The logic side has been known for a century, and every textbook states the condition. What the correspondence adds is that it’s the same condition.

4.5. Generics. The quantifier so far ranges over things. Let it range over *types* instead and you get something you’ve been writing for years. A *type variable* a stands for a type nobody has chosen yet, and $\Pi a A$ is the type of a program that works for every type a . Such a program is a *type abstraction* $\Lambda a. M$: it takes a type and returns a program. You use it by an *instantiation* $M[B]$, which hands it a type B . The rules are (III) and (IIE) of dependent types with types in the place of things,

$$(III) \frac{\Gamma, a \vdash M : A}{\Gamma \vdash \Lambda a. M : \Pi a A} \quad (IIE) \frac{\Gamma \vdash M : \Pi a A}{\Gamma \vdash M[B] : A(B/a)},$$

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

Example 4.22 (the identity). The function `id x = x` is $\Lambda a. \lambda h. h : \Pi a (a \rightarrow a)$: declare a , declare $h : a$, pop h with (abs), pop a with (III). Using it at `Int` is one (IIE),

$$(\Lambda a. \lambda h. h)[Int] \rightsquigarrow \lambda h. h : Int \rightarrow Int,$$

and Haskell inserts that instantiation for you at every call.

The type of `length` is `[a] -> Int`, and the `a` in it is a parameter, a type nobody has chosen yet. Haskell leaves the quantifier out, and writes it `forall` only when asked. Written in, it reads

$$length : \Pi a ([a] \rightarrow Int),$$

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. The first is one copy per element type,

`lengthOfIntList, lengthOfStringList, lengthOfBoolList, ...`

which is Hughes’s thirty-seven letters again. 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. 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, $A \rightarrow B \rightarrow A$, 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 $\Lambda a. \Lambda b. \lambda h. \lambda k. h : \Pi a \Pi b (a \rightarrow b \rightarrow a)$, and the typing “`(K) [Int, Int]`” is the term `K [Int] [Int]`. Java’s `static <A, B>` declares the two type variables in the same place.

The side condition on $(\forall I)$ 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 (III) 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.

Example 4.23 (a counter). There is a Σ over types too, and it’s a module. Suppose you want a counter: a thing with a starting state, a way to step it, and a way to read off a number. You don’t want to say what the state is, because that’s the implementation’s business.

```
{-# 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 $\Sigma a (a \times (a \rightarrow a) \times (a \rightarrow \text{Int}))$, *there is* a type with these three operations. `MkCounter` packs a witness, the state type, as (ΣI) packed a thing. The pattern match in `tick3` opens the pair, as (ΣE) did, and 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*. It’s the condition written on (ΣE) , in the compiler’s words. The implementation type may be used inside the match and may not leave it. A module boundary is a boundary because

of this. It's one side condition on one rule, not a convention anybody has to remember.¹⁰

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. The domain was types rather than people, though, which is why the type never amounted to a specification. Specifications need the first-order quantifier, over things.

4.6. Specifications. Read the consistency theorem with names in the language. The logic proves no atomic sentence: neither **Maint**(smith) nor **Pub**(reyes) nor $2 < 3$. Whatever it proves outright has a connective or a quantifier on the outside and is about how claims relate. Every fact about a particular person, element or day has to come from the hypotheses. A *specification* is a set of sentences taken as hypotheses, and what we prove, we prove *from* it. Three specifications, then, in increasing order of interest.

Example 4.24 (the release policy). From Γ_{pol} and **Maint**(smith) we derive **Pub**(smith) by the tree for Smith's ticket with its programs rubbed out: two ($\forall E$)s and two ($\rightarrow E$)s, with OH the two policy sentences and the one fact. So the tree witnesses $\Gamma_{\text{pol}}, \text{Maint}(\text{smith}) \vdash \text{Pub}(\text{smith})$, and it is a *certificate*. An auditor who disbelieves the answer needn't rerun the bot or understand it; they check four bars. And since the logic proves no atomic sentence on its own, the auditor knows that **Pub**(smith) came from MAINTAINERS.md and from nowhere else. From Γ_{pol} alone we can derive *every maintainer may publish*, which no policy engine settles by evaluation because it quantifies over everyone at once. It comes in two pieces, to fit the page. Call the first $\mathcal{E}$:

$$\mathcal{E} = \frac{\frac{\forall x (\text{Maint}(x) \rightarrow \text{Write}(x))}{\text{Maint}(u) \rightarrow \text{Write}(u)} \forall E \quad [\text{Maint}(u)]^1}{\text{Write}(u)} \rightarrow E$$

and then

$$\frac{\frac{\frac{\forall x (\text{Write}(x) \rightarrow \text{Pub}(x))}{\text{Write}(u) \rightarrow \text{Pub}(u)} \forall E \quad \mathcal{E}}{\text{Pub}(u)} \rightarrow E}{\text{Maint}(u) \rightarrow \text{Pub}(u)} \rightarrow I, 1}{\forall x (\text{Maint}(x) \rightarrow \text{Pub}(x))} \forall I$$

After the discharge, OH is the two policy sentences, and they're sentences, so u is free in neither and the last bar is legal. The tree witnesses $\Gamma_{\text{pol}} \vdash \forall x (\text{Maint}(x) \rightarrow \text{Pub}(x))$. Had we generalised before discharging, **Maint**(u) would still have been open with u free in it, and the clause wouldn't have applied. In plain words, we'd have been claiming the policy for all users on the strength of one. The tree is the typing of the same sentence in Example 4.10, with the programs gone, bar for bar.

¹⁰Mitchell 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.

Example 4.25 (a comparator). Here’s insertion sort, written so that the comparison is a parameter. Standard libraries write their sorting routines this way, because a library can’t know what you’re sorting.

```
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
```

Everything so far applies to that program. It has a type, and the type is polymorphic, like that of `length`,

$$\text{sort} : \Pi a ((a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]),$$

one program for every element type, instantiated at each call. The checker proves that type, and by the Curry–Howard correspondence the proof *is* the program. So the compiler has certified something real. This code won’t add a string to a number, won’t call a non-function, and will hand back a list of the type it was given. About sorting it has certified nothing whatever.

The gap is the parameter `lt`. Give `sort` a comparison that returns `True` when its two arguments are equal: `<=` where `<` was meant, about the commonest mistake there is. The type doesn’t change, because `<=` and `<` have the same type. The program compiles, runs and returns a list, which is sometimes not sorted. A library sort, given a longer input, may do something stranger. This is an exception Java’s sort really throws:

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

Read it carefully, because it’s unusual and exact. The sort hasn’t 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 had become impossible: it merged two runs it had recorded as ordered and found them not ordered, which can’t happen if the comparison behaves. It detected the failure of a *consequence* of the contract. To say what that means we have to be able to state the contract and derive the consequence, and now we can.

What `sort` needs of `lt` is that it be a *strict weak order*, the kind of order that may leave two different elements tied, as sorting people by surname does: three sentences, with $x < y$ for `lt(x,y)`.

What the library needs	As a sentence
nothing is less than itself	$\forall x \neg(x < x)$
if $x < y$ and $y < z$ then $x < z$	$\forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z)$
if $x < y$, every z is above x or below y	$\forall x \forall y \forall z (x < y \rightarrow x < z \vee z < y)$

Call those three Γ_{ord} . They say *for all*, and they’re about the elements, so no propositional type could hold them, or even write them down. They’re what the library’s documentation is trying to say in English. What a comparator built from `<=` violates is *asymmetry*. It reports $a < b$ and $b < a$ for a pair of

equal elements, and no comparator obeying the contract can, because

$$\Gamma_{\text{ord}} \vdash \forall x \forall y \neg(x < y \wedge y < x).$$

That isn't one of the three sentences, so it has to be derived. Informally the argument is 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. Here it is as a tree, in two pieces to fit the page. The first piece, which I'll call $\mathcal{D}$, takes transitivity down to $u < u$ on the supposition:

$$\mathcal{D} = \frac{\frac{\frac{\forall x \forall y \forall z (x < y \wedge y < z \rightarrow x < z)}{\forall y \forall z (u < y \wedge y < z \rightarrow u < z)} \forall E}{\forall z (u < v \wedge v < z \rightarrow u < z)} \forall E}{u < v \wedge v < u \rightarrow u < u} \forall E \quad [u < v \wedge v < u]^1 \rightarrow E$$

The second uses irreflexivity, discharges the supposition and generalises:

$$\frac{\frac{\frac{\forall x \neg(x < x)}{\neg(u < u)} \forall E \quad \mathcal{D}}{\perp} \rightarrow E}{\neg(u < v \wedge v < u)} \rightarrow I, 1}{\forall y \neg(u < y \wedge y < u)} \forall I}{\forall x \forall y \neg(x < y \wedge y < x)} \forall I$$

Read it from the top. The three ($\forall E$) bars of $\mathcal{D}$ strip the quantifiers off transitivity, choosing $t := u$, $t := v$ and $t := u$; the last choice is the step that says “then $u < u$ ”. The ($\rightarrow E$) bar fires the result on the supposition. In the second piece, irreflexivity at u meets $u < u$ and gives $\perp$. The ($\rightarrow I$) bar discharges the supposition, turning “the supposition leads to absurdity” into “not the supposition”, and the two ($\forall I$) bars generalise over v and then over u . The order is forced. The ($\forall I$) bar for v can't come above the discharge, because v is free in the supposition while it is open. This is the stack again, and it is why the proof couldn't have generalised first and supposed afterwards. When the last bar is drawn, OH is irreflexivity and transitivity, two of the three sentences of Γ_{ord} , so the two pieces together witness the claim, $\Gamma_{\text{ord}} \vdash \forall x \forall y \neg(x < y \wedge y < x)$.

Now decorate it. The program is one line:

$$\lambda u. \lambda v. \lambda h. \text{irrefl } u (\text{trans } u \ v \ u \ h) : \Pi x \Pi y \neg(x < y \times y < x),$$

with *trans* and *irrefl* two of the sentences of Γ_{ord} as variables, and the type written in the type theorist's letters. Given two elements and a claimed pair of comparisons h , it calls transitivity at u , v , u , hands it h , and hands what comes back, a proof of $u < u$, to irreflexivity, which returns $\perp$. Nine bars of proof, one line of program, and you can see in the program why the proof had the shape it had: three λ s, one for each *let ... be arbitrary* and one for the supposition, and the quantified hypotheses called as functions. So a comparator obeying the contract can never report both $a < b$ and $b < a$, and when the sort catches one doing just that, the exception is reporting a *theorem* being violated. The type checker couldn't state that theorem. A verifier, which is a type checker whose types may quantify over things, can state it and check the proof just given. That is the distance between *safe* and *correct*, on a program you've called a hundred times.

Example 4.26 (the sale). And the patch. With $\text{Disc}(d)$ for “the discount applies on day d ”, what a sale *is* comes to one sentence,

$$\forall d (start \leq d \wedge d \leq end \rightarrow \text{Disc}(d)).$$

The patched code applies the discount only when $start < d$ and $d < end$, so it fails the sentence at $d := start$, where the sentence demands a discount and the code withholds one. Nobody had written the sentence down, which is how the patch got approved. The bug report said “last day”, the tests said “last day”, and the only thing that said “every day” was in the head of whoever wrote the sale. Once the sentence is written, “is the patch correct?” has an answer a checker can verify, and the answer is no. What’s still missing is a logic in which “after this patch runs, the discount applies” is itself a formula. Chapter 13 has one, and comes back to this patch.

Witnesses. The quantifier $\exists$ is the $\forall$ whose tags are things, and by the existence property a proof of it carries its tag. “Is this file readable by anyone outside the team?” is an existence question, and a *yes* that can’t say *who* is no use to a security engineer. In this logic a *yes* proved from no hypotheses comes with the name attached.

The insistence on a name is also worth distrusting. “Either the program halts or it doesn’t” is the kind of thing you’d like to be able to assert without producing a halting run or a proof that there’s none. “Some input breaks this function” is the kind of thing you’d like to assert without producing the input. In this logic you may assert neither, because a proof of *or* says which and a proof of *there is* says what. Whether that’s a virtue or a restriction is a question for Part 2.

4.7. Second-order logic and System F (optional). *This section may be skipped: nothing later in the notes depends on it.* The section on generics met a quantifier over types from the programming side. Here it is found the other way round: start from a logic, decorate its derivations, and see what programs come out. On the way, $\wedge$, $\vee$ and $\perp$ turn out to be definable from $\rightarrow$ and $\forall$, and a quantifier over all propositions turns out to need a consistency proof of its own.

The first-order quantifiers range over things: people, pull requests, days. Look back at the propositional theorems, though, and you’ll find a quantifier of another kind that nobody wrote down. The theorem $p \rightarrow p$ was never about p in particular. Its proof works with any formula at all 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, which is Hughes’s thirty-seven letters once more. The cure is the same as before. Let a quantifier range over propositions, and write the claim once:

$$\forall X (X \rightarrow X),$$

“whatever proposition X is, X implies X ”.

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 **smith**, so we write propositional variables differently, as X, Y, Z .

Definition 4.27 (second-order propositional logic). Fix the propositional letters $p, q, r, \dots$ and, separately, a stock of *propositional variables* $X, Y, Z, \dots$. The formulas are

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

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, $(\rightarrow I)$ and $(\rightarrow E)$, and two more:

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

$$\frac{\mathcal{D} \quad \forall X A}{A(B/X)} \forall E \quad \text{for every formula } B \text{ substitutable for } X \text{ in } A,$$

each with the open hypotheses of $\mathcal{D}$.

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

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

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 $\text{OH} = \emptyset$, so they witness $\vdash \forall X (X \rightarrow X)$ and $\vdash \forall X \forall Y (X \rightarrow Y \rightarrow X)$. An instance of the schema is now one step: $(\forall E)$ at p turns $\forall X (X \rightarrow X)$ into $p \rightarrow p$.

The connectives, defined. The logic has $\rightarrow$ and $\forall$ and nothing else, and needs nothing else, because $\wedge, \vee$ and $\perp$ 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 4.29 (the connectives in second-order logic). *Define, with X a propositional variable not free in A or B ,*

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

With these definitions every rule of natural deduction for $\perp$, $\wedge$ and $\vee$ 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 $(\forall I)$ below is legal. $(\perp E)$: from $\Delta \vdash \forall X X$, $(\forall E)$ at C gives $\Delta \vdash C$. $(\wedge I)$: from $\Delta \vdash A$ and $\Delta \vdash B$, two $(\rightarrow E)$ s give $\Delta, A \rightarrow B \rightarrow X \vdash X$; $(\rightarrow I)$ gives $\Delta \vdash (A \rightarrow B \rightarrow X) \rightarrow X$, and $(\forall I)$ gives $\Delta \vdash A \wedge B$. $(\wedge E_1)$: from $\Delta \vdash A \wedge B$, $(\forall E)$ at A gives $\Delta \vdash (A \rightarrow B \rightarrow A) \rightarrow A$; and $\vdash A \rightarrow B \rightarrow A$, as in Example 4.28; so $(\rightarrow E)$ gives $\Delta \vdash A$. $(\wedge E_2)$ is the same with B and $A \rightarrow B \rightarrow B$. $(\vee I_1)$: from $\Delta \vdash A$, $(\rightarrow E)$ gives $\Delta, A \rightarrow X, B \rightarrow X \vdash X$; two $(\rightarrow I)$ s and $(\forall I)$ give $\Delta \vdash A \vee B$. $(\vee I_2)$ is the same from B . $(\vee E)$: from $\Delta \vdash A \vee B$, $(\forall E)$ at C gives $\Delta \vdash (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$; the two branches are $\Delta, A \vdash C$ and $\Delta, B \vdash C$, and $(\rightarrow I)$ turns them into $\Delta \vdash A \rightarrow C$ and $\Delta \vdash B \rightarrow C$; two $(\rightarrow E)$ s give $\Delta \vdash C$. $\square$

Each connective is defined by its elimination rule. $A \wedge B$ says “whatever follows from A and B together, follows”; $A \vee B$ says “whatever follows from A and also from B , follows”, which is proof by cases; and $\perp$ says “everything follows”. So propositional natural deduction sits inside this one, built from $\rightarrow$ and $\forall$ alone.

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 $\iota \rightarrow o$, 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 for which he typed his calculus. In it, induction on the natural numbers is one sentence,

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

where first-order logic needs a schema, one axiom for each formula. The power has a price: Chapter 15 shows that 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.

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: $\forall X (X \rightarrow X)$ may be used at $\forall X (X \rightarrow X)$, and $\perp$, which is $\forall X X$, at $\perp$. 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 Russell banned them from *Principia* for fear of the self-application that produced the paradoxes. So we have to ask whether this logic is consistent. For first-order logic the answer came from programs: decorate the derivations, run them, and look at what they stop at. Do the same here.

The polymorphic λ -calculus. Decorate a second-order derivation as propositional ones were decorated. The bars for $\rightarrow$ 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, and a $(\forall E)$ bar at B becomes an

instantiation $M[B]$, which hands it one. The formulas become types, with the renaming run backwards: a propositional variable X becomes a type variable a , and $\forall X$ becomes Πa , the Π of dependent types with types in the place of things. The two theorems of Example 4.28 decorate to

$$\Lambda a. \lambda h. h : \Pi a (a \rightarrow a) \quad \text{and} \quad \Lambda a. \Lambda b. \lambda h. \lambda k. h : \Pi a \Pi b (a \rightarrow b \rightarrow a).$$

Here is the calculus that results, the one the section on generics used informally.

Definition 4.30 (polymorphic λ -calculus). Fix a stock of *type variables* $a, b, c, \dots$. The *polymorphic types* and the *terms* are

$$\begin{aligned} A, B &::= a \mid A \rightarrow B \mid \Pi a A, \\ M, N &::= h \mid M N \mid \lambda h. M \mid \Lambda a. M \mid M[B], \end{aligned}$$

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 the clauses for substitution in formulas. A *context* is a finite list of declarations of type variables a and of typings $h : A$; no variable is declared twice, and every type variable free in a type in the context is declared earlier than the typing that mentions it. The rules are

$$\begin{aligned} (\text{var}) \quad & \frac{}{\Gamma \vdash h : A} \text{ if } h : A \text{ is in } \Gamma & (\text{abs}) \quad & \frac{\Gamma, h : A \vdash M : B}{\Gamma \vdash \lambda h. M : A \rightarrow B} \\ (\text{app}) \quad & \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \\ (\text{III}) \quad & \frac{\Gamma, a \vdash M : A}{\Gamma \vdash \Lambda a. M : \Pi a A} \\ (\text{PIE}) \quad & \frac{\Gamma \vdash M : \Pi a A}{\Gamma \vdash M[B] : A(B/a)} \text{ if } B \text{'s type variables are declared in } \Gamma \end{aligned}$$

and the steps are $(\lambda h. M) N \rightsquigarrow M(N/h)$ and $(\Lambda a. M)[B] \rightsquigarrow M(B/a)$.

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 $\perp$, for the reason the logic has already given: they can be defined. Erasure and decoration go through as before, 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.)

Data types, defined. Decorate the derivations in the proof of the proposition on the connectives, and the definitions of $\wedge$, $\vee$ and $\perp$ become programs,

with a for X and C for the result type:

$$\begin{aligned}
(M, N) &:= \Lambda a. \lambda k. k \ M \ N & (\wedge I), \text{ with } k : A \rightarrow B \rightarrow a, \\
fst \ P &:= P \ [A] \ (\lambda h. \lambda k. h) & (\wedge E_1), \\
snd \ P &:= P \ [B] \ (\lambda h. \lambda k. k) & (\wedge E_2), \\
Left \ M &:= \Lambda a. \lambda f. \lambda g. f \ M & (\vee I_1); \text{ Right } M \text{ has } g \ M, \\
\text{case } P \text{ of } \dots &:= P \ [C] \ (\lambda h. Q) \ (\lambda k. R) & (\vee E), \\
absurd \ P &:= P \ [C] & (\perp E).
\end{aligned}$$

They also run as they should:

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

and a **case** on $Left \ M$ 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

$$Bool := \Pi a \ (a \rightarrow a \rightarrow a), \quad Nat := \Pi a \ ((a \rightarrow a) \rightarrow a \rightarrow a),$$

with $true := \Lambda a. \lambda h. \lambda k. h$ and $false := \Lambda a. \lambda h. \lambda k. k$, and the numeral $\bar{n} := \Lambda a. \lambda f. \lambda h. f \ (\dots (f \ h))$ with n copies of f . Addition is

$$add := \lambda m. \lambda n. \Lambda a. \lambda f. \lambda h. m \ [a] \ f \ (n \ [a] \ f \ h) : Nat \rightarrow Nat \rightarrow Nat :$$

do n steps, then m more.

Remark (looking back: booleans and numerals). These are the booleans and numerals of Chapter 1, where $true$ was K , $false$ was KI , and $\bar{n} f x$ 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 $(A \rightarrow A) \rightarrow A \rightarrow A$ 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:

$$exp := \lambda m. \lambda n. \Lambda a. n \ [a \rightarrow a] \ (m \ [a]) : Nat \rightarrow Nat \rightarrow Nat$$

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. The counter of the section on generics had the type $\Sigma a \ (a \times (a \rightarrow a) \times (a \rightarrow Int))$. In the polymorphic calculus Σ over types is defined like $\vee$, by what you can do with it:

$$\Sigma a \ A := \Pi b \ (\Pi a \ (A \rightarrow b) \rightarrow b),$$

“whatever follows from A , whatever a is, follows”. The pattern match in `tick3` opens the pair at the result type `Int`: instantiate b at `Int` and hand it $\Lambda a. \lambda h. N$. That substitution puts `Int` under the Πa , so it is legal only because `Int` doesn’t mention a , which is the escape condition once more.

Normalisation. Now the question the logic left open. 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 4.31 (normalisation for the polymorphic λ -calculus; on loan). *Every typed term of the polymorphic λ -calculus runs to a normal form.*

Girard proved it in 1971, by extending Tait’s method of computable terms. It is much harder than the normalisation theorems it extends.¹¹ 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 $\perp$. Impredicativity is safe here. The same argument says more. A closed normal term of type $\Pi a\ (a \rightarrow a)$ is $\Lambda a.\ \lambda h.\ M$, with $M : a$ in the context $a, h : a$. The type a has no introduction form, so M is neutral (a variable with eliminations applied to it), with head h ; and nothing can be eliminated from $h : a$, so M is h . The only program of type $\Pi a\ (a \rightarrow a)$ is the identity. The type says what the program does, without your having to read the program. Reynolds called this *parametricity*, and Wadler turned it into a slogan, *theorems for free* (Wadler, 1989).

The language is also very 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.¹²

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. Martin-Löf’s first type theory, of 1971, took the last step: a type **Type** of all types, with **Type** : **Type**. Girard showed that 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.¹³ The repair, in Martin-Löf’s later theories, is a hierarchy of universes, $\text{Type}_0 : \text{Type}_1 : \text{Type}_2 : \dots$, none of them a member of itself.

4.8. History. The quantifier was invented twice, like natural deduction. Frege had it in 1879, in a notation nobody else used. Peirce and his student Mitchell had it by 1885, as Π and Σ , knowing nothing of Frege, and it was their version, carried by Schröder to Peano and by Peano to Russell, that became the logic everyone learned. Peano turned the Σ into $\exists$ in 1897 and Gentzen added $\forall$ to match in 1934. Hilbert and Ackermann’s textbook of 1928 was the first to set out the first-order logic of this chapter as a system on its own, and to ask the question about it that Part 2 answers. Then the letters came back as types: in Howard’s notes, in de Bruijn’s checker, in Martin-Löf’s foundation, and, over types rather than things, in Girard’s and Reynolds’s calculus, the one in your compiler. Church’s simple theory of types, the higher-order logic for which he typed the λ -calculus in 1940, is the ancestor of the provers that check mathematics in that logic today.

¹¹The 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).

¹²Girard’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.

¹³This is *Girard’s paradox* (Girard, 1972), a typed cousin of Russell’s.

Who	When	What
Frege	1879	function and argument; the quantifier, in two dimensions
Peirce, Mitchell	1883–85	the quantifier again, as Π and Σ
Peano	1897	$\exists$
Hilbert & Ackermann	1928	first-order logic as a system of its own
Gentzen	1934	$\forall$; the quantifier rules, with their side conditions
Church	1940	higher-order logic: the simple theory of types
Prawitz	1965	normalisation, quantifiers included
de Bruijn	1968	Automath: types that mention things, in a checker
Howard	1969	the quantifiers as a function from things and a pair
Girard; Reynolds	1971; 1974	Π over types: generics; second-order logic; normalisation
Girard	1972	Type : Type proves everything
Martin-Löf	1972–	Π and Σ ; a foundation on dependent types, with universes
Mitchell & Plotkin	1988	modules as existential types
Wadler	1989	parametricity as theorems for free

Summary. The quantifiers are $\wedge$ and $\vee$ with the list of cases made unbounded: $\forall x A$ is an *and* with a case for every thing, written once with a parameter, and $\exists x A$ an *or* whose tags are things. Said as types, a proof of *for all* is a function from things, whose result type mentions its argument; a proof of *there is* is a pair of a thing and evidence. These are $\Pi x A$ and $\Sigma x A$, Peirce’s letters, which the logic traded for $\forall$ and $\exists$ and the type theorists kept. Rub the programs out and the four rules are Gentzen’s: let x be arbitrary, in particular for t , take x to be t , let y be such a thing. The context enforces every freshness condition but the one on $(\exists E)$, which is why that one alone is written on its rule. The three ways of breaking the conditions are three bugs you’ve met, capture among them. Running a program still removes detours, every typed program still stops (proved by counting when there is no $\vee$ or $\exists$, on loan when there is), the logic proves no atomic sentence, and a proof of *there is* names the thing.

Let the quantifier range over types instead of things and you get generics: one program for every type, written once and checked once. Its side conditions are the error messages *rigid type variable* and *would escape its scope*. The optional section finds the same programs from the logic’s side, by decorating second-order logic. There a schema becomes a single theorem, and $\perp$, $\wedge$ and $\vee$ are definable from $\rightarrow$ and $\forall$. Every program still stops (on loan, from Girard), so that logic is consistent, and a type such as $\Pi a (a \rightarrow a)$ says what its one program does.

A **specification** is a set of sentences, and “ A follows from it” is a derivation, checkable by pattern matching: Hughes’s policy, a comparator’s contract, a sale. Since the logic proves no atomic sentence on its own, everything about Smith comes from the file. 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's *correct*: what was asked of it is provable from what was assumed.

Three questions to ask of every proof system, then. Could it prove something false? We've shown that this one doesn't prove *everything*, which is all that "false" can mean until the word has a meaning. Does it prove everything it should? We can't even ask. 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,

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

is a formula built from letters and arrows like any other. With type variables in place of the letters it is the type $\Pi a \Pi b (((a \rightarrow b) \rightarrow a) \rightarrow a)$, 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 4.1. Write the typing tree for Smith's ticket in full, from Hughes's five declarations, and then the tree the checker builds for `publish reyes ticket`, marking the bar at which it fails and what the error message says. Then say what the context would have to contain about Reyes for the handler to type.

Exercise 4.2. Take Γ_{pol} and the extra fact `Maint(brown)`.

- (1) Derive `Pub(brown)`, as a tree, and give its program.
- (2) Try to derive $\forall x (\text{Maint}(x) \rightarrow \text{Pub}(x))$ with the $(\forall I)$ bar above the $(\rightarrow I)$ bar, and say which condition stops you.
- (3) Add the policy line "nobody may read the audit log except administrators" to the signature and the policy, and say which of $\forall$, $\exists$ and $\neg$ you needed, and why the obvious first attempt says something weaker.

Exercise 4.3. Derive $\exists x \forall y R(x, y) \rightarrow \forall y \exists x R(x, y)$ and check that its program is

$$\lambda k. \lambda y. \text{let } \langle x, h \rangle = k \text{ in } \langle x, h y \rangle.$$

Then try the converse, $\forall y \exists x R(x, y) \rightarrow \exists x \forall y R(x, y)$, and say exactly which side condition stops you.

Exercise 4.4. Derive $\forall x (A \wedge B) \rightarrow \forall x A \wedge \forall x B$ and $\exists x (A \vee B) \rightarrow \exists x A \vee \exists x B$, and their converses, giving the program for each. (These are distributive laws one level up: a function into a pair is a pair of functions, and a pair with a tagged second component is a tagged pair.)

Exercise 4.5. Write a comparator, in any language, that violates exactly one of the three lines of the contract, and say which consequence of the contract it breaks. Which violation would a sort routine detect, and how?

Exercise 4.6.

- (1) Derive $\exists x (\text{Maint}(x) \rightarrow \text{Maint}(x))$ from the empty context over a signature with a constant c , and say why there is no derivation when the signature has none.
- (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 the existence property have to give up for it to go through?

Exercise 4.7. Suppose $(\forall I)$ carried no side condition, so that a tree could generalise over a variable free in an open hypothesis.

- (1) From the hypothesis $\text{Maint}(u)$ derive $\forall x \text{Maint}(x)$ in one step, then $\vdash \text{Maint}(u) \rightarrow \forall x \text{Maint}(x)$, and then $\vdash \forall y (\text{Maint}(y) \rightarrow \forall x \text{Maint}(x))$.
- (2) Conclude $\perp$ from the two hypotheses $\text{Maint}(c)$ and $\neg \text{Maint}(d)$, for constants c and d , which say only that one person is a maintainer and another isn't.
- (3) Say which of the steps in (1) the real rule forbids, and state the corresponding compiler error.

Exercise 4.8. Write the types of `length` and `map` with their quantifiers explicit. Then type the use `map show [1, 2]`, naming the instantiation at each quantifier. Finally, 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 write the error in the words of the section on generics.

Exercise 4.9. Try to derive $\exists x (\text{Maint}(x) \rightarrow \forall y \text{Maint}(y))$: “there is somebody such that, if they are a maintainer, everyone is”. Use the existence property to say what a closed derivation would have to hand over, and then say where every attempt sticks. Keep your attempt; Part 2 returns to this formula, together with $\neg \neg p \rightarrow p$ from the exercises of the last chapter.

Exercise 4.10. (*Optional section.*) With $A \wedge B$ and $A \vee B$ defined in second-order logic, type the programs (M, N) , `fst` and `case` of the proposition on the connectives, and run `case (Right M) of Left h => Q | Right k => R` to normal form.

Exercise 4.11. (*Optional section.*) Show that the closed normal terms of type $\Pi a (a \rightarrow a \rightarrow a)$ are exactly *true* and *false*. What are the closed normal terms of type $\Pi a a$, and what does that say about $\perp$?

Exercise 4.12. (*Optional section.*) Type `add` and `exp` on `Nat`, and give a program `mult : Nat → Nat → Nat`. Run `add 2 1 [a] f h` to normal form.

Reading. Sørensen and Urzyczyn (2006, chapter 8) is the first-order half of the correspondence at the level of this chapter, with the rules for Π and Σ written out and the normalisation theorem proved. Howard (1980), now for its second half, is where the quantifiers were first read as a function from things and a pair. Mitchell and Plotkin (1988) is the module side of abstract types; Wadler (2015) tells the whole story to programmers. Open Logic Project (2026, chapter 13) has the same rules at more leisure, and the Hilbert-style quantifier axioms this chapter did not need. de Bruijn (1970)

is a founder’s account of a checker whose types mention things. Girard et al. (1989, chapters 11, 14 and 15) is the polymorphic λ -calculus by its inventor: the definitions of the connectives, the normalisation proof and the representation theorem. Wadler (1989) is parametricity, with examples a programmer can use.

References.

- Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.
- N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer, Berlin, 1970.
- Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Louis Nebert, Halle, 1879. English translation by S. Bauer-Mengelberg in J. van Heijenoort (ed.), *From Frege to Gödel*, Harvard University Press, 1967, pp. 1–82.
- Gerhard Gentzen. Untersuchungen über das logische Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934. Appeared 1934–35; English translation in *The Collected Papers of Gerhard Gentzen*, ed. M. E. Szabo, North-Holland, 1969.
- 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.
- Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.
- William A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980. Written in 1969.
- John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
- Open Logic Project. The Open Logic Text. <https://openlogicproject.org>, 2026. Release 9620cc7, 12 July 2026. Licensed CC BY 4.0.
- Charles S. Peirce. On the algebra of logic: A contribution to the philosophy of notation. *American Journal of Mathematics*, 7(2):180–202, 1885.
- Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Almqvist & Wiksell, Stockholm, 1965.
- 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.
- Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry–Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, 2006.
- L. S. van Benthem Jutting. *Checking Landau’s “Grundlagen” in the AUTOMATH System*. PhD thesis, Eindhoven University of Technology, 1977.

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.

Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.