Type Theory by Example

Carl Joshua Quines

September 9, 2021

Contents

0 Intro

Here was my frustration on type theory. All the material I've seen either:

- covered the theory well, but was dense, research-level, and notation-heavy, with few examples I understood, or
- had plenty of good examples, but was specific to something like Coq or Lean, and gave only passing mention to the theory.

There was a hole here. I wanted a relatively accessible introduction to type theory, focusing on the *theory* behind it, without going through details specific to proof assistants. Despite that, I wanted to learn more about the theory that was relevant to proof assistants. I wanted to understand the relationship between types and propositions, and all the different kinds of types that made up the calculus of inductive constructions. I wasn't concerned too much about metatheory, or logics that aren't used in formal verification anyway. This is my attempt to fill this hole.

It's called *Type Theory by Example* because I believe in the value of having a "poster example" to have when thinking about a concept. I tried to make the examples as clear as possible, even if it meant sacrificing the finer details. In fact, we'll focus on examples so much that we'll only very rarely state any actual rules.

The intended pre-requisites:

- You're familiar with logical notation, enough that you know how to read $\forall x : \mathbb{R}, \exists y : \mathbb{R}, x + y = 0$ and have it make sense.
- You know some classical logic, enough that the statement "if A or B, and not A implies B, then B" makes sense.
- You know enough set theory to know the definition of a function $f : A \to B$ as a subset of $A \times B$ satisfying certain properties.
- You have solid experience reading other people's proofs and writing your own.

Thanks to Nir Elber and Jason Chen for comments and suggestions. There are probably typos, small errors, medium errors, and absolutely huge errors. If you spot anything wrong, please correct me, because I'm literally just an undergrad and I don't know anything about anything I'm writing about. You can contact me at cj@cjquines.com. Feel free to reach me too if you find something unclear or if you have questions.

1 Logic

1.1 Inference rules

Symbolic logic is the study of how sets of symbols and rules about them build up the foundation of mathematics. These rules are built up in levels, depending on how much restriction you're putting over things you can quantify—that is, say "for all" ($\forall$) and "there exists" ($\exists$) on. In higher-order logic, you can quantify over anything. In first-order logic, you can quantify over single variables, so you can't quantify over sets or functions. And in zeroth-order logic, there are no quantifiers at all!

If there aren't quantifiers in zeroth-order logic, what's left? There are propositions, like true ($\top$), false ($\bot$), and variables for propositions ($A, B, C, \dots$). There are connectives between propositions, like "and" ($\wedge$), "or" ($\vee$), "implies" ($\to$), and "not" ($\neg$). Propositions and connectives form other propositions (e.g. $A \to \neg B$). That's it. This is why zeroth-order logic is called **propositional calculus**. There are many kinds of propositional calculi, depending on what rules you use.

Propositional calculi are also built up in levels. On the top is classical logic, where we have all of our nice connectives, each proposition is either true or false, and the law of excluded middle holds. Removing the law of excluded middle gives us constructive logic. Removing most of the connectives, leaving only $\to$ and $\neg$, gives us implicational calculus. And below even *that*, where we *only* keep $\to$, we get **positive implicational calculus**. We'll study this first.

The rules in any symbolic logic are called **inference rules**. Given something that's true, and in a certain format, an inference rule tells us something else that's true. The essential one, in positive implicational calculus, is this one, **modus ponens**.

$$\frac{A \quad A \to B}{B}$$

The way this rule should be read is as "if A is true, and $A \to B$ is true, then B is true." The things above the bar are called hypotheses, and the thing below is the conclusion. Here A and B stand for any proposition. We can replace them with other propositions, like replacing A with $A \to B$ and B with $B \to C$.

$$\frac{A \to B \quad (A \to B) \to (B \to C)}{B \to C}$$

We can stack these diagrams to produce longer proofs. These can get rather long, so we can also label the rule we use to the side. Here MP stands for modus ponens.

$$\dfrac{B \qquad \dfrac{A \to B \quad (A \to B) \to (B \to C)}{B \to C}\ \mathsf{MP}}{C}\ \mathsf{MP}$$

Positive implicational calculus also gives us two axioms, which are inference rules that don't have "requirements". We'll call these rules rule K and S, for reasons that will become clearer later.

$$\frac{}{A \to (B \to A)}\ \mathsf{K} \qquad \frac{}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\ \mathsf{S}$$

Please convince yourself that MP, K, and S are true in classical logic as well. Now we can try to prove facts using just these rules. For example, consider the fact $A \to A$.

We know, from classical logic, that this is true for any A. Can we prove it using just these three rules? Our first attempt might proceed by just trying to substitute A for everything in S. We want to use MP, somehow, to get $A \to A$.

$$\frac{}{(A \to (A \to A)) \to ((A \to A) \to (A \to A))}\ \mathsf{S}$$

However, we can't immediately use this. We can make $A \to (A \to A)$ by substituting A for everything in K. Then we can use MP. But we don't get anything useful:

$$\dfrac{\dfrac{}{A \to (A \to A)}\ \mathsf{K} \qquad \dfrac{}{(A \to (A \to A)) \to ((A \to A) \to (A \to A))}\ \mathsf{S}\ \text{(from above)}}{(A \to A) \to (A \to A)}\ \mathsf{MP}$$

To get to the conclusion $A \to A$ with MP, we somehow need to have produced $A \to A$ already! So instead, we use S, substitute A for A, but substitute $A \to A$ for B instead. We can then use K, by substituting $A \to A$ for B. This allows us to use MP. To get the conclusion, we use K again. This gives us the following proof.

$$\dfrac{\dfrac{}{A \to ((A \to A) \to A)}\ \mathsf{K} \qquad \dfrac{}{(A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))}\ \mathsf{S}}{(A \to (A \to A)) \to (A \to A)}\ \mathsf{MP}$$

$$\dfrac{\dfrac{}{A \to (A \to A)}\ \mathsf{K} \qquad \dfrac{}{(A \to (A \to A)) \to (A \to A)}\ \text{(from above)}}{A \to A}\ \mathsf{MP}$$

There are lots and lots of parentheses. To reduce parentheses, we adopt the convention that $\to$ **is right-associative**. That is,

$$A \to B \to C \to D \text{ should be interpreted as } A \to (B \to (C \to D)).$$

Note that this allows us to write K and S with less parentheses.

$$\frac{}{A \to B \to A}\ \mathsf{K} \qquad \frac{}{(A \to B \to C) \to (A \to B) \to A \to C}\ \mathsf{S}$$

Why right-associative? The simple, perhaps unsatisfying answer, is that this means we write less parentheses than we do if we treated $\to$ as left-associative. We'll give a more satisfying explanation later on when we talk about currying.

1.2 The deduction theorem

These kinds of proofs get complicated. Can you use these three rules to prove the fact that $(A \to B) \to (B \to C) \to A \to C$? This intuitively feels like something that should be true: we take $A \to B$, and we take $B \to C$, and we should be able to "chain" them together to get $A \to C$. Translating this to a proof straight from the axioms is kind of hard, though.

It'd be really nice if, somehow, we had a sort of "metatheorem". Not a theorem in the implicational calculus, but a theorem *about* it, proven *above* it. If Δ is a list of propositions, we'd really like to prove this **deduction theorem**:

$$\begin{array}{c} \Delta \quad A \\ \vdots \\ B \end{array} \implies \begin{array}{c} \Delta \\ \vdots \\ A \to B \end{array}.$$

Here we're using $\vdots$ to represent multiple inferences. To prove this, we'll be a bit more specific about what we can do with inferences. One rule we've been using implicitly, without mentioning it, is substitution: if we know A, then we can change a part x to any y. We'll call this rule Sub., and abbreviate it as $A[x \mapsto y]$.

Another, even more basic rule, is inferring something we already have, which we'll call Ref. for reflexivity. We'll consider these two as inference rules we can *always* use, no matter what system we're using. With modus ponens, this gives us our three inference rules.

$$\frac{\Delta \quad A}{A}\ \textsf{Ref.} \qquad \frac{A}{A[x \mapsto y]}\ \textsf{Sub.} \qquad \frac{A \quad A \to B}{B}\ \textsf{MP}$$

To prove the deduction theorem, we'll induct on the number of inferences we've made. The base case is no inferences. Now suppose that the deduction theorem is true for any sequence of n inferences. Suppose that we started with Δ and A, and after $n+1$ inferences, end up with B. Then there are three cases for the last inference:

- We used Ref.. If B is A, we need to prove $A \to B = A \to A$, which we've already done. Else B is in Δ. Then

$$\frac{\dfrac{\Delta}{B}\ \textsf{Ref.} \quad \dfrac{}{B \to (A \to B)}\ \textsf{K}}{A \to B}\ \textsf{MP}.$$

- We used Sub.. Then $B = B'[x \to y]$ for some B', x, and y. We used at most n inferences to get B', so by induction, we can prove $A \to B'$. Then

$$\frac{\begin{array}{cc}\Delta & A \\ \vdots & \\ B' & \end{array} \overset{\text{ind. hyp.}}{\Longrightarrow} \begin{array}{c}\Delta \\ \vdots \\ A \to B'\end{array}}{(A \to B')[x \to y] = A \to B}\ \textsf{Sub.}$$

- We used MP. Then there exists some P such that the final step looks like

$$\frac{\begin{array}{cc}\Delta & A \\ \vdots & \\ P & \end{array} \quad \begin{array}{cc}\Delta & A \\ \vdots & \\ P \to B & \end{array}}{B}\ \textsf{MP}.$$

  By induction hypothesis,

$$\begin{array}{cc}\Delta & A \\ \vdots & \\ P & \end{array} \overset{\text{ind. hyp.}}{\Longrightarrow} \begin{array}{c}\Delta \\ \vdots \\ A \to P\end{array} \qquad \begin{array}{cc}\Delta & A \\ \vdots & \\ P \to B & \end{array} \overset{\text{ind. hyp.}}{\Longrightarrow} \begin{array}{c}\Delta \\ \vdots \\ A \to P \to B\end{array}.$$

  Finally,

$$\frac{\begin{array}{c}\Delta \\ \vdots \\ A \to P\end{array}\ \text{(above)} \qquad \dfrac{\begin{array}{c}\Delta \\ \vdots \\ A \to P \to B\end{array}\ \text{(above)} \quad \dfrac{}{(A \to P \to B) \to (A \to P) \to A \to B}\ \textsf{S}}{(A \to P) \to A \to B}\ \textsf{MP}}{A \to B}\ \textsf{MP}.$$

With the deduction theorem, it is now much easier to prove the fact $(A \to B) \to (B \to C) \to A \to C$. The idea is that we want to write $A \to B$, $B \to C$, and A as hypotheses, use MP to get C, then use the deduction theorem to bring down anything that's not a conclusion.

$$\dfrac{\dfrac{A \quad A \to B}{B}\ \text{MP} \qquad B \to C}{C}\ \text{MP}$$

$$\overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c} A \to B \quad B \to C \\ \vdots \\ A \to C \end{array}$$

$$\overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c} A \to B \\ \vdots \\ (B \to C) \to A \to C \end{array}$$

$$\overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c} \vdots \\ (A \to B) \to (B \to C) \to A \to C \end{array}$$

The great thing about our proof of the deduction theorem is that, if we wanted to, we can actually *construct* the inferences we used, by going backwards using the proof.

As an exercise, please prove K and S using MP, the deduction theorem, and Ref. and Sub.. This means that (MP, deduction theorem) can prove, and be proven with, (MP, K, S) where again we take Ref. and Sub. as implicit. This means these two systems are equivalent to each other!

1.3 Constructive negation

With positive implicational calculus behind us, we now proceed upward. The reason it's called *positive* is because there was no negation, or $\neg A$. To get implicational calculus, we add the proposition $\bot$, or false, to our system. Simply put, $\bot$ is a proposition that can never be proved. If we have $\bot$, we have a contradiction.

We now *define* $\neg A$ to mean $A \to \bot$. That way, we're still using $\to$ as our only connective, which is why it's called *implicational* calculus.

Here's an important philosophical point, even though it may sound simple. We know what it means for A to be true: if we can prove A, it is true. But what does it mean for A to be false? Take a moment to think about this before reading on.

Consider a proposition A, like "For all odd n, n is prime." To show A is false, one way would be to show a counterexample—that is, to bring out 9 and say, look, this integer is odd. If A was true, then 9 would be prime. We also know that as $9 = 3 \cdot 3$ then it is not prime. So we've gotten a contradiction, and so we know the A must be false. This means $A \to \bot$, or $\neg A$. In particular, $\neg A$ means "There exists some odd n such that n is not prime."

Now, consider a different proposition A, like "For all irrational a and b, a^b is irrational." To show A is false, we show a counterexample. Consider $\sqrt{2}^{\sqrt{2}}$. If A was true, then it would be irrational. Now consider $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}}$. It is 2, so it is rational. Contradiction. So A must be false, and therefore $\neg A$.

That's fine. But what is $\neg A$? Is it "There exists irrational a and b such that a^b is rational"? That was not what we proved. If that was what we proved, then where is the a and b? What we *did* prove is "it's not the case that for every irrational a and b, then a^b is irrational." That is what $\neg A$ is, nothing more.

We are getting at the heart of the distinction between **constructive logic** and classical logic. In classical logic, every proposition A must be true or false, so $\neg A$ is just the opposite. In classical logic, a proposition must be true or false, whether or not we can prove it. But in constructive logic:

- To say A is to prove A.
- To say $\neg A$ is to prove that A gives $\bot$.
- To say A is false is to prove that A gives $\bot$.
- To say $\neg A$ is false is to prove that $\neg A$ gives $\bot$.
- To say $\neg A$ is false is to prove that (proving that A gives $\bot$), gives $\bot$.
- To say $\neg A$ is false is to prove that there's no proof that A gives $\bot$.
- To say $\neg\neg A$ is true is to prove that there's no proof that A gives $\bot$.

In constructive logic, A and $\neg\neg A$ mean different things, and it's not always true that $\neg\neg A \to A$! This actually makes sense, given Gödel's incompleteness theorems, which say that there will always be some true statements we can't prove.

This means that, in constructive logic, statements like the law of excluded middle ($A \vee \neg A$) or double negation elimination ($\neg\neg A \to A$) aren't true for all A. On the other hand, even if double negation *elimination* is false, double negation *introduction* ($A \to \neg\neg A$) is true. Our constructive logic is positive implicational logic with $\neg$, so we can still use the deduction theorem. To prove $A \to \neg\neg A$, we need to prove $A \to (A \to \bot) \to \bot$. But this follows from MP, and deduction twice.

$$\frac{A \quad A \to \bot}{\bot}\ \mathsf{MP} \quad \overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c} A \\ \vdots \\ (A \to \bot) \to \bot \end{array} \quad \overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c} \vdots \\ A \to (A \to \bot) \to \bot \end{array}$$

Even more interesting: $\neg\neg\neg A \to A$ isn't always true, but $\neg\neg\neg A \to \neg A$ is! We invite the reader to think about it a little. You start with

$$\frac{((A \to \bot) \to \bot) \to \bot \qquad \dfrac{A \quad A \to \bot}{\bot}\ \mathsf{MP}}{}$$

then use deduction to bring $A \to \bot$ down one step. When we use deduction again, everything that was deduced from A gets removed with it, leaving

$$\frac{((A \to \bot) \to \bot) \to \bot \quad \begin{array}{c} A \\ \vdots \\ (A \to \bot) \to \bot \end{array}}{\bot}\ \mathsf{MP} \quad \overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c} ((A \to \bot) \to \bot) \to \bot \\ \vdots \\ A \to \bot \end{array}$$

from which another deduction finishes the proof.

1.4 The BHK interpretation

We just discussed what it means to prove that $\neg A$. But what about other connectives? What does it mean to prove $A \vee B$, $A \wedge B$? In fact, let's go back even further: what does $A \to B$ *mean*? We now hand it to Simplicio and Salviati.

SIMPLICIO: We've talked a lot about how to write these symbols down and use rules to change them. But what do they actually *mean*?

SALVIATI: I'm not sure I understand what *you* mean.

SIMPLICIO: I mean, we've worked with the symbols only by following rules blindly. The rules MP, K, and S don't "mean" anything beyond shuffling symbols around. We keep saying that $\to$ means "implies", but what *does* that word mean?

SALVIATI: Ah, you're wondering about the difference between *syntax* and *semantics*. So far, yes, we've only discussed syntax. Now let's talk about semantics, or what it means for a formula to be true.[1] Let's speak intuitively. If I asked you to prove $A \to B$, what would you do?

SIMPLICIO: Well, first I'd assume that A is true. Using that, I'd show that B is true.

SALVIATI: Yes. So $A \to B$ is a proof that, assuming A is true, proves that B is true. In particular, if you knew how to prove $A \to B$, and I gave you a proof of A, you would know how to prove B, right?

SIMPLICIO: Yeah, I just chain the proofs together. I write the proof of A. Then I write the proof of $A \to B$, which ends up proving B.

SALVIATI: Precisely! So $A \to B$ is a function.

SIMPLICIO: A function?

SALVIATI: Yes, a function. If your proof of $A \to B$ was a function f, and I gave you a proof p of A, then $f(p)$ would be "inserting" the proof of A into f. This results in a proof of B. Does that make sense?

SIMPLICIO: Kinda? It's weird thinking about proofs as things we can feed into functions. Or proofs *themselves* being functions.

SALVIATI: Well, you've done some programming before, right? What *is* a function, in a computer program? How does a function, say, reverse a list of words?

SIMPLICIO: Well, it's a list of instructions. You write down a list of instructions that define the function. You tell the function to take the list of words, write down the last word, then write down the word before that, and so on.

SALVIATI: Yes, exactly! Now think about a proof. A proof is just a list of words. A function *can* operate on lists of words, and a function *itself* is just a list of words.

SIMPLICIO: Certainly a function that operates on proofs written on paper would be rather complicated. But sure, in theory, it's possible.

SALVIATI: Let's move on. How would you prove $A \wedge B$?

SIMPLICIO: I'd hand you a proof of A, and then hand you a proof of B.

SALVIATI: Exactly. So $A \wedge B$ is an ordered pair of proofs (p, q), where p is a proof of A, and q is a proof of B. What about $A \vee B$?

SIMPLICIO: I'd tell you which of A or B I'm proving, then I prove it. So, is $A \vee B$ an ordered pair (P, p), where P is either A or B, and p is the proof of that?

SALVIATI: Yes! Now consider $\neg(A \wedge \neg A)$. This expands to $(A \wedge (A \to \bot)) \to \bot$. How would you prove this, intuitively?

SIMPLICIO: Well, the outermost thing is a $\to$. So I'll assume that $A \wedge (A \to \bot)$ is true, and I'm trying to prove $\bot$. But if I have $A \wedge (A \to \bot)$, I know A, and I also know

[1] Usually, "semantics" in logic refers to *formal semantics*. This section discusses a basis for proof-theoretic semantics. Okay, actually I'm not quite sure, but I think this is true?

$A \to \bot$. So I can conclude $\bot$, done.

SALVIATI: Now, how do we translate this in the perspective we've been using, with the functions and all? Let me start. The outermost thing is a $\to$. You want to give me a function that takes $A \wedge (A \to \bot)$ and outputs $\bot$.

SIMPLICIO: Okay. Well, the input to this function is (p, q), where p is a proof of A, and q is a proof of $A \to \bot$, right? Then I can—and this is kind of the weird part—I can write $q(p)$, which gives me a proof of $\bot$?

SALVIATI: Yes. So the proof is just the function that takes an ordered pair (p, q) and returns $q(p)$. Or $(p, q) \mapsto q(p)$.

SIMPLICIO: Wait a second! This function doesn't refer to A or $\bot$ or any of those at all?

SALVIATI: No, because we don't have to. *Your* proof didn't refer to the "meanings" of A or $A \to \bot$ or $\bot$. All your proof did was just take $A \wedge \neg A$, split it, then use MP. We wouldn't expect the function proof to be any different. Anyway, let's do one more example. How would you prove $A \to \neg\neg A$?

SIMPLICIO: Well, this is $A \to ((A \to \bot) \to \bot)$. So first, we assume A is true, we're trying to prove that $(A \to \bot) \to \bot$. Well, to prove *that*, let's assume that $A \to \bot$ is true too, right?

SALVIATI: Yep! Now we're assuming both A and $A \to \bot$.

SIMPLICIO: Alright. But since we're assuming both of those, we know that $\bot$ is true, just like we wanted to prove. Done!

SALVIATI: Okay, now let's speak of functions. You want to give me a function that takes A and outputs $(A \to \bot) \to \bot$.

SIMPLICIO: Let's say the input is p, where p is a proof of A. Then I need to return... another function?

SALVIATI: Yeah, another function. This function will take $A \to \bot$ and output $\bot$.

SIMPLICIO: So let's say this function's input is q. Then I need to return $q(p)$, right? So this is the function $q \mapsto q(p)$.

SALVIATI: So what's the whole function?

SIMPLICIO: Is it $p \mapsto (q \mapsto q(p))$? It's kind of weird having this "function that returns a function" thing, but I can see why it works. First the function takes in a proof of A, then it takes in a proof of $A \to \bot$, and then it outputs a proof of $\bot$.

This is the **Brouwer–Heyting–Kolmogorov interpretation**: interpreting $A \to B$ as a function, $A \wedge B$ as an ordered pair, and $A \vee B$ as another ordered pair. We'll say "BHK interpretation" from now on, or even simply "BHK".

Simplicio gave us a proof of $A \to \neg\neg A$. Compare it with the proof given to us by the deduction theorem. Now what about $\neg\neg A \to A$? Its proof would be a function that takes $(A \to \bot) \to \bot$ and returns A. But there isn't a function that does this. This corresponds to the fact that, in constructive logic, $\neg\neg A \to A$ isn't always true.

1.5 Lambda calculus

Before we finally begin our dive into types, we need to talk about the nature of functions. Already in our previous discussion we had to talk about functions that *return functions*, which is a weird concept to wrap your head around at first. So let's talk about the **lambda calculus**, which might help.

You might be used to writing functions as $f(x) = x^2$, which is read as "the function that takes x, and returns x^2". In lambda calculus, we'll often not give functions names, and instead write $x \mapsto x^2$. That arrow, $\mapsto$, is read as "maps to" or "goes to". And just like we write $f(3) = 9$ to apply the function to 3, we write $(x \mapsto x^2)(3) = 9$. If we *really* wanted to name our functions, we can write $f = x \mapsto x^2$.

Function creation is called **abstraction**. The reason it's called that is that a function "abstracts" a computation. Consider 3^2, $(-1)^2$, and π^2. These are all squaring some number. We can abstract the number into a function, like $x \mapsto x^2$.

In lambda calculus, we usually use the letter λ to write these functions. So instead of $x \mapsto x^2$, we'd have written $\lambda x.x^2$. I personally dislike this notation and we won't use it in these notes, but most treatments of lambda calculus will use λ.

Functions themselves can take functions as input. Consider the function $f \mapsto f(f(5))$. This takes in a function and applies it to 5, twice. So

$$(f \mapsto f(f(5)))(x \mapsto x^2) = (x \mapsto x^2)((x \mapsto x^2)(5)) = (x \mapsto x^2)(5^2) = (5^2)^2 = 625.$$

Note how we computed this. When we compute an **application** of a function, like $f \mapsto f(f(5))$ on $x \mapsto x^2$, we take the **bound variable** f, then change all the fs in the expression with whatever our input is, $x \mapsto x^2$. This is known as **beta reduction**.[2] We then applied beta reduction *again*, changing the bound variable x to the input 5. Then we apply beta reduction one more time.

We're already getting lots of parentheses, so when it's clear, we will omit parentheses in application. Instead of writing $f(x)$ to apply f to x, we will just write fx.

Functions themselves can also return functions. Consider the functions $y \mapsto 3 + y$, or $y \mapsto -1 + y$, or $y \mapsto \pi + y$. These are all the same computation! So we can abstract the number into a function, giving us $x \mapsto (y \mapsto x + y)$. The way we'd apply this would look like

$$(x \mapsto (y \mapsto x + y))(3)(5) = (y \mapsto 3 + y)(5) = 3 + 5 = 8.$$

Some functions take multiple inputs. For example, you might think of the addition function as $f(x, y) = x + y$. While we can write this as $(x, y) \mapsto x + y$, when we do lambda calculus, we prefer all of our functions to have only *one* input. To do functions with multiple inputs, we give our inputs one at a time. So we'd write the addition function as $x \mapsto (y \mapsto x + y)$. Hey, doesn't that look familiar?

When we turn a function with multiple inputs to several functions that take one input at a time, that's called **currying**. It's because of currying that we make $\mapsto$ **right-associative**. This means that

$$x \mapsto y \mapsto z \mapsto w \text{ should be interpreted as } x \mapsto (y \mapsto (z \mapsto w)),$$

because this is the curried version of $(x, y, z) \mapsto w$. When you see a chain of $\mapsto$s, we can think about it as "a function that returns a function", but we can also think of it

[2] There's also alpha conversion, and eta expansion. In the calculus of constructions, there's also iota reduction, delta reduction, and zeta reduction. We won't talk about those.

as "a function with multiple inputs". The BHK interpretation turns $\to$ to $\mapsto$, which is why they're both right-associative.

It's also because of currying that we make function application **left-associative**. This means that

$$fghx \text{ should be interpreted as } ((fg)h)x.$$

Putting it together, we'd say

$$\begin{aligned}(a \mapsto b \mapsto ab)(f \mapsto f5)(x \mapsto x+3) &= (b \mapsto (f \mapsto f5)b)(x \mapsto x+3)\\ &= (b \mapsto b5)(x \mapsto x+3)\\ &= (x \mapsto x+3)5,\end{aligned}$$

which is $5 + 3 = 8$.

1.6 Curry–Howard, part 1

Now we draw our first big connection. You may have noticed the way we proved $A \to \neg\neg A$ with the deduction theorem, and the BHK interpretation as $p \mapsto q \mapsto qp$, are related. We can make this precise. Here, we repeat the deduction proof, but we annotate each proposition with its BHK interpretation.

$$\dfrac{\overset{p}{A} \quad \overset{q}{A \to \bot}}{\dfrac{qp}{\bot}}\ \mathsf{MP} \quad \overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c}\overset{p}{A}\\ \vdots\\ q \mapsto qp\\ (A \to \bot) \to \bot\end{array} \quad \overset{\text{deduct.}}{\Longrightarrow} \quad \begin{array}{c}\vdots\\ p \mapsto q \mapsto qp\\ A \to (A \to \bot) \to \bot\end{array}$$

Please convince yourself that **MP is application** and **deduction is abstraction**. This is an important correspondence known as the **Curry–Howard correspondence**, and this won't be the first time we'll encounter it. It's important enough that it gets its own section, even if it's a short one.

As an exercise, consider our proof of $\neg\neg\neg A \to \neg A$, and write it out in BHK. The easiest way is to work from the deduction-style proof we wrote earlier. You should've gotten an answer like $p \mapsto q \mapsto p(r \mapsto rq)$; check that it also matches what we'd expect from the BHK interpretation.

2 Types

2.1 Curry–Howard, part 2

So far in the Curry–Howard correspondence, we've shown a relationship between deduction-style proofs and lambda calculus. In particular, given a deduction-style proof, we can come up with a corresponding proof in the BHK interpretation, that happens to be a function. But for this to be a correspondence, we want to be able to go the other way around as well! Given a function in the lambda calculus, does it represent a proof of some logical fact?

We can do this for some kinds of functions. Let's consider the example $p \mapsto q \mapsto qp$. The qp at the end, an application, must be a result of MP. Hence, if p is a proposition A, then q must be $A \to B$, for some B. By reversing the abstractions, the theorem must have been $A \to (A \to B) \to B$. Note that this is *more* general than the statement we originally used $p \mapsto q \mapsto qp$ to prove, which was $A \to \neg\neg A$, or $A \to (A \to \bot) \to \bot$.

On the other hand, we can't do this for all possible functions. Consider a function like $x \mapsto xx$. Again, we know from MP that if x was some proposition A, then x must be $A \to B$, for some B. But this is impossible! So if the correspondence isn't between deduction-style proofs and lambda calculus, what is it a correspondence between? This is solved by introducing types.

2.2 Inference and inhabitation

The main concept of **type theory** is that all mathematical objects, or **terms**, have a **type**. For example, we can say that π has type real, indicating that it's a real number. We write this as $\pi : \mathsf{real}$, where the : is read as "has the type".

As another example, $3 : \mathsf{int}$, where int is the type of integers. Two terms of different types have to be different, so $3 : \mathsf{real}$ is a different 3 than $3 : \mathsf{int}$. Functions are also terms, and have types determined by their input and output. For example, $x \mapsto \lfloor x \rfloor : \mathsf{real} \to \mathsf{int}$. Again, terms with different types are different; the previous function is different from $x \mapsto \lfloor x \rfloor : \mathsf{real} \to \mathsf{real}$.

Functions restrict the types of their inputs. This seemingly simple fact about functions is very, very important! We cannot call a function on inputs that are of different type than it accepts. By adding types to lambda calculus, we get what is known as the **simply typed lambda calculus**.

As we've seen in (untyped) lambda calculus, functions can also take functions as input. We can also assign types to these functions, like $f \mapsto ff5 : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$. Also, functions can return functions, so

$$x \mapsto y \mapsto x + y : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$$

is a possible function. Once again, $\to$ is right-associative, so $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ should be interpreted as $\mathsf{int} \to (\mathsf{int} \to \mathsf{int})$.

This allows us to do **type inference**, to determine what the type of a function term is, even if we're not told directly what it is. If we know $f\pi = 3$, we can infer $f : \mathsf{real} \to \mathsf{int}$. Even if all we're given is $f\alpha : B$, for some $\alpha : A$, we can tell that f must have type $A \to B$. Note that it's possible that A and B are the same type, but the most general possible type would be $A \to B$.

Now, let's go back to $p \mapsto q \mapsto qp$. What is the most general possible type of this function? We start inside. From qp, we know if $p : A$ and $qp : B$, then q must have

the type $A \to B$. This means the type is

$$(\text{type of } p) \to (\text{type of } q) \to (\text{type of } qp), \text{ or } A \to (A \to B) \to B.$$

Looks familiar? As an exercise, use a similar to procedure to find the type of $p \mapsto q \mapsto p(r \mapsto rq)$. You'd start with something like, "From rq, we know if $q : A$, and $rq : B$, then r must have type $A \to B \ldots$"

There are reasons why the simply typed lambda calculus is, in some ways, nicer than lambda calculus. In lambda calculus, there are certain kinds of functions that aren't well-behaved. Consider $\omega = x \mapsto xx$. What happens when we try to evaluate $\omega\omega$? We get an infinite loop: by calling ω with ω, the result is still $\omega\omega$. In some deep sense, this comes from the fact that we can't assign a valid type to ω. What happens when we try to infer its type? From xx, we see x is a function that takes in the type of x. Please convince yourself that this is impossible.

In a sense, the simply typed lambda calculus is what you get when you ask, *how can we enforce every function to be the result of a BHK interpretation*? The rule you get for application—the fact that function inputs must match types—is precisely what MP corresponds to. There is also a rule for abstraction, which prevents silly functions like $x \mapsto y$, because y isn't defined.

Type inference is the problem of figuring out a type based on the term. The opposite problem—figuring out whether a term of a certain type exists—is **type inhabitation**. If there is a term of a given type, we say the type is **inhabited**. Generally, type inhabitation is harder than inference, but we can still do it in some cases.

In particular, the previous section showed that if we have a deduction-style proof, we can convert it to a term. This means that facts like K and S, can be converted. Please check that

$$\mathsf{K} = p \mapsto q \mapsto p \text{ and } \mathsf{S} = p \mapsto q \mapsto r \mapsto pr(qr)$$

are functions that have the appropriate types.[3] On the other hand, if you try to find a term that has the type $((A \to B) \to A) \to A$, you'll see that you can't.

2.3 Curry–Howard, part 3

SIMPLICIO: Well, big deal. Theorems in zeroth-order implicational logic correspond to functions in simply typed lambda calculus, and vice versa. Is that all the Curry–Howard correspondence is about?

SALVIATI: Remember how we showed that MP, K, and S were enough to show the deduction theorem? That means that all inhabited types in simply typed lambda calculus can be written in Ks and Ss. For example, $\mathsf{SKK} : A \to A$.

SIMPLICIO: That's cool and all. But again, I'm left wondering: what does this actually *mean*? Once again, we've only established a connection between *syntax*. We haven't talked at all about what this correspondence means for *semantics*.

SALVIATI: Alright. What does it mean to say $\pi : \mathsf{real}$?

SIMPLICIO: It means that π is a term that is real.

[3] This is why these are named K and S, by the way! I think Schönfinkel named it K for the German for "constant", because it returns a constant function—no matter what q is, it always returns p. And the reason it's named S is from the German for "fuse".

SALVIATI: Right. Now, in $p \mapsto q \mapsto qp$, we saw that $p : A$. But what *is* A, other than just being some type? What does it represent, according to Curry–Howard?

SIMPLICIO: It represents the proposition A. In particular, it represents the proof of the proposition A. Wait. So that means that p, this term itself, *is* the proof of A?

SALVIATI: That's right. Propositions *are* types. A term of a certain type is a *proof* that the proposition is true. We can read : not only as "has the type" but "is a proof of". In particular, a proposition can be proven if and only if its type is inhabited.

SIMPLICIO: I guess this kinda makes sense, at least with the BHK interpretation. So one way to read $q : A \to B$ is a function that takes a term with type A and returns a term with type B. Another way is a proof that A implies B. So that's why we use $\to$ for both the function type and for "implies"! But how do you interpret a statement like $\pi : \mathsf{real}$?

SALVIATI: That means π is a proof of real—that there is a real number. The term is a **witness** that there is a term of type real. Showing me a term with a certain type is the same as showing that the corresponding proposition is true. This is why, to prove $A \to \neg\neg A$, it is enough to show me a function that has the type $A \to \neg\neg A$.

SIMPLICIO: Okay, sure. Well, what if I just say that R is the type of the statement of the Riemann hypothesis. And I say, oh look, here's $r : R$. Have I proven the Riemann hypothesis?

SALVIATI: Well, what is r? Are you building it up from other terms, or are you just creating it out of thin air? To create a type out of thin air is the same as making it an **axiom**. You have to start with *something*, right?

SIMPLICIO: Alright. We've discussed how to interpret a single type, and why "implies" corresponds to the type of functions. What about something like $\bot$?

SALVIATI: We define the type $\bot$ as a type that has no terms.

SIMPLICIO: If this is *really* $\bot$, then there should be a proof that $\bot \to A$, for any proposition A. What's the corresponding function, then?

SALVIATI: Well, we want $\bot \to A$ to be a function that, for every term of type $\bot$, gives us a term of type A. The empty function works, because there *are* no terms of type $\bot$. So, vacuously, it can have any output type we want, like A.

SIMPLICIO: Sure. But how do $\wedge$ and $\vee$ correspond to types?

SALVIATI: Well, in this case, you can go from the BHK interpretation to *create* a type. We talked about how the proof of $A \wedge B$ should be an ordered pair of proofs, (p, q), where $p : A$ and $q : B$. Using Curry–Howard to translate, the corresponding type should consist of the ordered pairs of terms (p, q), where $p : A$ and $q : B$.

SIMPLICIO: Oh, but there's a name for that! It's $A \times B$, right? At least when A and B are sets, the set $A \times B$ is the set of ordered pairs of elements.

SALVIATI: Yeah, and we use the same notation in type theory. We call it the **product type** $A \times B$. This is in analogy to the product of sets. Note that, when A and B are finite sets, the size of $A \times B$ is the product of the sizes of A and B. That's where the name "product" comes from.

SIMPLICIO: That makes sense. Except in this case, A and B are types, not sets. But what about $\vee$? It's either something like (A, p) where $p : A$ or (B, q) where $q : B$. These are ordered pairs, but there's no nice way to write this as a product.

SALVIATI: No. In this case we call it the **sum type** $A + B$, and just define it like that. Again, it's named "sum" in analogy to set theory, because when A and B are finite sets, the size of $A + B$ is the sum of the sizes of A and B. This operation is better known as disjoint union.

SIMPLICIO: So to recap: propositions are types, $\to$ is $\to$, $\bot$ is the empty type, $\wedge$ is $\times$, and $\vee$ is $+$? That settles everything we've discussed with the BHK interpretation. But that's still not all of logic! What about $\forall$ and $\exists$?

2.4 First-order logic

So far, we've only discussed zeroth-order logic. Now that we'll start discussing $\forall$ and $\exists$, we need to move to **first-order logic**.

In first-order logic, we can use $\forall$ and $\exists$, which are known as quantifiers. But we can only quantify over variables. In particular, we can't quantify over propositions. To make the distinction, we'll use $x, y, z, \ldots$ for variables and $A, B, C, \ldots$ for propositions. So $\forall x$ is allowed, in first-order logic, but $\forall A$ is not.

This is an important distinction, which we'll keep in mind as we go on. But it's also quite subtle, and I had a hard time wrapping my head around it at first. It's true that we didn't pay too much attention to this. We proved things like $(A \to B) \to (B \to C) \to A \to C$ without thinking too hard about what A, B, and C are. Strictly speaking, we didn't prove "For all propositions A, B, and C, ..." That is a fact of second-order logic. Instead, what we showed, was that for *these specific* A, B and C, this fact was true.

And yes, it's true that we *can* prove this for *any* possible combination of A, B, and C, simply by following the same steps. But the language of zeroth-order logic doesn't let us say anything about *all* A, B, and C. To do that, we'll have to build up more tools. To go to second-order logic, we first need to talk about first-order logic.

What *is* a variable? Well, imagine a proposition like "$2 \cdot 1$ is even." Now imagine another proposition, like "$2 \cdot 21$ is even." And imagine "$2 \cdot 3$ is even." These are all (zeroth-order) propositions of the same form: "$2x$ is even", where x is some integer. If we want to say something about all integers in general, we can say "For all integers x, $2x$ is even." Let $\mathsf{even}(y)$ be "y is even". A proposition like $\mathsf{even}(y)$, with a variable like y in it, is called a **predicate**. Our statement can now be written as $\forall x : \mathbb{Z}, \mathsf{even}(2x)$.

What's the BHK interpretation? Well, how would you prove $\forall x : S, Ax$? You need to produce a proof of Ax, no matter what x in S that I give you. This means $\forall x : S, Ax$ needs to be a *function*: it converts elements x of S to a proof of Ax. In particular, $\forall x : \mathbb{Z}, \mathsf{even}(2x)$ needs to be a function that converts an integer x to a proof that $2x$ is even. Now consider

$$(\forall x : S, Ax \to Bx) \to (\forall x : S, Ax) \to (\forall x : S, Bx).$$

How do we prove this?[4] Again, the proof will be a function, so let's name the inputs. Suppose that p is $\forall x : S, Ax \to Bx$ and q is $\forall x : S, Ax$. We need to show that

[4] We'll use the standard convention that $\forall$ and $\exists$ stretch out as far as they can. So $\forall x : S, Ax \to Bx$ should be interpreted as $\forall x : S, (Ax \to Bx)$, which is different from $(\forall x : S, Ax) \to Bx$.

$\forall x : S, Bx$. This itself should be a function, so let's name its input x. So far, we have $p \mapsto q \mapsto x \mapsto$. Now we need to get Bx. Well, we get that qx is Ax. Also, px is $Ax \to Bx$, so $px(qx)$ is Bx. Our proof is $p \mapsto q \mapsto x \mapsto px(qx)$. Looks familiar?

Similarly, how do we prove $\exists x : S, Ax$? To show that something exists, you have to give me the thing you say exists. A proof of $\exists x : S, Ax$ should be an ordered pair of x, and a proof of Ax.

Let's say that $\mathsf{prime}(n)$ is the predicate "n is prime", and $\mathsf{factor}(a, b)$ is the predicate "a is a factor of b". You can write statements like $\exists n : \mathbb{N}, \mathsf{prime}(n) \wedge \mathsf{factor}(n, 57)$, and a proof of this could be $(3, (p, q))$, where p is a proof of $\mathsf{prime}(3)$ and q is a proof of $\mathsf{factor}(3, 57)$. We can even write things like $\forall n : \mathbb{N}, \exists p : \mathbb{N}, \mathsf{prime}(p) \wedge \mathsf{factor}(p, n)$, the statement that every natural number has a prime factor.

As an example, consider $(\exists x, A \wedge Bx) \to (A \wedge \exists x, Bx)$. How do we prove this? The proof is a function. Its input would be something like $(x, (p, q))$, where p is a proof of A and q is a proof of Bx. What we want to return is this shuffled around. Please check that $(x, (p, q)) \mapsto (p, (x, q))$ makes sense as a proof of this fact.

We can prove relationships between these two. For example, how would we prove $(\exists x : S, \neg Ax) \to (\neg\forall x : S, Ax)$? This is a pretty complicated example, so please think about it first. It might help to interpret this intuitively.

So, the proof is a function. If we expand all the $\neg$s, we get

$$(\exists x : S, Ax \to \bot) \to (\forall x : S, Ax) \to \bot.$$

This function has two inputs. The first is an ordered pair, so we'll name it (x, p), where p is the proof of $Ax \to \bot$. The second input is $\forall x : S, Ax$, a function, which we'll name q. We need to get $\bot$. Indeed, we see that qx is Ax, and thus $p(qx)$ is $\bot$. So this function is $(x, p) \mapsto q \mapsto p(qx)$.

Interestingly, while it is true that $(\exists x : S, \neg Ax) \to (\neg\forall x : S, Ax)$, it is *not* true, at least in the BHK interpretation, that $(\neg\forall x : S, Ax) \to (\exists x : S, \neg Ax)$. This is another one of those things where classical logic is different from constructive logic. On the other hand, it *is* true that $\neg\exists x : S, Ax$ is equivalent to $\forall x : S, \neg Ax$; can you prove it?

In summary, in the BHK interpretation, $\forall x : S, Ax$ is a function that takes a variable x and returns a proof of Ax, and $\exists x : S, Ax$ is an ordered pair of a variable x and a proof of Ax.

2.5 Dependent types

Let's go back to Simplicio. What do $\forall$ and $\exists$ correspond to in type theory? Let's do what Salviati did, and use Curry–Howard to find the corresponding type.

We know that $\forall x : S, Ax$ should be a function. It takes in the variable x to produce a proof of Ax. Now we have to be very careful here. Ax is a proposition, so it must be a type. But x is a variable, not a proposition! So it shouldn't be a type. The natural way to interpret this is to make x a *term* of type S.

We get that $\forall x : S, Ax$ is a function that takes the term $x : S$ and converts it to $p : Ax$. What is the type of this function? We could try $S \to Ax$. But this isn't well-defined, because while we know what S and A are, we don't know what x is. We could try to write its type as $x \to Ax$. Now the information from x is contained in the type. But this isn't right either, because x is the *input*, not the type of the input.

So this is a different kind of function, one we can't write using our current notation. In this case, we say this function has a **dependent type**—its type *depends* on a certain

term. This specific case, where it depends on the function input, is a **dependent function type**. The notation we'll use for it is the same as the logical notation, so we'll write the type of such a function as $\forall x : S, Ax$.

I know this sounds pretty abstract, so let's think about an example. Let's define a type of integer tuples, which we'll call $\mathsf{inttup}(n)$. The type will include the length of the tuple, which is n. Now we can make a function like $n \mapsto (0, \ldots, 0)$, where there are n zeroes. This takes a natural number $n : \mathsf{nat}$ and returns the tuple $(0, \ldots, 0) : \mathsf{inttup}(n)$. It wouldn't be enough information to tell us that the function has type $\mathsf{nat} \to \mathsf{inttup}(n)$. What's the n in $\mathsf{inttup}(n)$? This is why we write

$$n \mapsto (0, \ldots, 0) : \forall n : \mathsf{nat}, \mathsf{inttup}(n).$$

Here's another example. Let's say there's a type $\mathsf{even}(n)$. Propositions are types, so we won't differentiate between $\mathsf{even}(n)$, the proposition that n is even, and the *type* $\mathsf{even}(n)$. The type means that, if $p : \mathsf{even}(n)$, then p is a *proof* that n is even.

Now the type $\forall n : \mathsf{int}, \mathsf{even}(2n)$ would be the type of functions that convert an integer $n : \mathsf{int}$ to a proof $p : \mathsf{even}(2n)$. We can't write down such a function yet, since we haven't defined $\mathsf{even}(n)$, but we will eventually.

In particular, this means that predicates like $\mathsf{even}(n)$ correspond to a dependent type. So what does a dependent type like $\mathsf{inttup}(n)$ correspond to? Just like how the type real is the proposition "there exists a real number", the type $\mathsf{inttup}(n)$ is the proposition "there exists a tuple of integers with length n." So **predicates correspond to dependent types**.

Note the general function type $A \to B$ is just a special case of the dependent function type $\forall x : A, B$, when the output type B does not depend on x. For example, you can consider the function $x \mapsto \lfloor x \rfloor : \mathsf{real} \to \mathsf{int}$. Please convince yourself that the type of this function can also be $\forall x : \mathsf{real}, \mathsf{int}$. The type of int just doesn't depend on the value of x. Because of this:

$$A \to B \text{ is an abbreviation for } \forall x : A, B.$$

What about $\exists x : S, Ax$? It's an ordered pair of the variable x, and a proof of Ax. Again, we can try writing its type as $S \times Ax$, but Ax depends on what x is. Neither does $x \times Ax$ work, because x is a term, and not a type. In this case, this is a **dependent product type**, where the type of the second element depends on the value of the first element.[5] We again use the notation $\exists x : S, Ax$ for this kind of ordered pair.

As an example, the type $\exists n : \mathsf{int}, \mathsf{even}(n)$ is the type of ordered pairs where the first element is $n : \mathsf{int}$, and the second element is $p : \mathsf{even}(n)$, a proof that n is even. Again, the dependent product type is a generalization of the normal product type. Please convince yourself that the definition

$$A \times B \text{ is an abbreviation for } \exists x : A, B$$

makes sense.

[5] Some sources will write $\Pi x : S, Ax$ for what we call the dependent function type, and call *that* the dependent product type. Then they'll write $\Sigma x : S, Ax$ for what we call the dependent product type, and call that the dependent sum type. I don't like this terminology. Our choice makes it so that function and product types are special cases of their dependent versions.

2.6 Polymorphism

We've been pretty careful about specifying we were only quantifying over variables, and not propositions. When we move up to **second-order logic**, we can now quantify over propositions. And this means *all* propositions; a proposition can quantify itself!

We'll use the same notation as previously, except we will write Prop to refer to the type of propositions. So for example, we can write $\forall A : \mathsf{Prop}, A \to A$. This is the statement "for all propositions A, A implies A." Again, *all* propositions.

Let's try to apply the BHK interpretation to $\forall A : \mathsf{Prop}, A \to A$. We want a function that takes in a proposition A, and then returns a proof of $A \to A$. Well, such a function could be $A \mapsto x \mapsto x$. It takes in a proposition A—the proposition itself, and not a proof of it—then takes in a proof of A, and then returns the same proof.

By Curry–Howard, this should correspond to some type, but what? We can try to write it as $A \to A \to A$. But that doesn't work, because the first input is A *itself*, not a term of type A. We know from Curry–Howard that propositions are types, so A is a general type. That means that this function takes in a type as input, which isn't something we've seen before.[6]

To write its type, we need to refer to the type of types. As propositions are types, we call this type Prop. Now, is the type $\mathsf{Prop} \to A \to A$? That doesn't work either, because we don't know what A is. Again, the solution is to use a dependent type:

$$A \mapsto x \mapsto x : \forall A : \mathsf{Prop}, A \to A.$$

So, what's the difference? Our dependent types only used to quantify over a single type, like $\forall x : S, Ax$. This allowed us to write functions where the output type depends on an input *term*. But now we can quantify over types themselves, like $\forall A : \mathsf{Prop}, A \to A$. This makes functions have output type depending on an input *type*. Again, note the difference we're drawing between variables, which correspond to terms, and propositions, which correspond to types.

A function that takes a type as input is **polymorphic**. This turns out to be a very natural concept for functions. You can consider, say, the identity function, $x \mapsto x$. Without polymorphism, you'd need to have an identity that's $\mathsf{int} \to \mathsf{int}$, and another identity that's $\mathsf{real} \to \mathsf{real}$, and another identity that's $\mathsf{nat} \to \mathsf{nat}$, and so on. But now, we can just use the polymorphic function $A \mapsto x \mapsto x$, with type $\forall A : \mathsf{Prop}, A \to A$. If we want an identity function for ints, we just plug it in: $(A \mapsto x \mapsto x)(\mathsf{int})$ would be $x \mapsto x : \mathsf{int} \to \mathsf{int}$.

Note that **quantifying over propositions corresponds to polymorphic functions**. We saw the example $\forall A : \mathsf{Prop}, A \to A$, which is both "for all propositions A, A implies A", and the type of the polymorphic function $A \mapsto x \mapsto x$. We can go back to the functions K and S, and see them as they truly are. They are both propositions that quantify over propositions, *and* polymorphic functions:

$$\begin{aligned}
\mathsf{K} &= A \mapsto B \mapsto p \mapsto q \mapsto p \\
&: \forall A : \mathsf{Prop}, \forall B : \mathsf{Prop}, A \to B \to A, \\
\mathsf{S} &= A \mapsto B \mapsto C \mapsto p \mapsto q \mapsto r \mapsto pr(qr) \\
&: \forall A : \mathsf{Prop}, \forall B : \mathsf{Prop}, \forall C : \mathsf{Prop}, (A \to B \to C) \to (A \to B) \to A \to C.
\end{aligned}$$

[6] In most sources, this would be written as $\Lambda A.\lambda x.x$, with the Λ indicating that A is a type, and not a term. We instead use the type of the function, in this case $\forall A : \mathsf{Prop}, A \to A$, to differentiate which inputs are types and terms.

3 The calculus of inductive constructions

3.1 Type constructors

Let's take a step back and think about what we've done so far.

- We began with the simply typed lambda calculus, with things like $x \mapsto \lfloor x \rfloor : \mathsf{real} \to \mathsf{int}$. Functions take a term (x) and return a term ($\lfloor x \rfloor$). These correspond to implication.
- We then added dependent types, with things like $n \mapsto (0, \ldots, 0) : \forall n : \mathsf{nat}, \mathsf{inttup}(n)$. Dependent types take a term (n) and return a type ($\mathsf{inttup}(n)$). These correspond to predicates.
- We then added polymorphism, with things like $A \mapsto x \mapsto x : \forall A : \mathsf{Prop}, A \to A$. Polymorphic functions take a type (A) and return a term ($x \mapsto x$). These correspond to quantifying propositions.

If we think about this table, we've filled in three out of the four entries:

(row) → (column)	terms	types
terms	(normal) functions	dependent types
types	polymorphic functions	?

The last piece we need are functions that go from types to types. We'll start by giving an example of why such a thing would be useful, then we'll try to recover what they mean logically. Note that this is the opposite of what we've been doing so far; usually we start with the logic and find the type theory equivalent. But Curry–Howard means we can do both directions, so we'll try doing the other direction this time.

We have dependent types, which are types that depend on terms, like $\mathsf{inttup}(n)$, the type of tuples of integers of length n. We can imagine a different approach to this type, like $\mathsf{triple}(A)$. Here A is a type, and $\mathsf{triple}(A)$ is the type of triples of terms of type A. For example, this would mean that $\mathsf{inttup}(3)$ and $\mathsf{triple}(\mathsf{int})$ are both triplets of integers. We can imagine a function like $A \mapsto x \mapsto (x, x, x)$, which takes in a type A, a term $x : A$, and then returns a triplet of xs. Please check that this makes sense:

$$A \mapsto x \mapsto (x, x, x) : \forall A : \mathsf{Prop}, A \to \mathsf{triple}(A).$$

A type that depends on another type is a **type constructor**. The name comes from the fact that it constructs a new type out of an old one, like $\mathsf{triple}(\mathsf{int})$ from int. Note the difference with a dependent type: a dependent type makes a new type out of a *term*, but a type constructor makes a new type out of a *type*. Two other examples of type constructors include the product and sum types: both of them take two types and return a new type. In fact, $\to$ itself is a type constructor!

Now, what would this correspond to logically? Let's compare type constructors to dependent types. Dependent types take terms and return types. By Curry–Howard, terms correspond to variables, and types correspond to propositions. That means dependent types correspond to something that takes a variable and returns a proposition. But this is precisely what a predicate is!

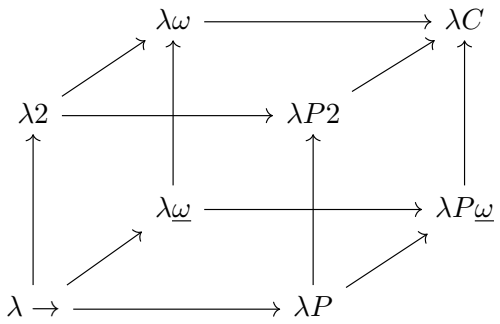
By analogy, a type constructor should be something that takes a *proposition* and returns another proposition. We want some kind of predicate over *other propositions*.

This requires us going up another step of logic, because so far we've only seen predicates over variables. For example, we could come up with a predicate like $\mathsf{excludedmiddle}(A)$, which means "$A \vee \neg A$." Then we could make statements like $\forall A : \mathsf{Prop}, \mathsf{excludedmiddle}(A)$, which is the law of excluded middle.

So we can think of predicates over propositions. I don't think there's a good name for these in the literature. Let's call these **higher-order predicates**, to differentiate them from predicates over variables. By adding higher-order predicates into our logic, we get what we call **higher-order logic**.[7]

3.2 The lambda cube

The three things we added—dependent types, polymorphism, and type constructors—can be combined in different sorts of ways to produce different sorts of systems. The three things are, in fact, fully independent of each other, leading to $2^3 = 8$ different possible systems. These systems are presented in a cube like this.



In the lower left corner, $\lambda \to$, is the simply typed lambda calculus. By going from left to right, we add dependent types. By going from bottom to top, we add polymorphism. By going from bottom-left to top-right, we add type operators. Each of the individual systems has a name, but that doesn't particularly matter for us—what matters more is the fact that these three things are independent, and can be combined.

For example, we can conceive of combining type constructors and dependent types, to produce a type that depends on both another type and a term. Let's say $A : \mathsf{Prop}$. Then we can consider the type of tuples with n terms of type A, which we'll call $\mathsf{tup}(A, n)$. Note that, as types, $\mathsf{tup}(\mathsf{int}, n)$ is the same type as $\mathsf{inttup}(n)$ from earlier. Now you can define a function like $A \mapsto n \mapsto x \mapsto (x, \ldots, x)$, which takes in A, $n : \mathsf{nat}$, and then a term $x : A$ returning a tuple of n xs. This function would have type $\forall A : \mathsf{Prop}, \forall n : \mathsf{nat}, A \to \mathsf{tup}(A, n)$.

We can also conceive of combining type constructors and polymorphism. For example, we can define the function

$$\mathsf{and} = A \mapsto B \mapsto (\forall C : \mathsf{Prop}, (A \to B \to C) \to C) : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}.$$

This is both a type constructor, because it takes two types and returns a type, as well as a polymorphic function, because it is a function that takes a type as input.[8] To

[7] As to why it's called higher-order, I asked a math.SE question about this.

[8] Notably, and isn't just an abbreviation that we're writing. It's an actual function we're declaring within the system, because we have type constructors! Without type constructors, and would just have to be an abbreviation that we're making "above" the actual system we're working in.

convince you why this can be called and, consider this function:

$$A \mapsto B \mapsto X \mapsto XA(\mathsf{K}AB) : \forall A : \mathsf{Prop}, \forall B : \mathsf{Prop}, \mathsf{and}(A, B) \to A.$$

Here, K is the function we defined earlier. We claim this function has type $\forall A : \mathsf{Prop}, \forall B : \mathsf{Prop}, \mathsf{and}(A, B) \to A$. First, note that X begins with type $\mathsf{and}(A, B)$, and so the type of X is $\forall C : \mathsf{Prop}, (A \to B \to C) \to C$. By finding XA, we get the type $(A \to B \to A) \to A$. Finally, note that $\mathsf{K}AB$ is a function with type $A \to B \to A$, so we can indeed feed it as input to XA, giving a result of type A. This shows that it indeed has that type. By Curry–Howard, this means this is also a proof that $\forall A : \mathsf{Prop}, \forall B : \mathsf{Prop}, \mathsf{and}(A, B) \to A$.

Another example. We can combine dependent types and polymorphism to prove a statement about relations. For example, $a < b$ is a relation on the reals, and $a \mid b$ is a relation on the integers. Generally, a relation on a type A is a predicate $R(a, b)$. So R is a dependent type, of type $\forall A : \mathsf{Prop}, A \to A \to \mathsf{Prop}$. Then we can write the following proposition:

$$\begin{aligned}&\forall A : \mathsf{Prop}, \forall R : A \to A \to \mathsf{Prop},\\ &(\forall x : A, \forall y : A, Rxy \to Ryx \to \bot) \to (\forall x : A, Rxx \to \bot).\end{aligned}$$

This proposition claims that an antisymmetric relation is also antireflexive. The proof needs to be a polymorphic function of this type. We can start out by writing the inputs, $A \mapsto R \mapsto p \mapsto x \mapsto q \mapsto$. Here, $p : \forall x : A, \forall y : A, Rxy \to Ryx \to \bot$, and $q : Rxx$. Please check that $pxxqq$ gives something of type $\bot$, so a function with this type is $A \mapsto R \mapsto p \mapsto x \mapsto q \mapsto pxxqq$.

By combining type constructors, we can write predicates on relations. A relation itself is something of type $\forall A : \mathsf{Prop}, A \to A \to \mathsf{Prop}$. So a predicate on a relation is of type $\forall A : \mathsf{Prop}, (A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$. So we can write things like:

$$\begin{aligned}\mathsf{relpred} &= \forall A : \mathsf{Prop}, (A \to A \to \mathsf{Prop}) \to \mathsf{Prop}\\ \mathsf{reflexive} &= A \mapsto R \mapsto \forall x : A, Rxx : \mathsf{relpred}\\ \mathsf{symmetric} &= A \mapsto R \mapsto \forall x : A, \forall y : A, Rxy \to Ryx : \mathsf{relpred}\\ \mathsf{transitive} &= A \mapsto R \mapsto \forall x : A, \forall y : A, \forall z : A, Rxy \to Ryz \to Rxz : \mathsf{relpred}.\end{aligned}$$

Even with all this notation, writing out propositions tend to get pretty long. As an exercise, please write out a proof for

$$\begin{aligned}&\forall A : \mathsf{Prop}, \forall R : A \to A \to \mathsf{Prop},\\ &\mathsf{symmetric}(A, R) \to \mathsf{transitive}(A, R) \to (\forall x : A, \exists y : A, Rxy) \to \mathsf{reflexive}(A, R),\end{aligned}$$

the proposition that a symmetric, transitive, and total relation is also reflexive.

3.3 Pure type systems

The **calculus of constructions** is λC, the system at the opposite corner of the cube from $\lambda \to$. This is the system we get when we add dependent types, polymorphism, and type constructors, all together with simply typed lambda calculus.

Now, the way we built up to the calculus of constructions was pretty ad hoc. We started with simply typed lambda calculus, and then added a bunch of random things.

But there is a connection between all the things we introduced! They all introduced some sort of function going from terms or types to terms or types. By thinking in the level of pure type systems, we can make this connection clear.

Please think about the relationship between $\mapsto$ and $\to$. For example, let's go back to the function $x \mapsto \lfloor x \rfloor : \mathsf{real} \to \mathsf{int}$. The description "$x \mapsto \lfloor x \rfloor$" describes the function in the level of terms: the function takes the term x to the term $\lfloor x \rfloor$. On the other hand, "$\mathsf{real} \to \mathsf{int}$" goes one step higher, and describes the function in the level of types: the function takes (something in real) to (something in int).

Similarly, consider a type constructor like triple, which we'll write as $A \mapsto (A, A, A) : \mathsf{Prop} \to \mathsf{Prop}$. Again, "$A \mapsto (A, A, A)$" describes the function in the level of types: the function takes the type A to the type of triplets (A, A, A). Then "$\mathsf{Prop} \to \mathsf{Prop}$" goes one step higher, and describes the function in a level *above* types: the function takes (something in Prop) to (something in Prop).

By going from $\mapsto$ to $\to$, we go "one level higher". The big idea of **pure type systems** is by going *another level higher*, from $\to$ to $\rightsquigarrow$.[9] To specify a pure type system, we list down the things we can $\rightsquigarrow$ on.

For example, in simply typed lambda calculus, we have Prop. The only thing we can $\rightsquigarrow$ on is $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}$. As $\mathsf{real} : \mathsf{Prop}$ and $\mathsf{int} : \mathsf{Prop}$, we can make $\mathsf{real} \to \mathsf{int}$, from the rule $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}$. The rule also means that $\mathsf{real} \to \mathsf{int} : \mathsf{Prop}$. Because $\mathsf{real} \to \mathsf{int}$ exists, we can make things of type $\mathsf{real} \to \mathsf{int}$, by taking, for example, $x : \mathsf{real}$ and $\lfloor x \rfloor : \mathsf{int}$ and saying $x \mapsto \lfloor x \rfloor : \mathsf{real} \to \mathsf{int}$.

More formally:

- A pure type system has a list of **sorts** A, B, etc. We specify relationships between the sorts, like $\mathsf{A} : \mathsf{B}$. We also list down several rules, like $\mathsf{A} \rightsquigarrow \mathsf{B}$.
- If $\mathsf{A} \rightsquigarrow \mathsf{B}$ is a rule, $A : \mathsf{A}$, and $B : \mathsf{B}$, we can form the type $\forall a : A, B$. This itself has type B.
- If $\forall a : A, B$ is a type, $a : A$, and $b : B$, we can form the function $a \mapsto b$. This has type $\forall a : A, B$.

Note the second rule here! This implies that, say, $\mathsf{Type} \to \mathsf{Prop} : \mathsf{Prop}$, while $\mathsf{Prop} \to \mathsf{Type} : \mathsf{Type}$.

With the language of pure type systems, we can now specify the calculus of constructions succinctly, instead of as a hodgepodge of three separate systems on top of simply typed lambda calculus:

- There are two sorts, Prop and Type.
- The only relationship is $\mathsf{Prop} : \mathsf{Type}$.
- The rules are $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}$, $\mathsf{Prop} \rightsquigarrow \mathsf{Type}$, $\mathsf{Type} \rightsquigarrow \mathsf{Prop}$, and $\mathsf{Type} \rightsquigarrow \mathsf{Type}$.

Here's an example usage of these rules:

- As $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}$ is a rule, $A : \mathsf{Prop}$ and $A : \mathsf{Prop}$, we can form the type $\forall x : A, A$. We can abbreviate this as $A \to A$. This itself has the type Prop, so $A \to A : \mathsf{Prop}$.
- As $\mathsf{Type} \rightsquigarrow \mathsf{Prop}$ is a rule, $\mathsf{Prop} : \mathsf{Type}$ and $A \to A : \mathsf{Prop}$, we can form the type $\forall A : \mathsf{Prop}, A \to A$.

[9] I am inventing notation here. As we've already discussed, in most sources, instead of $\mapsto$ and $\to$, they use λ and $\forall$ (or Π). And in literally every other source, instead of $\rightsquigarrow$, they use sets of triplets of sorts. But I think that obscures this relationship.

- As $A \to A$ is a type, $x : A$, and $x : A$, we can form the function $x \mapsto x$. This has type $A \to A$.
- As $\forall A : \mathsf{Prop}, A \to A$ is a type, $A : \mathsf{Prop}$, and $x \mapsto x : A \to A$, we can form the function $A \mapsto x \mapsto x$. This has the type $\forall A : \mathsf{Prop}, A \to A$.

Here's another example, showing relpred can be formed from the rules:

- As $\mathsf{Prop} \rightsquigarrow \mathsf{Type}$ is a rule, $A : \mathsf{Prop}$ and $\mathsf{Prop} : \mathsf{Type}$, we can form the type $\forall x : A, \mathsf{Prop}$. We can abbreviate this as $A \to \mathsf{Prop}$. This itself has the type Type.
- As $\mathsf{Prop} \rightsquigarrow \mathsf{Type}$ is a rule, $A : \mathsf{Prop}$ and $A \to \mathsf{Prop} : \mathsf{Prop}$, we can form the type $\forall x : A, A \to \mathsf{Prop}$. We can abbreviate this as $A \to A \to \mathsf{Prop}$. This itself has the type Type.
- As $\mathsf{Type} \rightsquigarrow \mathsf{Type}$ is a rule, $A \to A \to \mathsf{Prop} : \mathsf{Type}$ and $\mathsf{Prop} : \mathsf{Type}$, we can form the type $\forall x : A \to A \to \mathsf{Prop}, \mathsf{Prop}$. We can abbreviate this as $(A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$. This itself has the type Type.
- As $\mathsf{Type} \rightsquigarrow \mathsf{Type}$ is a rule, $\mathsf{Prop} : \mathsf{Type}$ and $(A \to A \to \mathsf{Prop}) \to \mathsf{Prop} : \mathsf{Type}$, we can form the type $\forall A : \mathsf{Prop}, (A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$.

Note the similarity between the first two applications of the rules, and the second two applications of the rules.

As an exercise, try to use the rules to show that reflexive indeed has the type relpred. You want to look at each of the rule applications we did, and then do a similar one to form reflexive.

3.4 Inductive types, part 1

Now that we have the calculus of constructions, the way to get the calculus of *inductive* constructions is to add inductive types. Adding inductive types leads to several other complications, but for now, let's just add the inductive types.

An **inductive type** consists of several **constructors**. Each constructor is either a term of the inductive type, or a function that returns a term of the inductive type. The inductive type consists of all the terms that can be made from the constructors, and *only* of the terms that can be made from the constructors.

Let's make our first inductive type, which we'll call riddle. We'll say riddle is a Type that has two constructors, foo and bar. We'll write it out like this:

$$\mathsf{riddle} : \mathsf{Type} = \begin{cases} \mathsf{foo} & : \mathsf{riddle} \\ \mathsf{bar} & : \mathsf{riddle}. \end{cases}$$

The first constructor says that foo is a term of type riddle. The second constructor says that bar is a different term of type riddle. Because these are the only two constructors, the only terms in riddle are foo and bar.

When we defined an inductive type, we gave its constructors. After defining the type, we also get its partner **destructor**.[10] If we want to define a function with type

[10] In most sources, these are called recursors or eliminators. We call them destructors as the opposite of constructors. Coq and Lean both have destructors, but they instead focus on pattern-matching and fixpoints, which are more convenient to use.

$\mathsf{riddle} \to T$, we have to use the destructor. The idea is that we need to give what things will get taken to in each possible constructor. That means if we want to specify a function P, we need to say what $P(\mathsf{foo}) : T$ is, and we need to say what $P(\mathsf{bar}) : T$ is. After specifying both of these, we now have a function $\mathsf{riddle} \to T$. So its destructor, which we'll name ndestruct for reasons to be clear later, is a function with the type

$$\mathsf{ndestruct}_{\mathsf{riddle}} : \underbrace{\forall T : \mathsf{Type}}_{\text{return type}}, \underbrace{T}_{P(\mathsf{foo})} \to \underbrace{T}_{P(\mathsf{bar})} \to \underbrace{\mathsf{riddle} \to T}_{\text{the function}}.$$

Let's define a function $\mathsf{riddle} \to \mathsf{int}$, that will take foo to 0 and bar to 1. First, we specify the type that the function will return, which is int, as the first input. This is the T in the type of $\mathsf{ndestruct}_{\mathsf{riddle}}$. Second, we specify where foo will go to, which is 0. Third, we specify where bar will go to, which is 1. After putting these in, we're left with a function $\mathsf{riddle} \to T$, where T is int, just like we wanted! So

$$\mathsf{ndestruct}_{\mathsf{riddle}}(\mathsf{int})(0)(1) = x \mapsto \begin{cases} 0 & \text{if } x \text{ is } \mathsf{foo} \\ 1 & \text{if } x \text{ is } \mathsf{bar} \end{cases} : \mathsf{riddle} \to \mathsf{int}.$$

As another example, we can define a function that takes *two* things of type riddle, and output bar if at least one of them is bar, and foo otherwise. In this case, we need two layers of functions. For the first layer, we want to return a function with type $\mathsf{riddle} \to \mathsf{riddle}$, and in the second layer, we just return something with type riddle. So

$$\mathsf{ndestruct}_{\mathsf{riddle}}(\mathsf{riddle} \to \mathsf{riddle})(y \mapsto y)(y \mapsto \mathsf{bar}) : \mathsf{riddle} \to \mathsf{riddle} \to \mathsf{riddle}$$

should work. Please spend a few moments thinking about why it should work. This is because this evaluates to

$$x \mapsto \begin{cases} y \mapsto y & \text{if } x \text{ is } \mathsf{foo} \\ y \mapsto \mathsf{bar} & \text{if } x \text{ is } \mathsf{bar} \end{cases}$$

The reason we named it riddle is because this type might be familiar to you! Here are better names for this type and its constructors:

$$\mathsf{bool} : \mathsf{Type} = \begin{cases} \mathsf{false} & : \mathsf{bool} \\ \mathsf{true} & : \mathsf{bool}. \end{cases}$$

The function we just defined is the or function on two bools: it returns true if either one is true, and false otherwise. So it turns out that as soon as we have inductive types, we already have booleans! As an exercise, think about other boolean operators.

If we want the output of the function to be all of the same type, then the destructor we have, the **non-dependent destructor**, is enough. But we can generalize this by having different output types for different inputs. If we were silly, we could want a function that brings false to $0 : \mathsf{int}$, and true to $\pi : \mathsf{real}$.

This is why, in the **dependent destructor**, instead of specifying the output type T, we give a function $T : \mathsf{bool} \to \mathsf{Type}$ that gives us the output type based on the input. So the dependent destructor has a dependent function type! It has the type

$$\mathsf{ddestruct}_{\mathsf{bool}} : \underbrace{\forall T : \mathsf{bool} \to \mathsf{Type}}_{\text{dependent return type}}, \underbrace{T(\mathsf{false})}_{P(\mathsf{false})} \to \underbrace{T(\mathsf{true})}_{P(\mathsf{true})} \to \underbrace{\forall x : \mathsf{bool}, Tx}_{\text{dependent function}}.$$

That means if we defined the function

$$T = x \mapsto \begin{cases} \mathsf{int} & \text{if } x \text{ is } \mathsf{false} \\ \mathsf{real} & \text{if } x \text{ is } \mathsf{true} \end{cases} : \mathsf{bool} \to \mathsf{Type},$$

perhaps using ndestruct, then the function we want is

$$\mathsf{ddestruct}_{\mathsf{bool}}(T)(0)(\pi) = x \mapsto \begin{cases} 0 & \text{if } x \text{ is } \mathsf{false} \\ \pi & \text{if } x \text{ is } \mathsf{true} \end{cases} : \forall x : \mathsf{bool}, Tx.$$

Please convince yourself that ndestruct is just a special case of ddestruct. The argument is that $\mathsf{ndestruct}(T) = \mathsf{ddestruct}(x \mapsto T)$, for any $T : \mathsf{Type}$. It may not be clear why we want dependent destructors now, but there's a good reason that'll become clearer later.

3.5 Inductive types, part 2

Let's create our second inductive type, which we'll call riddle. We'll say riddle is a Type that has two constructors, foo and bar. We'll write it out like this:

$$\mathsf{riddle} : \mathsf{Type} = \begin{cases} \mathsf{foo} & : \mathsf{riddle} \\ \mathsf{bar} & : \mathsf{riddle} \to \mathsf{riddle}. \end{cases}$$

The first constructor says that foo is a term of type riddle. The second constructor says that, if $x : \mathsf{riddle}$, then $\mathsf{bar}(x) : \mathsf{riddle}$. Because these are the only two constructors, the only terms in riddle are those that can be derived from these two rules. This means all the terms in riddle look like

$$\mathsf{foo}, \mathsf{bar}(\mathsf{foo}), \mathsf{bar}(\mathsf{bar}(\mathsf{foo})), \mathsf{bar}(\mathsf{bar}(\mathsf{bar}(\mathsf{foo}))), \ldots.$$

What is the type of $\mathsf{ndestruct}_{\mathsf{riddle}}$? Well, if we want to define a function P, we start with the output type $T : \mathsf{Type}$. Then we have to specify what $P(\mathsf{foo})$ is. What about $P(\mathsf{bar}(x))$? Please think for a moment.

A reasonable way to define $P(\mathsf{bar}(x))$ would be based on both x and Px. To compute Px, then, we need to recurse on x, by following all the steps again. This means we want a function that, given x and Px, gives us $P(\mathsf{bar}(x))$. The type of this function will be $T \to T$. Thus

$$\mathsf{ndestruct}_{\mathsf{riddle}} : \underbrace{\forall T : \mathsf{Type}}_{\text{return type}}, \underbrace{T}_{P(\mathsf{foo})} \to \underbrace{(\mathsf{riddle} \to T \to T)}_{x \mapsto Px \mapsto P(\mathsf{bar}(x))} \to \underbrace{\mathsf{riddle} \to T}_{\text{the function}}.$$

Let's write a function that doubles the number of bars in a term of type riddle. Say this function is double. First, T is riddle. Second, double(foo) would remain foo. Third is the tricky case: if we have the answer for $\mathsf{double}(x)$, how can we find $\mathsf{double}(\mathsf{bar}(x))$? Please think about this for a moment.

Let's say we have $x = \mathsf{bar}(\mathsf{bar}(\mathsf{foo}))$. Then $\mathsf{double}(x) = \mathsf{bar}(\mathsf{bar}(\mathsf{bar}(\mathsf{bar}(\mathsf{foo}))))$. And let's say we now want the doubled version of $\mathsf{bar}(x)$. Well, $\mathsf{bar}(x)$ has three xs, and we want six. We saw $\mathsf{double}(x)$ has four xs. So if we take $\mathsf{double}(x)$ and add two bars in front, we get $\mathsf{double}(\mathsf{bar}(x))$! That means the function we want, should be $x \mapsto p \mapsto \mathsf{bar}(\mathsf{bar}(p))$. So please check that

$$\mathsf{double} = \mathsf{ndestruct}_{\mathsf{riddle}}(\mathsf{riddle})(\mathsf{foo})(x \mapsto p \mapsto \mathsf{bar}(\mathsf{bar}(p))) : \mathsf{riddle} \to \mathsf{riddle}$$

is indeed the function we want. Please check that everything has the right type. Then, if we wanted to compute $\mathsf{double}(\mathsf{bar}(\mathsf{foo}))$, check that

$$\mathsf{double}(\mathsf{bar}(\mathsf{foo})) = (x \mapsto p \mapsto \mathsf{bar}(\mathsf{bar}(p)))(\mathsf{foo})(\mathsf{double}(\mathsf{foo}))$$

is the first step of the computation. The fact that the constructor is $\mathsf{bar}(x)$ selects the second part of the destructor. The inputs to that function are x and $\mathsf{double}(x)$, which in this case, are foo and $\mathsf{double}(\mathsf{foo})$.

We now ask the reader to think about the type of ddestruct:

$$\begin{aligned}\mathsf{ddestruct}_{\mathsf{riddle}} : {} & \forall T : \mathsf{riddle} \to \mathsf{Type}, \\ & T(\mathsf{foo}) \to (\forall x : \mathsf{riddle}, Tx \to T(\mathsf{bar}(x))) \to \forall x : \mathsf{riddle}, Tx.\end{aligned}$$

Mentally label each of the parts of this type with what they're supposed to be. What is T here? What is $\forall x : \mathsf{riddle}, Tx \to T(\mathsf{bar}(x))$? Why does this work?

The reason we named it riddle is because this type might be familiar to you! Here are better names for this type and its constructors:

$$\mathsf{nat} : \mathsf{Type} = \begin{cases} O & : \mathsf{nat} \\ S & : \mathsf{nat} \to \mathsf{nat}. \end{cases}$$

These are the natural numbers, where O is zero, and, say, $S(S(SO))$ is three. The letter S is chosen to stand for "successor", so adding S in front of a nat gives the next natural number. Then double indeed doubles a nat.

Now that we have nat, let's define add on it. It's easier to start with the special case $\mathsf{add}(x)$, where we're adding x to some nat. Please check that this works:

$$\mathsf{add}(x) = \mathsf{ndestruct}_{\mathsf{nat}}(\mathsf{nat})(\underbrace{x}_{\mathsf{add}(x)(O)})(\underbrace{y \mapsto p \mapsto Sp}_{y \mapsto \mathsf{add}(x)(y) \mapsto \mathsf{add}(x)(Sy)}) : \mathsf{nat} \to \mathsf{nat}$$

This means that

$$\mathsf{add} = x \mapsto \mathsf{ndestruct}_{\mathsf{nat}}(\mathsf{nat})(x)(y \mapsto p \mapsto Sp) : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.$$

3.6 Curry–Howard, part 4

SIMPLICIO: Inductive types sure are cool.

SALVIATI: Truly!

SIMPLICIO: But we haven't talked about Curry–Howard in a while. I really want to know: what's the logical interpretation of all of this? We talked about how $\pi : \mathsf{real}$ corresponds to "π is a proof that there exists a real number." And sure, I guess it makes sense that $O : \mathsf{nat}$ is "O is a proof that there is a natural number." But what about $S : \mathsf{nat} \to \mathsf{nat}$?

SALVIATI: What *about* $S : \mathsf{nat} \to \mathsf{nat}$? It is a proof that "there is a natural number" implies "there is a natural number."

SIMPLICIO: Not a very exciting fact.

SALVIATI: No, not really. It turns out that inductive types are better when it comes to *defining* things we can use Curry–Howard with. Consider, for example, our

previous predicate $\mathsf{even}(n)$. Let's say we wanted to define $\mathsf{even} : \mathsf{nat} \to \mathsf{Prop}$ as an inductive type. How would we do that?

SIMPLICIO: Well, to define an inductive type, we need to define constructors. Constructors are things of a given type we say exist. So I guess one reasonable constructor would be $\mathsf{evenO} : \mathsf{even}(O)$.

SALVIATI: Yes, that makes sense. This means that the term evenO is a proof that $\mathsf{even}(O)$. Now how could we get things like $\mathsf{even}(S(S(S(SO))))$?

SIMPLICIO: Hm. I guess we can do something similar to what we did for nat. Kind of like, if we knew $\mathsf{even}(x)$, we want to say $\mathsf{even}(S(Sx))$. Would the other constructor be $\mathsf{evenSS} : \mathsf{even}(x) \to \mathsf{even}(S(Sx))$?

SALVIATI: Close! You need to define what x is.

SIMPLICIO: Right, $\mathsf{evenSS} : \forall x : \mathsf{nat}, \mathsf{even}(x) \to \mathsf{even}(S(Sx))$.

SALVIATI: Yes, exactly. That gives us this definition:

$$\mathsf{even} : \mathsf{nat} \to \mathsf{Prop} = \begin{cases} \mathsf{evenO} & : \mathsf{even}(O) \\ \mathsf{evenSS} & : \forall x : \mathsf{nat}, \mathsf{even}(x) \to \mathsf{even}(S(Sx)). \end{cases}$$

SIMPLICIO: That actually makes a lot of sense! The first constructor is kind of like an axiom, saying "zero is even." The second constructor is another axiom, saying "for all natural numbers x, if x is even, then $S(Sx)$ is also even."

SALVIATI: Using this inductive type, how would you prove that $S(S(S(SO)))$ is even?

SIMPLICIO: Well, we need something of the type $\mathsf{even}(S(S(S(SO))))$. We know that $\mathsf{evenO} : \mathsf{even}(O)$. By applying evenSS, we get that $\mathsf{evenSS}(O)(\mathsf{evenO}) : \mathsf{even}(S(SO))$. And then we apply it again, giving us

$$\mathsf{evenSS}(S(SO))(\mathsf{evenSS}(O)(\mathsf{evenO})) : \mathsf{even}(S(S(S(SO)))).$$

SALVIATI: Right. By Curry–Howard, this corresponds to the following proof:

- We know that O is even by evenO.
- From evenSS applied to O, because O is even, we know $S(SO)$ is also even.
- From evenSS applied to $S(SO)$, because $S(SO)$ is even, we know $S(S(S(SO)))$ is also even.

SIMPLICIO: That's neat. But I wanted to prove less obvious things.

SALVIATI: Alright then. Let's go about proving $\forall x : \mathsf{nat}, \mathsf{add}(O)(x) = x$.

SIMPLICIO: Wait, but isn't that obvious?

SALVIATI: Not at all. By substituting O into add, we get

$$\mathsf{add}(O) = \mathsf{ndestruct}_{\mathsf{nat}}(\mathsf{nat})(O)(y \mapsto p \mapsto Sp) : \mathsf{nat} \to \mathsf{nat}.$$

What does this mean, again?

SIMPLICIO: Hm. This means $\mathsf{add}(O)$ is a function that takes some nat. If it's O, the result is O, which is what we want, so that's fine. Otherwise, it's Sx for some x.

Then the result will have to depend on what $\mathsf{add}(O)(x)$ is. And then the result of *that* will depend on whether x is O or not...

SALVIATI: Looks like you're getting stuck in a loop.

SIMPLICIO: Do we need induction here? I suppose we need induction. The "normal" proof would be, because $\mathsf{add}(O)(x) = x$ by inductive hypothesis, then $\mathsf{add}(O)(Sx) = S\mathsf{add}(O)(x) = Sx$, as desired. How do we do induction, then?

3.7 Curry–Howard, part 5

Recall $\mathsf{ddestruct}_{\mathsf{nat}}$:

$$\begin{aligned}\mathsf{ddestruct}_{\mathsf{nat}} : &\forall T : \mathsf{nat} \to \mathsf{Type},\\ &TO \to (\forall x : \mathsf{nat}, Tx \to T(Sx)) \to \forall x : \mathsf{nat}, Tx.\end{aligned}$$

This should be read like this: "If T is a function from naturals to types, and we give something of type TO, and for all x, we give a function that changes something of type Tx to something of type $T(Sx)$, then we have a function from each natural x to Tx."

This is restricted in the sense that $T : \mathsf{nat} \to \mathsf{Type}$. What if we wanted to have something $\mathsf{nat} \to \mathsf{Prop}$? Thankfully, as $\mathsf{Prop} : \mathsf{Type}$, a special case of $\mathsf{ddestruct}_{\mathsf{nat}}$ is what we'll call $\mathsf{pdestruct}_{\mathsf{nat}}$, where we have $P : \mathsf{nat} \to \mathsf{Prop}$ instead. Here, the p in the name stands for Prop:

$$\begin{aligned}\mathsf{pdestruct}_{\mathsf{nat}} : &\forall P : \mathsf{nat} \to \mathsf{Prop},\\ &PO \to (\forall x : \mathsf{nat}, Px \to P(Sx)) \to \forall x : \mathsf{nat}, Px.\end{aligned}$$

This should be read like this: "If P is a proposition about naturals, and we prove PO, and for all x, we prove Px implies $P(Sx)$, then P is true for all naturals."

Yes, propositions *are* types. These types all lie in Prop. But we want to maintain the difference between Prop, which in a sense are "small types", and Type, which are "big types". This is for technical reasons we won't talk about until later.

Going back to $\mathsf{pdestruct}_{\mathsf{nat}}$, we see it's the induction principle for the natural numbers. The fact that we can do induction on inductive types is why they're called inductive types in the first place. Let's get back to Salviati and Simplicio, as they use this to prove $\mathsf{add}(O)(x) = x$.

SALVIATI: Step back from thinking about how to do induction. Propositions are types. How do we prove a proposition?

SIMPLICIO: We write down a term with the type we want to prove.

SALVIATI: Right. We want to prove $\forall x : \mathsf{nat}, \mathsf{add}(O)(x) = x$. Think back to BHK. What kind of term is this? What does it do?

SIMPLICIO: This has to be a function. The function takes in any $x : \mathsf{nat}$, and returns a term with the type $\mathsf{add}(O)(x) = x$. Wait a second. I don't think we have *any* terms with $=$ in their type.

SALVIATI: No, no we don't. We'll have to make some. Let's use eq, to remind us that this is a predicate we're defining. We'll need to come up with something that gives us predicates of the type $\mathsf{eq}(x)(y)$. How do we define it? When's the last time we had to define a predicate?

SIMPLICIO: We defined even using an inductive type. So I guess we can define eq using an inductive type. I guess the really important constructor is $\mathsf{eq}(x)(x)$ for all x. We'll see if we can get by with that. So, this definition?

$$\mathsf{eq} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{Prop} = \left\{ \mathsf{reflexivity} \quad : \forall x : \mathsf{nat}, \mathsf{eq}(x)(x). \right.$$

SALVIATI: Sure. Alright, back to the problem. We need a function that takes any $x : \mathsf{nat}$, and returns a term with the type $\mathsf{eq}(\mathsf{add}(O)(x))(x)$. In particular, we need a function that goes from nat to something else.

SIMPLICIO: Ah, so we need a destructor! The proof should be something of the form

$$\mathsf{pdestruct}_{\mathsf{nat}}(x \mapsto \mathsf{eq}(\mathsf{add}(O)(x))(x))(\text{something})(\text{something}).$$

So *this* is why we wanted to have the more generally typed version of the destructor. The output type we want depends on what x is.

SALVIATI: What are the somethings? What are their types?

SIMPLICIO: Oh! Let me annotate it:

$$\mathsf{pdestruct}_{\mathsf{nat}}(\cdots)(\underbrace{\text{something}}_{\mathsf{eq}(\mathsf{add}(O)(O))(O)})(\underbrace{\text{something}}_{\forall x : \mathsf{nat}, \mathsf{eq}(\mathsf{add}(O)(x))(x) \to \mathsf{eq}(\mathsf{add}(O)(Sx))(Sx)}).$$

SALVIATI: Alright. Now let's construct terms of those types.

SIMPLICIO: Well, for the first one. The term $\mathsf{add}(O)(O)$ just simplifies to O, so we only need something of the type $\mathsf{eq}(O)(O)$, right? And a term of that type would be $\mathsf{reflexivity}(O) : \mathsf{eq}(O)(O)$!

SALVIATI: Yes! We can simplify terms and it'll stay the same. That fills in the blank for the first something. What about the second something? We need a function with the type $\forall x : \mathsf{nat}, \mathsf{eq}(\mathsf{add}(O)(x))(x) \to \mathsf{eq}(\mathsf{add}(O)(Sx))(Sx)$.

SIMPLICIO: We start with $x \mapsto p \mapsto$, so $p : \mathsf{eq}(\mathsf{add}(O)(x))(x)$. We need a term with type $\mathsf{eq}(\mathsf{add}(O)(Sx))(Sx)$. Now how do we deal with $\mathsf{add}(O)(Sx)$?

SALVIATI: Well, recall the definition for $\mathsf{add}(O)$. We're in the second case, where we have Sx. The first input's x, the second input, which is p, is the result for $\mathsf{add}(O)(x)$. The result is Sp, which is...

SIMPLICIO: Oh, $S\mathsf{add}(O)(x)$. Alright. So that means $\mathsf{add}(O)(Sx)$ simplifies to $S\mathsf{add}(O)(x)$. We need to produce a term of the type $\mathsf{eq}(S\mathsf{add}(O)(x))(Sx)$. We have p, which is a term of the type $\mathsf{eq}(\mathsf{add}(O)(x))(x)$. Hm. I'm not sure how to do this. I think I want a proposition about eq. Something like, $\mathsf{eq}(x)(y)$ means $\mathsf{eq}(fx)(fy)$, for a function f.

SALVIATI: Sure. For now, let's add that as a constructor for eq, and then we can get back to proving it later. Here:

$$\mathsf{eq} : \cdots = \begin{cases} \mathsf{reflexivity} & : \forall x : \mathsf{nat}, \mathsf{eq}(x)(x) \\ \mathsf{fequal} & : \forall x : \mathsf{nat}, \forall y : \mathsf{nat}, \\ & \quad \mathsf{eq}(x)(y) \to (\forall f : \mathsf{nat} \to \mathsf{nat}, \mathsf{eq}(fx)(fy)). \end{cases}$$

SIMPLICIO: Alright. So we have $p : \mathsf{eq}(\mathsf{add}(O)(x))(x)$. That means that

$$\mathsf{fequal}(\mathsf{add}(O)(x))(x)(p)(S) : \mathsf{eq}(S\mathsf{add}(O)(x))(Sx).$$

SALVIATI: So what's our final proof?

SIMPLICIO: Okay, this is pretty long. Unsurprising, I guess, given that it has to do all the work of a paper proof. But I'm pretty sure this is it:

$$\begin{aligned}&\mathsf{pdestruct}_{\mathsf{nat}}(x \mapsto \mathsf{eq}(\mathsf{add}(O)(x))(x))\\&\qquad(\mathsf{reflexivity}(O))\\&\qquad(x \mapsto p \mapsto \mathsf{fequal}(\mathsf{add}(O)(x))(x)(p)(S))\\&\quad : \forall x : \mathsf{nat}, \mathsf{eq}(\mathsf{add}(O)(x))(x).\end{aligned}$$

SALVIATI: Yeah. To summarize, we learned that each of the constructors of an inductive type correspond to an axiom about that type. We can do induction on inductive types—that's why it's called an *inductive* type. We also learned that the destructor isn't just used for defining functions from an inductive type, it's also used for induction.

These inductive proofs are really tricky and somewhat tedious to write on paper, especially compared to the previous CurryHoward things we've been doing, but I think it's instructive to work through one of these at least once in your life.

Proof assistants are called proof assistants because they help in making a lot of this work simpler. They keep track of the "somethings" that need to be filled in, and check that they have the types that they should have. Type theory provides a very natural language for this, not only giving us a good basis for formalizing what a proof means, but making it amenable to using a computer to work out.

4 Technical issues

4.1 Annotated types

We now dive into technical details for inductive types. We mentioned earlier that we'll talk about the proof of fequal, so we'll go through it here. In the process, we'll make the difference between ddestruct, the destructor that goes $\to$ Type, and pdestruct, the destructor that goes $\to$ Prop, even clearer.

Recall our definition for even:

$$\mathsf{even} : \mathsf{nat} \to \mathsf{Prop} = \begin{cases} \mathsf{evenO} & : \mathsf{even}(O) \\ \mathsf{evenSS} & : \forall x : \mathsf{nat}, \mathsf{even}(x) \to \mathsf{even}(S(Sx)). \end{cases}$$

This is a different kind of inductive type, one called an **annotated (inductive) type**. This is because, instead of defining something in A, we define something $\cdots \to \mathsf{A}$.

As another example of an annotated type, here's inttup, from earlier. We want inttup(3) to be the type of tuples with 3 integers. We can now define these using annotated types:

$$\mathsf{inttup} : \mathsf{nat} \to \mathsf{Type} = \begin{cases} \mathsf{tupO} & : \mathsf{inttup}(O) \\ \mathsf{tupS} & : \forall x : \mathsf{nat}, \mathsf{inttup}(x) \to \mathsf{int} \to \mathsf{inttup}(Sx). \end{cases}$$

The tuple $(-1, -2, -3)$ would now be encoded as

$$\mathsf{tupS}(2)(\mathsf{tupS}(1)(\mathsf{tupS}(0)(\mathsf{tupO})(-1))(-2))(-3).$$

Here is its ndestruct, with annotations indicating the function P:

$$\begin{aligned} \mathsf{ndestruct}_{\mathsf{inttup}} : & \underbrace{\forall T : \mathsf{Type}}_{\text{return type}}, \underbrace{T}_{P(\mathsf{tupO})} \to \\ & \underbrace{(\forall x : \mathsf{nat}, \mathsf{inttup}(x) \to T \to \mathsf{int} \to T)}_{x \mapsto t \mapsto Pt \mapsto a \mapsto P(\mathsf{tupS}(x)(t)(a))} \to \underbrace{\forall x : \mathsf{nat}, \mathsf{inttup}(x) \to T}_{\text{the function}}. \end{aligned}$$

So, if we had some function intplus : int $\to$ int $\to$ int, we can write a function total that takes the total of an inttup:

$$\begin{aligned} \mathsf{total} = \mathsf{ndestruct}_{\mathsf{inttup}}(& \underbrace{\mathsf{int}}_{\text{return type}})(\underbrace{0}_{\mathsf{total}(\mathsf{tupO})})(\underbrace{x \mapsto t \mapsto T \mapsto a \mapsto \mathsf{intplus}(T)(a)}_{x \mapsto t \mapsto \mathsf{total}(t) \mapsto a \mapsto \mathsf{total}(\mathsf{tupS}(x)(t)(a))}) \\ & : \forall x : \mathsf{nat}, \mathsf{inttup}(x) \to \mathsf{int}. \end{aligned}$$

Similarly, we can write its ddestruct. The tricky thing is that the dependent output type, T, wouldn't just be inttup$(x) \to$ Type. It needs to know what x is! So it's

$$\begin{aligned} \mathsf{ddestruct}_{\mathsf{inttup}} : & \forall T : (\forall x : \mathsf{nat}, \mathsf{inttup}(x) \to \mathsf{Type}), \\ & T O(\mathsf{tupO}) \to \\ & (\forall x : \mathsf{nat}, \forall t : \mathsf{inttup}(x), Txt \to \forall a : \mathsf{int}, T(Sx)(\mathsf{tupS}(x)(t)(a))) \to \\ & \forall x : \mathsf{nat}, \forall t : \mathsf{inttup}(x), Txt. \end{aligned}$$

The pdestruct would be the same, except with Type changed to Prop.

In the particular case of $\cdots \to \mathsf{Prop}$, we call it an **inductive predicate**, because it's a predicate that's also an inductive type. So even is an inductive predicate. The thing about inductive predicates is that they don't have $\mathsf{ndestruct}$ or $\mathsf{ddestruct}$. As predicates lie in $\cdots \to \mathsf{Prop}$, and $\mathsf{Prop} : \mathsf{Type}$, for technical reasons, we can't have a function that goes "up" to Type. For even *more* technical reasons, the $\mathsf{pdestruct}$ looks slightly different for inductive predicates. Here's the $\mathsf{pdestruct}$ for even:

$$\begin{aligned}&\mathsf{pdestruct}_{\mathsf{even}} : \forall P : \mathsf{nat} \to \mathsf{Prop},\\ &\qquad PO \to (\forall x : \mathsf{nat}, \mathsf{even}(x) \to Px \to P(S(Sx))) \to \forall x : \mathsf{nat}, \mathsf{even}(x) \to Px.\end{aligned}$$

This should be read as "If P is a proposition about naturals, and we prove PO, and for all x, we prove that Px and x being even implies $P(S(Sx))$, then P is true for all even naturals."

In particular, note that P isn't about $\mathsf{even}(x)$, because that *itself* is a proposition. Instead, P is about x, the input to the predicate even. This is in contrast to the destructors for our other inductive types, like $\mathsf{ddestruct}_{\mathsf{inttup}}$, where T gets *all* the information that goes into constructing the inductive type.

We now recall the definition for eq:

$$\mathsf{eq} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{Prop} = \begin{cases}\mathsf{reflexivity} & : \forall x : \mathsf{nat}, \mathsf{eq}(x)(x),\end{cases}$$

and write down its $\mathsf{pdestruct}$:

$$\begin{aligned}&\mathsf{pdestruct}_{\mathsf{eq}} : \forall P : \mathsf{nat} \to \mathsf{nat} \to \mathsf{Prop},\\ &\qquad (\forall x : \mathsf{nat}, Pxx) \to \forall x : \mathsf{nat}, \forall y : \mathsf{nat}, \mathsf{eq}(x)(y) \to Pxy.\end{aligned}$$

This now allows us to write a very short proof of fequal. It is simply

$$\mathsf{pdestruct}_{\mathsf{eq}}(\forall x : \mathsf{nat}, \forall y : \mathsf{nat}, \forall f : \mathsf{nat} \to \mathsf{nat}, \mathsf{eq}(fx)(fy))(\mathsf{reflexivity}(fx)).$$

We ask the reader to verify that it works. As further exercise, you can try to prove theorems like $\forall x : \mathsf{nat}, \forall y : \mathsf{nat}, \mathsf{eq}(\mathsf{add}(x)(y))(\mathsf{add}(y)(x))$, or $\forall x : \mathsf{nat}, \mathsf{even}(\mathsf{add}(x)(x))$, or even $\forall x : \mathsf{nat}, \mathsf{even}(x) \to \exists y : \mathsf{nat}, \mathsf{eq}(x)(\mathsf{add}(y)(y))$.

4.2 Parametrized types

Now that we've written inttup, we can try to write tup, a generalized version of inttup. We want, say, $\mathsf{tup}(\mathsf{int})(3)$ to be the type of tuples of 3 integers. Here's an attempt to define it with an annotated type:

$$\begin{aligned}&\mathsf{tup} : \mathsf{Type} \to \mathsf{nat} \to \mathsf{Type} =\\ &\quad \begin{cases}\mathsf{tupO} & : \forall T : \mathsf{Type}, \mathsf{tup}(T)(O)\\ \mathsf{tupS} & : \forall T : \mathsf{Type}, \forall x : \mathsf{nat}, \mathsf{tup}(T)(x) \to T \to \mathsf{tup}(T)(Sx).\end{cases}\end{aligned}$$

But there's a problem with this definition. Let's say you wanted to write a total function on $\mathsf{tup}(\mathsf{int})$. You wouldn't actually be able to do this, because $\mathsf{ndestruct}_{\mathsf{int}}$ requires whatever function you're defining to work on *all* possible tups, including things like $\mathsf{tup}(\mathsf{bool})$, for example.

Instead, we create a **parametrized (inductive) type**:

$$\begin{aligned}&\mathsf{tup} : \mathsf{Type} \to \mathsf{nat} \to \mathsf{Type} = \\ &\quad \forall T : \mathsf{Type}, \begin{cases} \mathsf{tupO} & : \mathsf{tup}(T)(O) \\ \mathsf{tupS} & : \forall x : \mathsf{nat}, \mathsf{tup}(T)(x) \to T \to \mathsf{tup}(T)(Sx). \end{cases}\end{aligned}$$

Now, $\mathsf{tup}(\mathsf{int})$ is its own inductive type, $\mathsf{tup}(\mathsf{bool})$ is its own inductive type, and so on.[11] Note that tupO and tupS still take T as an input. But it differs from the previous one in that the destructor is different, where there is only a single $\forall T$ on the outside, rather than multiple $\forall T$s inside each part of the destructor. Again, annotations indicate the function P:

$$\begin{aligned}\mathsf{ndestruct}_{\mathsf{tup}} : &\underbrace{\forall T : \mathsf{Type}}_{\mathsf{tup\ type}}, \underbrace{\forall R : \mathsf{Type}}_{\text{return type}}, \underbrace{R}_{P(\mathsf{tupO}(T))} \to \\ &\underbrace{(\forall x : \mathsf{nat}, \mathsf{tup}(T)(x) \to R \to T \to R)}_{x \mapsto t \mapsto Pt \mapsto a \mapsto P(\mathsf{tupS}(T)(x)(t)(a))} \to \underbrace{\forall x : \mathsf{nat}, \mathsf{tup}(T)(x) \to R}_{\text{the function}}.\end{aligned}$$

With annotated types, we now have all the ingredients to build up pretty much everything from scratch. For example, here is the sum type:

$$\mathsf{sum} : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type} = \forall A : \mathsf{Type}, \forall B : \mathsf{Type}, \begin{cases} \mathsf{suml} & : A \to \mathsf{sum}(A)(B) \\ \mathsf{sumr} & : B \to \mathsf{sum}(A)(B). \end{cases}$$

The first constructor, suml, says that if you have something of type A, then you can create something of type $\mathsf{sum}(A)(B)$. Similarly, the second constructor, sumr, says that if you have something of type B, then you can create something of type $\mathsf{sum}(A)(B)$. That means anything of type $\mathsf{sum}(A)(B)$ is either an object of type A or of type B, which is exactly what we want the sum type to be.

As another example, consider $\exists x : S, Ax$. Here, note that $S : \mathsf{Type}$ and $A : S \to \mathsf{Prop}$. We can define $\mathsf{exists}(S)(A)$ with a parametrized type:

$$\begin{aligned}&\mathsf{exists} : \forall S : \mathsf{Type}, (S \to \mathsf{Prop}) \to \mathsf{Prop} = \\ &\quad \forall S : \mathsf{Type}, \forall A : S \to \mathsf{Prop}, \begin{cases} \mathsf{example} & : \forall x : S, Ax \to \mathsf{exists}(S)(A). \end{cases}\end{aligned}$$

The only constructor, $\mathsf{example}$, says that "for all x, if Ax, then $\mathsf{exists}(S)(A)$." The destructor for exists could then give us that x. In particular, the function

$$\mathsf{ddestruct}_{\mathsf{exists}}(S)(A)(x \mapsto A \mapsto x) : \mathsf{exists}(S)(A) \to S$$

takes a claim $\mathsf{exists}(S)(A)$ and returns $x : S$ such that $Ax : \mathsf{Prop}$.

Annotated types are also the proper way to define eq, to extend over a general type T rather than just nat. In this manner we can build up all the mathematics we'd reasonably want. We encourage the reader to think about defining $\leq$, and proving things like $\forall x : \mathsf{nat}, \forall y : \mathsf{nat}, x \leq \mathsf{add}(x)(y)$.

[11] There are technicalities here. We're not quite allowed to use Type to construct another Type, for reasons we'll explain later. But we'll ignore that for now.

4.3 Constructors

Despite all this discussion, we've still hand-waved a lot of details about how exactly inductive types can be made.

There are genuine conditions on the constructors for inductive types. In order for a constructor of a type T to make sense, they have to be things that return T somehow. So $A \to T$ can be a constructor for T, but $T \to A$ cannot be a constructor for T.

Even more important is that not *anything* can be a constructor for T, just because it returns T. There's the requirement that the occurrences of T be *strictly positive*. That means that T cannot appear in an input to an input of a constructor, but it can appear anywhere else. In a constructor of type

$$(\forall ab : A \to B, C \to D) \to (\forall e : E, F) \to G \to H,$$

the strictly positive positions are D, F, G, and H.

The reason we have this requirement is because without it, we wouldn't have well-formed inductive types. That is, we can define inductive types that lead to contradictions. The simplest example is a constructor like $c : (T \to \bot) \to T$. If we allowed this constructor, then we can write a function $f : T \to (T \to \bot)$ using the destructor. Now $t \mapsto ft : T \to \bot$, and $c(t \mapsto ft) : T$ and thus $(t \mapsto ft)(c(t \mapsto ft)) : \bot$, and we have a contradiction.

If we follow these rules for making constructors, we *do* get well-formed inductive types. Here, "well-formed" means that we can do induction on them: we can take something formed by constructors, and get something "smaller and smaller", until eventually we reach a "base case". The reasoning for this is technical, so we'll skip it.

Given well-formed constructors, another technicality is how we form the corresponding parts of the destructor. We focus on ndestruct, because ddestruct is similar. Let's take this example constructor:

$$c = (\forall ab : A \to B, C \to D) \to (\forall e : E, T) \to T \to T,$$

and write its corresponding part of the destructor. Let R be our return type. First, we split up the outermost $\to$s, and work on each input separately, from left to right:

- The first input has no Ts at all, so it carries over entirely.
- The second input has a T, so it gets turned to $(\forall e : E, T) \to \forall e : E, R$. We carry the original, then carry it, changing T to R.
- The third input has a T, so it gets turned to $T \to R$.
- The last one is the output of the constructor, which is always turned to R.

The corresponding part of the destructor would have the type

$$\underbrace{(\forall ab : A \to B, C \to D)}_{x} \to \underbrace{(\forall e : E, T)}_{y} \to \underbrace{(\forall e : E, R)}_{e \mapsto P(ye)} \to \underbrace{T}_{z} \to \underbrace{R}_{Pz} \to \underbrace{R}_{P(cxyz)},$$

where we've annotated each input assuming the function was P.

There are again technical issues here. We assume that T is $\cdots \to$ Type, because if T was $\cdots \to$ Prop, the destructor would look slightly different. It is in inductive types that the difference between Prop and Type becomes more felt, as consequences of the relationship Prop : Type.

It is also possible for there to be mutually inductive types, which adds another layer to the technicalities. These allow several inductive types to have each other

as constructors. Checking that these are well-formed requires formulating a slightly different kind of strict positivity.

There are also things known as co-inductive types. Inductive types have to reach a "base case", but co-inductive types remove this restriction. As I understand it, the theory of inductive types is still relatively young compared to the rest of type theory.

4.4 Universes

We've already run into some difficulties thinking about Prop and Type and the difference between them, that we've mostly swept away. It is now time to think about these head-on. Recall our definition of pure type systems. Part of the rules say that, if $\mathsf{A} \rightsquigarrow \mathsf{B}$ is a rule, and $A : \mathsf{A}$ and $B : \mathsf{B}$, then $\forall a : A, B$ is a type. Consider:

$$\mathsf{id} = A \mapsto x \mapsto x : \forall A : \mathsf{Prop}, A \to A.$$

This is a perfectly normal function. We outlined in the section on pure type systems how to create this function using the rules. The two steps I want to highlight are how the type is made. I'll reproduce them here, with the thinking behind:

- We want to form $A \to A$. Well, $A : \mathsf{Prop}$ and $A : \mathsf{Prop}$, so we need the rule $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}$. We do! So:
  - As $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}$ is a rule, $A : \mathsf{Prop}$ and $A : \mathsf{Prop}$, we can form $A \to A : \mathsf{Prop}$.
- We want to form $\forall A : \mathsf{Prop}, A \to A$. Well, $\mathsf{Prop} : \mathsf{Type}$ and $A \to A : \mathsf{Prop}$, from earlier. So we need the rule $\mathsf{Type} \rightsquigarrow \mathsf{Prop}$. We do! So:
  - As $\mathsf{Type} \rightsquigarrow \mathsf{Prop}$ is a rule, $\mathsf{Prop} : \mathsf{Type}$ and $A \to A : \mathsf{Prop}$, we can form $\forall A : \mathsf{Prop}, A \to A$.

As we mentioned earlier, the $\forall A : \mathsf{Prop}$ here quantifies over *all* propositions, including itself! This means that Prop is an **impredicative** sort, as Props themselves can quantify over Props. The rule $\mathsf{Type} \rightsquigarrow \mathsf{Prop}$ is what allows this.

Now consider this function:

$$\mathsf{id?} = A \mapsto x \mapsto x : \forall A : \mathsf{Type}, A \to A.$$

This may seem like an okay function, but if we try to apply the same process, we run into a snag. We have $A \to A : \mathsf{Type}$, good so far. Then, to produce $\forall A : \mathsf{Type}, A \to A$, we'd need the type of Type. But the problem is that Type doesn't *have* a type!

How do we patch this up? The impulse is to say $\mathsf{Type} : \mathsf{Type}$. But this runs into a problem, where we can derive a contradiction, by using the ideas from Russell's paradox. Let's create this inductive type called box:

$$\mathsf{box} : \mathsf{Type} = \left\{ \mathsf{filtermap} : \forall T : \mathsf{Type}, (T \to \mathsf{Prop}) \to (T \to \mathsf{box}) \to \mathsf{box}. \right.$$

This is a kind of inductive type we've accepted so far. It actually implicitly uses $\mathsf{Type} : \mathsf{Type}$, because we are using T, which is a Type, to construct box, which also lies in Type. Something like $\mathsf{filtermap}(T)(P)(F)$ is intended to be the box that contains all $Ft : \mathsf{box}$ for all $t : T$ that satisfy $Pt : \mathsf{Prop}$.

We can now write what it means for $B : \mathsf{box}$ to be inside another box:

$$\mathsf{inside}(B) = \mathsf{ndestruct}_{\mathsf{box}}(\mathsf{Prop})(T \mapsto \underbrace{P}_{T \to \mathsf{Prop}} \mapsto \underbrace{F}_{T \to \mathsf{box}} \mapsto \mathsf{exists}(T)(t \mapsto Pt \wedge \mathsf{eq}(B)(Ft))).$$

Here we use the exists inductive type from earlier, as well as $\wedge$, which we could define with an inductive type.[12] Now, define

$$\mathsf{russell} = \mathsf{filtermap}(\mathsf{box})(t \mapsto \neg\mathsf{inside}(t)(t))(t \mapsto t).$$

It is possible to prove that, given a box $B : \mathsf{box}$, it's equivalent that $\neg\mathsf{inside}(B)(B)$ and $\mathsf{inside}(B)(\mathsf{russell})$. The forward direction follows from definition, after simplifying. In the backward direction, we have a proof that $\mathsf{inside}(B)(\mathsf{russell})$, which simplifies to $\mathsf{exists}(T)(t \mapsto \neg\mathsf{inside}(t)(t) \wedge \mathsf{eq}(B)(t))$, and using destructors to replace t with B yields the conclusion we want.

Given these, we now take B as russell and get both $\neg\mathsf{inside}(\mathsf{russell})(\mathsf{russell})$ and $\mathsf{inside}(\mathsf{russell})(\mathsf{russell})$, which gives $\bot$, a contradiction. So saying that Type : Type, while it seems nice, leads to a bad type theory.[13]

The way this is fixed, in the calculus of inductive constructions, is with **universes**. Along with Prop, we create the universe of sorts $\mathsf{Type}_0, \mathsf{Type}_1, \mathsf{Type}_2, \ldots$, one for each natural number. We also add Set, as a "small type" that is different from Prop. This is the pure type system of the calculus of inductive constructions:

- The sorts are $\mathsf{Set}, \mathsf{Prop}, \mathsf{Type}_0, \mathsf{Type}_1, \mathsf{Type}_2, \ldots$.
- The relationships are $\mathsf{Set} : \mathsf{Type}_0$, $\mathsf{Prop} : \mathsf{Type}_0$, and $\mathsf{Type}_i : \mathsf{Type}_{i+1}$.
- The rules are:
  - $\mathsf{Prop} \rightsquigarrow \mathsf{Prop}, \mathsf{Prop} \rightsquigarrow \mathsf{Set}, \mathsf{Set} \rightsquigarrow \mathsf{Prop}, \mathsf{Set} \rightsquigarrow \mathsf{Set}$,
  - $\mathsf{Type}_i \rightsquigarrow \mathsf{Prop}$,
  - and the special rule $\mathsf{Type}_i \rightsquigarrow \mathsf{Type}_j$, where we actually assign the result with type $\mathsf{Type}_{\max\{i,j\}}$ instead of just Type_j.

In the full calculus, instead of doing a bunch of definitions in Type, we mostly put our definitions in Set, unless we really need to go up to Type. Also note that the only sorts that can $\rightsquigarrow$ Type are Types in the first place.

Let's talk about the difference between Prop and Set. Again, let's consider the function id:

$$\mathsf{id} = A \mapsto x \mapsto x : \forall A : \mathsf{Prop}, A \to A.$$

The formation for $\forall A : \mathsf{Prop}, A \to A$, as $\mathsf{Prop} : \mathsf{Type}_0$ and $A \to A : \mathsf{Prop}$, relies on $\mathsf{Type}_0 \rightsquigarrow \mathsf{Prop}$. So this is still fine. Even in this system, Prop is still impredicative. On the other hand, we can't write

$$\mathsf{id} = A \mapsto x \mapsto x : \forall A : \mathsf{Set}, A \to A.$$

If we tried to form this type, as $\mathsf{Set} : \mathsf{Type}_0$ and $A \to A : \mathsf{Set}$, we'd need the rule $\mathsf{Type}_0 \rightsquigarrow \mathsf{Set}$. But we don't have that! So we *can't* write this function. That's the big difference between Set and Prop: Prop is impredicative and Set is not.

Now, what about this version of id?

$$\mathsf{id} = A \mapsto x \mapsto x : \forall A : \mathsf{Type}, A \to A.$$

[12] A technical note here is that we need a parametrized eq to say this, and this would technically be $\mathsf{eq}(\mathsf{box})(B)(Ft)$. But from here, and for the rest of the write-up, we'll omit the parameter on eq due to laziness.

[13] The reason we don't have this contradiction with Prop is that the destructor for Prop is different than the destructor for Type. We can't repeat this construction in Prop because of that.

Again, let's try to form the type. We didn't put the indices in the Type, because in practice, we omit the index and check that we can assign indices to make everything work. So, if we're making $A : \mathsf{Type}_0$, that means $A \to A : \mathsf{Type}_0$. As $\mathsf{Type}_0 : \mathsf{Type}_1$, we need the rule $\mathsf{Type}_1 \rightsquigarrow \mathsf{Type}_0$, which we do have. This places $\forall A : \mathsf{Type}_0, A \to A : \mathsf{Type}_1$. In general, we can see that if $A : \mathsf{Type}_i$, then $\forall A : \mathsf{Type}_i, A \to A : \mathsf{Type}_{i+1}$.

So this *is* a good definition, and we can write things like $\mathsf{id}(\mathsf{nat})$ and whatever. On the other hand, we *can't* write $\mathsf{id}(\mathsf{id})$, because if id takes in $A : \mathsf{Type}_i$, the type of id's type is Type_{i+1}, and we can't put Type_{i+1} into something that takes Type_i.

4.5 Equality

Let's talk about the type theory thing I understand the least, which is the notion of equality, and how it differs depending on the flavor of type theory you're using.

We first discuss the difference between intension and extension. And yes, that's intension with an 's'. Consider these two functions: $x \mapsto 3(2x + 6) : \mathsf{int} \to \mathsf{int}$, and $x \mapsto 2(3x + 9) : \mathsf{int} \to \mathsf{int}$. These two functions have a different **intension**: the way they compute their results are different, and we can tell that just by looking at it. On the other hand, we know that these functions have the same **extension**: for the same inputs, the functions will give the same outputs.

- First of all, there's **definitional equality**, or intensional equality. If we set one thing to be equal to another, by *definition*, then they are definitionally equal. So for example, 2 is definitionally equal to $S(SO)$.
- Looser is **computational equality**, or judgmental equality. Two things are computationally equal if they simplify to two things that are definitionally equal. So for example, $(x \mapsto x)2$ and 2 are computationally equal.
- Even looser is **propositional equality**, or extensional equality. For example, $\mathsf{add}(a)(b)$ is propositionally equal to $\mathsf{add}(b)(a)$. We represented this with eq, which is of type Prop, which is why it's called propositional equality.

There are two big flavors of type theory, intensional type theory, and extensional type theory. **Intensional type theory** is the one we've been building up so far. Definitional equality is a thing, by how we defined type theory. It is also by definition that, if two things are computationally equal, we can replace them with one another. On the other hand, even if things are propositionally equal, we can't always freely replace them for each other.

We've shown things like fequal, which state that if $\mathsf{eq}(x)(y)$, then $\mathsf{eq}(fx)(fy)$. But note that we started with propositional equality, and we ended once again with propositional equality. Sure, fequal can solve some of our problems with replacing x and y within *terms*. For example, we can use $f = t \mapsto \mathsf{plus}(t)(2)$ to show $\mathsf{eq}(\mathsf{plus}(x)(2))(\mathsf{plus}(y)(2))$. But what happens if they're in *types*? Suppose $P : \forall n : \mathsf{nat}, \mathsf{inttup}(n) \to \mathsf{Prop}$. If you knew $\forall a : \mathsf{inttup}(x), Pxa$, is it true that $\forall a : \mathsf{inttup}(y), Pya$? The answer is *yes*, but it takes effort to prove it.

In **extensional type theory**, we add the rule that if things are propositionally equal to each other, they are also computationally equal to each other. If we've found something of type $\mathsf{eq}(x)(y)$, then we can freely replace x for y *anywhere*. This makes our previous problem much easier. But it's also not a rule that means anything

computationally, unlike our other rules. It also makes type checking undecidable, which we don't want for many applications.[14]

Equality turns out to be an active area of research in type theory, and people are investigating what it means for two things to be equal. The field of **homotopy type theory** arises from taking intensional type theory, interpreting it using homotopy (whatever that means), and adding the univalence axiom, which is an axiom about equality. It's a fascinating field that I wish I knew more about.

4.6 Axioms

There are various strengths of axioms to add to our theory, all equality-related up until the end:

- There's **unicity of identity proofs**. This states that if $a : \mathsf{eq}(x)(y)$ and $b : \mathsf{eq}(x)(y)$, then $\mathsf{eq}(a)(b)$. Because the only constructor for eq is $\mathsf{reflexivity}$, that means that all proofs of equality are propositionally equal to $\mathsf{reflexivity}$.
- There's **proof irrelevance**. This states that if $P : \mathsf{Prop}$, $a : P$, and $b : P$, then $\mathsf{eq}(a)(b)$, or that any two proofs of the same proposition are propositionally equal. We can derive unicity of identity proofs from this.
- There's **propositional extensionality**. This states that if P and Q are Props, $P \to Q$, and $Q \to P$, then $\mathsf{eq}(P)(Q)$, or that equivalent propositions are propositionally equal. If two propositions imply each other, then they have the same extension, and that's how it gets its name. We can derive proof irrelevance from this.
- There's **predicate extensionality**. This states that if P and Q are both $A \to \mathsf{Prop}$, then if $\forall x : A, Px \to Qx$ and $\forall x : A, Qx \to Px$, we get $\mathsf{eq}(P)(Q)$. Equivalent predicates are propositionally equal. We can derive propositional extensionality from this.
- There's **functional extensionality**. This states that if f and g are $A \to B$ and $\forall x : A, \mathsf{eq}(fx)(gx)$, then $\mathsf{eq}(f)(g)$. This is the equality of functions we use in set theory. Propositional extensionality and functional extensionality can be used to prove predicate extensionality.
- There's **axiom of choice**. This states that if $R : A \to B \to \mathsf{Prop}$, and $\forall x : A, \mathsf{exists}(B)(y \mapsto Rxy)$, then $\mathsf{exists}(A \to B)(f \mapsto \forall x : A, Rx(fx))$. This is a functional way to state the more familiar axiom of choice: R is a family of sets with x as the indices and y as the elements, and f is the choice function that associates each index with an element.
- There's **excluded middle**. This states that for all $P : \mathsf{Prop}$, either P or $\neg P$. A result known as Diaconescu's theorem states that propositional extensionality, functional extensionality, and the axiom of choice, imply excluded middle.

So behold, the steady ladder that goes from seemingly "weak" extensionality axioms, all the way to classical logic. I think Diaconescu's theorem is really fascinating, and shows just how "classical" the axiom of choice is.

[14] I don't actually have a good example for why we don't usually do extensional type theory. If you have any you should tell me.

4.7 What next?

If you're interested in learning more, probably a good next step would be to "properly" learn how type theory is built. There's the concepts of contexts and judgments, and it's nice to read the conversion rules and typing rules in full rigor. There's the metatheory, with formal semantics and Heyting algebras and proofs as to when type checking is or isn't decidable. There are proof assistants like Coq and Lean in active development that use the theory we've built. There's homotopy type theory, which stretches out the Curry–Howard isomorphism even more.

And you can learn all of these by doing some more reading:

References

[1] Avigad, Jeremy, Leonardo de Moura, and Soonho Kong. "Theorem proving in Lean." (2015).

Intro-level textbook about working with Lean. Examples drawn from math.

[2] Barendregt, Henk P. "Lambda calculi with types." (1992).

Introduced the lambda cube. Talks about both Curry–style (implicitly-typed) and Church–style (explicitly-typed) lambda calculus.

[3] Bertot, Yves, and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science, Business Media, 2013.

Intro-level textbook about working with Coq. Draws many of its examples from program verification, and several from math too.

[4] Chlipala, Adam. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press, 2013.

More advanced introduction to working with Coq. A focus on program verification, but an excellent resource for Coq's theory too.

[5] Paulin-Mohring, Christine. "Inductive definitions in the system Coq rules and properties." Springer, Berlin, Heidelberg, 1993.

Introduction to inductive types. It explains the rules in more detail, and talks a bit about how inductive types actually work in Coq, where destructors aren't primitive, but built up from other language constructs.

[6] Sørensen, Morten Heine, and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism.* Elsevier, 2006.

Comprehensive textbook development from untyped lambda calculus to pure type systems. Goes over classical logic, sequent calculus, and formal semantics.

[7] The Univalent Foundations Program, Institute for Advanced Study. *Homotopy type theory: Univalent foundations of mathematics.* 2013.

The huge homotopy type theory book. I've barely read it, so I can't really comment.