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 $\text{nth}\,[1,2,3]\,3$ because it cannot be well typed. Either we choose $\text{nth}_3\,[1,2,3]\,3$ which is not well typed because 3 is not an element of $\text{Fin}\,3$ or $\text{nth}_4\,[1,2,3]\,3$ which is not well typed because $[1,2,3]$ is not an element of $\text{Vec}\,\mathbb{N}\,4$.

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

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

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

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

That is $\text{Fin}\,(1+n)$ constains a new element 0_n and all the elements $i : \text{Fin n}$ with a suc_n in front of them : $\text{suc}_n\,i$. 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:

$$[] : \text{Vec}\,A\,0$$
$$-::- : \Pi_{n:\mathbb{N}}A \to \text{Vec}\,A\,n \to \text{Vec}\,A\,(1+n)$$

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

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

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

$$\text{nth}\,(a :: v)\,0 :\equiv a$$
$$\text{nth}\,(a :: v)\,(1+n) :\equiv \text{nth}\,v\,n$$

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

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

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:

$$A + B \equiv \Sigma b : \mathrm{Bool}.\mathrm{if}\,b\,\mathtt{then}\,A\mathrm{else}\,B$$
$$A \times B \equiv \Pi b : \mathrm{Bool}.\mathrm{if}\,b\,\mathtt{then}\,A\mathrm{else}\,B$$

[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* $\mathrm{List}\, A$. But what is List if not a function on types, that is $\mathrm{List} : \mathbf{Type} \to \mathbf{Type}$. Here we introduce **Type** as a type of types and this sort of thing is called a *universe* in Type Theory. So for example $\mathbb{N} : \mathbf{Type}$ 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 $\mathrm{Fin} : \mathbb{N} \to \mathbf{Type}$ and $\mathrm{Vec} : \mathbb{N} \to \mathbf{Type} \to \mathbf{Type}$. 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 $\mathbf{Type} : \mathbf{Type}$. 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 $\mathbf{Type} : \mathbf{Type}$. This time it