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 plus : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with 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 plus : $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ as plus $:\equiv \lambda x.\lambda y.x + y$. plus is a function which applied to a natural number returns a function. That is 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 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 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 plus 3 4. Here another convention comes into play namely that we don't have to write this as (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:

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