Chapter 1

What this book is about

The *Tao Te Ching* is a classical book by Laotse which is about the way humans should follow in their life. I am borrowing the word *Tao* which means *the way* to talk about an alternative **way** to do *Mathematics*, namely using Intuitionistic Type Theory instead of Classical Set Theory.

Most Mathematicians nowadays accept Set Theory as the foundations of Mathematics. To be precise most Mathematicians don't really care very much about foundations, they just do Mathematics using sets because that's what they were taught do it. And it works. To a degree.

I am not a Mathematician but my education took place in Computer Science. Hence maybe I lack some of the skills of a good Mathematician but also some of the biases which go along with it. Having said this since I work in theoretical Computer Science most people would describe the papers I publish as *Mathematics.*

What is Set Theory? I will give a more detailed historic overview in chapter 2. Mathematicians organize mathematical objects in sets like the set of natural numbers $\mathbb{N}$ which are the numbers you use for counting $\mathbb{N} = \{0, 1, 2, \dots\}$ [1] Another example is the set of real numbers $\mathbb{R}$ which corresponds to the points on a line and for example the number $\pi = 3.1415\dots$ is a real number, we write $\pi \in \mathbb{R}$.

One of the nice things with set theory is the fact that you only need sets to construct mathematical objects. How does this work? We start with the empty set which can be written as $\{\}$ and now we can construct a new set, namely the set containing the empty set $\{\{\}\}$. Now we have these two different sets we can put them together $\{\{\}, \{\{\}\}\}$ to make a new set. Yes if you think of russian dolls that's the right idea. I don't want to write it down but continuing with this russian doll idea we can combine all the sets we have made to make a set with 3 elements and so on and thus we can create natural numbers as sets.

Ok this is the basic idea. They are some precise rules how to construct sets and how to reason about them. This is called Zermelo-Fraenkel set theory and

[1] That I start with 0 shows that I am a computer scientist. Real mathematicians start with 1.

it is a list of 8 axioms which are written in the language of predicate logic and which only use the symbol $\in$ in addition to the basic language of predicate logic. Actually there are a few extra axioms which are usually assumed such as the *axiom of choice* or you can also omit some axioms and work in a weaker system. So there is actually more than one set theory. I will explain the axioms and predicate logic in more detail in the next chapter.

Now what is Type Theory and how is it different? First of all when I say Type Theory with capital letters I mean an alternative foundation of Mathematics which is often associated with the Swedish Mathematician and Philosopher Per Martin-Löf. I say *associated* because while it is fair to say that he started it there are now lots of ideas in contemporary Type Theory which go beyond his original conception. People also use the term *type theory* (without capitals) as the theory of types in programming languages and there are some university courses with this title. The uncaptialized type theory is related to the capitalized Type Theory but it is not the same topic.

The basic idea of Type Theory is to organize mathematical objects into types instead of sets. So for example we have a type of natural numbers $\mathbb{N}$ and a type of real numbers $\mathbb{R}$ and to say that π is a real number we write $\pi : \mathbb{R}$.

Ok, I see what you are thinking: I have just replaced the word *set* by *type* and the symbol $\in$ by $:$ and that's all. Not so. In Type Theory we can only make objects of a certain type, that is the type is first and then we can construct the element. In Set Theory we think of all the objects (which are in turn sets) are there already and then we can organize them into different sets. Hence in set theory you can just have an arbitrary object x (which is a set again because everything is a set) and then you can ask yourself wether this object is is a natural number $x \in \mathbb{N}$ or a real number $x \in \mathbb{R}$. In Type Theory if I have $x : \mathbb{N}$ then this object is a natural number by birth and we can even ask the question wether it is a real number. We say that $x : \mathbb{N}$ is a *judgement* while $x \in \mathbb{N}$ is a proposition.

That sounds rather restrictive and so it is. Type Theory is a more disciplined approach to Mathematics but this pays off in the end. There are things we cannot do in Type Theory and we are better off because of it because Type Theory allows us to view mathematical objects more abstractly. What do I mean? In set theory there is actually more than one way to define natural numbers in some of them the empty set is include $\{\} \in \mathbb{N}$ in others it is not, we write $\{\} \notin \mathbb{N}$ to express that the empty set is not a natural number. However, this question hasn't much to do with natural numbers it is about our encoding of natural numbers. And because we can talk about the details of the encoding we cannot just replace one definition of natural numbers with another equivalent one because somewhere in our reasoning we may have used properties of the encoding. In Computer Science we would say there is a lack of information hiding.

In Type Theory we cannot ask these silly questions we cannot talk about the encoding of natural numbers. Either something is a natural number or it isn't. The empty type certainly isn't a natural number it is a type! We may compare this difference to static and dynamic typing in programming languages. For example Python is a dynamically typed language, we get an object and it may be a number. We can query this at runtime but we can also have runtime errors because we assume something is a number but then it turned out to be a string. So in a way Python is a bit like set theory. In contrast in a statically typed language like Haskell we know at compile time what is the type of an object. As a consequence we cannot have runtime errors caused by trying to add a number and a string. So Haskell is more like Type Theory.

In Programming the statically typed approach pays off because we can catch more errors earlier. In Mathematics we have to prove things so we shouldn't make errors. However, the advantage of Type Theory is that we cannot talk about the encoding of objects and as a consequence we can replace one type by another which is *equivalent*. I am going to make this precise later but in the moment you can think of *equivalent* as meaning that there is a one-to-one correspondence of objects in two types. This is the essence of the *univalence axiom* which was introduced by Vladimir Voevodsky and which basically says that two types which are equivalent are actually equal. You couldn't do this in set theory because in set theory we can actually distinguish equivalent sets by talking about details of their encodings (e.g. wether the empty set is a natural number).

It is interesting that Voevodsky came up with the univalence axiom by thinking about an abstract version of geometry which is called Homotopy Theory. To oversimplify things: Homotopy Theory classifies geometric objects by the paths you can follow on their surface. That is a ball is different from a bicycle tube because on the tube you can walk from one point to itself by going through around the inside. You can also walk around a ball but you can transform this path by many little steps into the empty path i.e. where you just stand still. Hence upto deformation of paths there is only one path on a ball. The bicycle tube is different because we cannot deform every path to the empty path because we are only allowed to deform paths on the surface we are not allowed to go through the middle.

Hence this new version of Type Theory is often called *Homotopy Type Theory* and there is a nice book (ok I was involved in writing it) about this topic but this requires a bit of mathematical background [**?**].

There is another important difference between Type Theory and Set Theory which I would like to explain. As I said above set theory uses predicate logic which codifies the rules of reasoning. It also introduces a formalism how to write logical statements (these are called propositions), e.g. using $\wedge$ for *and* and $\forall$ to say *for all*. The rules are justified by an explanation when a proposition is true. Now this makes sense if we talk about the real world because hopefully we can just check wether a statement is true. Ok, this can also be difficult but this is a different story. However, the sets do not exist in the real world they only exist in our heads. Now what does it really mean for a statement about sets to be true?

At this point we are getting philosophical and refer to the greek philosopher Plato. Plato had the view that the world of ideas is as real as the real world and that the things we see are mere shadows of ideal objects. That is a real table is just a shadow of the ideal table. In this *Platonic universe* mathematical objects like sets are real and hence we can talk about.

Ok that doesn't sound very convincing. Maybe the reason is that I am not a Platonist. But Mathematicians would say that it just makes sense to pretend that we can talk about mathematical objects as if the were real and who cares about philosophy anyway.

The nice thing about Type Theory is that we don't need to refer to any Platonic universe or even to have the notion of truth precise. Instead of truth we should rather think of evidence. We can explain what is the evidence for a proposition by associating a type with every proposition which is the type of evidence for this proposition. This is called the *proposition-as-types* principle or the *Curry-Howard-Equivalence*.

How does this work? If A and B are propositions and we know what is evidence for A and for B but what is evidence for $A \wedge B$ that is A **and** B? We say evidence for $A \wedge B$ is a pair (a, b) where $a : A$ and $b : B$. We can also write $A \wedge B = A \times B$ where $A \times B$ is the type of pairs or tuples made from A and B. An interesting example is the explanation of if-then that is *If* A *then* B which we write as $A \Rightarrow B$. Now evidence for $A \Rightarrow B$ is a function from A to B. I write the type of functions from A to B as $A \to B$ and we can now say $A \Rightarrow B = A \to B$ that is the reasons that A imples B are the functions from the reasons for A to the reaosns for B.

I should say what I mean by a function. Unlike in set theory functions in Type Theory are a primitive concept. The functions in Type Theory are the same as functions in functional programming languages like Haskell but with the proviso that they have to terminate, that is they can't run forever. And indeed Type Theory is a programming language and you can execute your type-theoretic programs. This is not true in Set Theory where there a things which are called functions which you cannot implement on a computer. I always say that this is a bad name because a function which doesn't function shouldn't be called a function.

The logic of evidence we get from Type Theory is different from the logic of truth we get in predicate logic. The culprit is the *principle of excluded middle* which says that every proposition is either true or false. We write $A \vee \neg A$ where $\vee$ means *or* and $\neg$ means *not* and A is just any proposition. We can prove this by constructing a truth table and going through the possibilities that A is either true or false. In the evidence based semantics of Type Theory we cannot justify this principle because it basically says that for any proposition we have either to give evidence that is true or we have to give evidence that it is not true. But we cannot in general do this because there are certainly propositions of which we don't know wether it is true or false. Actually it is even worse: because propositions are given by types and we cannot look into a type any proof of the principle of excluded middle would either to have to prove or disprove all propositions, which makes no sense at all.

The logic we get is called *Intuitionistic Logic* for which the Dutch Mathematician Brouwer was famous. It is called intuitionistic because the basic idea are based on intuitive justification and not the platonic universe. Hence there is also a version of predicate logic which does not include the principle of the excluded middle and this is called *Intuitionistic Predicate Logic*, while the one with excluded middle is called *Classical Predicate Logic*. I think Type Theory is much nicer because we don't need any predicate logic on top. We just have to explain the translation of propositions to types (which is straightforward) and then there is no need for a logical system on top. I think this is really cool: Type Theory is a programming language and you get logic for free.

The restriction to intuitionistic logic is not only nice philosophically (and we can always be a bit blaze about philosophy) but it is also important practically. This follows from what I said already: because our language is a programming langauge we can compute with it. So if we prove that there is a number with some property $\exists x : \mathbb{N}.P(x)$ then we can actually compute the number from the proof. That is not the case in classical logic because we cannot execute our functions. Also in an intuitionistic system you can make some useful differences: while the principle of excluded middle doesn't always hold it may hold in some particular case. For example we may have a property P of the natural numbers so that for each instance we can prove the particular instance of the excluded middle. We write $\forall x : \mathbb{N}.P(x) \vee \neg P(x)$ that is for each natural number either P holds or it doesn't. In Computer Science terms this means that the property P is decidable. As we know thanks to Turing not every property is decidable. Many are, so for example we can decide wether a number is a prime number that is greater than 1 and only divisible by 1 and itself. But we cannot decide that if we view the number as the bit pattern of a program on a computer wether this program will stop or run forever. Hence in an intuitionistic system we can express this and other properties useful in Computer Science internally.

Ok, I realize that I am trying to sell Type Theory here. Not everybody would agree and we can have long discussions about it. Also for the clarity of presentations I have avoided to hum and err too much but as we know there are two sides to everything. Mathematicians are no fools and many of them think that using intuitionistic logic would complicate things too much and that classical reasoning is easier. I don't agree but I leave it to them to make their case.

Chapter 3

Simple types

The set theory in the last chapter was getting quite technical. So let's go for something easier, let's move on to *simple types*. One important difference between Set Theory and Type Theory that instead of encoding the constructs we need in terms of sets they just become basic constructs. As a consequence Type Theory has more primitive concepts which you may think is not so good. However, I think that some of the coding tricks in set theory are quite artificial and actually they are a number of basic constructs which shouldn't be encoded in terms of each other.

The first central construction of Type Theory is the function type which we also write $\to$. I will in general use the same symbols in set theory and type theory which are based on the same intuitive ideas. The same applies to the type of natural numbers $\mathbb{N}$, cartesian product $\times$ etc. In this chapter we will look at *simple types* as opposed to *dependent types* which have to wait until the next chapter. This is also historically correct because simple types where first introduced by Alonzo Church in the 1940ies also with the intention to use it for foundation of Mathematics but with a slightly different slant then the Type Theory which we will be looking at. This theory which is also called *simply typed λ-calculus* is at the base of type systems of typed functional programming languages, examples are SML, CAML and Haskell.

Before we start to look at the definition of functions and other basic constructs I would like to expand on what I have already said in chapter **??**. There is a fundamental difference between $3 \in \mathbb{N}$ and $3 : \mathbb{N}$. The first one is a proposition, that is something we may want to prove, while the 2nd is a judgement which is something which just holds by looking at it. In Type Theory there is another important judgement namely definitional equality which we denote $\equiv$. Definitional equalities are not something we want to prove but something that should follow from the definitions, e.g. that $3 + 3 \equiv 6$ should follow from the definition of $+$. We also have $=$ in Type Theory and it plays a similar role as $=$ in predicate logic, even though things are going to be a bit more subtle once we look at Homotopy Type Theory. In any case we won't introduce $=$ before we introduce dependent types in the next chapter.

3.1 Functions that function

A function to me is not a set of pairs but a machine where you can put something in and get something out. And this is an important basic construct irreducible to anything else. As an example consider the function $f : \mathbb{N} \to \mathbb{N}$ which is defined as $f(x) :\equiv x + 2$. Note that we use $\equiv$ because this is the definition of f. Now if we apply the function to an argument like $f(3)$ we can calculate $f(3) \equiv 3 + 2 \equiv 5$. The first step is the interesting one: if we apply a function to an argument we can compute the answer by replacing the parameter with the actual argument in the definition of the function. The 2nd step uses the definition of $+$ which I haven't given but I thought you know $+$ already. Actually I am going to define it explicitly later in this chapter. The equality we obtain by substituting the parameter in the definition of a function is called β-equality and it is the core to computing with functions in Type Theory.

However, functions should be mathematical objects on their own, whereas here we combine the idea of defining a name f and the idea of a function. We should be able to separate the two and just write down a function without assigning it to a name at the same time. This is achieved by the λ-notation which was introduced by Church. The λ-notation enables us to write the function without giving it a name: $\lambda x.x{+}3 : \mathbb{N} \to \mathbb{N}$. We can read the previous definition of f as $f(x) :\equiv x + 2$ as a shorthand for $f :\equiv \lambda x.x + 2$. And the β-rule states that $(\lambda x.x + 2)(3) \equiv 3 + 2$.

λ allows us to define a function with one argument. How can we capture functions with several arguments like $+$ itself? One approach would be to introduce cartesian products (which we are going to do soon anyway) and then consider $\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with $\mathrm{plus}((x, y)) \equiv x + y$ However, in Type Theory as in functional programming we adopt a different approach following an idea of Haskell Curry which is called currying. We define $\mathrm{plus} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ as $\mathrm{plus} :\equiv \lambda x.\lambda y.x + y$. plus is a function which applied to a natural number returns a function. That is $\mathrm{plus}(3) \equiv (\lambda x.\lambda y.x + y)(3) \equiv \lambda y.3 + y$, the result is the function that adds 3. We can further apply this function as in $\mathrm{plus}(3)(5) \equiv (\lambda y.3 + y)(5) \equiv 3 + 5 \equiv 8$.

There are some notational conventions which I should mention: since we always use currying to define functions with several parameters we avoid the proliferation of brackets in the type of a function with several parameters by reading $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ as $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. Also when we define a function with several parameters we do not repeat the λ but write $\lambda x, y.x + y$ (as we have already done for $\forall$ in the previous chapter). Finally, an expression like $\mathrm{plus}(3)(4)$ looks a bit strange and hence to avoid having to write brackets each time when applying a function we adopt the convention from functional programming and write function application just as empty space that is $\mathrm{plus}\,3\,4$. Here another convention comes into play namely that we don't have to write this as $(\mathrm{plus}\,3)\,4$ but can omit these brackets. In computer science slang we say the $\to$ associates to the right while application associates to the left. This combination of conventions achieves that when we define curried functions we don't need to write brackets in the types and when we apply them we don't need to write brackets to group the arguments.

On the other hand it does make sense to consider a function $g : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ which is a function that gets a function as input and returns a number. An example would be $g :\equiv \lambda k.(k\,2) + (k\,3)$ or we may just write $g\,k :\equiv (k\,2) + (k\,3)$. So what is $g\,f$ where $f : \mathbb{N} \to \mathbb{N}$ was defined earlier as $f\,x = x + 2$? We can calculate this just using the β-law and our knowledge of $+$.

$$\begin{aligned} g\,f &\equiv (f\,2) + (f\,3) \\ &\equiv (2+2) + (2+3) \\ &\equiv 9 \end{aligned}$$

A function like g is called a *higher order function* because it accepts functions as input.

The particular names of variables we choose doesn't really matter, all what a variable is doing is to create an invisible link between the place where it is introduced via λ and the place where it is used. This is echoed by the rule that functions which only differ in the choice of variable names are considered to be definitionally equal. That is for example $\lambda x.x + 3 \equiv \lambda y.y + 3$ - this is called α-equality.

When calculating with functions we have to be careful that the link between the introduction and the use of a variable is not destroyed. Consider again plus $:\equiv \lambda x.\lambda y.x + y$ and now define $h :\equiv \lambda y.\mathrm{plus}\,y$. It seems that we can calculate that $\mathrm{plus}\,y \equiv (\lambda x.\lambda y.x + y)\,y \equiv \lambda y.y + y$ and hence $h \equiv \lambda y.\lambda y.y + y$. This is confusing because one of the ys should refer to the inner and one to the outer y. Actually there is a convention that a variable always refers to the latest introduction and hence both ys should refer to the inner y. But this is wrong because we have destroyed the connection between the introduction of y and the its use. Indeed, this phenomenon is called *variable capture* and it has to be avoided. But then what is $(\lambda x.\lambda y.x + y)\,y$? A way out is to use α-equality to rename the bound variable before β-reducing $(\lambda x.\lambda y.x+y)\,y \equiv (\lambda x.\lambda z.x+z)\,y \equiv \lambda z.y + z$ and hence $h \equiv \lambda y.\lambda z.y + z$.

Actually you may have observed that the result of reducing h is α-equivalent to plus. Because all we have done is transformed a function f into $\lambda x.f\,x$ which is just the same function. This rule is often considered to hold in general and is called η-equality. In the case of the example above η-equality doesn't tell us anything new but we can also apply this to functions which are just variables as in $\lambda f.\lambda x.f\,x$ which is η-equal to $\lambda f.f$. In this case we really need to use η. However, we should not call this a definitional equality, if η is included we say it is *judgemental equality*.

There are many important functions that work for all types - this is called polymorphism. I think to properly understand polymorphism we need dependent types, hence we will get back to this in the next chapter. However, we can fake it by just assuming we have some types in prose. Here are two important functions like this:

the identity function This is a function which just echoes its input. Let A be a type then we can define $\mathrm{id}_A : A \to A$ as $\mathrm{id}_A :\equiv \lambda x.x$,

composition Given types A, B, C and two functions $f : A \to B$ and $g : B \to C$ we can combine them to form a new function $g \circ f : A \to C$ which on an input $a : A$ first runs f and then g. More precisely we define $\mathrm{cmp}_{A,B,C} : (B \to C) \to (A \to C) \to A \to C$ as $\mathrm{cmp}_{A,B,C} :\equiv \lambda f, g.\lambda x.f\,(g\,x)$ but to improve readability we write $\mathrm{cmp}_{A,B,C}\, f\, g$ as $f \circ g$.

You may notice the strange change of direction in when looking at the type of composition: first comes the function which is run last. The reason for tis is that already function application is the wrong way around that is when calculating $f\,(g\,a)$ we first evaluate $g\,a$ and then feed the result into f, but we write f before g reading from left to right. The point is that composition just doesn't change the order of functions that is $(f \circ g)\,a \equiv f\,(g\,a)$. If we could reinvent mathematical notation from scratch we would write the argument before the function. This is as easy as convincing English people to drive on the right side instead of the left.

There are some basic equalities governing these two functions: If we compose any function with the identity function then nothing changes. This is expressed as $f \circ \mathrm{id} \equiv f$ and $\mathrm{id} \circ f \equiv f$. Actually to be clear these equalities follow from the β and η-laws:

$$\begin{aligned} f \circ \mathrm{id} &\equiv \lambda x.f\,(\lambda y.y)\,x \\ &\equiv \lambda x.f\,x \\ &\equiv f \end{aligned}$$

and for the other direction

$$\begin{aligned} \mathrm{id} \circ f &\equiv \lambda x.(\lambda y.y)\,(f\,x) \\ &\equiv \lambda x.f\,x \\ &\equiv f \end{aligned}$$

Another equation tells us that if we compose f with the composition of g and h then this will give us the same result as composing the composition of f and g with h, that is $f \circ (g \circ h) \equiv (f \circ g) \circ h$. This law follows from the β-law. We show this by showing that both expressions are equal to $\lambda x.f\,(g\,(h\,x))$:

$$\begin{aligned} f \circ (g \circ h) &\equiv \lambda x.f\,(\lambda y.g\,(h\,y))\,x \\ &\equiv \lambda x.f\,(g\,(h\,x)) \end{aligned}$$

and

$$\begin{aligned} (f \circ g) \circ h &\equiv \lambda y.(\lambda x.f\,(g\,x))\,(g\,y) \\ &\equiv \lambda x.f\,(g\,(h\,x)) \end{aligned}$$

Note that I am changing the name of variables all the time not just to avid capture but also to improve readability, because if you see the same variable twice you may mistakenly assume that the refer to the same thing but as we have seen this isn't always the case.

I am not saying that $g \circ f$ is the same as $g \circ f$, first of all this doesn't *type-check* because if $g : B \to C$ and $f : A \to C$ with all three types being different then only $g \circ f : A \to C$ makes sense but $g \circ f$ doesn't. For this equality to make sense we need to assume that $f, g : A \to A$ and in this case we can compare $f \circ g$ and $g \circ f$ because both have the type $A \to A$. However, it is easy to find a counterexample: let $f : \mathbb{N} \to \mathbb{N}$ be $f\,x \equiv x + 2$ and $g : \mathbb{N} \to \mathbb{N}$ be $g\,x \equiv 2 \times x$ then $f \circ g \equiv \lambda x.f\,(g\,x) \equiv \lambda x.(2 \times x) + 2$ while $g \circ f \equiv \lambda x.g\,(f\,x) \equiv 2 \times (x + 2)$. These functions are different, since for example $(f \circ g)\,1 \equiv 4$ while $(g \circ f)\,1 \equiv 6$.

A structure satisfying the three laws $f \circ \mathrm{id} \equiv f$, $\mathrm{id} \circ f \equiv f$ and $f \circ (g \circ h) \equiv (f \circ g) \circ h$ is called a category. Category theory grew out of very abstract algebra and was established since the 1940ies by Saunders MacLane and others. There is a very good fit between category theory and Type Theory but they have different purposes. Type theory is based on some intuitive insight what laws should hold, while category theory has some very effective mechanisms to classify laws and support abstract reasoning.

3.2 Products without multiplication

Products in simple type theory are very easy, given $a : A$ and $b : B$ we can form $(a, b) : A \times B$. How do we construct a function out of a product, that is $f : A \times B \to C$? We just need to say what it does on tuples, e.g. we can define the alternative version of

$\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

by defining

$\mathrm{plus}\,(x, y) :\equiv x + y$

This induces a definitional equality $\mathrm{plus}(3, 4) :\equiv 3 + 4$.

We can define two generic functions, which are called *projections* to extract the components of a tuple: $\pi_0 : A \times B \to A$ and $\pi_1 : A \times B \to B$ which are defined:

$\pi_0(x, y) :\equiv x$
$\pi_1(x, y) :\equiv y$

Using only projection there is a different way to define plus:

$\mathrm{plus}\,p :\equiv (\pi_0\,p) + (\pi_1\,p)$

Indeed all functions on $\times$ can be defined using only projections. Hence the defining equations for the projections are the β-equality for products.

Given a tuple $p : A \times B$ with $p :\equiv (a, b)$ what happens if we take it's projections and then put it back together? That is $(\pi_0\,p, \pi_1\,p) : A \times B$. Exactly we end up where we started $(\pi_0\,p, \pi_1\,p) \equiv (a, b)$. Generalising this to any term gives us the η-rule for products:

$(\pi_0\,p, \pi_1\,p) \equiv p$

Exercises for Naive Type Theory
Part 1

Thorsten Altenkirch

April 11, 2016

1. Given

   $$\begin{aligned}
   &f : \mathbb{N} \to \mathbb{N}\\
   &f :\equiv \lambda x.x + x\\
   &g : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}\\
   &g :\equiv \lambda h.\lambda n.h\,(h\,n) + n
   \end{aligned}$$

   Evaluate $g\,f\,(f\,1)$ justifying each step and underline the affected subterm.

2. For any types A, B, C we define identity and composition as follows:

   $$\begin{aligned}
   \mathrm{id}_A &: A \to A\\
   \mathrm{id}_A &:\equiv \lambda x.x\\
   \mathrm{cmp}_{A,B,C} &: (B \to C) \to (A \to B) \to A \to C\\
   \mathrm{cmp}_{A,B,C} &:\equiv \lambda f, g.\lambda x.f\,(g\,x)
   \end{aligned}$$

   Writing $\mathrm{cmp}_{A,B,C}\,f\,g$ as $f \circ g$ show that this data forms a category by establishing

   (a) $f \circ \mathrm{id} \equiv f$

   (b) $\mathrm{id} \circ f \equiv f$

   (c) $(f \circ g) \circ h \equiv f \circ (g \circ h)$

   for appropriate f, g, h. You can use η-equality - where do we need it?

3. For any types A, B, C find an element of

   $$((A \to C) \to C) \to (A \to B) \to ((B \to C) \to C)$$

I am not saying that $g \circ f$ is the same as $g \circ f$, first of all this doesn't *type-check* because if $g : B \to C$ and $f : A \to C$ with all three types being different then only $g \circ f : A \to C$ makes sense but $g \circ f$ doesn't. For this equality to make sense we need to assume that $f, g : A \to A$ and in this case we can compare $f \circ g$ and $g \circ f$ because both have the type $A \to A$. However, it is easy to find a counterexample: let $f : \mathbb{N} \to \mathbb{N}$ be $f\,x \equiv x + 2$ and $g : \mathbb{N} \to \mathbb{N}$ be $g\,x \equiv 2 \times x$ then $f \circ g \equiv \lambda x.f\,(g\,x) \equiv \lambda x.(2 \times x) + 2$ while $g \circ f \equiv \lambda x.g\,(f\,x) \equiv 2 \times (x + 2)$. These functions are different, since for example $(f \circ g)\,1 \equiv 4$ while $(g \circ f)\,1 \equiv 6$.

A structure satisfying the three laws $f \circ \mathrm{id} \equiv f$, $\mathrm{id} \circ f \equiv f$ and $f \circ (g \circ h) \equiv (f \circ g) \circ h$ is called a category. Category theory grew out of very abstract algebra and was established since the 1940ies by Saunders MacLane and others. There is a very good fit between category theory and Type Theory but they have different purposes. Type theory is based on some intuitive insight what laws should hold, while category theory has some very effective mechanisms to classify laws and support abstract reasoning.

3.2 Products without multiplication

Products in simple type theory are very easy, given $a : A$ and $b : B$ we can form $(a, b) : A \times B$. How do we construct a function out of a product, that is $f : A \times B \to C$? We just need to say what it does on tuples, e.g. we can define the alternative version of

$\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

by defining

$\mathrm{plus}\,(x, y) :\equiv x + y$

This induces a definitional equality $\mathrm{plus}(3, 4) :\equiv 3 + 4$.

We can define two generic functions, which are called *projections* to extract the components of a tuple: $\pi_0 : A \times B \to A$ and $\pi_1 : A \times B \to B$ which are defined:

$\pi_0(x, y) :\equiv x$
$\pi_1(x, y) :\equiv y$

Using only projection there is a different way to define plus:

$\mathrm{plus}\,p :\equiv (\pi_0\,p) + (\pi_1\,p)$

Indeed all functions on $\times$ can be defined using only projections. Hence the defining equations for the projections are the β-equality for products.

Given a tuple $p : A \times B$ with $p :\equiv (a, b)$ what happens if we take it's projections and then put it back together? That is $(\pi_0\,p, \pi_1\,p) : A \times B$. Exactly we end up where we started $(\pi_0\,p, \pi_1\,p) \equiv (a, b)$. Generalising this to any term gives us the η-rule for products:

$(\pi_0\,p, \pi_1\,p) \equiv p$

As for functions you may wonder why we add this law because can't we derive it. Indeed if p is of the form (a, b) (or equal $\equiv$ to it) then it follows from the β-rule. The η rule extends this to expressions which are not of this form, e.g. the identity function on a products $\mathrm{id} : A \times B \to A \times B$ with $\lambda x.x$ is judgementally equal to $\lambda x.(\pi_0\, x, \pi_1\, x)$.

I have now presented two versions of plus, one had the type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and the other $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. In functional programming we prefer the 2nd version, which is called a curried function. Indeed, we can derive some generic operations to curry and uncurry a function and hence to move between the two ways to represent functions with two arguments: $A \times B \to C$ and $A \to B \to C$. We can define two operations which translate between the two representations:

$$\begin{aligned} \text{curry} &: (A \times B \to C) \to (A \to B \to C) \\ \text{uncurry} &: (A \to B \to C) \to (A \times B \to C) \end{aligned}$$

They are not hard to define:

$$\begin{aligned} \text{curry}\, f &:\equiv \lambda x.\lambda y.f\,(x, y) \\ \text{uncurry}\, g &:\equiv \lambda z.g\,(\pi_0\, z))\,(\pi_1\, z) \end{aligned}$$

What happens if we curry and then uncurry? If we just expand the definitions of curry and uncurry in [1]:

$$\text{uncurry}\,(\text{curry}\, f)$$

we get

$$\lambda z.(\lambda x.\lambda y.f\,(x, y))\,(\pi_0\, z)\,(\pi_1\, z)$$

and we can use the β-law for functions to reduce this to

$$\lambda z.f\,(\pi_0\, z, \pi_1\, z)$$

now we have a wonderful opportunity to use the η-law for products to obtain $\lambda z.f\, z$ and by the η-law for functions this is judgementally equal to f - hence we get back to where we started.

Vice versa what happes if we uncurry and then curry? Expanding

$$\text{curry}\,(\text{uncurry}\, g)$$

we obtain

$$\lambda x.\lambda y.\,(\lambda z.g\,(\pi_0\, z))\,(\pi_1\, z))\,(x, y)$$

as before we use the β-rule for functions to reduce this to

$$\lambda x.\lambda y.g\,(\pi_0\,(x, y)))\,(\pi_1\,(x, y)))$$

[1] Actually we apply the β-rule for functions

and now applying the definition of the projections, that is the β-law for products we can simplify this to $\lambda x.\lambda y.g\,x\,y$ and now after applying the η-law for functions twice we are left with g, hence again where we started.

We say that the operations curry and uncurry are inverse to each other. After all this shouldn't come as a surprise because I was saying that the two representations of functions with two arguments are equivalent and this equivalence is witnessed by these two inverse operations between the two representations. In the language of category this equivalence is used to express the relationship between $\times$ and $\to$.

3.3 Coproducts : Products in the mirror

So what is the *mirror image* of products? They are called coproducts and the *co-* is the categorical terminology for mirror. Given types A and B we construct a new type $A + B$ whose elements are $\mathrm{inl}\,a : A + B$ for $a : A$ and $\mathrm{inr}\,b : A + B$ for $b : B$. These operations are called *injections*, inl stands for inject left and inr for inject right. In set-theory this operation is called *disjoint union* and it is defined using the $\cup$ operation, that is

$x + y = \{0\} \times x \cup \{1\} \times y$

assuming now that x, y are sets and using the representations of $0, 1, \times$ introduced in the previous section.

$+$ is different from $\cup$ in that it labels the arguments and hence makes them disjoint. To see the difference, consider the type of booleans with $\mathrm{true}, \mathrm{false} : \mathrm{Bool}$ corresponding to the set $\mathrm{Bool} = \{\mathrm{true}, \mathrm{false}\}$. Now in set theory $\mathrm{Bool} \cup \mathrm{Bool} = \mathrm{Bool}$ since multiplicity of elements doesn't matter. On the other hand in Type Theory $\mathrm{Bool} + \mathrm{Bool}$ has four elements, namely

$\mathrm{inl}\,\mathrm{false}, \mathrm{inl}\,\mathrm{true}, \mathrm{inr}\,\mathrm{false}, \mathrm{inr}\,\mathrm{true} : \mathrm{Bool} + \mathrm{Bool}$

It should also be no surprise that this type has exactly $4 = 2 + 2$ elements, since the coproduct of two finite types has the sum of numer of elements of the two types. Hence the use of the $+$-symbol, using the same analogy as for $\times$.

$+$ is a sensible operation in Type Theory while $\cup$ isn't. $\cup$ exposes the internal encoding used, e.g consider $\mathbb{N} \cup \mathrm{Bool}$ in set theory: if we represent $\mathrm{Bool} = \{0, 1\}$ then $\mathbb{N} \cup \mathrm{Bool} \subseteq \mathbb{N}$. However, if we use a different encoding, lets say $\mathrm{Bool} = \{(0, 0), (1, 1)\}$ then this is not the case. This is bad because it exposes the internal implementation of Bool, and this hinders abstraction. On the other hand there is not problem with $+$, $\mathbb{N} + \mathrm{Bool}$ is always the same type upto equivalence, independent of the choice of representation of Bool and $\mathbb{N}$. This is the basic intuition behind the univalence principle which will discuss later.

To summarize: $\cup$ is not a type-theoretic operation because it is too intensional. On the other hand $+$ is an extensional operation but it is not defined via $\cup$ as in set theory. Hence I think that the name *disjoint union* is misleading and I prefer the term *coproduct* which expresses the categorical duality to products.

How to define a function out of a coproduct? As an example consider $f : \mathrm{Nat} + \mathrm{Nat} \to \mathrm{Nat}$ which is defined by saying how it behaves for left and how for right injections:

$$\begin{aligned} &f\,(\mathrm{inl}\,n) :\equiv 2 \times n \\ &f\,(\mathrm{inr}\,n) :\equiv 2 \times n + 1 \end{aligned}$$

This function maps the left injections to the even numbers and the right injections to the odd numbers. Indeed it is part of an equivalence between $\mathbb{N} + \mathbb{N}$ and $\mathbb{N}$ but we don't yet have the means to express let alone prove it.

We can also devise a generic operations to define functions out of coproducts which we call case because it performs a form of case analysis. Given types A, B, C we define $\mathrm{case}_{A,B,C} : (A \to C) \to (B \to C) \to (A + B \to C)$ by

$$\begin{aligned} &\mathrm{case}\,f\,g\,(\mathrm{inl}\,a) :\equiv f\,a \\ &\mathrm{case}\,f\,g\,(\mathrm{inr}\,b) :\equiv g\,b \end{aligned}$$

As an example we could have defined the function f above using case:

$$f :\equiv \mathrm{case}\,(\lambda n.2 \times n)\,(\lambda n.2 \times n + 1)$$

The defining equations for case are the β-laws for coproducts. What about the η-rule? We may think that

$$\mathrm{case}\,\mathrm{inl}\,\mathrm{inr}\,x \equiv x$$

is enough but it turns out that this isn't strong enough to prove many equalities on coproducts. As a example consider $\mathrm{swap} : A + B \to B + A$:

$$\begin{aligned} \mathrm{swap}\,(\mathrm{inl}\,a) &:\equiv \mathrm{inr}\,a \\ \mathrm{swap}\,(\mathrm{inr}\,b) &:\equiv \mathrm{inl}\,b \end{aligned}$$

Now we would like to show that swapping twice doesn't do anything:

$$\mathrm{swap}\,(\mathrm{swap}\,x) \equiv x$$

However, the β and η-rules for products are not strong enough to do this. It is possible to add additional equations to fix this and we have shown that the resulting theory is decidable, that is can be checked by a computer, but this wasn't so easy. Even worse the algorithm is quite inefficient and not really useful in practice.

In what sense are coproducts the mirror image of products? From the perspective of Category Theory products are characterized by an equivalence of $(C \to A) \times (C \to B)$ and $C \to A \times B$ given by the following functions:

$$\begin{aligned} \mathrm{pair} &: ((C \to A) \times (C \to B)) \to C \to A \times B \\ \mathrm{pair}\,f\,g\,c &\equiv (f\,c, g\,c) \\ \mathrm{unpair} &: (C \to A \times B) \to (C \to A) \times (C \to B)) \\ \mathrm{unpair}\,h &:\equiv (\lambda c.\pi_0\,(h\,c), \pi_1\,(h\,c)) \end{aligned}$$

Just using the β and η-laws for products and functions we can show that these two functions re inverse to each other, i.e. $\mathrm{unpair}\,(\mathrm{pair}\,x) \equiv x$ and $\mathrm{pair}\,(\mathrm{unpair}\,x) \equiv x$).

Now for coproducts we have another equivalence which can be obtained by turning all the arrows around. That is $(A \to C) \times (B \to C)$ and $A + B \to C$ are equivalent. From left to right this is just a curried version of case and other one $\mathrm{uncase} : (A + B \to C) \to (A \to C) \times (B \to C)$ which is defined using the injections:

$$\mathrm{uncase}\, f :\equiv (\lambda a.f\,(\mathrm{inl}\,a), f\,(\lambda b.\mathrm{inr}\,b))$$

We can show that $\mathrm{case}\,(\mathrm{uncase}\,x) \equiv x$ but the other equality $\mathrm{uncase}\,(\mathrm{case}\,x) \equiv x$ corresponds exactly to the strong theory which we have discussed previously. However, once we introduce the equality type we can show that this equality holds upto provable equality $\mathrm{case}\,(\mathrm{uncase}\,x) = x$ which is sufficient. Indeed as soon as we move on to natural numbers in the next section the strong theory is undecidable, i.e. it cannot be implemented on a computer.

So, indeed the mirror we are using is a bit broken. If we want to be principled we really should reject all η-laws. The η-laws are not really definitional equalities, they are extensional equalities which we can implement on a computer. The coproducts are a borderline case which is implementable but not practically. Hence should we just get rid of all the η-laws. Maybe but they are actally convenient to have in implementations of Type Theory. We will get back to this question later.

I had briefly discussed the laws of a monoid, maybe you remember that 1 is the neutral element for multiplication and 0 is the neural element for addition. Indeed we have types corresponding to 1 and 0 and they are called unit and empty type. We have an element of unit $() : 1$. This notation should remind you in an empty tuple because 1 is in a way the extreme case of a product. The empty type has, as the name says, no element. To define a function out of the empty type like $f : 0 \to A$ we have to give no defining equation because this function will never be applied to an argument. Hence there is no work involved defining functions out of the empty type.

Getting back to η-laws the η-rule for the empty type has the particular wierd consequence that any two terms are equal as soon as we can construct an element of the empty type. This is maybe ok for simple types but it is impossible to decide (on a computer) wether the empty tye is inhabited as soon as we have got dependent types. Hence while we could possibly have η-laws for binary coproduct, we can't implement them for the 0-ary coproduct that is the empty type.

Using 1 and + we can define $\mathrm{Bool} :\equiv 1 + 1$ with $\mathrm{false} :\equiv \mathrm{inl}()$ and $\mathrm{true} :\equiv \mathrm{inr}()$.

3.4 Propositions as types: propositional logic

In the introduction I said that in Type Theory we explain the meaning of propositions by assigning to each proposition the type of reasons or proofs that we accept this proposition. This idea is associated with Haskell Curry (after whom the programming language Haskell is named) and William A. Howard and hence is called the Curry-Howard correspondence. We will look at this idea first for propositional logic which is the part of predicate logic which corresponds to simple types. That is what we get looking only at $\wedge$ (and), $\vee$ (or), $\neg$ (negation), $\rightarrow$ (if-then) and so on but ignoring for the moment predicates and relations and $\forall$ (for all) and $\exists$ (exists). Propositional logic is a bit strange while on the one hand it is simpler than predicate logic on the other hand it is a bit useless because we can't say anything interesting in Mathematics just using propositional logic. Instead we talk about arbitrary propositions similar as I talked about arbitrary types in the previous sections. Hence while propositional logic is a bit easier than predicate logic it always is a bit abstract. To overcome this I find it always useful to think about specific examples from everyday life, like *the sun shines* and *we go to the zoo* and so on but you shouldn't think that propositional logic is mainly about everyday reasoning.

What is the type of reasons to believe that *the sun shines* **and** *we go to the zoo*? We better have reasons to believe both. Writing P, Q for arbitrary propositions (but think about the two I just gave) we say that the reasons to believe in $P \wedge Q$ are pairs of reasons that is $P \times Q$, we identifying a proposition with the reasons to believe in it means also that we say $P \wedge Q \equiv P \times Q$. If you say *the sun shines* **or** *we go to the zoo* you either have a reason to believe in one or the other proposition and this corresponds to using coproducts: $P \vee Q \equiv P + Q$. I think most interesting is the translation of $\Rightarrow$: The reason to believe $P \Rightarrow Q$ is a function that transforms reasons for P in reasons for Q. Once I have such a function and you give me a reason for Q I just apply it and have a reason for Q. Hence we identify $P \Rightarrow Q \equiv P \rightarrow Q$.

How do we translate $\neg P$? $\neg P$ means that there is no reason to believe P, hence we can translate $\neg P$ as *If P then False*, that is $\neg P \equiv P \rightarrow \text{false}$. We don't really say false to mean impossible in natural language. Hence I suggest to read $P \rightarrow \text{false}$ as *If P then pigs have wings.* We translate false with the empty type, because there is no reason to believe that pigs have wings: false $\equiv 0$. Imagine a man asking a woman to marry her and she replies: *If I marry you then pigs have wings.* I would take this as a *no.*

There are many possible choices for true because any type with an element would do. The simplest choice is 1 the type with one element: true $\equiv 1$. Finally we translate logical equivalence $P \Leftrightarrow Q$ as implications in both directions $P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

How can we use this translation to see wether a proposition is a tautology? For example $P \wedge (Q \vee R)$ is logically equivalent to $(P \wedge Q) \vee (P \wedge R)$, this is similar to the law of distributivity in arithmetic $x \times (y + z) = x \times y + x \times z)$.

$$\begin{aligned}
\text{true} &:\equiv 1\\
P \wedge Q &:\equiv P \times Q\\
\text{false} &:\equiv 0\\
P \vee Q &:\equiv P + Q\\
P \Rightarrow Q &:\equiv P \to Q\\
\neg P &:\equiv P \Rightarrow \text{false}\\
&\equiv P \to 0\\
P \Leftrightarrow Q &:\equiv (P \Rightarrow Q) \wedge (Q \Rightarrow Q)\\
&\equiv (P \to Q) \times (Q \to P)
\end{aligned}$$

Figure 3.1: Propositions as types for propositional logic

We derive both directions separately:

$\mathrm{lr} : P \times (Q + R) \to (P \times Q) + (P \times R)$
$\mathrm{lr}\,(x, \mathrm{inl}\,y) :\equiv \mathrm{inl}\,(x, y)$
$\mathrm{lr}\,(x, \mathrm{inr}\,z) :\equiv \mathrm{inr}\,(x, z)$

$\mathrm{rl} : (P \times Q) + (P \times R) \to P \times (Q + R)$
$\mathrm{rl}\,(\mathrm{inl}(x, y)) :\equiv (x, \mathrm{inl}\,y)$
$\mathrm{rl}\,(\mathrm{inr}(x, z)) :\equiv (x, \mathrm{inr}\,y)$

$\mathrm{distr} : P \times (Q + R) \leftrightarrow (P \times Q) + (P \times R)$
$\mathrm{distr} \equiv (lr, rl)$

To avoid repeating formulas in the translation of $\Leftrightarrow$, I write $P \leftrightarrow Q$ for $(P \to Q) \times (Q \to P)$.

The reason to accept that $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$ is a tautology using propositions as types is quite different from the classical explanation which uses a truthtable:

P	Q	R	$P \wedge (Q \vee R)$	$(P \wedge Q) \vee (P \wedge R)$	$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
false	false	false	false	false	true
false	false	true	false	false	true
false	true	false	false	false	true
false	true	true	false	false	true
true	false	false	false	false	true
true	false	true	true	true	true
true	true	false	true	true	true
true	true	true	true	true	true

It is clear that either side is true if P and either Q or R are true. The truthtable for $\Leftrightarrow$ is very simple, it is true if both inputs agree., false otherwise. Hence we always obtain true, no matter what the inputs are: this is the definition of a tautology.

Here we have two different ways to observe that

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$

is a tautology: either by writing a program which shows us that we will believe it no matter what is the input and the other that it is true no matter what is the truth of the inputs.

However, this is not always the case. In section 2.4 I showed how in classical logic connectives can be defined from each other, so for example $P \wedge Q$ can be defined as $\neg(\neg P \vee \neg Q)$. One ingredient was the de Morgan formula $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q)$. However, in propositions as types only one direction is valid, namely $\neg P \vee \neg Q \Rightarrow \neg(P \wedge Q)$ which translates to

$$\mathrm{demorgan} : (P \to 0 + Q \to 0) \to ((P \times Q) \to 0)$$

The idea is to use reasoning by cases on the assumption $P \to 0 + Q \to 0$: in either case we can derive a contradiction if we assume $P \times Q$. As a program this looks like this:

$$\mathrm{demorgan}\,(\mathrm{inl}\,f)\,(x, y) :\equiv f\,x$$
$$\mathrm{demorgan}\,(\mathrm{inr}\,g)\,(x, y) :\equiv g\,y$$

However, there is no program for the other direction $((P \times Q) \to 0) \to (P \to 0 + Q \to 0)$. To define such a function we assume as input $x : (P \times Q) \to 0$ we need to construct an element of $P \to 0 + Q \to 0$, in particular we have to decide wether to use inl or inr. However, there is no information available to make that choice: all we know is that having both P and Q is inconsistent but not that one of them is and then which one?

However, when we draw the truthtable it is clear that $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q)$ is a tautology:

P	Q	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$	$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
false	false	true	true	true
false	true	true	true	true
true	false	true	true	true
true	true	false	false	true

In this case the propositions as types view and the classical truth based view diverges. It is interesting to note that other de Morgan formula $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$ is valid in propositions as types and in the classical view. The reason is that both formulas $\neg(P \vee Q)$ and $\neg P \wedge \neg Q$ are *negative*, that is they contain no information. hence the problem where we had to make our mind up for the de Morgan rule $\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q)$ doesn't arise.

The difference between the intuitionistic (just a another word for the propositions as types view) and the classical view can be brought down to a basic principle: the *law of excluded middle* which expresses that every proposition is either true or false, which can be translated to $P \vee \neg P$. The truth table for this one should be obvious:

P	$\neg P$	$P \vee \neg P$
false	true	true
true	false	true

On the other hand how would we prove $P + (P \to 0)$? Again we have to make up our mind wether to use inl or inr, but which one? P may in general be a proposition for which w simply don't know wether it is true or false. But even worse: we would expect a uniform solution which doesn't depend on P and this means we have to make up our mind in advance wether all propositions are true or not.

The principle of the excluded middle describes exactly the difference between classical and intuitionistic logic: once we assume it we can derive all classical tautologies. Maybe the following question comes to your mind: we cannot prove $P \vee \neg P$ maybe we can find a particular proposition Q for which we can prove $\neg(Q \vee \neg Q)$? This turns out is impossible and we can even do better and show that the negation holds for all propositions $\neg\neg(P \vee \neg P)$, that is we can show the that it is for any proposition P it is not the case that the excluded middle doesn't hold. This clearly implies that we cannot have a counterexample because if we had one we can put it together with the proof I am just about to present and derive a contradiction.

To convince you: applying our translation we need to come up with $\mathrm{nnem} : ((P + P \to 0) \to 0) \to 0$. Let's do this step by step: we assume $f : (P + P \to 0) \to 0$ and we want to derive an element $?_0 : 0$. The only way to construct an element of the empty type is by using our assumption $?_0 \equiv f\,?_1 : 0$ where $?_1 : P + P \to 0$. But now we have to make a choice we have to go left or right. How can we decide? Going left we would be stuck immediately hence let's see what happens if we go right: $?_1 :\equiv \mathrm{inr}\,?_2$ where $?_2 : P \to 0$. Now there is only one way to implement $?_2$ we introduce a function using λ: $?_2 :\equiv \lambda x.?_3$. Now $?_3 : 0$ – are we running in a circle? But note that we have a new assumption $x : P$ which we can use to define $?_3$. Again the only thing we can use is f, that is $?_3 :\equiv f\,?_4$ with $?_4 : P + P \to 0$. We went right last time, this time we go left: $?_4 :\equiv \mathrm{inl}\,?_5$ with $?_5 : P$. But this is easy because we can just set $?_5 :\equiv x$ and voila, we are done. To summarize:

$\mathrm{nnem}\,f :\equiv f\,(\mathrm{inr}\,(\lambda x.f\,(\mathrm{inl}\,x)))$

This proof can be illustrated by the following story which is a slight modification on one that I have heard from Peter Selinger. The story shows that double negation is the same as time travel:

> Once upon a time there was a poor king who had no gold. At this time it was unknown wether ipads existed and the King was keen

> to find out. He offered the hand of this daughter to anybody who could answer this question. Then a magician arrived in the court who seemed to have an answer: he promised the king that he would either get him an ipad or a machine that would turn an ipad into gold. In truth the magician didn't have an ipad and he certainly had no gold but he had a time machine. So once the King agreed, the magician said that he had indeed a machine which turns an ipad into gold which was the time machine with his friend sitting inside. Now the King was satisfied and the magician married the daughter. However, te next day an ipad appeared at court. As soon as the greedy King put the ipad into the machine, his friend took it and travelled back in time. Now the magician was able to fullfil his promise by presenting the ipad and he could still marry the daughter.

I leave it to you to figure out the relationship between the fairy tale and the proof of nnem.

There is an alternative to excluded middle when we want to add classical reasoning to propositions as types. This is based on the double negation operator we have just encountered: the principle of indirect proof says that to show P is is enough to show that P cannot be false, that is $\neg\neg P \to P$ Indeed, this is a very common situation in a classical proof where to prove P you assume $\neg P$ and try tp derive a contradiction. $\neg\neg P \to P$ as not provable using propositions as types which reflects the intuition that indirect evidence is not as good as direct evidence. E.g. to know that the key cannot be outside the house is not as good as actally having it. However, we can derive $\neg\neg P \to P$ from assuming excluded middle $h : P + P \to 0$ we assume $f : (P \to 0) \to 0$ now to show $? : P$ we do case analysis over h: either $h \equiv \mathrm{inl}\, x$ with $x : P$ and in this case we are done with $? \equiv x$ or we have $h \equiv \mathrm{inr}\, g$ with $g : P \to 0$ and in this case we can derive $f\, g : 0$ and now using that we can prove everything form false $\mathrm{efq} : 0 \to P$ we have $? \equiv \mathrm{efq}\,(f\, g)$ To summarize we define $\mathrm{em2ip} : (P + P \to 0) \to ((P \to 0) \to 0) \to P$ as follows

$\mathrm{em2ip}\, f\, h :\equiv \mathrm{case}\,(\lambda x.x)\,(\lambda g.\mathrm{efq}\,(\,f\, g))$

This also works the other way around but not if we keep the proposition P the same. To prove $P \vee \neg P$ from indirect proof we use the fact that we have just shown *nnem* : $\neg\neg(\neg P \vee P)$ for any P and hence we only have to apply indirect proof for $\neg P \vee P$ to derive it. Hence the two principles: excluded middle and indirect proof are equivalent and both of them enable us to prove all classical tautologies. However, we don't really want to assume either of them globally but sometimes we may be able to prove them in specific instances.

3.5 Counting for dummies

The natural numbers are based on the basic intuition of counting. As a computer scientist I prefer to start counting with 0 this is also useful as the answer to the

Exercises for Naive Type Theory
Part 2

Thorsten Altenkirch

April 12, 2016

1. Define

   $\mathrm{swap}_{A,B} : A \times B \to B \times A$

   using only

   $\mathrm{uncurry} : (X \to Y \to Z) \to (X \times Y \to Z)$

2. Using if-then-else

   $$\begin{aligned} \mathrm{ifThenElse} &: \mathrm{Bool} \to A \to A \\ \mathrm{ifThenElse}\,\mathrm{true}\,a\,b &:\equiv a \\ \mathrm{ifThenElse}\,\mathrm{false}\,a\,b &:\equiv b \end{aligned}$$

   define

   $f : (\mathrm{Bool} \to A) \to A \times A$
   $g : A \times A \to (\mathrm{Bool} \to A)$

   which are inverse to each other (you don't need to prove this).

3. Using the propositions as types translation, try to verify the de Morgan-laws:

   $\neg(P \land Q) \Leftrightarrow \neg P \lor \neg Q$
   $\neg(P \lor Q) \Leftrightarrow \neg P \land \neg Q$

   Which one(s) are provable? And which one(s) are partially provable (one direction of $\Leftrightarrow$)?

4. Show that the principle of exclude middle, that is for all propositions P

   $P \lor \neg P$

   is equivalent to the principle of indirect proof, that is for all Q

   $\neg\neg Q \implies Q$

Chapter 4

Dependent types

Now we come to the heart of the matter: dependent types. This was the main insight of Per Martin-Löf when he started to develop Type Theory in 1972. Per knew about the propositions as type equivalence or Curry-Howard equivalence and indeed just before he published his first paper on Type Theory he had visited Howard in Chicago. This type-theoretic interpretation of logic turned out to be more than just a curiosity, it is possible to do all mathematical constructions in Type Theory once one adds this essential ingredient: dependent types.

4.1 The power of Π and Σ

The basic idea of dependent types is quite easy: we have seen lists in the last chapter as sequences of arbitrary length. Now we want to refine this and instead consider sequences of a fixed length: we write $\mathrm{Vec}\,A\,n$ for the type of sequence of elements of the type A of length $n : \mathbb{N}$ which we call vectors. So for example $[1,2,3] : \mathrm{Vec}\,\mathbb{N}\,3$. This is a dependent type because $\mathrm{Vec}\,A\,n$ *depends* on an a natural number $n : \mathbb{N}$. Once we have dependent types we need to look at dependent versions of function types and products: What is the type of a function that has a natural number $n : \mathbb{N}$ as the input and which produces a vector of numbers of this length as output? We write $\Pi n : \mathbb{N}.\mathrm{Vec}\,\mathbb{N}\,n$, the Π is the capital greek letter *pi* and we call them Π-types. An example is the function $\mathrm{zeros} : \Pi n : \mathbb{N}.\mathrm{Vec}\,\mathbb{N}\,n$ which produces a sequence of zeros of a given length, e.g. $\mathrm{zeros}\,3 \equiv [0,0,0]$. Here is the primitive recursive definition of zeros:

$$\begin{aligned}\mathrm{zeros}\,0 &:\equiv []\\ \mathrm{zeros}\,(1+n) &:\equiv 0 :: (\mathrm{zeros}\,n)\end{aligned}$$

Another example would be the type of pairs where the first component is a natural number $n : \mathbb{N}$ and the second component is a vector of natural numbers of of that length $v : \mathrm{Vec}\,\mathbb{N}\,n$. This we express as $\Sigma n : \mathbb{N}.\mathrm{Vec}\,\mathbb{N}\,n$, the Σ is the capital greek letter *sigma* and such a type is called a Σ-type. For example $[1,2,3] : \Sigma n : \mathbb{N}.\mathrm{Vec}\,\mathbb{N}\,n$. It is interesting to notice that $\Sigma n : \mathbb{N}.\mathrm{Vec}\,A\,n$ is actually equivalent to $\mathrm{List}\,A$ because we can represent sequences of arbitrary length.

While dependent types were originally invented for mathematical constructions and they enable us to extend the propositions as types equivalence to predicate logic, they are also extremely useful in programming. As an example consider the problem of accessing an arbitrary element of a sequence that is given $l : \mathrm{List}\,A$ and a number $n : \mathbb{N}$ we want to extract the nth element of the list $\mathrm{nth}\,l\,n$, e.g. $\mathrm{nth}\,[1,2,3]\,2 \equiv 3$ (we start counting at 0). I hope that you immediately realize that there is a problem because it could be that n is out of range as in $\mathrm{nth}\,[1,2,3]\,3$, moreover it could be that A is empty hence we cannot return a default element which would be bad programming style anyway. As before for the predecessor we have to adjust the type of nth to allow for an error:

$$\mathrm{nth} : \mathbb{N} \to \mathrm{List}\,A \to 1 + A$$

The idea is that $\mathrm{nth}\,[1,2,3]\,2 \equiv \mathrm{inr}\,3$ while $\mathrm{nth}\,[1,2,3]\,3 \equiv \mathrm{inr}\,()$ indicating a runtime error. Here is the definition of nth:

$$\begin{aligned}
\mathrm{nth}\,[]\,n &:\equiv \mathrm{inl}\,() \\
\mathrm{nth}\,(a :: l)\,0 &:\equiv \mathrm{inr}\,a \\
\mathrm{nth}\,(a :: l)\,(1+n) &:\equiv \mathrm{nth}\,l\,n
\end{aligned}$$

I leave it to the concerned reader to translate this program into one only using fold and iter.

The introduction of the explicit runtime-error may appear invonvenient but using a well-known technique this can be hidden - I already mentioned Haskell's notation for monadic programs which can be applied in this case. However, we may have a carefully constructed program which we know will only use nth safely. We still have to handle the impossible error at some point the programmer will write a comment *impossible* at this bit of code and maybe interrupt the excution assuming that this will never happen. However, if she made a mistake very bad things can happen form crashing planes to exploding nuclear power stations.

Dependent types offer a more satisfying solution to this problem: we use a dependent type $\mathrm{Fin}\,n$ where $n : \mathbb{N}$, which contains exactly n-elements, which we write $0, 1, \ldots, n-1 : \mathrm{Fin}\,n$. Now we can give a more precise type

$$\mathrm{nth} : \Pi_{n:\mathbb{N}} \mathrm{Vec}\,A\,n \to \mathrm{Fin}\,n \to A$$

The need for an error disappears because the elements of $\mathrm{Fin}\,n$ correspond exactly to the positions in $\mathrm{Vec}\,A\,n$. Before examining the construction of the dependently typed nth in detail I need to explain some notational conventions: I have put the $n : \mathbb{N}$ in $\Pi_{n:\mathbb{N}} \ldots$ in subscript because it can be inferred from the context and I will usually not write it. That is I will just write $\mathrm{nth}\,[1,2,3]\,2$ as I orginally intended which is short for $\mathrm{nth}_3\,[1,2,3]\,2$, where $[1,2,3] : \mathrm{Vec}\,\mathbb{N}\,3$ and $2 : \mathrm{Fin}\,3$. If I want to make a hidden argument explicit I write at as a subscript as in nth_3. I do this to avoid clutter wich very quickly makes dependently typed programs unreadable. Note that there is no problem with $\mathrm{nth}\,[1,2,3]\,3$ because it cannot be well typed. Either we choose $\mathrm{nth}_3\,[1,2,3]\,3$ which is not well typed because 3 is not an element of $\mathrm{Fin}\,3$ or $\mathrm{nth}_4\,[1,2,3]\,3$ which is not well typed because $[1,2,3]$ is not an element of $\mathrm{Vec}\,\mathbb{N}\,4$.

How do we define the dependently typed nth? Let's start with Fin: as we have constructed $\mathbb{N}$ from 0 and suc we can only construct Fin this way using

$$\begin{aligned} 0 &: \Pi_{n:\mathbb{N}}\mathrm{Fin}\,1+n \\ \mathrm{suc} &: \Pi_{n:\mathbb{N}}\mathrm{Fin}\,n \to \mathrm{Fin}\,(1+n) \end{aligned}$$

Maybe the following table helps to understand the definition of Fin:

n	$\mathrm{Fin}\,n$
0	
1	0_0
2	$0_1, \mathrm{suc}_1\,0_0$
3	$0_2, \mathrm{suc}_2\,0_1, \mathrm{suc}_2\,(\mathrm{suc}_1\,0_0)$
$\vdots$	$\vdots$

That is $\mathrm{Fin}\,(1+n)$ constains a new element 0_n and all the elements $i : \mathrm{Fin}\,\mathrm{n}$ with a suc_n in front of them : $\mathrm{suc}_n\,i$. But this is exactly what the types of the constructor express, isn't it?

So far I have relied on an intuitive understanding of Vec but we can do better and play a similar game as we have just played with changing $\mathbb{N}$ into Fin be coming up with dependent types for the constructors 0 and suc. We can do the same with $[]$ and $- :: -$ and declare:

$$\begin{aligned} [] &: \mathrm{Vec}\,A\,0 \\ - :: - &: \Pi_{n:\mathbb{N}}A \to \mathrm{Vec}\,A\,n \to \mathrm{Vec}\,A\,(1+n) \end{aligned}$$

As an example we can construct $1 ::_2 2 ::_1 3 ::_0 [] : \mathrm{Vec}\,A\,3$.

We now have all the ingredients in place to define the dependently typed

$$\mathrm{nth} : \Pi_{n:\mathbb{N}}\mathrm{Vec}\,A\,n \to \mathrm{Fin}\,n \to A$$

$$\begin{aligned} \mathrm{nth}\,(a :: v)\,0 &:\equiv a \\ \mathrm{nth}\,(a :: v)\,(1+n) &:\equiv \mathrm{nth}\,v\,n \end{aligned}$$

We haven't given a definition for $\mathrm{nth}_0\,[]$ because this case is impossible - there is no element of $\mathrm{Fin}\,0$. In an actual implementation of Type Theory like Agda this is explicitly indicated by writing

$$\mathrm{nth}\,[]\,i()$$

I am going to introduce the dependent version of fold later and then we will revisit this definition to see that I am not cheating and we can actually construct nth using dependent primitive recursion.

The idea behind definitions like nth is that we can avoid run-time errors by introducing more precise types for our programs. Thus replacing the simply typed nth which used the error monad with the dependently typed one enables us to guarantee that there is no error when accessing a list statically. However, this benefit doesn't come for free: in general we have to work harder to construct a dependently typed program.

The name of this section is inspired by a nice paper by Wouter Swierstra and Nicolas Oury [**?**] where they give some interesting programming applications of dependent types. [1]

4.2 Type Arithmetic

Function types are a special case of Π-types and cartesian products are a special case of Σ-types. To be be precise, given types A, B we have

$$A \to B \equiv \Pi x : A.B$$
$$A \times B \equiv \Sigma x : A.B$$

Why do we use Π and Σ? In Mathematics Σ is used for sums. For example the sum of the numbers from 1 until 5 we can write as $\Sigma_{i=1}^{5} i$ which just stands for

$$1 + 2 + 3 + 4 + 5 \equiv 15$$

Π is used for products, for example the product of the numbers from 1 unil 5 (that is 5!, the number of ways to place 5 people on 5 chairs) can be written as $\Pi_{i=1}^{5} i$ which just stands for

$$1 \times 2 \times 3 \times 4 \times 5 \equiv 120$$

In the previous chapter we have already observed that for finite sets the type-theoretic operations coincide with the arithmetical ones, we can express this more succinctly use $\mathrm{Fin} : \mathbb{N} \to \mathbf{Type}$:

$$\mathrm{Fin}\, m + \mathrm{Fin}\, n = \mathrm{Fin}\,(m + n)$$
$$\mathrm{Fin}\, m \times \mathrm{Fin}\, n = \mathrm{Fin}\,(m \times n)$$

The = here stands for equivalence of types, that means here *having the same number of elements*. This notation is actually justified in Homotopy Type Theory where equality of types is equivalence - we will discuss this in more detail in the next chapter.

This can be extended to function types using exponentation:

$$\mathrm{Fin}\, m \to \mathrm{Fin}\, n = \mathrm{Fin}\,(n^m)$$

[1] I am not impartial: Wouter was my PhD student, and Nicolas was a Marie-Curie fellow in my group when they wrote this paper.

For example how many functions of type $\mathrm{Fin}\,3 \to \mathrm{Fin}\,4$ are there? For each of the 3 inputs we have a choice of 4 answers, hence there are $4 \times 4 \times 4 \equiv 4^3 \equiv 64$. In Mathematics people often use exponetial notation to write function types, that is instead of $\mathbb{N} \to \mathrm{Bool}$ they write $\mathrm{Bool}^{\mathbb{N}}$.

This analogy extends to Π and Σ. We can translate the expression $\Sigma_{i=1}^{5}\, i$ into a type

$$\Sigma i : \mathrm{Fin}\,4.\mathrm{Fin}\,(1+i)$$

Hang on this doesn't type check because Fin expects a natural number but $1 + i : \mathrm{Fin}\,4$. However, there is a function $\mathrm{fin2nat} : \Pi_{n:\mathbb{N}}\mathrm{Fin}\,n \to \mathbb{N}$ which maps every element of a finite type to a corresponding natural number. Hence the correct version is

$$\Sigma i : \mathrm{Fin}\,4.\mathrm{Fin}\,(\mathrm{fin2nat}\,(1+i))$$

These coercions are very common and I often prefer not to write them. [2] As the arithmetical Σ corresponds to iterated addition, the type-theoretic Σ corresponds to iterated coproduct

$$\begin{aligned}&\Sigma i : \mathrm{Fin}\,4.\mathrm{Fin}\,(1+i)\\ &= \mathrm{Fin}\,1 + (\mathrm{Fin}\,2 + (\mathrm{Fin}\,3 + (\mathrm{Fin}\,4 + (\mathrm{Fin}\,5))))\\ &= \mathrm{Fin}\,15\end{aligned}$$

Here I write $=$ instead of $\equiv$ because this is not the definition of Σ. This seems to be add odds with my explanation of Σ as the type of dependent pairs, namely an element of $\Sigma i : \mathrm{Fin}\,4.\mathrm{Fin}\,(1+i)$ is of the form (i, j) where $i : \mathrm{Fin}\,10$ and $j : \mathrm{Fin}\,(i+1)$. Luckily this matches the iterated sum interpretation because the i selects the summand and the j the element of the summand. So for example $(2,1)$ is translated to $\mathrm{inr}, (\mathrm{inr}\,(\mathrm{inl}\,1)))$.

The same analogy applies to Π: As the arithmetical Π corresponds to iterated multiplication, the type-theoretic Π corresponds to iterated products

$$\begin{aligned}&\Pi i : \mathrm{Fin}\,4.\mathrm{Fin}\,(1+i)\\ &= \mathrm{Fin}\,1 \times (\mathrm{Fin}\,2 \times (\mathrm{Fin}\,3 \times (\mathrm{Fin}\,4 \times (\mathrm{Fin}\,5))))\\ &= \mathrm{Fin}\,120\end{aligned}$$

And again this fits with the view of Π-types as dependent functions, applying a function $f : \Pi i : \mathrm{Fin}\,4.\mathrm{Fin}\,(1+i)$ to an input $2 : \mathrm{Fin}\,4$ leading to $f\,2 : \mathrm{Fin}\,3$ correspnds to applying as many second projections to the nested tuple view of f, i.e. $\pi_0\,(\pi_1\,(\pi_1\,f))$.

We can recover the binary products and coproducts from Π and Σ by using Bool as indexing type:

$$\begin{aligned}A + B &\equiv \Sigma b : \mathrm{Bool}.\mathrm{if}\,b\,\texttt{then}\,A\mathrm{else}\,B\\ A \times B &\equiv \Pi b : \mathrm{Bool}.\mathrm{if}\,b\,\texttt{then}\,A\mathrm{else}\,B\end{aligned}$$

[2] This sort of convention is supported in Coq using canonical structures.

Did you notice that $\times$ actually appears twice: we can either view $A \times B$ as a non-dependent Σ-type ($\Sigma x : B.B$) or as a Bool-indixed Π-type. This dual nature of products indeed leads to slightly different approaches to products: either based on tupling or based on projections.

To summarize these observations we can say that we can classify the operaors in 2 different ways: either along depdnent / non-dpendent or along binary or iterated:

non-dep.	dep.
$\times$	Σ
$\to$	Π

binary	iterated
$+$	Σ
$\times$	Π

This can lead to some linguistic confusions: what is a dependent product type? Hence I prefer just to say Π-type and Σ-type.

4.3 One Universe is not enough

So far we have avoided to consider types as explicit objects but instead they were part of our prose. Remember, I was saying: *given a type A we define the type of lists* List A. But what is List if not a function on types, that is List : **Type** $\to$ **Type**. Here we introduce **Type** as a type of types and this sort of thing is called a *universe* in Type Theory. So for example $\mathbb{N}$: **Type** but also $3 : \mathbb{N}$ hence $\mathbb{N}$ can appear on either side of the colon. A universe is a type whose elements are types again. This also works for Fin : $\mathbb{N} \to$ **Type** and Vec : $\mathbb{N} \to$ **Type** $\to$ **Type**. We call Fin and Vec families of types or dependent types and we should also call List a dependent type but the terminology is a bit vague here and some people would call it a *type indexed type*.

The reason is that functional programming languages support types like List but not *properly dependent types* like Fin or Vec. This has mainly to do with pragmatic reasons, namely that programming language implementers don't want to evaluate programs at compile time but they would have to do check that applying a function $f : \mathrm{Fin}\,7 \to \mathbb{N}$ to $i : \mathrm{Fin}\,(3+4)$ as in $f\,i$ is well typed. In this case this is easy and just involves checking that $7 \equiv 3 + 4$ but there are no limits what functions we could use here. In the case of a language like Haskell this could even involve a function that loops which would send the compiler into a loop. However, implementations of Type Theory like Agda or Coq aren't really so much better, at least from a pragmatic point of view, since we may use the Ackermann function here which may mean that while the function will eventually terminate this may only happen after the heat death of the universe.

The obvious question is what is the type of **Type**? Is it **Type** : **Type**. Hang on wasn't this the sort of thing where Frege got in trouble'? Does Russell's paradox applies here? The type of all type which does not contain itself? Not immediately because : isn't a proposition hence we cannot play this sort of trick here. Maybe this was what Per Martin-Löf was thinking when he wrote his first paper about Type Theory which did include **Type** : **Type**. This time it

Just using were elementary reasoning we can show that

$\mathrm{el}\, R\, R \leftrightarrow (\mathrm{el}\, R\, R) \to 0$

And it follows from propositional logic that $\neg(P \Leftrightarrow \neg P)$ from simple reasoning in propositional logic. But this means that the empty type in inhabited.

Where did we actually use **Type** : **Type**? We didn't explicitly but implicitly when we introduced a constructor makeTree whose argument is a type again.

So, if **Type** : **Type** doesn't work what is now the type of **Type**? To avoid the problem we introduce not one namely infinitely many universes.

$\mathbf{Type}_0 : \mathbf{Type}_1 : \mathbf{Type}_2 : \ldots$

Hence the answer depends on the level, if we ask about the first universe $\mathbf{Type}_0$ then its type is $\mathbf{Type}_1$ and what is the type of $\mathbf{Type}_1$? Right is is $\mathbf{Type}_2$. And so on.

The numbers $0, 1, 2, \ldots$ which as the index of types are not our natural numbers (even though the look very much like them) because otherwise we would have to make our mind up was is the type of a function that assigns to a number a type and what is the type of this function?

All our usual garden variety types like $\mathbb{N}, \mathrm{Bool}, \mathrm{List}\,\mathbb{N}, \mathbb{N} \to \mathrm{Bool}, \mathrm{InfTree}, \ldots$ are elements of $\mathbf{Type}_0$, but they are also elements of all higher universes - this is called cummulativity. The thing that is new in $\mathbf{Type}_1$ is $\mathbf{Type}_0$ itself. We say that $\mathbf{Type}_0$ is larger then all the types in $\mathbf{Type}_0$ because if there would be a type in $\mathbf{Type}_0$ which is equivalent to $\mathbf{Type}_0$ we can derive the paradox. There are more new types in $\mathbf{Type}_1$ for example $\mathrm{List}\,\mathbf{Type}_0$ or $\mathbf{Type}_0 \to \mathbb{N}$. This story continues: the new types in $\mathbf{Type}_2$ are $\mathbf{Type}_1$ and all the types which can be built from it.

While this solves the problem with **Type** : **Type** it does introduce a lot of bureaucracy: for example we have many copies of List namely $\mathrm{List}_0 : \mathbf{Type}_0 \to \mathbf{Type}_0$ and $\mathrm{List}_1 : \mathbf{Type}_1 \to \mathbf{Type}_1$ etc. This is not covered by cummulativity which only tells us that $\mathrm{List}_0\,\mathbb{N} : \mathbf{Type}_1$. What is the best way to deal with this problem is a research question: while there are a number of candidates and implementations there is no general agreement what is the best way. On paper we can be lazy and adopt the convention that we just pretend that we had **Type** : **Type** but then check that there is a consistent assignment of indices to each occurence of **Type**. This is close to what the Coq system is doing but there are cases where the inference algorithm is too weak.

4.4 Propositions as Types: Predicate logic

I am now going to deliver on the promise to extend the propositions as types explanation to predicate logic using dependent types. Actually it is rather straightforward and I couldn't avoid giving away the secret in the last section - maybe you noticed. We translate $\forall$ with Π-types. That is for example the statement [3] that

$\forall x : \mathbb{N}.x + 0 = x$

is translated into a type

$\Pi x : \mathbb{N}.x + 0 = x$

That is a reason is a function that assigns to any natural number $n : \mathbb{N}$ a reason that $x + 0 = x$. I haven't yet explained what equality is as a type but I will do this soon for the moment I will appeal on some basic intuition about equality. How would an element of this type look like? I am defining $f : \Pi x : \mathbb{N}.x + 0 = x$ using primitive recursion:

$$\begin{aligned} f\,0 &:\equiv \mathrm{refl}\,0 \\ f\,(\mathrm{suc}\,n) &:\equiv \mathrm{resp}\,\mathrm{suc}\,(f\,n) \end{aligned}$$

Some explanation is required I am using $\mathrm{refl} : \Pi n : \mathbb{N}.n = n$ as the proof of reflexivity and

$\mathrm{resp} : \Pi_{A,B:\mathbf{Type}}\Pi f : A \to B.\Pi_{a,a':A} a = a' \to f\,a \to f\,a'$

as a proof that any function respects equality. That is in particular

$\mathrm{resp}\,\mathrm{suc} : \Pi m, n : \mathbb{N}.m \equiv n \to \mathrm{suc}\,m = \mathrm{suc}\,n$

What does this function actually do? What is for example $f\,5$? Now that is easy $f\,5 \equiv \mathrm{refl}\,5$ since $5 + 0 \equiv 5$ and hence $\mathrm{refl}\,5 : 5 + 0 \equiv 5$. So f is just a constant function returning refl! Could we not have just written

$f\,n :\equiv \mathrm{refl}\,n$?

No, because $n + 0$ is not by definition the same as n, and hence this definition of the function doesn't obviously typecheck. The more complicated version above does, but this requires some thought: the result of $\mathrm{resp}\,\mathrm{suc}\,(f\,n)$ has the type $\mathrm{suc}\,(n + 0) = \mathrm{suc}\,n$ but we needed an element of $(\mathrm{suc}\,n + 0 = \mathrm{suc}\,n)$ In this case however, we can exploit the definition of $+$ which tells us that $(\mathrm{suc}\,n) + m \equiv \mathrm{suc}\,(n + m)$. The definition of f uses primitive recursion for dependent types which generalized iter, but really this is a proof by induction. We will look into this relationship soon.

Similarly we translate $\exists$ with Σ-type. For example we translate

$\exists x : \mathbb{N}.x = x + x$

into

$\Sigma x : \mathbb{N}.x = x + x$

[3] To be consistent I am using here predicate logic with types whereas previously we assumed that there always is only one type we talk about.

That is a reason for an existential statement is a pair: the first component is the actual element we use (sometimes called the witness) and the 2nd component is the reason that this element satisfies the predicate. In this case it is easy to write down a proof:

$$(0, \mathrm{refl}\,0) : \Sigma x : \mathbb{N}.x = x + x$$

Here I am using that $\mathrm{refl}\,0 : 0 = 0 + 0$ since $0 + n \equiv n$ because that is the way we have defined $+$.

What is a predicate? Predicates are properties, since we now work with types we can only define properties of elements of a given type. For example Even is the property of a natural number to be even. So given a natural number $n : \mathbb{N}$ we can construct $Even\, n$ which is a proposition, that is using propositions as types, a type. Hence even is a function from natural numbers to types:

$$\mathrm{Even} : \mathbb{N} \to \mathbf{Type}$$

That s using the terminology from the previous sections: Even is a dependent type. Indeed, we need depndent types to extend propositions as types to predicate logic, we use dependent types to interpret predicates. This also works for relations using currying for any given type $A : \mathbf{Type}$ we have $- =_A - : A \to A \to \mathbf{Type}$ the equality relation. Or more generally $- = - : \Pi_{A:\mathbf{Type}} A \to A \to \mathbf{Type}$ since equality works polymorphically for any type.

Using Σ-types we can make sense of the comprehension notation of set theory. If we have a type $A : \mathbf{Type}$ and a predicate P over A that is $P : A \to \mathbf{Type}$ we can represent $\{x : A \mid P\,x\}$ as $\Sigma x : A.P\,x$ so an element is an element of A together with a proof that it satsifies the predicate. For example

$$\{x : \mathbb{N} \mid \mathrm{Even}\,x\} \equiv \Sigma x : \mathbb{N}.\mathrm{Even}\,x$$

represents the type of even numbers as pairs of a number and a proof that this number is even. However, there can be a mismatch here: if we have two ways to prove that for example 2 is even then there are two different versions of 2 in $\{x : \mathbb{N} \mid \mathrm{Even}\,x\}$. This is undesirable and we should demand that there is at most one proof that a certain number is even - in this particular instance this is not difficult to achieve.

Earlier I promised that we can make sense of $\cap$ and $\cap$ and other operations from set theory. As explained earlier they are not operation on types but they can be understood as operations on predicates. That is given a type $A : \mathbf{Type}$ this is often called a universe in set theory but should not be confused with type theoretic universes. Now given two predicates $P, Q : A \to \mathbf{Type}$ we can define new predicates:

$$P \cap Q, P \cap Q : A \to \mathbf{Type}$$
$$P \cap Q :\equiv \lambda x.P\,x \wedge Q\,x$$
$$P \cup Q :\equiv \lambda x.P\,x \vee Q\,x$$

We can also make sense of $P \subseteq Q$ which is a proposition interpreted as a type:

$$P \subseteq Q :\equiv \Pi x : A.P\, x \to Q\, x$$

This interpretation of the operations covers most of matehmatical practice where $\cup, \cap, \subseteq$ are applied to subsets, i.e. predicates, of a given set.

We can use the interpretation of $\forall$ and $\exists$ to verify tautologies of predicate logic. Maybe you remember the slightly surprising tautology

$$(\forall x.\Phi(x) \Rightarrow p) \Leftrightarrow (\exists x.\Phi(x) \Rightarrow p)$$

from section 2.4 which I illustrated using student performance and lecturer happiness. Instead of this sketch of a logical justification we can now construct functions which show us that the reasons can be translated both ways. Let's fix a type $A : \mathbf{Type}$ the statement becomes:

$$(\forall x : A.Q\, x \Rightarrow P) \Leftrightarrow (\exists x.Q\, x \Rightarrow P)$$

where Q is a predicate over A that is $Q : A \to \mathbf{Type}$ and P is a proposition which we represent as a type $P : \mathbf{Type}$. Propositions as types here means we replace the symbols:

$$(\Pi x : A.Q\, x \Rightarrow P) \Leftrightarrow ((\Sigma x : A.Q\, x) \Rightarrow P)$$

that means we need to construct 2 functions:

$$\begin{aligned} &f : (\Pi x : A.Q\, x \Rightarrow P) \to ((\Sigma x : A.Q\, x) \Rightarrow P) \\ &f\, h :\equiv \lambda y.h\, (\pi_0\, y, \pi_1\, y) \end{aligned}$$

and for the other direction:

$$\begin{aligned} &g : ((\Sigma x : A.Q\, x) \Rightarrow P) \to (\Pi x : A.Q\, x \Rightarrow P) \\ &g\, k :\equiv \lambda x, q.k\, (x, q) \end{aligned}$$

If you have a dejavu when looking at this definitions you are right. In section 3.2 we defined the functions curry and uncurry:

$$\begin{aligned} \text{curry} &: (A \times B \to C) \to (A \to B \to C) \\ \text{uncurry} &: (A \to B \to C) \to (A \times B \to C) \end{aligned}$$

and their definitions exactly match g and f. This should be no surprise: all we have done is to replace the product $\times$ by a Σ-type and the function $\to$ by a Π-type. That is we replaced the non-dependent version of an operation on types by the dependent one. And now we can use this to justify a tautology from predicate logic.

We can use propositions as types to extend simple Type Theory from the previous chapter by reasoning using predicate logic. Indeed, this is the way the Coq system is often used in practice. It seems to me that it is a shame using dependent types only in this fashion but indeed a much tighter integration is often possible and desirable. As an example let's verify that the equality of natural numbers is decidable. As a first step we could implement a decision function $\mathrm{eq} : \mathbb{N} \to \mathbb{N} \to \mathrm{Bool}$ which can be defined as follows:

$$\begin{aligned}
\mathrm{eq}\,0\,0 &:\equiv \mathrm{true}\\
\mathrm{eq}\,0\,(\mathrm{suc}\,n) &:\equiv \mathrm{false}\\
\mathrm{eq}\,(\mathrm{suc}\,m)\,0 &:\equiv \mathrm{false}\\
\mathrm{eq}\,(\mathrm{suc}\,m)\,(\mathrm{suc}\,n) &:\equiv \mathrm{eq}\,m\,n
\end{aligned}$$

The strategy is that we look at all combinations of successor and zero that are possible: if both are 0 then there are equal; if one is zero and the other is a successor then they are obviously not equal and if both are successor we recursively compare the predeccessors. While this is obviously correct we should be paranoid and demand a proof anyway, which could be given by proving

$$\forall m, n : \mathbb{N}.\mathrm{eq}\,m\,n \Leftrightarrow m = n$$

using the propositions as types translation. However, there is a nicer way using dependent types directly. The type of eq above wasn't very informative: it only told us that we have a function from two natural numbers to bool. Using dependent types we can be more explicit and replace bool by a more informative type. That is for any natural numbers $m, n : \mathbb{N}$ we want to prove $(m = n) \vee \neg(m = n)$. Indeed, this is an instance of the law of excluded middle, which doesn't hold in general but it does in this case, witnessing decidability. That is replacing $\vee$ with $+$ and $\neg$ with $\to 0$ we get

$$\mathrm{eq} : \Pi m, n : \mathbb{N}.(m = n) + (m = n) \to 0$$

If we can prove an instance of the law of the exclude middle we say that the coresponding proposition has beed decided, that is we have established wether it is true or false. If we have a predicate we say that this predicate is decidable if we can decide all its instances, and similar for relations. To avoid writing long and unreadable formulas below it is useful to define

$$\begin{aligned}
&\mathrm{Dec} : \mathbf{Type} \to \mathbf{Type}\\
&\mathrm{Dec}\,P :\equiv P + (P \to 0)
\end{aligned}$$

where we read Dec as decided. Then the type of eq becomes

$$\mathrm{eq} : \Pi m, n : \mathbb{N}.\mathrm{Dec}\,(m = n)$$

Now let me sketch the definition of the depdnently typed eq only leaving out the bits where I need to use equality reasoning which I haven't yet introduced.

The general structure resembles the previous definition with Bool:

$$\begin{aligned}
\mathrm{eq}\,0\,0 &:\equiv \mathrm{inl}\,(\mathrm{refl}\,0)\\
\mathrm{eq}\,0\,(\mathrm{suc}\,n) &:\equiv \mathrm{inr}\,\mathrm{noConf}\,n\\
\mathrm{eq}\,(\mathrm{suc}\,m)\,0 &:\equiv \mathrm{inr}\,(\lambda p.\mathrm{noConf}\,m\,(\mathrm{sym}\,p))\\
\mathrm{eq}\,(\mathrm{suc}\,m)\,(\mathrm{suc}\,n) &: \mathrm{eqAux}\,(\mathrm{eq}\,m\,n)
\end{aligned}$$

where we use

$$\begin{aligned}
\mathrm{noConf} &: \Pi n : \mathbb{N}.(0 = \mathrm{suc}\,n) \to 0\\
\mathrm{sym} &: \Pi_{A:Type}\Pi_{a,b:A} a = b \to b = a\\
\mathrm{eqAux} &: \Pi_{m,n:\mathbb{N}}\mathrm{Dec}\,(m = n) \to \mathrm{Dec}\,(\mathrm{suc}\,m)\,(\mathrm{suc}\,n)
\end{aligned}$$

noConf stands for *no confusion* and corresponds to the principle in Peano Arithmetic that 0 is not equal to any successor. In the second case I need the same inequality but the other way around which can be derived using a proof sym that equality is symmetric. Here is the definition of eqAux using yet another auxilliary function:

$$\begin{aligned}
\mathrm{eqAux}\,(\mathrm{inl}\,p) &:\equiv \mathrm{inl}\,(\mathrm{resp}\,\mathrm{suc}\,p)\\
\mathrm{eqAux}\,(\mathrm{inr}\,h) &:\equiv \mathrm{inr}\,(\lambda q.h\,(\mathrm{injSuc}\,q)
\end{aligned}$$

What is happening? eqAux preserves the injection becuase clearly the equality of m and n is the same as the equality of $\mathrm{suc}\,m$ and suc, n which is why we didn't need to do anything in the boolean version. However, we have to massage the reason: if $m = n$ then $\mathrm{suc}\,m = \mathrm{suc}\,n$ we have seen this before we can prove it using resp suc. In the onther direction we have to show that $m \neq n$ implies $\mathrm{suc}\,m \neq \mathrm{suc}\,n$. That means from assuming $h : m = n \to 0$ we have to derive $\mathrm{suc} m = \mathrm{suc} n \to 0$, that is given $p : \mathrm{suc}\,m = \mathrm{suc}\,n$ we need to derive a hypothetical element of the empty type. Are only chance is to use h and to close the gap we need

$$\mathrm{injSuc} : \Pi_{m,n:\mathbb{N}}\mathrm{suc}\,m = \mathrm{suc}\,n \to m = n$$

injSuc thans for injectivity of successor and is another principle from Peano Arithmetic if the successors of two numbers are equal then the numbers are equal. Unlike resp this doesn't hold for all functions but it does hold for successor.

I am not sure I have convinced you that this is a nicer way to define this function. The nice thing is that this version of the function gives us both the computation and the correctness proofs. As Conor McBride expressed it: *It says on the tin what is inside the tin.* That is the type tells us everything we want to know about the function: it decides equality. Certainly this refined version of the function is technically more work but then I didn't actually spell out the details of the correctness proof of the boolean version of eq.

Exercises for Naive Type Theory
Part 3

Thorsten Altenkirch

April 14, 2016

1. Given a type A and $n : \mathbb{N}$ we define the type $\mathrm{Vec}\, A\, n$ as given by the constructors:

$$\begin{aligned} [\,] &: \mathrm{Vec}\, A\, 0 \\ - :: - &: \Pi_{n:\mathbb{N}} A \to \mathrm{Vec}\, A\, n \to \mathrm{Vec}\, A\, (1+n) \end{aligned}$$

   We define lists over A the following way:

   $\mathrm{List}\, A :\equiv \Sigma n : \mathbb{N}. \mathrm{Vec}\, A\, n$

   Given this derive the following operations on lists:

$$\begin{aligned} \mathrm{nil} &: \mathrm{List}\, A \\ \mathrm{cons} &: A \to \mathrm{List}\, A \to \mathrm{List}\, A \\ \mathrm{fold} &: X \to (A \to X \to X) \to \mathrm{List}\, A \to X \end{aligned}$$

   using general recursive definitions by pattern matching.

2. Given $n : \mathbb{N}$ we define the type $\mathrm{Fin}\, n$ as given by the following constructors

$$\begin{aligned} 0 &: \Pi_{n:\mathbb{N}} \mathrm{Fin}\, 1+n \\ \mathrm{suc} &: \Pi_{n:\mathbb{N}} \mathrm{Fin}\, n \to \mathrm{Fin}\, (1+n) \end{aligned}$$

   Define the following functions

$$\begin{aligned} \mathrm{max} &: \Pi_{n:\mathbb{N}} \mathrm{Fin}(1+n) \\ \mathrm{emb} &: \Pi_{n:\mathbb{N}} \mathrm{Fin}\, n \to \mathrm{Fin}\, (1+n) \\ \mathrm{inv} &: \Pi_{n:\mathbb{N}} \mathrm{Fin}\, n \to \mathrm{Fin}\, (1+n) \end{aligned}$$

   where max computes the maximal element in a non-empty finite type; emb embedds $\mathrm{Fin}\, n$ into $\mathrm{Fin}\, (1+n)$ without changing the value, e.g. $\mathrm{emb}\, 3_5 \equiv 3_6$. The function inv inverts the order of elements in a finite type in particular it maps 0 to the maximal element and the maximal element to 0.

3. Using propositions as types, can you prove the translation of the axiom of choice? Let A, B be type and R be a relation between A and B the axiom of choice is:

   $(\forall x : A.\exists y : B.R\,x\,y) \Rightarrow (\exists f : A \to B.\forall x : A.R\,x\,(f\,x))$

4. Using propositions as types try to prove the de Morgan laws of predicate logic:

   $\neg(\forall x : A.P\,x) \Leftrightarrow \exists x : A.\neg(P\,x)$
   $\neg(\exists x : A.P\,x) \Leftrightarrow \forall x : A.\neg(P\,x)$

Did you notice that $\times$ actually appears twice: we can either view $A \times B$ as a non-dependent Σ-type ($\Sigma x : B.B$) or as a Bool-indexed Π-type. This dual nature of products indeed leads to slightly different approaches to products: either based on tupling or based on projections.

To summarize these observations we can say that we can classify the operators in 2 different ways: either along dependent / non-dependent or along binary or iterated:

non-dep.	dep.
$\times$	Σ
$\to$	Π

binary	iterated
$+$	Σ
$\times$	Π

This can lead to some linguistic confusions: what is a dependent product type? Hence I prefer just to say Π-type and Σ-type.

4.3 One Universe is not enough

So far we have avoided to consider types as explicit objects but instead they were part of our prose. Remember, I was saying: *given a type A we define the type of lists* List A. But what is List if not a function on types, that is List : **Type** $\to$ **Type**. Here we introduce **Type** as a type of types and this sort of thing is called a *universe* in Type Theory. So for example $\mathbb{N}$: **Type** but also $3 : \mathbb{N}$ hence $\mathbb{N}$ can appear on either side of the colon. A universe is a type whose elements are types again. This also works for Fin : $\mathbb{N} \to$ **Type** and Vec : $\mathbb{N} \to$ **Type** $\to$ **Type**. We call Fin and Vec families of types or dependent types and we should also call List a dependent type but the terminology is a bit vague here and some people would call it a *type indexed type*.

The reason is that functional programming languages support types like List but not *properly dependent types* like Fin or Vec. This has mainly to do with pragmatic reasons, namely that programming language implementers don't want to evaluate programs at compile time but they would have to do check that applying a function $f : \mathrm{Fin}\,7 \to \mathbb{N}$ to $i : \mathrm{Fin}\,(3+4)$ as in $f\,i$ is well typed. In this case this is easy and just involves checking that $7 \equiv 3 + 4$ but there are no limits what functions we could use here. In the case of a language like Haskell this could even involve a function that loops which would send the compiler into a loop. However, implementations of Type Theory like Agda or Coq aren't really so much better, at least from a pragmatic point of view, since we may use the Ackermann function here which may mean that while the function will eventually terminate this may only happen after the heat death of the universe.

The obvious question is what is the type of **Type**? Is it **Type** : **Type**. Hang on wasn't this the sort of thing where Frege got in trouble'? Does Russell's paradox applies here? The type of all type which does not contain itself? Not immediately because : isn't a proposition hence we cannot play this sort of trick here. Maybe this was what Per Martin-Löf was thinking when he wrote his first paper about Type Theory which did include **Type** : **Type**. This time it wasn't an English philosopher who pointed out the problem but a french one: Jean-Yves Girard. But then Per Martin-Löf is from Sweden not Germany.

Girard wasn't interested in dependent types at the time but rather in the type indexed by type variety. Actually he wasn't interested in types for programming but types for logic. His goal was to prove Takeuti's conjecture who was looking for a *syntactic proof of the consistency of 2nd order logic*, which is predicate logic extended by extra quantifiers which allow you to quantify over all propositions. Girard's approach was to introduce a λ-calculus which he called *System F* which corresponds to 2nd order logic in the same way as Gentzen's System T corresponds to Peano Arithmetic. I should add that the power of 2nd order logic and hence System F vastly exceeds Peano's Arithmetic / System T. By the power of a logic we mean the ability of one logic to *eat* another one: we can prove the consistency of Peano's arithmetic in 2nd order logic but not the other way around. Hence Takeuti's conjecture was asking for a relative simple, i.e. syntactic, proof of the consistency of a very powerful system. I believe it wasn't very clear what exactly he meant by this and it is maybe doubtful wether Girard's ingenious proof of the strong normalisation of System F fits this bill. Girard proved that all programs in System F terminate no matter how you evaluate them, this is called strong normalisation.

The main feature of System F is that we can construct Π-types that quantify over all types as in $\Pi X : \mathbf{Type}.X \to X$. This is the type of the *polymorphic identity function*:

$$I \equiv \lambda X.\lambda x : X.x : \Pi X : \mathbf{Type}.X \to X$$

The nice feature of the polymorphic identity is that it works for any type, e.g. $I\,\mathbb{N} : \mathbb{N} \to \mathbb{N}$ is the identity for natural numbers. It even works for its own type as in

$$I\,(\Pi X : \mathbf{Type}.X \to X) : (\Pi X : \mathbf{Type}.X \to X) \to (\Pi X : \mathbf{Type}.X \to X)$$

which means that in particular we can apply it to itself:

$$I\,(\Pi X : \mathbf{Type}.X \to X)\,I : \Pi X : \mathbf{Type}.X \to X$$

I find this rather mind boggling and it also looks dangerous since in the untyped λ-calculus self-apply was the main ingredient of the fixpoint combinator which enables us to run any program including some which don't terminate. Hence it is reassuring to know that strong normalisation holds and that this is indeed impossible.

There are some interesting games one can play with System F: there is a clever way to encode the natural numbers which is called *Church numerals*:

$$\mathbb{N} :\equiv \Pi X.X \to (X \to X) \to X$$

The idea is that a number is represented by a higher order function that repeats another function, e.g. $3\,a\,f \equiv f\,(f\,(f\,a))$. We can represent zero and successor:

$0 : \mathbb{N}$
$0 \equiv \lambda X, a, f.a$
$\mathrm{suc} : \mathbb{N} \to \mathbb{N}$
$\mathrm{suc} \equiv \lambda n, X, a, f.f\,(n\,X\,a\,f)$

and there also clever ways to encode the standard arithmetic functions:

$\mathrm{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathrm{plus} : \lambda m, n.\lambda X, a, f.mXf(nXfa)\mathrm{mult} \qquad : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathrm{mult} : \lambda m, n.\lambda X, a, f.mX(nXf)a$
$\mathrm{pow} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathrm{pow} : \lambda m, n.\lambda X, a, f.n(X \to X)(mX)$

Unlike our primitive recursive definitions, plus doesn't use suc, mult doesn't use plus and pow doesn't use mult. Instead we use the power to repeat: plus repeats the function f first n-times and then m-times, mult repeats the repetition of n, m-times and pow? I don't know anybody who understand pow without getting a headache but it is interesting that this is the first function which exploits the *polymorphic* nature of the natural numbers and instantiates the 2nd argument to a type different from X.

The idea of System F was so natural that it got reinvented in Computer Science by John Reynolds who introduced the polymorphic λ-calculus only to find out that Girard had invented basically the same calculus a few years earlier under the name System F. However, it is certainly Reynolds influence which lead to the polymorphic λ-calculus aka System F to be used as the base of type systems for many modern programming languages.

Ok, this was a long diversion. What has all this to do with Martin-Löf's Type Theory using **Type** : **Type**. I am almost there! One generalisation of System F is to allow not only to use **Type** but also functions over types, that is something like

$\Pi F : (\mathbf{Type} \to \mathbf{Type}).\Pi X.F\,(F\,X) \to X$

This extension which allows to quantify over functions over types is called System F^ω and it is still strongly normalizing. But Girard wondered wether this level could be polymorphic again, i.e. using System F instead just the simply typed λ-calculus to define functions over types, such as

$\Pi G : (\Pi X : \mathbf{Type}.X \to X).\Pi X : \mathbf{Type}.GX \to X$

But here Girard hit a brick wall - this system which he called System U is not strongly normalising anymore. Girard's truly ingenious paradox used an encoding of the Burali-Forti Paradox in System U: the collection of all well-orders cannot be well-ordered itself.

Now when Girard saw Martin-Löf's system he quickly realized that this system is powerful enough to encode System U, and hence it couldn't be strongly normalising itself. Even worse it turns out that in a system with **Type** : **Type** every type is inhabited including the empty type. This makes it useless as a logic because now we can prove everything.

To explain why **Type** : **Type** doesn't work I prefer a construction by Thierry Coquand [**?**] who showed that we can encode Russell's paradox in such a theory using trees. This is less general than Girard's paradox because it relies on the availability of trees and it doesn't work for System U (at least I am not aware how it could be adapted to System U). But Girard's paradox is very technical and hard to understand intuitively (at least for me).

Thierry's idea was to use an idea by Peter Aczel and encode the sets of set theory as trees. That is he defined a type Tree : **Type** with a constructor

$\mathrm{mkTree} : \Pi I : \mathbf{Type}(I \to \mathrm{Tree}) \to \mathrm{Tree}$

The idea is that mkTree works like the curly brackets from set theory. That is if we have any collection of trees indexed by any type we can form a new tree. For example we can define the empty tree empty : Tree using the empty type,

$\mathrm{empty} :\equiv \mathrm{mkTree}\, 0\, \mathrm{efq}$

where $\mathrm{efq} : 0 \to \mathrm{Tree}$ is a function that needs no definition. Also given a, b : Tree we can define a new tree $\mathrm{pair}\, a\, b$: Tree representing $\{a, b\}$. This implements the axiom of pairing.

$\mathrm{pair}\, a\, b :\equiv \mathrm{mkTree}\, \mathrm{Bool}\, (\lambda x.\mathrm{if}\, x\, \mathrm{then}\, a\, \mathrm{else}\, b)$

We don't get the axiom of extensionality because $\mathrm{pair}\, a\, b$ and $\mathrm{pair}\, b\, a$ are not equal. However, this doesn't matter for Russell's paradox which also works for trees. We do get unlimited comprehension for trees that is given a property of trees encoded as a family of types $P : \mathrm{Tree} \to \mathbf{Type}$ we can define $\mathrm{compr}\, P$: Tree representing $\{t \mid P\, t\}$ as

$\mathrm{compr}\, P :\equiv \mathrm{mkTree}\, (\Sigma t : \mathrm{Tree}.P\, t)\pi_0$

That is the index type is a pair of a tree and a proof that it satisfies the property and the function which assigns trees to indices is just the first projection.

To define the property of trees that they are not an element of themselves we have to explain what an element of a tree is, namely a direct subtree. That is we define $\mathrm{el} : \mathrm{Tree} \to \mathrm{Tree} \to \mathbf{Type}$ with $\mathrm{el}\, a\, b$ expressing $a \in b$.

$\mathrm{el}\, a\, (\mathrm{mkTree}\, I\, f) \equiv \Sigma i : I.f\, i = a$

Here we use Σ to encode $\exists$ and we use the equality type which we haven't discussed yet but I hope this is clear anyway. Now we can express the property of a tree t not containing itsef $(\mathrm{el}\, t\, t) \to 0$ and we define Russell's tree R : Tree as

$R :\equiv (\mathrm{compr}(\lambda t.(\mathrm{el}\, t\, t) \to 0)$

Just using were elementary reasoning we can show that

$\mathrm{el}\, R\, R \leftrightarrow (\mathrm{el}\, R\, R) \to 0$

And it follows from propositional logic that $\neg(P \Leftrightarrow \neg P)$ from simple reasoning in propositional logic. But this means that the empty type in inhabited.

Where did we actually use **Type** : **Type**? We didn't explicitly but implicitly when we introduced a constructor makeTree whose argument is a type again.

So, if **Type** : **Type** doesn't work what is now the type of **Type**? To avoid the problem we introduce not one namely infinitely many universes.

$\mathbf{Type}_0 : \mathbf{Type}_1 : \mathbf{Type}_2 : \ldots$

Hence the answer depends on the level, if we ask about the first universe $\mathbf{Type}_0$ then its type is $\mathbf{Type}_1$ and what is the type of $\mathbf{Type}_1$? Right is is $\mathbf{Type}_2$. And so on.

The numbers $0, 1, 2, \ldots$ which as the index of types are not our natural numbers (even though the look very much like them) because otherwise we would have to make our mind up was is the type of a function that assigns to a number a type and what is the type of this function?

All our usual garden variety types like $\mathbb{N}, \mathrm{Bool}, \mathrm{List}\, \mathbb{N}, \mathbb{N} \to \mathrm{Bool}, \mathrm{InfTree}, \ldots$ are elements of $\mathbf{Type}_0$, but they are also elements of all higher universes - this is called cummulativity. The thing that is new in $\mathbf{Type}_1$ is $\mathbf{Type}_0$ itself. We say that $\mathbf{Type}_0$ is larger then all the types in $\mathbf{Type}_0$ because if there would be a type in $\mathbf{Type}_0$ which is equivalent to $\mathbf{Type}_0$ we can derive the paradox. There are more new types in $\mathbf{Type}_1$ for example $\mathrm{List}\, \mathbf{Type}_0$ or $\mathbf{Type}_0 \to \mathbb{N}$. This story continues: the new types in $\mathbf{Type}_2$ are $\mathbf{Type}_1$ and all the types which can be built from it.

While this solves the problem with **Type** : **Type** it does introduce a lot of bureaucracy: for example we have many copies of List namely $\mathrm{List}_0 : \mathbf{Type}_0 \to \mathbf{Type}_0$ and $\mathrm{List}_1 : \mathbf{Type}_1 \to \mathbf{Type}_1$ etc. This is not covered by cummulativity which only tells us that $\mathrm{List}_0\, \mathbb{N} : \mathbf{Type}_1$. What is the best way to deal with this problem is a research question: while there are a number of candidates and implementations there is no general agreement what is the best way. On paper we can be lazy and adopt the convention that we just pretend that we had **Type** : **Type** but then check that there is a consistent assignment of indices to each occurence of **Type**. This is close to what the Coq system is doing but there are cases where the inference algorithm is too weak.

4.4 Propositions as Types: Predicate logic

I am now going to deliver on the promise to extend the propositions as types explanation to predicate logic using dependent types. Actually it is rather straightforward and I couldn't avoid giving away the secret in the last section - maybe you noticed. We translate $\forall$ with Π-types. That is for example the

What has really happened is that from the lie that we can recover information we have just hidden we can extract information as long as we hide the function doing the extraction. And this has nothing to do with Σ-types and existentials but with the behaviour of the hiding operation, or inhabitance $||-||$. Hence we can formulate a simpler version of the axiom of choice in Type Theory:

$$(\Pi x : A.||B\,x||) \to ||\Pi x : A.B\,x||$$

This implies the revised translation of the axiom. [4] It is interesting that the reverse direction

$$||\Pi x : A.B\,x|| \to (\Pi x : A.||B\,x||)$$

is actually provable - for the details we need a better understanding of $||-||$. However, it should be intuitively clear that from a hidden function we can generate hidden information but not vice versa.

4.6 Induction is recursion

In section 3.7 we analysed primitive recursion for simple types. That is we explained that for example to define a function from the natural numbers to another type let's say $f : \mathbb{N} \to A$, we need to say what the function is doing for 0 hence we need an $z : A$ and we need to say what the function is doing for the successor assuming we already know the result for the previous number, that menas we need a function $s : A \to A$ and we can define f by primitive recursion as:

$$\begin{aligned} f\,0 &:\equiv z \\ f\,(\mathrm{suc}\,n) &:\equiv s\,(f\,n) \end{aligned}$$

We also discovered that we can turn this principle into a higher order function which I called iter which for any type A has the type

$$\mathrm{iter}_A : A \to (A \to A) \to \mathbb{N} \to A$$

actually now that we know about universes we can get rid of the ad hoc treatment of the type and just say

$$\mathrm{iter} : \Pi_{A:\mathbf{Type}} A \to (A \to A) \to \mathbb{N} \to A$$

which is defined as

$$\begin{aligned} \mathrm{iter}_A\,a\,f\,0 &:\equiv a \\ \mathrm{iter}_A\,a\,f\,(\mathrm{suc}\,n) &:\equiv f\,(\mathrm{iter}\,a\,f\,n) \end{aligned}$$

[4]It is actually equivalent if we drop the restriction that R is propositional.

We can define f just using iter:

$f :\equiv \mathrm{iter}\, z\, s$

Now let's switch tack and say we want to define a dependent function $f : \Pi n : \mathbb{N}.A\, n$ where $A : \mathbb{N} \to \mathbf{Type}$ is a dependent type. As before we need to say what f does for 0 hence we need $z : A\, 0$ and we need to say how to compute $f\,(\mathrm{suc}\, n)$ from $f\, n$ and that means we need $s : \Pi n : \mathbb{N}.A\, n \to A\,(\mathrm{suc}\, n)$. Given these ingredients we can define f by dependent primitive recursion:

$$\begin{aligned} f\, 0 &:\equiv z \\ f\,(\mathrm{suc}\, n) &:\equiv s\, n\,(f\, n) \end{aligned}$$

Indeed, this is almost the same as before: the only difference is that we need to communicate the natural number which we use to s, which if you remember makes it more like prec. I often hide the first argument to s in which case the definition looks exactly the same.

We have already seen some examples. In the beginning of the chapter we defined zeros : $\Pi n : \mathbb{N}.\mathrm{Vec}\, \mathbb{N}\, n$ using dependent primitive recursion:

$$\begin{aligned} \mathrm{zeros}\, 0 &:\equiv [] \\ \mathrm{zeros}\,(1 + n) &:\equiv 0 :: (\mathrm{zeros}\, n) \end{aligned}$$

In this case $z \equiv []$ and $s\, n\, x :\equiv 0 ::_n x$. Another example using propositions as types I need to define a dependent function to show $\Pi n : \mathbb{N}.n = n + 0$. We defined $f : \Pi n : \mathbb{N}.n = n + 0$ as:

$$\begin{aligned} f\, 0 &:\equiv \mathrm{refl}\, 0 \\ f\,(\mathrm{suc}\, n) &:\equiv \mathrm{resp}\, \mathrm{suc}\,(f\, n) \end{aligned}$$

So here $z \equiv \mathrm{refl}\, 0$ and $s\, n\, x :\equiv \mathrm{resp}\, \mathrm{suc}\, x$.

And again as before we can define one define one higher order function which implements dependent primitive recursion: this is called elim which is short for eliminator:

$\mathrm{elim} : \Pi A : \mathbb{N} \to \mathbf{Type}.A\, 0 \to (\Pi n : \mathbb{N}.A\, n \to A\,(\mathrm{suc}\, n)) \to \Pi n : \mathbb{N}.A\, n$

which is defined in a way very similar to iter:

$$\begin{aligned} \mathrm{elim}\, A\, z\, s\, 0 &:\equiv z \\ \mathrm{elim}\, A\, z\, s\,(\mathrm{suc}\, n) &:\equiv s\, n\,(\mathrm{elim}\, A\, z\, s\, n) \end{aligned}$$

We can put elim to work to derive the previous examples, in the case of zeros : $\Pi n : \mathbb{N}.\mathrm{Vec}\, \mathbb{N}\, n$ we use

$\mathrm{zeros} :\equiv \mathrm{elim}\,(\mathrm{Vec}\, \mathbb{N})\, []\, (\lambda n, x.0 ::_n x)$

and in the case of $f : \Pi n : \mathbb{N}.n = n + 0$:

$f :\equiv \mathrm{elim}\,(\lambda n.n = n + 0)\,(\mathrm{refl}\, 0)\,(\lambda n, x.\mathrm{resp}\, \mathrm{suc}\, x)$

It is worthwhile to look at the type of elim and notice that it exactly corresponds to the principle of induction for natural numbers using the propositions as types view. Or using our refined version we could say that induction is the special case of dependent primitive recursion in the case of a propositional family $A : \mathbb{N} \to \mathbf{Prop}$, which is what we used in the 2nd case.

While this is not a hard theorem it is an important observation that the proposition as types views shows us that induction is just a special case of dependent primitive recursion. This hopefully correspond to our naive understanding of induction, which is true because we can use recursion to repeat the inductive step as many times as needed before resorting to the base case.

As we were able to extend primitive recursion to other datatypes we can do the same in the for the other datatypes we looked at. To define a function out of a datatype we only need to say what it is doing for the constructors. We can condense this into an eliminator which is a higher order function implementing this reduction.

What is the point of an eliminator? Using eliminators we can reduce the definability of a function to a formal criterion, the same way as we reduced primitive recursion to just using one constant, iter. In practice we don't want to use eliminators all the time but come up with nice and readable definitions which can be reduced to eliminators. But if in doubt we better provide a translation into the use of basic eliminators to make sure that we are not cheating.

As an example let's derive the eliminator for lists, which with our propositions as types glasses will correspond to an induction principle for lists. On the other hand it should generalize the fold we have already seen. But let's just go through the basic motions. To define a dependent function $f : \Pi x : \text{List}\, A.B\, x$ where $B : \text{List}\, A \to \mathbf{Type}$ we need to say what f is doing for the constructors of List A that is we have to complete the lines

$$\begin{aligned} f\,[] &:\equiv ?_0 \\ f\,(a :: l) &:\equiv ?_1 \end{aligned}$$

Clearly $?_0 : B[]$ so lets just assume we have $?_0 \equiv n : B\,[]$. To complete $?_1 : B\,(a :: l)$ we can use the result of the recursive call $f\,l : B\,l$ and we can also use $a : A$, that is we need a function $c : \Pi a : A, \Pi l : \text{List}\, A.B\, l \to B\,(a :: l)$ and we set $?_1 \equiv c\,a\,l\,(f\,l)$. To put it all together we arrive at the following definition of f

$$\begin{aligned} f\,[] &:\equiv n \\ f\,(a :: l) &:\equiv c\,a\,l\,(f\,l) \end{aligned}$$

given

$$\begin{aligned} &n : B\,[] \\ &c : \Pi a : A, \Pi l : \text{List}\, A.B\, l \to B\,(a :: l) \end{aligned}$$

Now to condense everything into an eliminator we make all the parameters explicit which includes the type parameters:

$$\begin{aligned}\text{elim}^{\text{List}} :\quad & \Pi_{A:\mathbf{Type}}\Pi B : \text{List}\, A \to \mathbf{Type}.B\,[\,] \\ & \to (\Pi a : A, l : \text{List}\, A.B\, l \to B\,(a :: l)) \\ & \to \Pi l : \text{List}\, A.B\, l\end{aligned}$$

$$\begin{aligned}\text{elim}^{\text{List}}\, B\, n\, c\, [\,] &:\equiv n \\ \text{elim}^{\text{List}}\, B\, n\, c\, (a :: l) &:\equiv c\, a\, l\, (\text{elim}^{\text{List}}\, B\, n\, c\, l)\end{aligned}$$

There is one more example I would like to cover and that is to show an eleminator for an inductively defined family like Fin : $\mathbb{N} \to \mathbf{Type}$. To remind us: Fin is inductively generated by:

$$\begin{aligned}0 &: \Pi_{n:\mathbb{N}}\text{Fin}\, 1 + n \\ \text{suc} &: \Pi_{n:\mathbb{N}}\text{Fin}\, n \to \text{Fin}\, (1 + n)\end{aligned}$$

Now how to define a function out of Fin. Such a function will need to inputs: the index $n : \mathbb{N}$ and the element $i : \text{Fin}n$. That is we want to define a function $f : \Pi_{n:\mathbb{N}}\Pi i : \text{Fin}\, n.A\, n\, i$ where $A : \Pi_{n:\mathbb{N}}\text{Fin}\, i \to \mathbf{Type}$. The idea is the same as before: we have to say what f is returning for 0_n and we have to say what it returns for $\text{suc}_n\, i$ assuming we know the result of $f_n\, i$. That is we have to complete

$$\begin{aligned}f_{1+,n}\, 0_n &:\equiv ?_0 \\ f_{1+n}(\text{suc}_n\, i) &:\equiv ?_1\end{aligned}$$

Before we fill in the ?s - do you note something? The hidden parameter, that is the index $n : \mathbb{N}$ is always $1 + n$! That is because the constructors of Fin always produce elements in $\text{Fin}\, (1 + n)$ and never any in $\text{Fin}\, 0$ - which is only right because $\text{Fin}\, 0$ is intentionally left empty. And indeed we are analyzing the constructors of Fin not the indizes.

Now back to the ?s. $?_0 : A_{1+n}\, 0_n$ hence we need $z : \Pi_{n:\mathbb{N}}A_{1+n}\, 0_n$ and $?_1 : A_{n+1}\, (\text{suc}_n\, i)$ but here we can use the recursive result of $f_n\, i : A_n\, i$. Hence we stipulate a function $s : \Pi_{n:\mathbb{N}}A_n\, i \to A_{1+n}\, (\text{suc}_n\, i)$ and we can fill in

$$\begin{aligned}f_{1+,n}\, 0_n &:\equiv z \\ f_{1+n}(\text{suc}_n\, i) &:\equiv s_n\, (f\, n\, i)\end{aligned}$$

Now we package this all into one complicated looking eliminator:

$$\begin{aligned}\text{elim}^{\text{Fin}} : \Pi A : (\Pi_{n:\mathbb{N}}\text{Fin}\, n \to \mathbf{Type}).&(\Pi n : \mathbb{N}.A_{n+1}\, 0_n) \\ & \to (\Pi_{n:\mathbb{N}}\Pi i : \text{Fin}\, n.A_n\, i \to A_{1+n}\, (\text{suc}_n\, i)) \\ & \to \Pi_{n:\mathbb{N}}\Pi i : \text{Fin}\, n \to A_n\, i\end{aligned}$$

Let us use the eliminator to fullfill an earlier promise, namely to give a justification why

$$\text{nth} : \Pi_{n:\mathbb{N}}\text{Vec}\, A\, n \to \text{Fin}\, n \to A$$

$$\begin{aligned}\mathrm{nth}\,(a :: v)\,0 &:\equiv a\\ \mathrm{nth}\,(a :: v)\,(1+n) &:\equiv \mathrm{nth}\,v\,n\end{aligned}$$

is a reasonable definition, even though it doesn't seem to cover the index zero. To translate this we need head and tail for vectors which are inverting the cons operation for vectors. They are defined as follows:

$$\begin{aligned}&\mathrm{hd} : \Pi_{n:\mathbb{N}}\mathrm{Vec}\,A\,(1+n) \to A\\ &\mathrm{hd}\,(a :: v) :\equiv a\\ &\mathrm{tl} : \Pi_{n:\mathbb{N}}\mathrm{Vec}\,A\,(1+n) \to \mathrm{Vec}\,A\,n\\ &\mathrm{tl}\,(a :: v) :\equiv v\end{aligned}$$

I leave it as an exercise to derive the eliminator for Vec and then show that hd and tl can be defined using it. To make it work we need to commute the arguments, because we will need to make sure that the index of the vector is determined by the index of the element of Fin. That is we define

$$\mathrm{nth}' : \Pi_{n:\mathbb{N}}\mathrm{Fin}\,n \to \mathrm{Vec}\,A\,n \to A$$

This way we can use $\mathrm{Vec}\,A\,n \to A$ depending on n as the motive:

$$\mathrm{nth}' :\equiv \mathrm{elim}^{\mathrm{Fin}}\,(\lambda n.\mathrm{Vec}\,A\,n \to A)\,(\lambda n.\mathrm{hd}_n)\,?_0\,?_1$$

The type of $?_0$ is $\Pi n : \mathbb{N}.\mathrm{Vec}\,A\,(1+n) \to A$, the 1+ here has the origin in the 1+ in the type of 0_n. This fits perfectly with the type of hd which is what we want to do here, hence $?_0 \equiv \lambda n.\mathrm{hd}_n$. In the other case we have

$$?_1 : \Pi_{n:\mathbb{N}}\Pi i : \mathrm{Fin}\,n.(\mathrm{Vec}\,A\,n \to A) \to (\mathrm{Vec}\,A\,(1+n)) \to A)$$

which perfectly fits with calling the recursively computed result with the tail of the input vector:

$$?_1 \equiv \lambda n, i, h.\lambda v.h\,(tl_n v)$$

We notice that we aren't actually using the i parameter to the motive, which means that we don't use the eliminator in its full generality here. Indeed, the result type only depnds on the index not on the element of Fin itself. Putting everything together we can define nth also inlining the parameter swap:

$$\begin{aligned}\mathrm{nth}\,v\,i \quad &:\equiv \mathrm{elim}^{\mathrm{Fin}}\,(\lambda n.\mathrm{Vec}\,A\,n \to A)\\ &\quad(\lambda n.\mathrm{hd}_n)\\ &\quad(\lambda n, i, h.\lambda v.h\,(\mathrm{tl}_n v))\\ &\quad i\,v\end{aligned}$$

OK, this is maybe more principled then the first version of nth but hardly readable. This is a common tension: if we try to justify a construction from first principles we loose readability but on the other hand it is clear that we are not cheating. To navigate in this space we usually present the readable version but when in doubt make sure that it is actually derivable using the spartan world of eliminators. Also implementations of Type Theory should work like this and should translate elegant presentations into type-theoretic assembly language. Alas very few do and it is often considered easier to directly interpret a fairly high level language. This also leads to unintended consequences in form of unsoundness (we can prove False) which diminish the trustworthiness of the system. We will get back to this in the last chapter.

Also you may remark that while I have presented the idea how to derive eliminators from the constructor type, this can hardly be called systematic. What exactly are the restrictions on constructor types (see our discussion in 3.7)? And shouldn't we present the derivation of the eliminator in a more systematic way

There are a number of possible answers here. Both the restrictions and the derivation of the eliminators can be presented in the form of schematic syntax rules. They are precise but do get quite complicated and it is hard to design them so that we call cover all the cases. And if the instances of the eliminators look complicated they are nothing compared to the rules. While there is a point to represent a system based on some precise rules, these rules are hardly readable and they don't give us much insight.

More insight can be obtained by using the language of category theory. As we have seen in the section on simple types data types can be described by functors and the actual type is the initial algebra of a functor. Actually not completely, because we couldn't really capture the η-rules for more interesting datatypes like the natural numbers or lists. This is in a way fixed once we add the dependent eliminator because we can prove the η-rules using the equality type which I will introduce in full gory detail in the next section. What about dependent types like Fin? Here we have to change the category we live in and move from the category of types and functions, to the category of families of types and families of functions. That is types are replaced by families $A : \mathbb{N} \to \mathbf{Type}$ and given $A, B : \mathbb{N} \to \mathbf{Type}$ a family of functions is given by $f : \Pi n : \mathbb{N}. A\, n \to B\, n$. Now Fin can be also specified by a functor that is on type families:

$$F : (\mathbb{N} \to \mathbf{Type}) \to (\mathbb{N} \to \mathbf{Type})$$
$$F\, A :\equiv \lambda n.(\Sigma m : \mathbb{N}. m = 1 + n) + (\Sigma m : \mathbb{N}. m = 1 + n \times A\, m)$$

I leave it as an exercise (which does involve a better understanding of the yet unspecified equality type) to show that this operation comes with a well behaved map operation on family of functions. And indeed, Fin arises as an initial algebra of this functor with the same asymmetry between definitional β-rules and provable η-rules.

The fact that datatypes and even dependent datatypes can be understood as initial algebras is certainly a nice fit between category theory and Type Theory. However, category theory doesn't answer the question what datatypes are acceptable. Already earlier, in section 3.7 I argued that some functors do not correspond to reasonable datatypes, e.g. the functor given by $F : \mathbf{Type} \to$

Type with $F\,X :\equiv (X \to 2) \to 2$ seems problematic. We want to restrict ourselves to strictly positive datatypes but what does this mean exactly?

The answer is that we only want to consider functors which correspond to *containers*, where a container is given by a type of shapes S : **Type** and a family of positions $P : \mathbb{N} \to$ **Type** which gives rise to a functor [5]

$$\begin{aligned} &T : \mathbf{Type} \to \mathbf{Type} \\ &T\,X :\equiv \Sigma s : S.P\,s \to X \end{aligned}$$

The intuition is that an instance of a container $T\,A$ is given by a choice of shape $s : S$, and an assignment of a *payload*, that is an element of A to each position, i.e. a function $P\,s \to A$. An intuitive example is given by the container representation of the list functor: the shape of a list is its length hence $S :\equiv \mathbb{N}$ and the positions is the finite set with n elements because this is how much payload a list of length n can take, hence $P :\equiv \mathrm{Fin} : S \to$ **Type**. Indeed, we can show that any sytactically strict positive datatype can be represented as a container.

Ever container gives rise to a datatype, that is the type of trees whose nodes are labelled by elements of S and the subtrees of a node labelled by $s : S$ is indexed by $P\,s$. This datatype is called a W-type and given a container that is S : **Type** and $P : S \to$ **Type** and now W : **Type** is generated by the constructor

$$\mathrm{node} : \Pi s : S \to (P\,s \to W) \to W$$

As we have already observed the natural numbers are generated by the functor $T\,X = 1 + X$ which is a container given by $S = 2$ and $P\,false :\equiv 0$ and $P\,true :\equiv 1$. The trees generated by $W\,S\,P$ correspond to the natural numbers where $\mathrm{node}\,\mathrm{false} : (0 \to W) \to W$ represents the number 0 all we need to do is to add the function $\mathrm{efq} : 0 \to W$ (which does need no definition) as a parameter $0 \equiv \mathrm{node}\,\mathrm{false}\,\mathrm{efq}$. Successors are represented using nodes of the other shape $\mathrm{node}\,\mathrm{true} : (1 \to W) \to W$ hence we define $\mathrm{suc}\,n :\equiv \mathrm{noe}\,\mathrm{true}\,(\lambda x.n)$.

We can do this with other datatypes we have encountered so far, for example lists over A : **Type** are generated by the W-type specified by

$$\begin{aligned} S &:\equiv 1 + A \\ P\,(\mathrm{inl}\,()) &:\equiv 0 \\ P\,(\mathrm{inr}\,a) &:\equiv 1 \end{aligned}$$

To better understand the general idea let's anaylze the expression trees which were given by

$$\begin{aligned} \mathrm{const} &: \mathbb{N} \to \mathrm{Expr} \\ \mathrm{plusOp} &: \mathrm{Expr} \to \mathrm{Expr} \to \mathrm{Expr} \\ \mathrm{timesOp} &: \mathrm{Expr} \to \mathrm{Expr} \to \mathrm{Expr} \end{aligned}$$

[5] As an exercise try to derive the map part of this functor.

Here we hace 3 constructors the first one parametrised by $\mathbb{N}$. Hence we choose

$S :\equiv \mathbb{N} + 2$

How many recursive occurences of Expr can we count for each of the three constructors: none for the first and 2 for the 2nd and the 3rd. Hence we define P as follows:

$$\begin{aligned} P\,(\mathrm{inl}\,n) &:\equiv 0 \\ P\,(\mathrm{inr}\,b) &:\equiv 2 \end{aligned}$$

We can also represent InfTree which was given by

$$\begin{aligned} \mathrm{leaf} &: \mathrm{InfTree} \\ \mathrm{node}(\mathbb{N} \to \mathrm{InfTree}) &\to \mathrm{InfTree} \end{aligned}$$

here we have a situation where P can be infinite. That is we define

$$\begin{aligned} S &:\equiv 2 \\ P\,\mathrm{false} &:\equiv 0 \\ P\,\mathrm{true} &:\equiv \mathbb{N} \end{aligned}$$

What about dependent datatypes like Fin and Vec? With some more work and excessive use of equality they can be done as well. What about coinductive types like streams? It turns out that they can be represented as well by viewing same as a limit of approximations of trees of finite depth. There are some issues here which have to do with the nature of equality which I will cover in the next section.

Ok, we can represent datatypes using W-types. What is the point? The point is that we now only need to specify one datatype namely the W-type. To be precise W is a parametrised type because S and P are really parameters hence

$W : \Pi S : \mathbf{Type}.(S \to \mathbf{Type}) \to \mathbf{Type}$

it comes with an eliminator (to make it readable lets fix S and P again) that is $W \equiv W\,S\,P$.

$$\begin{aligned} \mathrm{elim}^W &: \Pi M : W \to \mathbf{Type}. \\ &\quad (\Pi s : S.\Pi f : P\,s \to W.(\Pi p : P\,s.A\,(f\,p) \to W\,(\mathrm{node}\,s\,f))) \\ &\quad \to \Pi w : W.A\,s \end{aligned}$$

$$\mathrm{elim}^W\,M\,m\,(\mathrm{node}\,s\,f) :\equiv m\,s\,f\,(\lambda p.\mathrm{elim}^W\,M\,m\,(f\,p))$$

Ok, this is ugly enough but since we can reduce all other datatypes to this one, this is the only eliminator we have to write down to specify what we mean by dependent primitive recursion for any datatype.

Ok, this is not completely true: we also need to specify eliminators some more basic types: the finite types $0, 1, 2$, Σ-types and the mysterious equality type which I have saved for the next section. This way Type Theory can be turned into a closed system like set theory and everything we need can be derived which maybe quite hard work. Hopefully we can rely on the help of computers here. [6]

4.7 The mystery of equality

I am keeping the equality type to the end because there are some issues with it which is a good preparation for the next chapter. The basic idea is quite easy: for any type $A : \mathbf{Type}$ equality is a dependent type $- =_A - : A \to A \to \mathbf{Type}$ whose only constructor is reflexivity:

$\mathrm{refl} : \Pi_{a:A} A \to A \to \mathbf{Type}$

After having seen many examples of dependent primitive recursion it shouldn't be hard to derive the eliminator for this type. Even though the word recursion is certainly misapplied here because there is no recursion going on. Let's go through the motions: How do we define a function out of an equality type? As before for Fin we also have to abstract over the indices, that is the question is how do we define a function

$f : \Pi_{x,y:A} \Pi p : x = y . M\, x\, y\, p$

where $M : \Pi_{x,y:A} x = y \to \mathbf{Type}$. Since the only constructor is equality we only need to prescribe what is the result for

$f_{x\,x}\,(\mathrm{refl}\, x \equiv ?$

and ? has the type $M\, x\, x\, (\mathrm{refl}\, x)$. That is to define f we need

$m : \Pi x : A . M\, x\, x\, (\mathrm{refl}\, x)$

and given m we define

$f_{x\,x}\,(\mathrm{refl}\, x = m\, x$

We can condense this into one of the horrible eliminators

$$\begin{aligned} \mathrm{elim}^{=} : \Pi M : (\Pi_{x,y:A} x = y \to \mathbf{Type}). \\ (\Pi_{x:A} M_{x\,x}\, \mathrm{refl}_x) \qquad \to \Pi_{x,y:A} \Pi p : x = y . M\, x\, y\, p \end{aligned}$$

$\mathrm{elim}^{=}\, M\, m\, \mathrm{refl}_x :\equiv m_x$

[6] Just in case my clever colleagues read this and shake their heads: This is not completely true if we want to get the same definitional equalities as we get from the naive definition. I will discuss this particular can of worms in the final chapter.

Exercises for Naive Type Theory
Part 4

Thorsten Altenkirch

April 14, 2016

1. Given the definition of $\mathrm{T}ree : \mathbf{Type}$ by the following constructor:

   $\mathrm{mkTree} : \Pi I : \mathbf{Type}.(I \to \mathrm{Tree}) \to \mathrm{Tree}$

   Implement the following axioms of set theory as operations on trees:

   (a) Empty set axiom

   $\exists x.\forall y.\neg(y \in x)$

   (b) Axiom of pairing

   $\forall x, y.\exists z.x \in z \wedge y \in z$

   (c) Axiom of union

   $\forall x.\exists y.(\forall z.z \in x.\forall w.w \in z \Rightarrow w \in y)$

   (d) Axiom of infinity

   $\exists x.\emptyset \in x \wedge \forall y.y \in x \Rightarrow y \cup \{y\} \in x$

   (e) Powerset axiom

   $\forall x.\exists y.\forall z.z \subseteq x \Rightarrow z \in y$

2. Define the function

   $\min : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

   which computes the minimum of two natural numbers first using pattern matching style and then only using

   $\mathrm{rec}_A : A \to (\mathbb{N} \to A \to A) \to \mathbb{N} \to A$

$$\begin{aligned} \mathrm{rec}\, z\, s\, 0 &:\equiv z \\ \mathrm{rec}\, z\, s\, (\mathrm{suc}\, m) &:\equiv s\, n\, (\mathrm{rec}\, z\, s\, m) \end{aligned}$$

Prove that

$\forall x, y : \mathbb{N}. \mathrm{min}\, x\, y = \mathrm{min}\, y\, x$

using propositions as types. First give a proof using pattern matching style, then give a proof only using the eliminator:

$\mathrm{elim} : \Pi A : \mathbb{N} \to \mathbf{Type}. A\, 0 \to (\Pi n : \mathbb{N}. A\, n \to A\, (\mathrm{suc}\, n)) \to \Pi n : \mathbb{N}. A\, n$

$$\begin{aligned} \mathrm{elim}\, A\, z\, s\, 0 &:\equiv z \\ \mathrm{elim}\, A\, z\, s\, (\mathrm{suc}\, n) &:\equiv s\, n\, (\mathrm{elim}\, A\, z\, s\, n) \end{aligned}$$

3. We define the dependent type

   $\mathrm{In} : A \to \mathrm{List}\, A \to \mathbf{Type}$

   as given by the constructors:

   $$\begin{aligned} \mathrm{here} &: \Pi a : A. \Pi l : \mathrm{List}\, A. \mathrm{In}\, a\, (a :: l) \\ \mathrm{later} &: \Pi a, b : A. \Pi l : \mathrm{List}\, A. \mathrm{In}\, a\, l \to \mathrm{In}\, a\, (b :: l) \end{aligned}$$

   Derive the recursor (non dependent eliminator) and the dependent eliminator for this In.