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. NJ with quantifiers 10
4.3. Side conditions 20
4.4. Dependent types 23
4.5. Curry–Howard with quantifiers 30
4.6. Theories 33
4.7. The λ-cube 36
History 38
Exercises 40

4. Quantifiers

Suppose you write drivers for a phone maker, and on a Monday morning a security researcher sends you a thirty-second screen recording. An app with no permissions at all, installed from the store, asks the system for a camera and is handed one, and a moment later it is reading from the microphone, which it was never granted.

None of it is a bug in the usual sense. Every handle in the kernel, whether to a camera, a socket or a file, is the same sort of thing: a small token the app holds, with operations `read`, `write` and `close` that the kernel carries out on the app's behalf. Behind each token is a private record that only the token's own driver understands.

The camera call accepted the app's token and set to work on the record behind it as though it were a camera's. It never asked whether it was. The bytes were a microphone handle's, and the call read them as camera settings, one of which was a pointer the app had chosen.

What did the token say? A handle is a claim about *some resource*: the holder may operate on it. But the token the camera call checked doesn't say *which kind* of resource it is for, so to the kernel a handle opened for a microphone is just a handle. Hand the camera call a handle of the wrong kind and nothing in the code has anything to say.

Security people have a name for this, *type confusion*: a value of one type used where another was meant, the two kept apart only in the programmer's head.[1] The camera driver was not careless by the usual standard. "This record is a camera's" was a fact its author knew when she wrote the call, and the code had nowhere to write it down.

Your first thought is the right one, and I want to watch it fail. In the last chapters a propositional letter stood for a whole claim: let p be "the holder may read a camera". If the trouble is that p doesn't say *camera*, write a letter that does. Let p_{cam} be "the holder has a camera capability", q_{cam} "the holder has a camera handle" and r_{cam} "the holder may read a camera". Do the same for the microphone and every other kind, and write the rule once per kind:

$$
\begin{aligned}
p_{\text{cam}} &\to q_{\text{cam}},\\
q_{\text{cam}} &\to r_{\text{cam}},\\
p_{\text{mic}} &\to q_{\text{mic}},\\
q_{\text{mic}} &\to r_{\text{mic}},\\
&\vdots
\end{aligned}
$$

Now a camera handle proves r_{cam}, which isn't r_{mic}, and the wrong-kind read fails. So far so good.

But look at what you've had to write. A phone's kernel carries dozens of device kinds, so the rule is now dozens of letters and twice as many implications, every one of them a copy of the same two sentences with a

[1] Catalogued as CWE-843, "Access of Resource Using Incompatible Type" (The MITRE Corporation, 2024). It is a staple of kernel exploits: a handle, a message or an object of one kind is fed to code written for another, and the second reads the first's bytes as a structure of its own, pointers and all.

different subscript. The same thing written forty times is wrong in one of them. Worse, the list changes: a driver is loaded for a new sensor, and the rule that was complete on Monday is short a kind on Tuesday.

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_{cam} and r_{cam} are two letters, as unrelated as p and r were. Nothing in his list of shapes says that the "cam" in the one is the "cam" in the other, and $p_{\text{cam}} \to r_{\text{mic}}$ is as well formed as any formula on the page. You have written the kind inside the letter, and inside the letter is where the clerk can't look.

What the kernel's design actually says is one sentence: *every kind of handle supports the same three operations.* Every kind, including the sensor whose driver loads on Tuesday: nothing so far can say that.

The word keeps coming up. A payment runbook says what happens on *every* payment, and is used for the one on a 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 `<=` 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 the chapter asks two questions:

- how do you say *every* and *some*, in a proof and in a type?
- what does saying it cost?

The language is Frege's (Section 4.1). The rules of proof are Gentzen's, four of them, and each is a phrase you've said aloud in an argument (Section 4.2). Two of the four carry a condition on a variable, and Section 4.3 shows what each condition prevents by running a proof without it.

Then the programs go back on (Section 4.4), and the quantifiers turn out to be two things you've programmed with for years: a proof of *for all* is a function whose result type mentions its argument, and a proof of *there is* is a pair of a thing and evidence about it. These are *dependent types*, and they are what the camera call needed. By then you'll know the trick well enough to guess the typing rules before I give them, and I'd like you to try. Section 4.5 extends the Curry–Howard correspondence to them, and normalisation then gives two facts: the logic proves no atomic sentence on its own, and a proof of *there is* names the thing, the *existence property.*

Section 4.6 ends where the chapter began, with sentences about things written down as theories and proved from, the driver theory among them. Section 4.7 puts this chapter's type system and Chapter 5's in one picture, the λ-cube, and Chapter 5 then points the same quantifier at propositions instead of things, and from there at types, which gives generics.

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

$$p_{\text{cam}} \quad \text{becomes} \quad \mathsf{Cap}(\mathsf{cam}),$$

"the holder has a camera capability", where Cap is a *predicate*, a claim with a hole in it, and cam is a name that fills the hole. Now $\mathsf{Cap}(\mathsf{cam})$ and $\mathsf{Read}(\mathsf{cam})$ visibly share a part, and the clerk, who compares symbols, can see that they do. The sentence $\mathsf{Cap}(\mathsf{cam}) \to \mathsf{Read}(\mathsf{mic})$ is still well formed, but it's now visibly a different sentence from $\mathsf{Cap}(\mathsf{cam}) \to \mathsf{Read}(\mathsf{cam})$, and anything that said the first would be seen to say it.

This is the move the *Begriffsschrift* of 1879 was built around. Logic before it analysed a sentence into a subject and a predicate, "the camera" and "is live", which is the grammar of Latin; the book analysed a sentence instead into a *function and an argument*, $\mathsf{Live}(\ )$ and cam, which is the grammar of mathematics. Its author 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 $\mathsf{Cap}(k) \to \mathsf{Hand}(k)$ is the rule with the kind left out, a claim about whatever kind k is, and it's one formula, not forty. On the type side the same move turns `Handle` from a type into a function from kinds to types, `Handle cam`, `Handle k`, a type with a hole for a kind.

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 the forty subscripted letters were trying to write. The first is to say that the claim holds *whatever* fills it. For the forty kinds that would be the conjunction

$$(\mathsf{Cap}(\mathsf{cam}) \to \mathsf{Hand}(\mathsf{cam})) \wedge (\mathsf{Cap}(\mathsf{mic}) \to \mathsf{Hand}(\mathsf{mic})) \wedge \cdots$$

with one conjunct per kind, 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":

$$\forall k\, (\mathsf{Cap}(k) \to \mathsf{Hand}(k)).$$

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. "Some driver has registered the camera" is the disjunction

$$\mathsf{Reg}(\mathsf{v4l}, \mathsf{cam}) \vee \mathsf{Reg}(\mathsf{usb}, \mathsf{cam}) \vee \cdots$$

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

$$\exists d\, \mathsf{Reg}(d, \mathsf{cam}).$$

That's *there is*, an *or* with one case for every thing, and the thing is the tag. Logicians call it the *witness*.[3]

---

[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.

[3] Quine put the point as a slogan in 1948: *to be is to be the value of a variable*. If you want to know what a theory says exists, he argued, don't look at its nouns; look at what its quantifiers range over. Philosophers have argued about the slogan ever since, which, for a philosopher, is a mark of success.

The quantifiers are $\wedge$ and $\vee$ with the list of cases made unbounded. Everything else about them (the proof rules, the side conditions, the programs) 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, the forty subscripted letters again, and it says nothing about input forty-one. 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.

Why it mattered. What has just happened is larger than it looks. For two thousand years logic meant the syllogism: *every A is B, every B is C, so every A is C*. Each sentence of a syllogism has one *every* or one *some*, at the front, and nothing in the system can put one inside another. Late in the eighteenth century the subject was declared finished.[4]

Here is an argument that every reader accepts and no syllogism can reach: *every horse is an animal, so every head of a horse is the head of an animal.*[5] The conclusion has a *some* inside an *every*: for every y, if y is the head of some horse, then y is the head of some animal. With predicates Horse, Animal and $\mathsf{HeadOf}(y, x)$, "y is the head of x", that is

$$\forall y \left(\exists x \left(\mathsf{Horse}(x) \wedge \mathsf{HeadOf}(y, x)\right) \to \exists x \left(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y, x)\right)\right),$$

and Example 4.23 derives it from $\forall x \left(\mathsf{Horse}(x) \to \mathsf{Animal}(x)\right)$ in nine bars. Quantifiers nest, and nesting was the whole difficulty.

Mathematics and software are made of nested sentences:

- *every number has a larger one*;
- *between any two fractions there is a third*;
- *every request gets a response*;
- *for every input there is an output.*

A logic that can't put a *some* inside an *every* can't state any of these, let alone check a proof of one.

The notation was built to say all of arithmetic, and it could. Within fifty years it had raised the question whether a machine could decide which formulas follow from which, and the machine invented in 1936 to answer it, the Turing machine, is our clerk.[6] And the query language of every relational database is the same logic in different clothes: an SQL query

[4] By Kant, in the preface to the second edition of the *Critique of Pure Reason* (1787): since Aristotle, logic "has been unable to advance a step and, thus, to all appearance has reached its completion". Ninety-two years later the *Begriffsschrift* appeared, a booklet of eighty-eight pages that van Heijenoort, who edited the standard collection of the papers that followed it, called "perhaps the most important single work ever written in logic" (van Heijenoort, 1967, p. 1). Philosophers who announce that a subject is finished have a poor record, and Kant's is among the worst.

[5] The example is traditionally credited to Augustus De Morgan, who in the 1840s collected inferences that the syllogism could not handle and argued that logic needed relations. He was right, and was largely ignored.

[6] Turing (1937).

with `EXISTS` and `NOT EXISTS` in it is a formula with quantifiers, written in another notation.

The 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 $\to$, $\wedge$ and $\vee$;
   - the constant $\bot$;
   - the quantifiers $\forall$ and $\exists$;
   - a stock of *variables* $x, y, z, \ldots$.
2. The *non-logical symbols*, which make up the *signature* and change from one language to the next:
   - some *constant symbols* $c, d, \ldots$;
   - some *function symbols* $f, g, \ldots$;
   - some *predicate symbols* $P, Q, \ldots$.

   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 $p, q, r, \ldots$.

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 $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ 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. $\bot$ is a formula.
2. If P is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula. (When $n = 0$ it is a propositional letter p, written without brackets.)
3. If A and B are formulas, then $(A \to B)$, $(A \wedge B)$ 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.

1. $\neg A$ abbreviates $A \to \bot$.
2. Familiar symbols such as $<$ and $+$ are written between their arguments, so that $x + 1 < y$ is the atomic formula $<(+(x, 1), y)$.

3. Brackets are left out where nothing is lost:
   - the outermost pair goes;
   - $\wedge$ and $\vee$ take precedence over $\to$;
   - the arrow groups to the right;
   - $\neg$ and the quantifiers take precedence over everything else.

   So $\forall x\, A \to B$ is $(\forall x\, A) \to B$, and a quantifier whose scope is meant to be larger is followed by brackets, as in $\forall x\, (A \to B)$.

With the conventions in force, the two definitions fit on two lines:

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

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.

The kernel's signature, which the chapter uses throughout, has

- constants for the resource kinds and the drivers, cam, mic, net, v4l;
- a function symbol owner(k), the driver that owns kind k;
- five predicates:

Predicate	**Reads**
Cap(k)	the holder has a capability for kind k
Hand(k)	the holder has an open handle to kind k
Read(k)	the holder may read from kind k
Reg(d, k)	driver d has registered kind k
Live(k)	kind k has a driver loaded

Kinds and drivers are things of one sort as far as the language goes. A variable ranges over both; the predicates say which is which. So Cap(v4l) 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, the kernel's design comes down to three sentences, which I'll call the *driver theory*, Γ_{pol} (a set of sentences used as hypotheses is a *theory*, and Section 4.6 defines it):

What the theory says	As a sentence
a capability for a kind opens a handle to it	$\forall k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k))$
a handle to a kind may be read	$\forall k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k))$
a registered kind has a driver loaded	$\forall d\,\forall k\,(\mathsf{Reg}(d,k) \to \mathsf{Live}(k))$

The open rule as the kernel states it, "a kind may be opened once some driver has registered it, and its owner has a capability for it", is $\exists d\,\mathsf{Reg}(d,k) \wedge \mathsf{Cap}(\mathsf{owner}(k))$. The term $\mathsf{owner}(k)$ names a driver without saying which; that is what a function symbol is for. The sentence is the kind a kernel team writes in a comment and would rather a machine checked.

Reading quantified sentences. The order of the quantifiers matters, and so does the connective under them. The sentence $\forall k\,\exists d\,\mathsf{Reg}(d,k)$ says that every kind has some driver registered for it, perhaps each a different one. The sentence $\exists d\,\forall k\,\mathsf{Reg}(d,k)$ says that one driver registered every kind. The symbols are the same and the claims are very different.

The connective under a quantifier follows three rules of thumb.

- A condition on *every* goes on the left of an arrow. The theory's first sentence is $\forall k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k))$; written with $\wedge$, as $\forall k\,(\mathsf{Cap}(k) \wedge \mathsf{Hand}(k))$, it would say that *every* kind is one the holder has a capability and a handle for.
- A description under *some* is joined with $\wedge$. Written with an arrow, $\exists k\,(A \to B)$ is derivable as soon as some kind is known to fail A, whatever B says.
- An *any* in the condition of a rule is an *every* over the whole rule: when x is not free in B, each of $\exists x\,A \to B$ and $\forall x\,(A \to B)$ derives the other, while $\exists x\,(A \to B)$ says much less.

A negation can cross a quantifier only at a price. In this logic $\neg\forall x\,A$ does not give $\exists x\,\neg A$: a proof of *there is* has to name the thing, and a proof that not everything is A needn't name anything. Part 2 comes back to the gap.

Turning English into formulas is where most mistakes with quantifiers are made, before a single rule has been applied. Five sentences from a driver writer's week show these points at work.

Example 4.4 (Ivanova: a rule about every kind)**.** Ivanova writes drivers for the phone maker of the opening, and her week is spent writing the kernel's rules down as formulas. The kernel's documentation says: *a kind with no driver loaded can't be read.* That is a rule about every kind, so its shape is *for every k, if … then …*:

$$\forall k\,(\neg\mathsf{Live}(k) \to \neg\mathsf{Read}(k)).$$

The easy mistake is to write $\wedge$ where the sentence has *if*. The formula $\forall k\,(\neg\mathsf{Live}(k) \wedge \neg\mathsf{Read}(k))$ says that no kind has a driver loaded and none can be read, which describes a phone that's switched off.

Example 4.5 (Ivanova, cont'd: a report about some kind)**.** The researcher's report says: *some kind can be read by an app that holds no capability for it.* That's a claim that a particular thing exists, so its shape is *there is a k such*

that ... and ...:

$$\exists k\,(\mathsf{Read}(k) \wedge \neg \mathsf{Cap}(k)).$$

Here the easy mistake runs the other way: $\exists k\,(\neg \mathsf{Cap}(k) \to \mathsf{Read}(k))$. That formula follows from $\mathsf{Cap}(\mathsf{cam})$ alone.

1. Suppose the app has no camera capability.
2. That contradicts $\mathsf{Cap}(\mathsf{cam})$, and from a contradiction anything follows, $\mathsf{Read}(\mathsf{cam})$ included.
3. So $\neg \mathsf{Cap}(\mathsf{cam}) \to \mathsf{Read}(\mathsf{cam})$, and the camera is the witness.

The formula would be established by an app that holds a camera capability and reads nothing at all.

Example 4.6 (Ivanova, cont'd: a named driver and an unnamed one)**.** Two engineers summarise the boot sequence. One writes *every kind is registered by its owner*; the other, *every kind is registered by some driver*. The first uses the function symbol and the second a quantifier:

$$\forall k\,\mathsf{Reg}(\mathsf{owner}(k), k) \qquad \text{and} \qquad \forall k\,\exists d\,\mathsf{Reg}(d, k).$$

They aren't the same claim. The first names the driver for each kind, by a rule anyone can apply: hand it the camera and it says $\mathsf{owner}(\mathsf{cam})$. The second says only that each kind has a driver, perhaps a different one each time, and gives no way of finding it.

The first gives the second, as Example 4.20 below shows. Nothing in the second says that the driver it promises is the one the symbol owner names, so it doesn't give the first.

Example 4.7 (Ivanova, cont'd: where the *not* goes)**.** Two lines from a security review of an untrusted app: *not every kind may be read*, and *no kind may be read*. They are

$$\neg\forall k\,\mathsf{Read}(k) \qquad \text{and} \qquad \forall k\,\neg\mathsf{Read}(k).$$

The first is a mild complaint: some kinds are protected, perhaps only one. The second is a sandbox: every kind is protected. Moving the $\neg$ across the $\forall$ changes the claim, and the second is much the stronger. The sandbox can also be written $\neg\exists k\,\mathsf{Read}(k)$, *there's no kind it may read*, and Example 4.21 below gets the per-kind form from that one.

Example 4.8 (Ivanova, cont'd: a *some* that means *every*)**.** The incident runbook says: *if any kind can be read without a capability, page the security team.* Write s for "the security team is paged", and $\mathsf{Bad}(k)$ as short for $\mathsf{Read}(k) \wedge \neg \mathsf{Cap}(k)$, *$k$ can be read without a capability* (an abbreviation, not a new predicate). The obvious formula is

$$\exists k\,\mathsf{Bad}(k) \to s.$$

The monitoring code checks one kind at a time, so it implements a rule that looks different: *for every kind, if it can be read without a capability, page the team*,

$$\forall k\,(\mathsf{Bad}(k) \to s).$$

By the third rule of thumb the two say the same thing, and Example 4.22 below derives each from the other. What doesn't say the same thing is $\exists k\,(\mathsf{Bad}(k) \to s)$, with the quantifier moved over the arrow as well. That one

would be established by any single kind known not to be readable, whether or not anyone is ever paged.

Definition 4.9 (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 $\mathrm{var}(t)$ of variables of a term t is defined by recursion on the shape of t,

$$\mathrm{var}(t) \coloneqq \begin{cases} \{x\} & \text{if } t \text{ is a variable } x, \\ \varnothing & \text{if } t \text{ is a constant } c, \\ \mathrm{var}(t_1) \cup \cdots \cup \mathrm{var}(t_n) & \text{if } t \text{ is } f(t_1, \ldots, t_n), \end{cases}$$

and the set $\mathrm{FV}(A)$ of *free variables* of a formula A by recursion on the shape of A, with $\circ$ standing for any of $\to$, $\wedge$ and $\vee$, and Q for either quantifier:

$$\mathrm{FV}(A) \coloneqq \begin{cases} \varnothing & \text{if } A \text{ is } \bot, \\ \mathrm{var}(t_1) \cup \cdots \cup \mathrm{var}(t_n) & \text{if } A \text{ is } P(t_1, \ldots, t_n), \\ \mathrm{FV}(B) \cup \mathrm{FV}(C) & \text{if } A \text{ is } B \circ C, \\ \mathrm{FV}(B) \setminus \{x\} & \text{if } A \text{ is } Qx\, B. \end{cases}$$

Definition 4.10 (sentence)**.** 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 $\forall x\, (x < y) \to \exists y\, (x < y)$ 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 theory of every kind rather than a remark about one. The open rule is not: its k is free, and it becomes a claim about a particular kind only when a term is put in for k. That operation needs care, because a quantifier inside the formula can capture a variable of the term put in.

Definition 4.11 (substitution)**.** Let t be a term and x a variable. For a term s, the term $s[t/x]$, "s with t for x", is defined by recursion on the shape of s:

$$s[t/x] \coloneqq \begin{cases} t & \text{if } s \text{ is } x, \\ s & \text{if } s \text{ is a variable } y \neq x \text{ or a constant}, \\ f(s_1[t/x], \ldots, s_n[t/x]) & \text{if } s \text{ is } f(s_1, \ldots, s_n). \end{cases}$$

For a formula A, the formula $A[t/x]$ is defined by recursion on the shape of A:

$$A[t/x] \coloneqq \begin{cases} \bot & \text{if } A \text{ is } \bot, \\ P(t_1[t/x], \ldots, t_n[t/x]) & \text{if } A \text{ is } P(t_1, \ldots, t_n), \\ B[t/x] \circ C[t/x] & \text{if } A \text{ is } B \circ C, \\ Qx\, B & \text{if } A \text{ is } Qx\, B, \\ Qy\, B[t/x] & \text{if } A \text{ is } Qy\, B \text{ with } y \neq x. \end{cases}$$

Definition 4.12 (substitutable)**.** A term t is *substitutable* for x in a formula A when no free occurrence of x in A lies in the scope of a quantifier on a variable of t.

We write $A[t/x]$ only when t is substitutable for x in A. So $A[t/x]$ puts t in for the *free* occurrences of x, and only those: the fourth line of the last display stops at a quantifier on x, because every x in its scope is bound. Substitutability is what makes the last line 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 cam 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 $\bot$.

Remark (looking back: substitution in the λ-calculus). The lines are those of Definition 3.18, substitution in λ-terms, with a quantifier in the place of a λ, and the last line for FV is the one for an abstraction, $\mathrm{FV}(\lambda x.\, M) = \mathrm{FV}(M) \setminus \{x\}$. 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. **NJ with quantifiers.** The rules for the connectives came from listening to how people argue: *suppose*, *so*, *either way*. The quantifier rules came the same way, and you've said all four. Here is someone showing that every kind with a capability may be read: "Let u be any kind, and suppose a capability for u. Then there's a handle to u, so u may be read. And u was any kind." Here is someone using the register rule: "Some driver registered the camera; call it y. Then by the rule, at y, the camera is live." Four phrases carry the quantifiers in those two arguments, and the four phrases are the four rules.

Said aloud	**From**	**To**
let y be arbitrary	$A[y/x]$, nothing assumed about y	$\forall x\, A$
in particular, for t	$\forall x\, A$	$A[t/x]$
take x to be t	$A[t/x]$	$\exists x\, A$
let y be such a thing; then …	$\exists x\, A$; and C, supposing $A[y/x]$	C

The first and the last come with a condition, and in each case the condition is what the phrase means. *Arbitrary* means that nothing assumed says anything about y. *Such a thing* means that y is a name made up for the duration of the argument, so the conclusion C, which outlives the argument, mustn't mention it. Those two conditions are the chapter's other subject, and the next section takes them away to see what breaks.

You've used all four rules 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 driver registered this kind" and works with whichever it was, without being told which, is *let y be such a thing*.

Introduction (to prove it)	**Elimination** (to use it)
$\dfrac{A[y/x]}{\forall x\, A}\ \forall\mathrm{I}$ *y free in no open hypothesis, nor in* $\forall x\, A$	$\dfrac{\forall x\, A}{A[t/x]}\ \forall\mathrm{E}$
$\dfrac{A[t/x]}{\exists x\, A}\ \exists\mathrm{I}$	$\dfrac{\exists x\, A \qquad \begin{array}{c}[A[y/x]]\\ \vdots\\ C\end{array}}{C}\ \exists\mathrm{E}$ *y free in no other open hypothesis, nor in* C *or* $\exists x\, A$

FIGURE 4.1. The quantifier rules of NJ.

NJ was first-order from the day it was invented, in 1934: the propositional rules are the half of the system without quantifiers, and the four rules below are the other half.[7]

Written as bars, in the two columns of Figure 4.1, the four phrases are four rules.

- *($\forall$E). In particular*: from $\forall x\, A$, conclude A with any term t put in for x.
- *($\exists$I). Take x to be t*: from A with t put in, conclude $\exists x\, A$.
- *($\forall$I). Let y be arbitrary*: from A with a variable y put in, conclude $\forall x\, A$, provided nothing still assumed mentions y.
- *($\exists$E). Let y be such a thing*: to get a conclusion C out of $\exists x\, A$, get it from the supposition that A holds of a new name y, which C must not mention. Like ($\vee$E), it discharges its supposition, and it is ($\vee$E) with one case for every thing there is.

One tree before the definition, to read aloud rather than check: the argument this section began with, that a capability for any kind gives the right to read it, from the first two sentences of the driver theory. Its first piece is needed again and again in this chapter, so it gets a name. For a term t, the *handle tree* $\mathcal{H}(t)$ gets a handle to t from a capability for t:

$$\mathcal{H}(t) \;=\; \dfrac{\dfrac{\forall k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k))}{\mathsf{Cap}(t) \to \mathsf{Hand}(t)}\ \forall\mathrm{E} \qquad \mathsf{Cap}(t)}{\mathsf{Hand}(t)}\ \to\mathrm{E}$$

When a bar further down discharges its leaf $\mathsf{Cap}(t)$, the leaf is written in brackets above the name, with the number of that bar. The argument uses

[7] Gentzen (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. Peano, who gave logic $\exists$ and $\in$, also believed that mathematicians should share one language, and that it should be Latin with the grammar taken out. He invented it, called it *Latino sine flexione*, and wrote the last edition of his great compendium of mathematics, the *Formulario Mathematico* of 1908, in it. The symbols caught on; the Latin did not.

$\mathcal{H}(u)$, with its leaf discharged by the bar marked 1:

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k))}{\mathsf{Hand}(u) \to \mathsf{Read}(u)}\,\forall\mathrm{E} \qquad \begin{matrix}[\mathsf{Cap}(u)]^1 \\ \mathcal{H}(u) \\ \mathsf{Hand}(u)\end{matrix}}{\mathsf{Read}(u)}\,{\to}\mathrm{E}}{\mathsf{Cap}(u) \to \mathsf{Read}(u)}\,{\to}\mathrm{I},1}{\forall k\,(\mathsf{Cap}(k) \to \mathsf{Read}(k))}\,\forall\mathrm{I}$$

1. Let u be any kind, and suppose a capability for it.
2. The first sentence, *in particular* for u, gives a handle.
3. The second, in particular for u, gives the right to read.
4. Discharge the supposition: if a capability for u, then u may be read.
5. The kind u was arbitrary: the leaves still open are two sentences of the theory, and neither mentions u, so generalise.

Here is the whole system as trees; the definition says in general what this tree did.

Definition 4.13 (derivations, with quantifiers)**.** The *derivations* are defined inductively; each has a *conclusion*, the formula at its root. 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 that of Definition 4.11. Two clauses refer to the open hypotheses $\mathrm{OH}(\mathcal{D})$ of a tree, which are computed from its shape alone, by cases on its last rule; the next definition gives the computation.

1. (*hypothesis*) For every formula A, the one-node tree whose only node is labelled A is a derivation, with conclusion A.

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

$$\dfrac{\begin{matrix}[A] \\ \mathcal{D} \\ B\end{matrix}}{A \to B}\,{\to}\mathrm{I} \qquad \text{is a derivation.}$$

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

$$\dfrac{\begin{matrix}\mathcal{D}_1 \\ A \to B\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ A\end{matrix}}{B}\,{\to}\mathrm{E} \qquad \text{is a derivation.}$$

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

$$\dfrac{\begin{matrix}\mathcal{D}_1 \\ A\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ B\end{matrix}}{A \wedge B}\,\wedge\mathrm{I} \qquad \text{is a derivation.}$$

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

$$\dfrac{\begin{matrix}\mathcal{D} \\ A \wedge B\end{matrix}}{A}\,\wedge\mathrm{E}_1 \qquad \text{and} \qquad \dfrac{\begin{matrix}\mathcal{D} \\ A \wedge B\end{matrix}}{B}\,\wedge\mathrm{E}_2$$

are derivations.

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 $\vee\mathrm{I}_1$, and symmetrically from a derivation of B, labelled $\vee\mathrm{I}_2$.

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

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

is a derivation.

8. ($\bot$E) If $\mathcal{D}$ concludes $\bot$, then for every formula C the tree with root C and immediate subtree $\mathcal{D}$ is a derivation.

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

$$\dfrac{\begin{array}{c}\mathcal{D} \\ \forall x\, A\end{array}}{A[t/x]}\ \forall\mathrm{E} \qquad \text{is a derivation.}$$

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 $\mathrm{OH}(\mathcal{D})$,

then

$$\dfrac{\begin{array}{c}\mathcal{D} \\ A[y/x]\end{array}}{\forall x\, A}\ \forall\mathrm{I} \qquad \text{is a derivation.}$$

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

$$\dfrac{\begin{array}{c}\mathcal{D} \\ A[t/x]\end{array}}{\exists x\, A}\ \exists\mathrm{I} \qquad \text{is a derivation.}$$

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 $\mathrm{OH}(\mathcal{D}_1) \setminus \{A[y/x]\}$,

then

$$\dfrac{\begin{array}{ccc}\mathcal{D}_0 & [A[y/x]] \\ \exists x\, A & \mathcal{D}_1 \\ & C\end{array}}{C}\ \exists\mathrm{E}$$

is a derivation.

13. Nothing else is a derivation.

The variable y of ($\forall$I) and ($\exists$E) is the *eigenvariable* of the step.

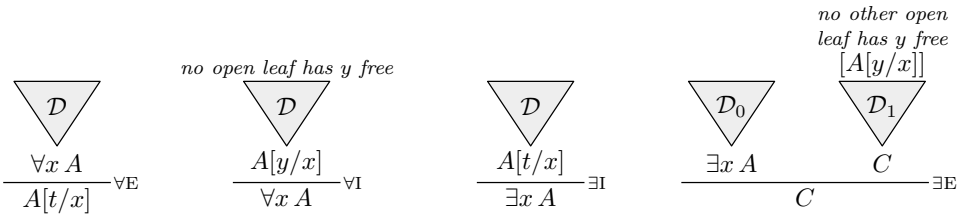


FIGURE 4.2. The quantifier rules as ways of combining trees.

Definition 4.14 (open hypotheses, with quantifiers)**.** The set $\mathrm{OH}(\mathcal{D})$ of *open hypotheses* of a derivation $\mathcal{D}$ is defined by cases on its last rule:

$$\mathrm{OH}(\mathcal{D}) \coloneqq \begin{cases} \{A\} & \text{a leaf } A, \\ \mathrm{OH}(\mathcal{D}') \setminus \{A\} & (\to\mathrm{I}) \text{, discharging } A, \\ \mathrm{OH}(\mathcal{D}_1) \cup \mathrm{OH}(\mathcal{D}_2) & (\to\mathrm{E}),\ (\wedge\mathrm{I}), \\ \mathrm{OH}(\mathcal{D}') & (\wedge\mathrm{E}),\ (\vee\mathrm{I}),\ (\bot\mathrm{E}), \\ & (\forall\mathrm{E}),\ (\forall\mathrm{I}),\ (\exists\mathrm{I}), \\ \mathrm{OH}(\mathcal{D}_0) \cup (\mathrm{OH}(\mathcal{D}_1) \setminus \{A\}) & (\vee\mathrm{E}) \text{, discharging } A,\ B, \\ \quad \cup\, (\mathrm{OH}(\mathcal{D}_2) \setminus \{B\}) & \\ \mathrm{OH}(\mathcal{D}_0) \cup (\mathrm{OH}(\mathcal{D}_1) \setminus \{A[y/x]\}) & (\exists\mathrm{E}) \text{, discharging } A[y/x], \end{cases}$$

where $\mathcal{D}'$ is the one subderivation of a one-premiss rule, $\mathcal{D}_1$, $\mathcal{D}_2$ the two of a two-premiss rule, and $\mathcal{D}_0$, $\mathcal{D}_1$, $\mathcal{D}_2$ those of $(\vee\mathrm{E})$ and $(\exists\mathrm{E})$, left to right.

Definition 4.15 (derivability, with quantifiers)**.** We write $\Gamma \vdash_{\mathrm{NJ}} A$, and say that A is *derivable from* Γ *in NJ*, when there is a derivation $\mathcal{D}$ whose conclusion is A and whose open hypotheses are a subset of Γ, $\mathrm{OH}(\mathcal{D}) \subseteq \Gamma$; such a derivation *witnesses* $\Gamma \vdash_{\mathrm{NJ}} A$.

The four new clauses combine trees as the old ones did (Figure 4.2). $(\forall\mathrm{E})$, $(\forall\mathrm{I})$ and $(\exists\mathrm{I})$ each take one tree and draw a new conclusion beneath it. $(\exists\mathrm{E})$ is the shape of $(\vee\mathrm{E})$ with a single case: a tree for $\exists x\, A$, and a tree from $A[y/x]$ to C, combine into one tree for C, discharging $A[y/x]$ in the second.

What is new is that two of the four come with an eigenvariable condition, and both conditions are read off the trees: $(\forall\mathrm{I})$ looks at the open leaves of its one tree, and $(\exists\mathrm{E})$ at the open leaves of its second tree, apart from those it discharges. The notes on the triangles in the figure are these conditions. The clauses ask two more things, which the figure leaves out: that y not be free in $\forall x\, A$ (for $\forall\mathrm{I}$), or in C or $\exists x\, A$ (for $\exists\mathrm{E}$), and that the term put in, t or y, be substitutable for x in A.

Three clauses now discharge: $(\to\mathrm{I})$, $(\vee\mathrm{E})$ and $(\exists\mathrm{E})$, under the conventions already in force: a bar takes every open copy of its formula, perhaps none, and a leaf is discharged by the first bar below it that discharges its formula from the part of the tree the leaf stands in. For $(\exists\mathrm{E})$ that part is the second premiss: $A[y/x]$ leaves the second premiss only, and nothing in the first is discharged.

The two eigenvariable clauses are the two conditions of the table of phrases, made exact. For (∀I), *arbitrary* comes to two checks: y is free in no open hypothesis, which is read off the leaves of the tree, and y isn't free in $\forall x\, A$, so the formula being generalised doesn't already talk about it. For (∃E), *such a thing* comes to three: the made-up name y is free neither in the conclusion C, which outlives it, nor in $\exists x\, A$, nor in any other open hypothesis of the branch, so that nothing already in play is quietly identified with it.

Set the two clauses beside the turnstile, writing Δ for the open hypotheses of the tree above the bar, and the conditions read as one: nothing to the left of the turnstile mentions y. On the left is the tree; on the right, what it witnesses:

$$\dfrac{\begin{array}{c}\mathcal{D}\\ A[y/x]\end{array}}{\forall x\, A}\ \forall\mathrm{I} \qquad\qquad \dfrac{\Delta \vdash_{\mathrm{NJ}} A[y/x]}{\Delta \vdash_{\mathrm{NJ}} \forall x\, A}$$

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

$$\dfrac{\begin{array}{ccc}\mathcal{D}_0 & & [A[y/x]]\\ \exists x\, A & & \mathcal{D}_1\\ & & C\end{array}}{C}\ \exists\mathrm{E} \qquad\qquad \dfrac{\Delta_0 \vdash_{\mathrm{NJ}} \exists x\, A \qquad \Delta_1,\ A[y/x] \vdash_{\mathrm{NJ}} C}{\Delta_0,\ \Delta_1 \vdash_{\mathrm{NJ}} C}$$

with y free in none of C, $\exists x\, A$ and the formulas of Δ_1.

One consequence is the order of the bars. (∀I) looks at the leaves still open, so a supposition that mentions y must be discharged before the tree generalises over y.

Worked derivations. Quantifier proofs look hard on the page, and most of the difficulty is in seeing which rule to use next. So here are seven questions about the driver theory, and the head of a horse, each done in the same order: the situation, the claim, the reasoning in words, and the tree. The hypotheses are mostly the driver theory Γ_{pol}. How trees like these are found comes after normalisation, which says what shape to look for.

Example 4.16 (Ivanova, cont'd: the boot log)**.** The boot log has the line `v4l: registered camera`. Is the camera live? The register rule says that any kind any driver registers is live, so yes. The claim is

$$\forall d\,\forall k\,(\mathsf{Reg}(d,k) \to \mathsf{Live}(k)),\ \mathsf{Reg}(\mathsf{v4l},\mathsf{cam}) \ \vdash_{\mathrm{NJ}}\ \mathsf{Live}(\mathsf{cam}).$$

In words:

1. the rule holds for every driver, in particular for `v4l`; and
2. for every kind, in particular the camera; and
3. the log says its condition is met.

$$\dfrac{\dfrac{\dfrac{\forall d\,\forall k\,(\mathsf{Reg}(d,k) \to \mathsf{Live}(k))}{\forall k\,(\mathsf{Reg}(\mathsf{v4l},k) \to \mathsf{Live}(k))}\ \forall\mathrm{E}}{\mathsf{Reg}(\mathsf{v4l},\mathsf{cam}) \to \mathsf{Live}(\mathsf{cam})}\ \forall\mathrm{E} \qquad \mathsf{Reg}(\mathsf{v4l},\mathsf{cam})}{\mathsf{Live}(\mathsf{cam})}\ \to\mathrm{E}$$

The two (∀E) bars come off in the order the quantifiers were written, outermost first: $t := \mathsf{v4l}$ for d, then $t := \mathsf{cam}$ for k. The open hypotheses are the rule and the log line, so the tree witnesses the claim.

Example 4.17 (Ivanova, cont'd: one rule for the documentation)**.** Ivanova wants a single sentence for the driver guide: *a capability for a kind gives you a handle to it and the right to read it.* Does the theory support it? The claim is $\Gamma_{\text{pol}} \vdash_{\text{NJ}} \forall k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k) \wedge \mathsf{Read}(k))$.

In words:

1. Let u be any kind, and suppose a capability for it.
2. Then there's a handle, by the theory's first sentence, and so the right to read, by its second. That's both halves.
3. Discharge the supposition, and since u was any kind, generalise.

The handle is needed twice, once for itself and once for the read, so the tree uses the handle tree $\mathcal{H}(u)$ twice:

$$\dfrac{\dfrac{\dfrac{[\mathsf{Cap}(u)]^1 \quad \dfrac{\dfrac{\forall k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k))}{\mathsf{Hand}(u) \to \mathsf{Read}(u)}\,\forall\mathrm{E} \quad \begin{array}{c}[\mathsf{Cap}(u)]^1\\ \mathcal{H}(u)\\ \mathsf{Hand}(u)\end{array}}{\mathsf{Read}(u)}\,{\to}\mathrm{E}}{\mathsf{Hand}(u) \wedge \mathsf{Read}(u)}\,\wedge\mathrm{I}}{\mathsf{Cap}(u) \to \mathsf{Hand}(u) \wedge \mathsf{Read}(u)}\,{\to}\mathrm{I},1}{\forall k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k) \wedge \mathsf{Read}(k))}\,\forall\mathrm{I}$$

The bar marked 1 discharges both copies of $[\mathsf{Cap}(u)]^1$ at once. After it, OH is the theory's first two sentences, in which u isn't free, so the ($\forall$I) bar is legal and the tree witnesses the claim.

Example 4.18 (Ivanova, cont'd: can it read anything?)**.** A reviewer of the camera app asks a blunt question: can it read anything at all? The app holds a camera capability, so yes, and a good answer says what.

The claim is $\Gamma_{\text{pol}},\ \mathsf{Cap}(\mathsf{cam}) \vdash_{\text{NJ}} \exists k\,\mathsf{Read}(k)$. In words:

1. with the capability the camera has a handle,
2. so the camera can be read,
3. so some kind can be read, namely the camera.

The tree uses the handle tree $\mathcal{H}(\mathsf{cam})$, whose leaf $\mathsf{Cap}(\mathsf{cam})$ stays open:

$$\dfrac{\dfrac{\dfrac{\forall k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k))}{\mathsf{Hand}(\mathsf{cam}) \to \mathsf{Read}(\mathsf{cam})}\,\forall\mathrm{E} \quad \begin{array}{c}\mathcal{H}(\mathsf{cam})\\ \mathsf{Hand}(\mathsf{cam})\end{array}}{\mathsf{Read}(\mathsf{cam})}\,{\to}\mathrm{E}}{\exists k\,\mathsf{Read}(k)}\,\exists\mathrm{I}$$

The last bar takes $t := \mathsf{cam}$, since $\mathsf{Read}(\mathsf{cam})$ is $\mathsf{Read}(k)$ with cam put in for k. Ask the proof *which kind?* and it tells you: the witness is written on that bar. The open hypotheses are two of the theory's sentences and the capability, so the tree witnesses the claim.

Example 4.19 (Ivanova, cont'd: a capability for something)**.** Another app's install manifest has been cut short. It records that the app was granted a capability for some kind, and the line saying which is missing. Can the app read anything? Yes: whatever kind it was, the theory lets it read that one.

The claim is $\Gamma_{\text{pol}},\ \exists k\,\mathsf{Cap}(k) \vdash_{\text{NJ}} \exists k\,\mathsf{Read}(k)$. In words:

1. Call the kind y, so that $\mathsf{Cap}(y)$.

2. Then y has a handle and can be read, so some kind can be read.
3. That conclusion doesn't mention y, so it follows whichever kind y was.

Here is the branch in which y is supposed, which I'll call $\mathcal{B}$; it uses the handle tree $\mathcal{H}(y)$:

$$\mathcal{B} \;=\; \dfrac{\dfrac{\dfrac{\forall k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k))}{\mathsf{Hand}(y) \to \mathsf{Read}(y)}\,\forall\mathrm{E} \qquad \begin{matrix}[\mathsf{Cap}(y)]^1 \\ \mathcal{H}(y) \\ \mathsf{Hand}(y)\end{matrix}}{\mathsf{Read}(y)}\,{\to}\mathrm{E}}{\exists k\,\mathsf{Read}(k)}\,\exists\mathrm{I}$$

and here is the whole:

$$\dfrac{\exists k\,\mathsf{Cap}(k) \qquad \begin{matrix}[\mathsf{Cap}(y)]^1 \\ \mathcal{B} \\ \exists k\,\mathsf{Read}(k)\end{matrix}}{\exists k\,\mathsf{Read}(k)}\,\exists\mathrm{E},1$$

Check the three conditions on the ($\exists$E) bar: y is free in none of

- the conclusion $\exists k\,\mathsf{Read}(k)$,
- $\exists k\,\mathsf{Cap}(k)$, and
- the theory's two sentences, the other open hypotheses of $\mathcal{B}$.

The open hypotheses of the whole tree are those two sentences and $\exists k\,\mathsf{Cap}(k)$, so it witnesses the claim. The argument works with the kind it was given a name for and offers the same kind back as the witness, without ever learning which kind it was.

Example 4.20 (Ivanova, cont'd: every kind has a driver)**.** Back to the two summaries of Example 4.6. Does *every kind is registered by its owner* give *every kind is registered by some driver*? Yes: for any kind, its owner is such a driver.

The claim is $\forall k\,\mathsf{Reg}(\mathsf{owner}(k), k) \vdash_{\mathrm{NJ}} \forall k\,\exists d\,\mathsf{Reg}(d, k)$. In words:

1. Let u be any kind.
2. Its owner registered it, so some driver registered it, namely $\mathsf{owner}(u)$; and
3. The kind u was arbitrary.

$$\dfrac{\dfrac{\dfrac{\forall k\,\mathsf{Reg}(\mathsf{owner}(k), k)}{\mathsf{Reg}(\mathsf{owner}(u), u)}\,\forall\mathrm{E}}{\exists d\,\mathsf{Reg}(d, u)}\,\exists\mathrm{I}}{\forall k\,\exists d\,\mathsf{Reg}(d, k)}\,\forall\mathrm{I}$$

The ($\exists$I) bar takes $t := \mathsf{owner}(u)$. A witness needn't be a constant; here it's a term built from the variable. The ($\forall$I) bar is legal because the one open hypothesis is a sentence, and that hypothesis is the claim's, so the tree witnesses it.

Example 4.21 (Ivanova, cont'd: the sandbox)**.** An untrusted app runs in a sandbox whose policy is one sentence: *there's no kind it may read*, $\neg\exists k\,\mathsf{Read}(k)$. The kernel enforces the sandbox one kind at a time: each time the app asks for a kind, it checks that the app may not read that one. Does the policy license the check, whatever kind is asked for?

The claim is $\neg\exists k\,\mathsf{Read}(k) \vdash_{\mathrm{NJ}} \forall k\,\neg\mathsf{Read}(k)$. In words:

1. Let u be any kind, and suppose the app may read u.
2. Then there's a kind it may read, which the policy denies.
3. So it may not read u; and u was any kind.

$$\dfrac{\dfrac{\dfrac{\neg\exists k\,\mathsf{Read}(k) \qquad \dfrac{[\mathsf{Read}(u)]^1}{\exists k\,\mathsf{Read}(k)}\,\exists\mathrm{I}}{\bot}\,{\to}\mathrm{E}}{\neg\mathsf{Read}(u)}\,{\to}\mathrm{I},1}{\forall k\,\neg\mathsf{Read}(k)}\,\forall\mathrm{I}$$

Remember that $\neg A$ is $A \to \bot$, so the ($\to$E) bar applies the policy to a witness, and the ($\to$I) bar makes a $\neg$. The one open hypothesis is the policy, so the tree witnesses the claim. The other direction, from $\forall k\,\neg\mathsf{Read}(k)$ to $\neg\exists k\,\mathsf{Read}(k)$, needs ($\exists$E), and is part 2 of Exercise 4.6 with its hypotheses in the other order.

Example 4.22 (Ivanova, cont'd: the pager)**.** Example 4.8 claimed that the runbook's rule and the monitoring code's rule say the same thing. First, the runbook gives the code's rule: $\exists k\,\mathsf{Bad}(k) \to s \vdash_{\mathrm{NJ}} \forall k\,(\mathsf{Bad}(k) \to s)$. In words:

1. Let u be any kind, and suppose it can be read without a capability.
2. Then some kind can,
3. so the runbook says page.

$$\dfrac{\dfrac{\dfrac{\exists k\,\mathsf{Bad}(k) \to s \qquad \dfrac{[\mathsf{Bad}(u)]^1}{\exists k\,\mathsf{Bad}(k)}\,\exists\mathrm{I}}{s}\,{\to}\mathrm{E}}{\mathsf{Bad}(u) \to s}\,{\to}\mathrm{I},1}{\forall k\,(\mathsf{Bad}(k) \to s)}\,\forall\mathrm{I}$$

That's the sandbox tree again, with s where it had $\bot$, since $\neg\exists k\,\mathsf{Read}(k)$ is $\exists k\,\mathsf{Read}(k) \to \bot$; its one open hypothesis is the runbook's rule.

Second, the code's rule gives the runbook's: $\forall k\,(\mathsf{Bad}(k) \to s) \vdash_{\mathrm{NJ}} \exists k\,\mathsf{Bad}(k) \to s$. In words:

1. Suppose some kind can be read without a capability, and call it y.
2. The code's rule at y says page.
3. Paging doesn't mention y, so it follows whichever kind it was.

$$\dfrac{\dfrac{[\exists k\,\mathsf{Bad}(k)]^2 \qquad \dfrac{\dfrac{\forall k\,(\mathsf{Bad}(k) \to s)}{\mathsf{Bad}(y) \to s}\,\forall\mathrm{E} \quad [\mathsf{Bad}(y)]^1}{s}\,{\to}\mathrm{E}}{s}\,\exists\mathrm{E},1}{\exists k\,\mathsf{Bad}(k) \to s}\,{\to}\mathrm{I},2$$

The ($\exists$E) bar is legal because y is free in none of s, $\exists k\,\mathsf{Bad}(k)$ and the code's rule, and once the supposition is discharged the code's rule is the one open hypothesis. The two trees together are an equivalence.

Example 4.23 (the head of a horse)**.** Last, the argument that two thousand years of syllogisms couldn't make: *every horse is an animal, so every*

head of a horse is the head of an animal. The claim is that the hypothesis $\forall x\,(\mathsf{Horse}(x) \to \mathsf{Animal}(x))$ derives

$$\forall y\,(\exists x\,(\mathsf{Horse}(x) \wedge \mathsf{HeadOf}(y,x)) \to \exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x))).$$

Read the conclusion first, as the habit from the propositional case says.

1. It starts with $\forall$, so the argument begins *let y be any thing.*
2. Under that is an arrow, so *suppose y is the head of some horse.*
3. That supposition starts with $\exists$, so open it: *call the horse z.*
4. Then z is an animal, because every horse is, and y is still its head, so y is the head of some animal, namely z.

Write $H(z)$ for $\mathsf{Horse}(z) \wedge \mathsf{HeadOf}(y,z)$, the supposition about z. The middle of the argument, in which z is known, I'll call $\mathcal{Z}$. It comes in two pieces, to fit the page. The first, $\mathcal{A}$, shows that z is an animal:

$$\mathcal{A} \;=\; \dfrac{\dfrac{\forall x\,(\mathsf{Horse}(x) \to \mathsf{Animal}(x))}{\mathsf{Horse}(z) \to \mathsf{Animal}(z)}\,\forall\mathrm{E} \qquad \dfrac{[H(z)]^1}{\mathsf{Horse}(z)}\,\wedge\mathrm{E}_1}{\mathsf{Animal}(z)}\,\to\mathrm{E}$$

and the second uses it:

$$\mathcal{Z} \;=\; \dfrac{\dfrac{\mathcal{A}}{\mathsf{Animal}(z)} \qquad \dfrac{[H(z)]^1}{\mathsf{HeadOf}(y,z)}\,\wedge\mathrm{E}_2}{\mathsf{Animal}(z) \wedge \mathsf{HeadOf}(y,z)}\,\wedge\mathrm{I}}{\exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x))}\,\exists\mathrm{I}$$

The whole argument wraps it in three more bars, one for each of *call the horse z*, *suppose* and *let y be any thing*, again in two pieces. The first bar, for *call the horse z*, makes a tree I'll call $\mathcal{W}$:

$$\dfrac{[\exists x\,(\mathsf{Horse}(x) \wedge \mathsf{HeadOf}(y,x))]^2 \qquad \overset{\mathcal{Z}}{\exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x))}}{\exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x))}\,\exists\mathrm{E},1$$

and the other two finish the argument:

$$\dfrac{\dfrac{\overset{\mathcal{W}}{\exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x))}}{\exists x\,(\mathsf{Horse}(x) \wedge \mathsf{HeadOf}(y,x)) \to \exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x))}\,\to\mathrm{I},2}{\forall y\,(\exists x\,(\mathsf{Horse}(x) \wedge \mathsf{HeadOf}(y,x)) \to \exists x\,(\mathsf{Animal}(x) \wedge \mathsf{HeadOf}(y,x)))}\,\forall\mathrm{I}$$

The ($\exists$E) bar is legal because z appears in neither its conclusion, nor $\exists x\,(\mathsf{Horse}(x) \wedge \mathsf{HeadOf}(y,x))$, nor the hypothesis about horses. The ($\forall$I) bar is legal because, once both suppositions are discharged, the only open hypothesis is that sentence, in which y isn't free. Nine bars, and every one of them is a rule of this chapter.

Checking a tree. Checking a tree takes the four checks of the propositional case, with ($\exists$E) added to the bars that discharge, and one new check for the quantifiers.

1. Every bar has premisses of the shape its clause requires.

2. Every bracket carries a number, and the bar with that number is the first bar below the leaf, on its path to the root, that discharges its formula from the part of the tree the leaf stands in.
3. An ($\to$I) bar discharges every open copy of its formula above it, and is vacuous only if there is none. An ($\vee$E) bar discharges the left side in its second premiss and the right side in its third; an ($\exists$E) bar discharges $A[y/x]$ in its second premiss; neither discharges anything in its first.
4. Every ($\forall$I) bar has y free neither in $\forall x\, A$ nor in any leaf still open above it. Every ($\exists$E) bar has y free in none of C, $\exists x\, A$, and the leaves of its second premiss still open there other than $A[y/x]$. In all four quantifier bars the term put in, t or y, is substitutable.
5. The leaves left without brackets are the open hypotheses. The tree witnesses $\Gamma \vdash_{\mathrm{NJ}} A$ for every Γ that contains them.

The fourth check is new, and the next section is about what breaks without it.

Consistency and the existence property. Two things are worth knowing about these rules before anything is built on them. Neither is a derivation; both are theorems *about* derivations, as Theorem 3.38 was for the propositional rules.

Theorem 4.24 (consistency)**.** *In NJ with quantifiers, no atomic formula is derivable from no hypotheses:*

- $\nvdash_{\mathrm{NJ}} \bot$*;*
- $\nvdash_{\mathrm{NJ}} P(t_1, \ldots, t_n)$ *for every predicate symbol* P *and terms* $t_1, \ldots, t_n$*.*

Corollary 4.25 (disjunction property)**.** *If* $\vdash_{\mathrm{NJ}} A \vee B$*, then* $\vdash_{\mathrm{NJ}} A$ *or* $\vdash_{\mathrm{NJ}} B$*.*

Corollary 4.26 (existence property)**.** *If* $\vdash_{\mathrm{NJ}} \exists x\, A$*, then there is a term* t *with* $\vdash_{\mathrm{NJ}} A[t/x]$*.*

The three are proved in Section 4.5, by putting the programs back on the trees and running them, as Theorem 3.38 was proved for the propositional rules.

The theorem says that the rules add nothing of their own: every fact about a particular kind, driver or day in a conclusion was put in as a hypothesis. The existence property is the quantifier's character in one line. A theorem of the form *there is* doesn't merely assert that something exists; it can be made to say *what*, and the worked derivations show where to look: at the ($\exists$I) bar.[8]

4.3. **Side conditions.** Three things can go wrong with a quantifier rule, and each of them is a bug you've met in a program. You remember a side condition once you've watched its absence prove everything, so the third

[8]All three are about theorems, with no hypotheses. From a hypothesis $\exists x\, A$ you can't name a witness, since a hypothesis is assumed and not derived, and shows nothing of how it came to hold, which is why a log may say "declined or approved" without saying which. And with no constant in the signature, $\vdash_{\mathrm{NJ}} \exists k\, (\mathsf{Cap}(k) \to \mathsf{Cap}(k))$ 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.

bug is run all the way to $\bot$ below; one of the exercises does the same for the first.

Generalising over a fixed variable. Suppose the kernel is handling a request for some kind u, and the hypothesis in force is $\mathsf{Cap}(u)$: *this* caller has a capability for kind u. One application of ($\forall$I) would give $\forall k\, \mathsf{Cap}(k)$, the caller has a capability for every kind, from which, with the driver theory, it may read every kind. The ($\forall$I) clause refuses it: u is free in the open hypothesis $\mathsf{Cap}(u)$, so u isn't arbitrary, whatever the argument calls it. Discharge the hypothesis first, by ($\to$I), and generalising becomes legal. But what you've generalised then is $\mathsf{Cap}(u) \to \ldots$, and that is harmless.

The clause asks only that u be free in no hypothesis *used*. A hypothesis about u lying in Γ untouched doesn't block the bar, since it isn't open. In the illegal step the leaf $\mathsf{Cap}(u)$ is open above the bar and has u free, and that is exactly what the condition on ($\forall$I) forbids.

Escaping witnesses. "Some driver registered the camera; call it y; so y registered the camera." The tree refuses this 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, each of them a way of naming a thing that was already in play.

You might choose as the fresh name a variable that $\exists x\, A$ already mentions. From "some number is below y" you would then get "some number is below itself":

$$\dfrac{\exists x\,(x < y) \qquad \dfrac{[y < y]^1}{\exists x\,(x < x)}\,\exists\mathrm{I}}{\exists x\,(x < x)}\,\exists\mathrm{E},1$$

The conclusion is clean of y, but y is free in $\exists x\,(x < y)$, so the bar is illegal, and rightly so: with $y := 1$ the hypothesis is true of the natural numbers and the conclusion is false.

Or you might choose a variable that some other open hypothesis already says something about. From "some kind has a capability" and "kind y has none" you would then get a contradiction:

$$\dfrac{\exists k\, \mathsf{Cap}(k) \qquad \dfrac{\neg\mathsf{Cap}(y) \quad [\mathsf{Cap}(y)]^1}{\bot}\,\to\mathrm{E}}{\bot}\,\exists\mathrm{E},1$$

Here y is free in neither $\bot$ nor $\exists k\, \mathsf{Cap}(k)$, but it is free in the other open hypothesis $\neg\mathsf{Cap}(y)$, and the two hypotheses are consistent: the kind with the capability need not be y. Together the three conditions say that you may use the thing and may not learn which thing it is, which is what a module boundary says. In the first of the three cheats it is the conclusion that fails: $\mathsf{Reg}(y, \mathsf{cam})$ mentions the made-up name y, which would outlive the argument that made it up.

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

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

"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.

$$\dfrac{\dfrac{\forall x\,\exists y\,(x < y)}{\exists y\,(y < y)}\,\forall\mathrm{E} \qquad \dfrac{\dfrac{\forall x\,\neg(x < x)}{\neg(z < z)}\,\forall\mathrm{E} \quad [z < z]^1}{\bot}\,{\to}\mathrm{E}}{\bot}\,\exists\mathrm{E},1$$

Four bars, and OH is the two harmless hypotheses, so the tree would witness $\forall x\,\exists y\,(x < y),\ \forall x\,\neg(x < x) \vdash_{\mathrm{NJ}} \bot$. The system has proved $\bot$ 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 $\bot$, $\exists y\,(y < y)$ 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. The one illegal bar is the left-hand ($\forall$E): the term y would land under $\exists y$ in $\exists y\,(x < y)$, so it is not substitutable for x there, and every other bar of the tree is legal.

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 $A \to B$ from the hypothesis B, and the blind result $\lambda y.\,y$ is a proof of $A \to A$ 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.

4.4. **Dependent types.** Back, now, to the code where the trouble started, and to the programs. What type should the open call have? In the kernel's old code it was an arrow from `Capability` to `Handle`, and we've seen what that cost.

The open function has to do two things the old type couldn't say. It has to work for *every* kind, Tuesday's included. And it has to say, in its result, *which* kind it was given, so that the handle it makes for the camera is a handle for the camera.

The same move, function and argument, gives the types. With `Capability k` and `Handle k` as types with a hole in them, the declarations become

```
cam            : Kind
camCap         : Capability cam                  -- granted at install
openHandle     : (k : Kind) -> Capability k -> Handle k
readHandle     : (k : Kind) -> Handle k -> Bytes k
record         : (k : Kind) -> Handle k -> IO ()
```

Read the third line slowly, because it's the only new thing here. The function `openHandle` takes a kind k, then a capability for k, and returns a handle to k. The result type mentions the argument.

In the old code the kind and the token were one value, `h :: Handle`, and that was the bug. Here they're two: the kind, and a separate value whose type has the kind 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.[9]

On the program side, the long *and* is a record with one field per kind: forty handles, one for each device. A function from kinds to handles holds the same information (hand it the camera and it returns the camera'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 forty-way `Either`: a value with a tag saying which case it is. Here the tags are drivers, so the value is a pair of a driver and evidence about that driver. 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 Π, Σ, $\times$ and $+$ for the same four things. The last two sections used the logician's letters; this one uses the type theorist's, and the table at the end of the section lines them up.

The letters have a history, which the chapter's History tells. Π and Σ were the quantifiers' first letters, chosen for the reason given in the first section: a long *and* is a product and a long *or* a sum. Logic traded them for $\forall$ and $\exists$, and nearly every logic book since has used those. Type theory took them back, ninety years later, because in it a proof of *for all* is a function, an

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

element of a product with one factor for each thing, and a proof of *there is* a pair, an element of a sum with one summand for each thing.[10]

So putting the programs back on means renaming as well as decorating: $\forall$ becomes Π and $\exists$ becomes Σ, along with $\wedge$ to $\times$ and $\vee$ to $+$. Before I write the rules down I'd like you to find them.

Problem 4.27. Take the four quantifier rules and decorate them with programs, as ($\to$I) and ($\to$E) were decorated with λ and application. What is the program for each of *let x be arbitrary*, *in particular, for t*, *take x to be t* and *let y be such a thing*? And what becomes of the eigenvariable and its conditions?

The definitions below are the answer. The hint has already been given: a proof of *for all* is a function from things, and a proof of *there is* is a pair.

Definition 4.28 (dependent types)**.** Fix a signature. Its terms t name things, and its variables $x, y, z, u, v, k, d, \ldots$ stand for things. The *types* are the formulas over the signature, written as a type theorist writes them:

$$\begin{aligned} A, B, C \ ::= \ & P(t_1, \ldots, t_n) \mid \bot \mid A \to B \mid A \times B \mid A + B \\ & \mid \Pi x\, A \mid \Sigma x\, A, \end{aligned}$$

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 $A \to \bot$.

Definition 4.29 (terms of λP)**.** A second stock of variables $h, j, a, e, f, g, \ldots$, disjoint from the variables for things, stands for evidence. The *terms* are

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

where h and j 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. In the third, the braces hold the two branches of a **case**, which looks at the tag on M: if it is *Left*, the evidence M carries is called h and P runs; if it is *Right*, the evidence is called j and Q runs. So h is in scope in P only, and j in Q only. The fourth line has the four new forms:

- an abstraction over a thing;

[10] Peirce (1885) for the first letters; Martin-Löf, from 1972, for the dependent types. Martin-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.

- an application to a thing;
- a pair of a thing and evidence;
- the match that opens such a pair.

Definition 4.30 (context)**.** A *context* Γ is a finite list of entries of two kinds,

- a variable x on its own, which *declares* a thing, or
- a typing $h : A$, which declares evidence,

such that

- 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 $\to$, $\times$, $+$ and $\bot$ are the propositional ones, except that a context may now contain declarations.

Definition 4.31 (typing rules)**.** The judgement $\Gamma \vdash_{\lambda P} M : A$ is defined by the rules of Figure 4.3, the propositional ones above the dotted line and the four new ones below it; the subscript λP is the name the λ-cube at the end of the chapter gives this system. In every rule, each context written must be a context in the sense of Definition 4.30; in (abs), for example, every variable free in A must be declared in Γ.

Read (ΠE) 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.

(ΠI) 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 $\{\mathit{Left}, \mathit{Right}\}$ 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. The ($\forall$I) clause asked that its eigenvariable be free in no open hypothesis, and the section on side conditions showed what goes wrong without the condition. Yet (ΠI) in Figure 4.3 carries no

$$\frac{}{\Gamma \vdash_{\lambda P} h : A}\ \text{var} \quad (\text{if } h : A \text{ is in } \Gamma) \qquad \frac{\Gamma,\ h : A \vdash_{\lambda P} M : B}{\Gamma \vdash_{\lambda P} \lambda h.\, M : A \to B}\ \text{abs}$$

$$\frac{\Gamma \vdash_{\lambda P} M : A \to B \qquad \Gamma \vdash_{\lambda P} N : A}{\Gamma \vdash_{\lambda P} M\, N : B}\ \text{app}$$

$$\frac{\Gamma \vdash_{\lambda P} M : A \qquad \Gamma \vdash_{\lambda P} N : B}{\Gamma \vdash_{\lambda P} (M, N) : A \times B}\ \times\text{I} \qquad \frac{\Gamma \vdash_{\lambda P} M : \bot}{\Gamma \vdash_{\lambda P} \mathit{absurd}\, M : C}\ \bot\text{E}$$

$$\frac{\Gamma \vdash_{\lambda P} M : A \times B}{\Gamma \vdash_{\lambda P} \mathit{fst}\, M : A}\ \times\text{E}_1 \qquad \frac{\Gamma \vdash_{\lambda P} M : A \times B}{\Gamma \vdash_{\lambda P} \mathit{snd}\, M : B}\ \times\text{E}_2$$

$$\frac{\Gamma \vdash_{\lambda P} M : A}{\Gamma \vdash_{\lambda P} \mathit{Left}\, M : A + B}\ +\text{I}_1 \qquad \frac{\Gamma \vdash_{\lambda P} M : B}{\Gamma \vdash_{\lambda P} \mathit{Right}\, M : A + B}\ +\text{I}_2$$

$$\frac{\Gamma \vdash_{\lambda P} M : A + B \qquad \Gamma,\ h : A \vdash_{\lambda P} P : C \qquad \Gamma,\ j : B \vdash_{\lambda P} Q : C}{\Gamma \vdash_{\lambda P} \mathbf{case}\ M\ \mathbf{of}\ \{\ \mathit{Left}\, h \Rightarrow P \mid \mathit{Right}\, j \Rightarrow Q\ \}\ :\ C}\ +\text{E}$$

. . . . . .

$$\frac{\Gamma,\ x \vdash_{\lambda P} M : A}{\Gamma \vdash_{\lambda P} \lambda x.\, M : \Pi x\, A}\ \Pi\text{I} \qquad \frac{\Gamma \vdash_{\lambda P} M : \Pi x\, A}{\Gamma \vdash_{\lambda P} M\, t : A[t/x]}\ \Pi\text{E} \quad (\text{if } t \text{ is in scope in } \Gamma)$$

$$\frac{\Gamma \vdash_{\lambda P} M : A[t/x]}{\Gamma \vdash_{\lambda P} \langle t, M \rangle : \Sigma x\, A}\ \Sigma\text{I} \quad (\text{if } t \text{ is in scope in } \Gamma)$$

$$\frac{\Gamma \vdash_{\lambda P} M : \Sigma x\, A \qquad \Gamma,\ x,\ h : A \vdash_{\lambda P} N : C}{\Gamma \vdash_{\lambda P} \mathbf{let}\ \langle x, h \rangle = M\ \mathbf{in}\ N : C}\ \Sigma\text{E} \quad (\text{if } x \notin \mathrm{FV}(C))$$

FIGURE 4.3. The typing rules of the dependent type system λP.

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.[11]

The first bug of the section on side conditions fails here in another form. The kernel is handling a request for kind u with $h : \mathsf{Cap}(u)$ in force; u was declared before h, so it isn't on top of the stack, and (ΠI) doesn't apply.

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

[11] Two 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.

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; Chapter 5 shows how.

Now some typings, as sequent-style trees, in which every node is a whole judgement with its context. The declarations above become one context. The kinds are constants of the signature, which are always in scope, so they need no declaring, and what is left is the evidence: ch and hr for the first two sentences of the driver theory (the functions `openHandle` and `readHandle`), lv for its register sentence, rec for `record`, and cc for `camCap`. With the types written in the predicates of the signature, that context is

$$\begin{aligned}\Gamma \;=\; & \mathsf{ch} : \Pi k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k)),\\ & \mathsf{hr} : \Pi k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k)),\\ & \mathsf{lv} : \Pi d\,\Pi k\,(\mathsf{Reg}(d,k) \to \mathsf{Live}(k)),\\ & \mathsf{rec} : \Pi k\,(\mathsf{Read}(k) \to \mathit{IO}\,()),\\ & \mathsf{cc} : \mathsf{Cap}(\mathsf{cam}).\end{aligned}$$

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

Example 4.32 (Ivanova, cont'd: the camera handle)**.** The camera's read ticket is built in two stages. First the handle, from ch and the capability:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\lambda P} \mathsf{ch} : \Pi k\,(\mathsf{Cap}(k) \to \mathsf{Hand}(k))}\,\text{var}}{\Gamma \vdash_{\lambda P} \mathsf{ch}\,\mathsf{cam} : \mathsf{Cap}(\mathsf{cam}) \to \mathsf{Hand}(\mathsf{cam})}\,\Pi\text{E} \qquad \dfrac{}{\Gamma \vdash_{\lambda P} \mathsf{cc} : \mathsf{Cap}(\mathsf{cam})}\,\text{var}}{\Gamma \vdash_{\lambda P} \mathsf{ch}\,\mathsf{cam}\,\mathsf{cc} : \mathsf{Hand}(\mathsf{cam})}\,\text{app}$$

and then the read ticket, from hr and the tree just drawn, whose root judgement, marked (1), is a premiss:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\lambda P} \mathsf{hr} : \Pi k\,(\mathsf{Hand}(k) \to \mathsf{Read}(k))}\,\text{var}}{\Gamma \vdash_{\lambda P} \mathsf{hr}\,\mathsf{cam} : \mathsf{Hand}(\mathsf{cam}) \to \mathsf{Read}(\mathsf{cam})}\,\Pi\text{E} \qquad (1)}{\Gamma \vdash_{\lambda P} \mathsf{hr}\,\mathsf{cam}\,(\mathsf{ch}\,\mathsf{cam}\,\mathsf{cc}) : \mathsf{Read}(\mathsf{cam})}\,\text{app}$$

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

Example 4.33 (Ivanova, cont'd: the wrong-kind read)**.** The app's read handler passes the camera ticket to `record mic`,[12] and the checker gets as far as the ticket:

$$\dfrac{\dfrac{\Gamma \vdash_{\lambda P} \mathsf{rec} : \Pi k\,(\mathsf{Read}(k) \to \mathit{IO}\,())}{\Gamma \vdash_{\lambda P} \mathsf{rec}\,\mathsf{mic} : \mathsf{Read}(\mathsf{mic}) \to \mathit{IO}\,()}\,\Pi\text{E} \qquad \dfrac{\vdots}{\Gamma \vdash_{\lambda P} \mathit{ticket} : \mathsf{Read}(\mathsf{cam})}}{\text{(app) fails: expected } \mathsf{Read}(\mathsf{mic}),\text{ got } \mathsf{Read}(\mathsf{cam})}$$

Here *ticket* is the term at the root of the camera's tree, $\mathsf{hr}\,\mathsf{cam}\,(\mathsf{ch}\,\mathsf{cam}\,\mathsf{cc})$, copied across. The type confusion is a type error, and the error says which kind the handle was for. Nothing in the kernel's code had to be clever.

[12] $\mathit{IO}\,()$ is the type of an action that returns nothing of interest.

Remark (looking back: Jones). The tree for the camera handle is the typing of the same ticket chain in Chapter 2, with the two (ΠE) bars added. And the wrong-kind read is refused for the reason Jones was refused there, when he asked to rotate the signing key: a mismatch of evidence. What has changed is that the type can now say which kind the evidence is about.

Example 4.34 (Ivanova, cont'd: every kind may be read)**.** The theory's first two sentences, composed and then generalised, give the typing that answers Ivanova's original question: may every kind with a capability be read? In the context $\Gamma, k, h : \mathsf{Cap}(k)$, with k declared and then a hypothesis about it, two (ΠE)s and two (app)s give $\mathsf{hr}\, k\, (\mathsf{ch}\, k\, h) : \mathsf{Read}(k)$, by the trees of Example 4.32 with k for cam and h for cc. Then two bars close it, and the order they come in is the side condition:

$$\dfrac{\dfrac{\Gamma,\ k,\ h : \mathsf{Cap}(k) \vdash_{\lambda P} \mathsf{hr}\, k\, (\mathsf{ch}\, k\, h) : \mathsf{Read}(k)}{\Gamma,\ k \vdash_{\lambda P} \lambda h.\, \mathsf{hr}\, k\, (\mathsf{ch}\, k\, h) : \mathsf{Cap}(k) \to \mathsf{Read}(k)}\ \text{abs}}{\Gamma \vdash_{\lambda P} \lambda k.\, \lambda h.\, \mathsf{hr}\, k\, (\mathsf{ch}\, k\, h) : \Pi k\, (\mathsf{Cap}(k) \to \mathsf{Read}(k))}\ \Pi\text{I}$$

The (abs) bar pops h, the hypothesis about k. Then k is on top and (ΠI) pops it. Try them the other way round and the (ΠI) bar has nothing to apply to, because k isn't the last entry. The program is function composition with a kind threaded through it, and its type, read as a sentence, is *every kind with a capability may be read*, which isn't in the theory but follows from it.

Example 4.35 (Ivanova, cont'd: the register rule)**.** At boot the `v4l` driver registers the camera, and the kernel records the fact as $\langle \mathsf{v4l}, \mathsf{rc} \rangle : \Sigma d\, \mathsf{Reg}(d, \mathsf{cam})$, by (ΣI) from $\mathsf{rc} : \mathsf{Reg}(\mathsf{v4l}, \mathsf{cam})$: the driver and the evidence, packed together, in a value whose type says only that *some* driver registered the camera.

Before it hands out camera handles, the kernel needs $\mathsf{Live}(\mathsf{cam})$, and the register rule $\mathsf{lv} : \Pi d\, \Pi k\, (\mathsf{Reg}(d, k) \to \mathsf{Live}(k))$ gives it, once the pair is opened. The kernel holds the record as a, so its context is $\Gamma,\ a : \Sigma d\, \mathsf{Reg}(d, \mathsf{cam})$. Opening the pair adds two entries, a driver y and the evidence e about it, and in that context the rule applies. Two (ΠE)s and an (app) give

$$\Gamma,\ a : \Sigma d\, \mathsf{Reg}(d, \mathsf{cam}),\ y,\ e : \mathsf{Reg}(y, \mathsf{cam}) \ \vdash_{\lambda P}\ \mathsf{lv}\, y\, \mathsf{cam}\, e : \mathsf{Live}(\mathsf{cam}),$$

which, marked (1), is a premiss of the tree that opens the pair. In that tree, S is short for $\Sigma d\, \mathsf{Reg}(d, \mathsf{cam})$:

$$\dfrac{\dfrac{}{\Gamma,\ a : S \vdash_{\lambda P} a : S}\ \text{var} \qquad (1)}{\Gamma,\ a : S \vdash_{\lambda P} \mathbf{let}\ \langle y, e \rangle = a\ \mathbf{in}\ \mathsf{lv}\, y\, \mathsf{cam}\, e : \mathsf{Live}(\mathsf{cam})}\ \Sigma\text{E}$$

The right-hand branch calls the rule at y and cam and applies it to e, where y is a name for whichever driver it was. The side condition holds because $\mathsf{Live}(\mathsf{cam})$ doesn't mention y.

Suppose instead that the handler tried to conclude, from the register record, which driver registered the camera:

$\mathbf{let}\ \langle y, e \rangle = a\ \mathbf{in}\ e\ :\ \mathsf{Reg}(y, \mathsf{cam})$ — refused: y is free in $\mathsf{Reg}(y, \mathsf{cam})$.

Inside the match, $e : \mathsf{Reg}(y, \mathsf{cam})$ 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. This is the escaping witness of the section on side conditions, caught by the result type.

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 4.36 (β-reduction)**.** A step rewrites an introduction met by its elimination, anywhere inside a term. The relation $M \rightsquigarrow N$ is given by one rule for each introduction form met by its elimination,

$$\frac{}{(\lambda h.\, P)\, Q \;\rightsquigarrow\; P[Q/h]}\ \beta \qquad \frac{}{\mathit{fst}\,(P, Q) \;\rightsquigarrow\; P}\ \text{fst} \qquad \frac{}{\mathit{snd}\,(P, Q) \;\rightsquigarrow\; Q}\ \text{snd}$$

$$\frac{}{\mathbf{case}\ (\mathit{Left}\, R)\ \mathbf{of}\ \{\, \mathit{Left}\, h \Rightarrow P \mid \mathit{Right}\, j \Rightarrow Q \,\} \;\rightsquigarrow\; P[R/h]}\ \mathit{Left}$$

$$\frac{}{\mathbf{case}\ (\mathit{Right}\, R)\ \mathbf{of}\ \{\, \mathit{Left}\, h \Rightarrow P \mid \mathit{Right}\, j \Rightarrow Q \,\} \;\rightsquigarrow\; Q[R/j]}\ \mathit{Right}$$

$$\frac{}{(\lambda x.\, P)\, t \;\rightsquigarrow\; P[t/x]}\ \beta \qquad \frac{}{\mathbf{let}\ \langle x, h\rangle = \langle t, P\rangle\ \mathbf{in}\ Q \;\rightsquigarrow\; Q[t/x][P/h]}\ \mathbf{let}$$

together with a rule for each place a subterm can sit, as (app_1), (app_2) and (abs). There is no rule for *absurd*, because $\bot$ 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)**
$A \to B$	$A \to B$	$\lambda h.\, M$	application $M\, N$
$A \wedge B$	$A \times B$	(M, N)	*fst*, *snd*
$A \vee B$	$A + B$	$\mathit{Left}\, M$, $\mathit{Right}\, M$	`case` with two branches
$\bot$	$\bot$	—	*absurd*
$\neg A$	$A \to \bot$	$\lambda h.\, M$	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 the kernel checks. The first two columns are a dictionary between two notations: $\times$ for $\wedge$, $+$ for $\vee$, Π for $\forall$ and Σ for $\exists$, with $\to$ and $\bot$ the same on both sides.

Types that mention values weren't invented for kernels. In 1967 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 a classic textbook of analysis,[13] every lemma,

[13] Landau's *Foundations of Analysis*: a good choice, since nobody could accuse Landau of skipping steps. The book builds the real numbers from the natural numbers in 301 numbered theorems, and its preface for the student begins: "Please forget everything you

and it checked.[14] 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.

4.5. **Curry–Howard with quantifiers.** Rub the programs out of the typing rules of Figure 4.3, keeping the contexts and the formulas, and rename as you go: Π to $\forall$, Σ to $\exists$, $\times$ to $\wedge$, $+$ to $\vee$. Here is what's left of the four quantifier rules, with the declarations kept, since a declaration is a name and not evidence:

$$\frac{\Gamma,\ x \vdash_{\mathrm{NJ}} A}{\Gamma \vdash_{\mathrm{NJ}} \forall x\, A} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} \forall x\, A}{\Gamma \vdash_{\mathrm{NJ}} A[t/x]} \qquad \frac{\Gamma \vdash_{\mathrm{NJ}} A[t/x]}{\Gamma \vdash_{\mathrm{NJ}} \exists x\, A}$$

$$\frac{\Gamma \vdash_{\mathrm{NJ}} \exists x\, A \qquad \Gamma,\ x,\ A \vdash_{\mathrm{NJ}} C}{\Gamma \vdash_{\mathrm{NJ}} C}$$

with the side conditions carried over unchanged (t in scope in the second and third, x not free in C in the fourth). They are the four quantifier rules of NJ, written on the turnstile: *let x be arbitrary*, *in particular, for t*, *take x to be t*, *let y be such a thing*. The Curry–Howard correspondence of Theorem 3.15 extends.

Theorem 4.37 (Curry–Howard, with quantifiers)**.** *Typings in λP and derivations in NJ with quantifiers correspond one to one.*

- *Erasing the terms from a typing, keeping its declared variables, 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 ($\exists E$).*
- *Decorating a derivation with terms turns it into a typing, and erasure and decoration undo each other.*

Writing $|\Gamma|$ for the set of types of the evidence entries of Γ, read as formulas,

$$\Gamma \vdash_{\lambda P} M : A \text{ for some term } M \quad \text{if and only if} \quad |\Gamma| \vdash_{\mathrm{NJ}} A,$$

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

In words: a derivation with quantifiers is a program whose type mentions things, and the side conditions are the same on both sides, read off the context on one and off the leaves on the other.

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

---

have learned in school; for you haven't learned it." The preface for the teacher explains that the book is written, "as befits such easy material, in merciless telegram style". A clerk could ask for no better source.

[14] de 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.

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.

A tree has no declarations. What the context records, the tree recovers by computing OH, the stack included. 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. Read on the turnstile, the tree's 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.

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 theorem. But the tree states the condition that matters and nothing more, and that is one reason proof theory is written in trees.

Read through the correspondence, each quantifier rule is a shape of program. A ($\forall$E) bar is a call on a thing, so two of them followed by ($\to$E) are a function called on two things and then on evidence. An ($\exists$I) bar packs a pair whose first half is the witness, so a proof of *there is* answers *which one?* with its first half. An ($\exists$E) bar is a **let** that opens such a pair and works with the thing it finds without learning which it was. And a supposition used twice is an evidence variable used twice.

Example 4.38 (the worked derivations, decorated)**.** Decorated, each of Ivanova's seven questions and the head of a horse becomes a program of one line. The theory's sentences go under the names the context Γ of Section 4.4 gave them, ch, hr and lv, with cc the camera capability.

Example	**Program**
4.16, the boot log	$\mathsf{lv}\ \mathsf{v4l}\ \mathsf{cam}\ r$, with r the log line
4.17, the documentation	$\lambda u.\, \lambda h.\ \big(\mathsf{ch}\, u\, h,\ \mathsf{hr}\, u\, (\mathsf{ch}\, u\, h)\big)$
4.18, anything at all?	$\langle \mathsf{cam},\ \mathsf{hr}\, \mathsf{cam}\, (\mathsf{ch}\, \mathsf{cam}\, \mathsf{cc})\rangle$
4.19, a capability for something	$\mathbf{let}\ \langle y, h\rangle = a\ \mathbf{in}\ \langle y,\ \mathsf{hr}\, y\, (\mathsf{ch}\, y\, h)\rangle$, with a the manifest's record
4.20, every kind has a driver	$\lambda u.\ \langle \mathsf{owner}(u),\ o\, u\rangle$, with o the first summary
4.21, the sandbox	$\lambda u.\, \lambda h.\ n\, \langle u, h\rangle$, with n the sandbox policy
4.22, the pager	$\lambda u.\, \lambda h.\ f\, \langle u, h\rangle$ and $\lambda a.\ \mathbf{let}\ \langle y, h\rangle = a\ \mathbf{in}\ g\, y\, h$, with f and g the two rules
4.23, the head of a horse	$\lambda y.\, \lambda a.\ \mathbf{let}\ \langle z, p\rangle = a\ \mathbf{in}$ $\big\langle z,\ (ha\, z\, (\mathit{fst}\, p),\ \mathit{snd}\, p)\big\rangle$, with ha the hypothesis about horses

Ask the third program *which kind?* and it says cam. The fourth opens the manifest's record and hands back the kind it found, without learning which it was. The sandbox packs a kind and a supposed proof together and

hands them to the policy, which turns any such pair into $\bot$. The pager's two programs are a function on pairs and its curried form, currying one level up. And the horse:

1. open the pair;
2. keep the horse;
3. swap the evidence that it's a horse for evidence that it's an animal;
4. pack it up again.

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 4.39 (subject reduction)**.** *If* $\Gamma \vdash_{\lambda P} M : A$ *and* $M \rightsquigarrow N$*, then* $\Gamma \vdash_{\lambda P} N : A$*.*

In words: running a program never changes its type. The proof is the propositional one with two more cases, one for each new step; each needs that putting a term in for a declared variable, in a program and in its type at once, preserves typing, which is an induction on the typing tree.

Theorem 4.40 (normalisation[15])**.** *Every typed term runs to a normal form. Equivalently, every derivation can be rewritten into one without detours.*

Proof for terms without $+$ *and* Σ*.* The counting proof of Theorem 3.33 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 fact the count rests on, that a step creates only redexes of lower degree than the one it removes, 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.
- 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. □

The type 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.

The theorem holds all the same for every connective and both quantifiers; the proofs that reach that far add commuting conversions to the count, or argue by computable terms instead of counting, and for terms with $+$ or Σ we take the theorem on trust.[16]

[15] Prawitz (1965).

[16] Prawitz (1965) for the conversions, Tait (1967) for computable terms.

Proposition 4.41 (closed normal forms)**.** *A closed typed term in normal form is an introduction form:* $\lambda h.\, M$*,* (M, N)*,* $\mathit{Left}\, M$*,* $\mathit{Right}\, M$*,* $\lambda x.\, M$ *or* $\langle t, M\rangle$*.*

In words: a closed program that has finished running is a constructor, and its type says which.

The consistency theorem (Theorem 4.24) and its two corollaries, left unproved when they were stated, now follow from these three results, as Theorem 3.38 did.

Proof of the consistency theorem and its corollaries. By the Curry–Howard theorem a closed derivation is a closed typed term, and by normalisation and subject reduction it runs to a closed normal term of the same type, which is an introduction form.

- *Consistency.* No introduction form has type $\bot$ or an atomic type, so there is no closed normal term of either.
- *The disjunction property.* A closed normal term of type $A + B$ is $\mathit{Left}\, M$ or $\mathit{Right}\, M$, and its tag says which disjunct holds.
- *The existence property.* A closed normal term of type $\Sigma x\, A$ is $\langle t, M\rangle$ with $\vdash_{\lambda P} M : A[t/x]$, and its first half is the witness.

□

Finding a tree, with quantifiers. In the propositional case the method for finding a tree rested on the shape of a normal derivation, and the proposition on closed normal forms says the shape survives the quantifiers: introductions near the root, building the conclusion, over eliminations that hang off hypotheses and take them apart. So the method gains a line for each quantifier.

Read the conclusion.

- If it starts with $\forall$, the last bar is probably ($\forall$I), and the argument begins *let u be any kind.*
- If it starts with $\exists$, the last bar is probably ($\exists$I), and the question is which witness to choose; often the hypotheses have to be taken apart before you know, so leave that bar till last, as with ($\vee$I).
- If a hypothesis starts with $\exists$, open it early with ($\exists$E) and give the thing a name, as with ($\vee$E).

Example 4.19 was found that way: its ($\exists$E) bar is the root, the first decision made, and the witness y for its ($\exists$I) bar was chosen only after that bar had named it.

4.6. **Theories.** Read the consistency theorem with names in the language. The logic proves no atomic sentence: neither $\mathsf{Cap}(\mathsf{cam})$ nor $\mathsf{Read}(\mathsf{mic})$ 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 kind, element or day has to come from the hypotheses.

Definition 4.42 (theory)**.** A *theory* is a set of sentences, used as hypotheses. A formula A *follows from* a theory Γ in NJ if $\Gamma \vdash_{\mathrm{NJ}} A$.

A derivation from a theory is a *certificate*. Anyone who doubts its conclusion need neither rerun nor understand the system the theory describes: they check the bars, one rule at a time. And since the logic proves no atomic sentence on its own, every atomic fact in the conclusion came from the theory and from nowhere else.

A theory can also say what a program's parameter must satisfy, and is then called a *contract*. A type checker certifies that a program is safe to run and says nothing about the contract. A program whose parameter breaks the contract fails by violating a consequence of it, and a *verifier*, a type checker whose types may quantify over things, can state the contract and check derivations of its consequences. That is the distance between *safe* and *correct*. Three theories follow, in increasing order of interest.

Example 4.43 (Ivanova, cont'd: the driver theory)**.** From Γ_{pol} and $\mathsf{Cap}(\mathsf{cam})$, the tree of Example 4.18 without its last bar derives $\mathsf{Read}(\mathsf{cam})$: two ($\forall$E)s and two ($\to$E)s, with OH the theory's first two sentences and the one fact. An auditor who doubts that the camera may be read checks four bars, and knows that $\mathsf{Read}(\mathsf{cam})$ came from the driver theory.

From Γ_{pol} alone, the tree that opened Section 4.2 derives *every kind with a capability may be read*, $\forall k\,(\mathsf{Cap}(k) \to \mathsf{Read}(k))$, which no runtime check settles, because it speaks of every kind at once. It is the typing of Example 4.34 with the programs gone, bar for bar.

Example 4.44 (a comparator)**.** Here's insertion sort, written so that the comparison is a parameter, as library sorting routines are, since 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
```

Its type is polymorphic, a quantifier over the element type (Chapter 5 works such types out),

$$sort : \Pi a\,((a \to a \to Bool) \to [a] \to [a]),$$

and the checker proves it. So 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.

Give `sort` a comparison that returns `True` when its arguments are equal, `<=` where `<` was meant. The type doesn't change, and the program compiles, runs and returns a list that is sometimes not sorted. On a longer input a library sort may do something stranger; this is an exception Java's sort really throws:

```
java.lang.IllegalArgumentException:
```

```
Comparison method violates its general contract!
```

The sort hasn't checked the contract, which would cost more than the sort. It found two runs it had recorded as ordered that were not, which is the failure of a *consequence* of the contract.

The contract is that `lt` 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 $x < z$	$\forall x\, \forall y\, \forall z\, (x < y \wedge y < z \to x < z)$
if $x < y$, every z is above x or below y	$\forall x\, \forall y\, \forall z\, (x < y \to x < z \vee z < y)$

Call this theory Γ_{ord}. A comparator built from `<=` reports both $a < b$ and $b < a$ for a pair of equal elements, and no comparator obeying the contract can, because

$$\Gamma_{\mathrm{ord}} \;\vdash_{\mathrm{NJ}}\; \forall x\, \forall y\, \neg(x < y \wedge y < x).$$

If $u < v$ and $v < u$, then transitivity gives $u < u$, which irreflexivity forbids. As a tree, in two pieces to fit the page, the first piece, $\mathcal{D}$, takes transitivity down to $u < u$ on the supposition:

$$\mathcal{D} \;=\; \dfrac{\dfrac{\dfrac{\dfrac{\forall x\, \forall y\, \forall z\, (x < y \wedge y < z \to x < z)}{\forall y\, \forall z\, (u < y \wedge y < z \to u < z)}\,\forall\mathrm{E}}{\forall z\, (u < v \wedge v < z \to u < z)}\,\forall\mathrm{E}}{u < v \wedge v < u \to u < u}\,\forall\mathrm{E} \qquad [u < v \wedge v < u]^1}{u < u}\,{\to}\mathrm{E}$$

and the second uses irreflexivity, discharges the supposition and generalises:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x\, \neg(x < x)}{\neg(u < u)}\,\forall\mathrm{E} \qquad \mathcal{D}}{\bot}\,{\to}\mathrm{E}}{\neg(u < v \wedge v < u)}\,{\to}\mathrm{I},1}{\forall y\, \neg(u < y \wedge y < u)}\,\forall\mathrm{I}}{\forall x\, \forall y\, \neg(x < y \wedge y < x)}\,\forall\mathrm{I}$$

The three ($\forall$E) bars of $\mathcal{D}$ choose $t := u$, $t := v$ and $t := u$. The discharge comes before the ($\forall$I) bar for v, since v is free in the supposition, and at the root OH is irreflexivity and transitivity, so the tree witnesses the claim. Decorated, it is one line,

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

with *trans* and *irrefl* two sentences of Γ_{ord} as variables. When the sort catches a comparator reporting both $a < b$ and $b < a$, the exception is reporting this theorem violated, on a program you've called a hundred times.

Example 4.45 (the sale)**.** And the patch. With $\mathsf{Disc}(d)$ for "the discount applies on day d", what a sale *is* comes to one sentence,

$$\forall d\, (\mathit{start} \le d \wedge d \le \mathit{end} \to \mathsf{Disc}(d)).$$

The patched code applies the discount only when $\mathit{start} < d$ and $d < \mathit{end}$, so it fails the sentence at $d := \mathit{start}$, 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 14 has one, and comes back to this patch.

Remark 4.46 (witnesses). The quantifier $\exists$ is the $\vee$ 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.

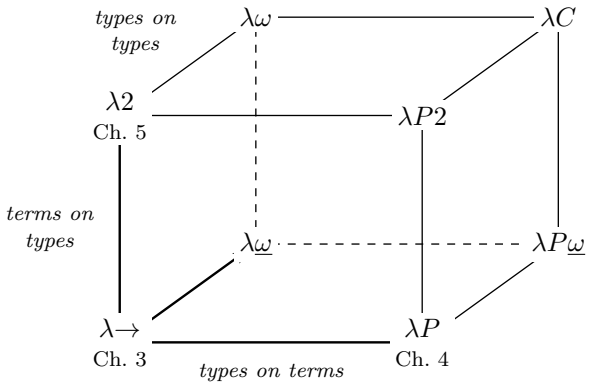
4.7. **The λ-cube.** By the end of the 1980s there were a dozen typed λ-calculi in the literature, each with its own notation, its own rules and its own proof that its programs stop, and nobody could see at a glance how they were related. The *λ-cube* is the observation that they differ in one respect only: what is allowed to depend on what.[17]

In every typed calculus a term can depend on a term: that's a function, $\lambda h.\, M$, and it's all the simply typed calculus had. Three more dependencies are possible, and you've now met each of them.

- *Terms that depend on types*: a program that takes a type, $\Lambda a.\, M$. That's generics, the `a` in `length :: [a] -> Int`, and Chapter 5 works it out.
- *Types that depend on terms*: a type that takes a thing, such as $\mathsf{Hand}(k)$. That's the dependent types of the section on dependent types.
- *Types that depend on types*: a type that takes a type, such as the list type, which takes a and gives $[a]$. You use one every time you write `List<String>`; a calculus that has them lets you write your own and check them.

[17] Barendregt (1991). Barendregt's earlier book, *The Lambda Calculus: Its Syntax and Semantics* (1981), is still the reference for the untyped calculus of Chapter 3. It also fixed a convention that this chapter has been quietly relying on: choose the bound variables of a term different from its free ones, and renaming is never needed, so capture can't happen. Every author since has adopted it. In 2007 Urban, Berghofer and Norrish showed that the convention, used carelessly in a proof by rule induction, can be used to prove a wrong theorem. The bug of the section on side conditions is a patient one.

Each of the three can be switched on or off independently of the others, so there are $2^3 = 8$ calculi, drawn as the corners of a cube. Each arrow switches one dependency on.



$\lambda\!\to$, the corner nearest you, is the simply typed λ-calculus of Chapter 3. Go up and you reach $\lambda 2$, System F, the subject of Chapter 5. Go right and you reach λP, types that mention things: this chapter, the heart of Automath, and of the logical framework LF, designed in 1987 for writing down logics.[18] Go back and you reach $\lambda\underline{\omega}$, types that take types. Up and back together is $\lambda\omega$, System Fω of 1972. The far top corner, with all three switched on, is λC, the *calculus of constructions*, and it is the calculus at the core of a proof assistant that thousands of mathematicians and engineers now use.[19]

Read through the Curry–Howard correspondence, every corner is a logic as well as a type system, and the cube becomes a cube of logics. $\lambda\!\to$ is the logic of implication, the logic of Chapters 2 and 3. λP is first-order logic, this chapter's predicates and quantifiers over things. $\lambda 2$ is second-order propositional logic, Chapter 5's quantifiers over propositions. λC is higher-order logic of every kind at once.

The edges are the moves of Part 1. Chapter 3 stood at one corner. This chapter walked one edge, to types that mention things. Chapter 5 walks another, to generics and then to the logic behind them. Nothing in this course leaves the edges that meet at $\lambda\!\to$; the rest of the cube is where proof assistants live.

The picture did more than tidy up. Every corner is a part of λC, so one proof that the programs of λC all stop covers all eight calculi at once, and with it the consistency of all eight logics.

And the cube marks its own edge. Take one more step, beyond λC: let the type of all types be a type, $\mathsf{Type} : \mathsf{Type}$. That is Girard's paradox, which Chapter 5 tells, and there everything is provable.

---

[18] The corners have their inventors: $\lambda\!\to$ is Church (1940), $\lambda 2$ is Girard (1972) and Reynolds (1974), LF is Harper, Honsell and Plotkin.

[19] The calculus is Coquand and Huet (1988)'s. The proof assistant was called Coq, after the calculus of constructions, after Thierry Coquand, and after the rooster that is the national symbol of France. In 2025 it was renamed Rocq, officially in honour of Inria's Rocquencourt site, where it was first built. Anyone who has said the old name aloud in an English-speaking lecture theatre can guess at the unofficial reasons.

History. The quantifier was invented twice, like natural deduction. Frege had it in 1879, in a notation nobody else used.[20]

Peirce and his student Mitchell had it by 1885, as Π and Σ,[21] 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.[22] 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, whose story is Chapter 5's. Figure 4.4 collects the dates.

Summary. For two thousand years logic couldn't put a *some* inside an *every*, and so couldn't say most of what mathematics and specifications say. Quantifiers can. They 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. The four quantifier rules are four phrases: let x be arbitrary, in particular for t, take x to be t, let y be such a thing. Two of them carry a condition on their eigenvariable, and the three ways of breaking the conditions are three bugs you've met, capture among them. The logic proves no atomic sentence, and a proof of *there is* names the thing.

Put the programs back and the quantifiers are two things you've programmed with: a proof of *for all* is a function from things, whose result type mentions its argument, and a proof of *there is* is a pair of a thing and evidence. These are $\Pi x\, A$ and $\Sigma x\, A$, the quantifiers' first letters, which logic traded for $\forall$ and $\exists$ and type theory kept. The context, a stack, enforces every freshness condition but the one on (ΣE), which is why that one alone is written on its rule. Erase the programs and the rules are NJ's, bar for bar. Running a program removes detours, every typed program stops (proved by

[20] Frege's own system didn't outlive its first decade. In June 1902, with the second volume of his *Basic Laws of Arithmetic* at the printer, he received a polite letter from the young Bertrand Russell, showing that one of its laws, about the extension of a concept, led to a contradiction. Frege's logic quantified over concepts as well as objects, the second-order kind of Chapter 5, and the contradiction lived there. Frege added an appendix, which begins: "Hardly anything more unfortunate can befall a scientific writer than to have one of the foundations of his edifice shaken after the work is finished." The quantifier survived; the edifice didn't.

[21] Some historians think the student taught the teacher, and that Mitchell's notation of 1883 is where Peirce's came from. Peirce's career went less well than his logic. The Johns Hopkins lectureship that gave him Mitchell was the only university post he ever held; he lost it in 1884 after a scandal over his remarriage, and died poor in 1914, after which Harvard bought his papers from his widow.

[22] Hilbert was sure that every such question would be answered. Closing an address at Königsberg in September 1930, broadcast on the radio, he said: *Wir müssen wissen. Wir werden wissen*, "we must know; we will know". The words are on his gravestone in Göttingen. The day before, at a round table in the same meeting, a twenty-four-year-old named Kurt Gödel had announced a theorem saying that no system of proofs for arithmetic that a clerk can check settles every question it can ask. Part 3 tells that story.

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
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
Martin-Löf	1972–	Π and Σ; a foundation on dependent types
Mitchell & Plotkin	1988	modules as existential types
Barendregt	1991	the λ-cube

FIGURE 4.4. Quantifiers and dependent types: who did what.

counting when there is no $\vee$ or $\exists$, on loan when there is), and consistency and the existence property are the dividend.

A **theory** is a set of sentences, and "A follows from it" is a derivation, checkable by pattern matching: the driver theory, a comparator's contract, a sale. Since the logic proves no atomic sentence on its own, everything about the camera comes from the theory. 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.

The λ-cube puts the type systems of Part 1 in one picture, sorted by what may depend on what. Chapter 3 is its nearest corner; this chapter's dependent types are one step along one edge, and Chapter 5's generics one step along another.

Exercises.

Exercise 4.1. Write the typing tree for the camera's read ticket in full, from the context Γ of the section on dependent types, and then the tree the checker builds for `record mic ticket`, marking the bar at which it fails and what the error message says. Then say what the context would have to contain about the microphone for the handler to type.

Exercise 4.2. Take Γ_{pol} and the extra fact $\mathsf{Cap}(\mathsf{mic})$.

(1) Derive $\mathsf{Read}(\mathsf{mic})$, as a tree, and give its program.
(2) Try to derive $\forall k\,(\mathsf{Cap}(k) \to \mathsf{Read}(k))$ with the ($\forall$I) bar above the ($\to$I) bar, and say which condition stops you.
(3) Add the sentence "no kind may be read unless some driver has registered it" to the driver theory, and say which of $\forall$, $\exists$ and $\neg$ you needed, and why the obvious first attempt says something weaker.

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

$$\lambda a.\,\lambda y.\ \mathbf{let}\ \langle x, h\rangle = a\ \mathbf{in}\ \langle x, h\,y\rangle.$$

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

Exercise 4.4. Derive $\forall x\,(A \wedge B) \to \forall x\,A \wedge \forall x\,B$ and $\exists x\,(A \vee B) \to \exists x\,A \vee \exists x\,B$, 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 $\exists x\,(\mathsf{Cap}(x) \to \mathsf{Cap}(x))$ 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 $\vdash_{\mathrm{NJ}} \exists x\,A \to \neg\forall x\,\neg A$, using ($\exists$E), and say what program it is.
(3) Try to derive the converse, $\neg\forall x\,\neg A \to \exists x\,A$, 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 $\mathsf{Cap}(u)$ derive, in turn,

$$\forall x\,\mathsf{Cap}(x) \qquad \text{in one step,}$$
$$\vdash_{\mathrm{NJ}}\ \mathsf{Cap}(u) \to \forall x\,\mathsf{Cap}(x),$$
$$\vdash_{\mathrm{NJ}}\ \forall y\,(\mathsf{Cap}(y) \to \forall x\,\mathsf{Cap}(x)).$$

(2) Conclude $\bot$ from the two hypotheses $\mathsf{Cap}(c)$ and $\neg\mathsf{Cap}(d)$, for constants c and d, which say only that the holder has a capability for one kind and not for another.

(3) Say which of the steps in (1) the real rule forbids, and state the corresponding compiler error.

Exercise 4.8. Try to derive $\exists x\,(\mathsf{Cap}(x) \to \forall y\,\mathsf{Cap}(y))$: "there is a kind such that, if the holder has a capability for it, it has one for every kind". 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 $\neg\neg p \to p$, which NJ does not derive either.

References.

Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

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

Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988.

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.

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.

William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

The MITRE Corporation. CWE-843: Access of resource using incompatible type ('type confusion'). Common Weakness Enumeration, 2024. `https://cwe.mitre.org/data/definitions/843.html`.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System*. PhD thesis, Eindhoven University of Technology, 1977.

Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.