# 4 Dependent types

By a dependent type we mean a type indexed by elements of another type. For example the types of $n$-tuples $A^n : \mathbf{Type}$ their elements are $(a_0, a_1, \ldots a_{n-1}) : A^n$ where $a_i : A$, or the finite type $\overline{n} : \mathbf{Type}$. Indeed, tuples are also indexed by $A : \mathbf{Type}$. We can use functions into $\mathbf{Type}$ to represent these dependent types:

$$\begin{aligned} &\mathrm{Vec} : \mathbf{Type} \to \mathbb{N} \to \mathbf{Type} \\ &\mathrm{Vec}\, A\, n :\equiv A^n \end{aligned}$$

$$\begin{aligned} &\mathrm{Fin} : \mathbb{N} \to \mathbf{Type} \\ &\mathrm{Fin}\, n :\equiv \overline{n} \end{aligned}$$

In the propositions as types view dependent types are used to represent predicates, e.g. $\mathrm{Prime} : \mathbb{N} \to \mathbf{Type}$ assigns to any natural number $n : \mathbb{N}$ the type of evidence $\mathrm{Prime}\, n : \mathbf{Type}$ that $n$ is a prime number. This does not need to be inhabited, e.g $\mathrm{Prime}\, 4$ is empty. Using currying we can use this also to represent relations, e.g. $\leq : \mathbb{N} \to \mathbb{N} \to \mathbf{Type}$, $m \leq n : \mathbf{Type}$ is the type of evidence that $m$ is less or equal to $n$.

## 4.1 $\Pi$-types and $\Sigma$-types

$\Pi$-types generalize function types to allow the codomain of a function to depend on the domain. For example consider the function zeroes that assigns to any natural number $n : \mathbb{N}$ a vector of $n$ zeroes

$$\underbrace{(0, 0, \ldots, 0)}_{n} : \mathbb{N}^n$$

We use $\Pi$ to write such a type:

$$\begin{aligned} &\mathrm{zeroes} : \Pi n : \mathbb{N}.\mathbb{N}^n \\ &\mathrm{zeroes}\, n :\equiv \underbrace{(0, 0, \ldots, 0)}_{n} \end{aligned}$$

The non-dependent function type can now be understood as a special case of $\Pi$-types, $A \to B \equiv \Pi - : A.B$.

In the same vain, $\Sigma$-types generalize product types to the case when the 2nd component depends on the first. For example we can represent tuples of arbitrary size as a pair of a natural number $n : \mathbb{N}$ and a vector of this size $A^n$ as an element of $\Sigma n : \mathbb{N}.A^n$. So for example $(3, (1, 2, 3)) : \Sigma n : \mathbb{N}.\mathbb{N}^n$ because $(1, 2, 3) : \mathbb{N}^3$. Indeed, this type is very useful so we give it a name:

$$\begin{aligned} &\mathrm{List} : \mathbf{Type} \to \mathbf{Type} \\ &\mathrm{List}\, A :\equiv \Sigma n : \mathbb{N}.A^n \end{aligned}$$

In the propositions as types translation we use Π-types to represent evidence for universal quantification. For example a proof of $\forall x : \mathbb{N}.1 + x = x + 1$ is a dependent function of type $f : \Pi x : \mathbb{N}.1 + x = x + 1$, such that applying it as in $f\,3$ is evidence that $1 + 3 = 3 + 1$.

Similar, we use Σ-types to represent evidence for existential quantification where the first component is the instance for which the property is supposed to hold and the second component a proof that it holds for this particular instance. For example the statement $\exists n : \mathbb{N}.\mathrm{Prime}\,n$ is translated to $\Sigma n : \mathbb{N}.\mathrm{Prime}\,n$ and a proof of this is $(3, p)$ where $p : \mathrm{Prime}\,3$.

As for Π-types the non-dependent products arise as a special case of Σ-types: $A \times B :\equiv \Sigma\_ : A.B$.

To avoid clutter we sometimes want to omit arguments to a Π-type when it is derivable from later arguments or the first component of a Σ-type. In this case we write the argument in subscript as in $\Pi_{x:A} B\,x$ or $\Sigma_{x:A} B\,x$. For example if we define $\mathrm{List}\,A :\equiv \Sigma_{n:\mathbb{N}}.A^n$ we can omit the length and just write $(1, 2, 3) : \mathrm{List}\,\mathbb{N}$.

We can also define a dependent recursor or eliminator for Σ-types which allows us to define any dependent function out of a Σ-type. This eliminator is not just parametrized by a type but by a family $C : \Sigma x : A.B\,x \to \mathbf{Type}$:

$$\mathrm{E}^{\Sigma} : (\Pi x : A, \Pi y : B\,x.C\,(x, y)) \to \Pi p : \Sigma x : A.B\,x.C\,p$$
$$\mathrm{E}^{\Sigma}\,f\,(a, b) :\equiv f\,a\,b$$

**Exercise 5** *As in exercise 3 we can define projections out of a Σ-type, let* $A : \mathbf{Type}$ *and* $B : A \to \mathbf{Type}$*:*

$$\mathrm{fst} : \Sigma x : A.B\,x \to A$$
$$\mathrm{fst}\,(a, b) :\equiv a$$
$$\mathrm{snd} : \Pi p : \Sigma x : A.B\,x.B\,(\mathrm{fst}\,p)$$
$$\mathrm{snd}\,(a, b) :\equiv b$$

*Note that the type of the 2nd projections is a dependent function type using the first projection.*

*Derive the projections using only the eliminator* $\mathrm{E}^{\Sigma}$*. Vice versa, can you derive the eliminator from the projections without making further assumptions?*

**Exercise 6** *Using the propositions as types translation for predicate logic try to derive the following tautologies:*

1. $(\forall x : A.P\,x \wedge Q\,x) \Leftrightarrow (\forall x : A.P\,x) \wedge (\forall x : A.Q\,x)$
2. $(\exists x : A.P\,x \vee Q\,x) \Leftrightarrow (\exists x : A.P\,x) \vee (\exists x : A.Q\,x)$
3. $(\exists x : A.P\,x) \implies R \Leftrightarrow \Pi x : A.P\,x \implies R$
4. $\neg\exists x : A.P\,x \Leftrightarrow \forall x : A.\neg P\,x$
5. $\neg\forall x : A.P\,x \Leftrightarrow \exists x : A.\neg P\,x$

*where* $A, B : \mathbf{Type}$ *and* $P, Q : A \to \mathbf{Type}$, $R : \mathbf{Type}$, *represent predicates and a proposition.*

We have seen that $\Sigma$-types are related to products but they are also related to sums. Indeed we can derive $+$ from $\Sigma$ using as the first component an element of $\mathrm{Bool} = \overline{2}$ and the second component is either the first or the 2nd component of the sum (assuming $A, B : \mathbf{Type}$:

$$\begin{aligned} &A + B : \mathbf{Type} \\ &A + B :\equiv \Sigma x : \mathrm{Bool}.\mathrm{if}\, x\, \mathrm{then}\, A\, \mathrm{else}\, B \end{aligned}$$

In the same way we can also derive $\times$ from $\Pi$ by using dependent functions over the booleans which returns either one or the other component of the product.

$$\begin{aligned} &A \times B : \mathbf{Type} \\ &A \times B :\equiv \Pi x : \mathrm{Bool}.\mathrm{if}\, x\, \mathrm{then}\, A\, \mathrm{else}\, B \end{aligned}$$

It is interesting to note that $\times$ can be viewed in two different ways: either as a non-dependent $\Sigma$-type or as a dependent function-type over the booleans.

**Exercise 7** *Show that injections, pairing, non-dependent eliminators can be derived for these encodings of sums and products.*

Finally, we notice that the arithmetic interpretation of types extends to $\Sigma$ and $\Pi$ giving a good justification for the choice of their names, let $m : \mathbb{N}$ and $f : \overline{m} \to \mathbb{N}$:

$$\begin{aligned} \overline{\Sigma x : m.f\, x} &= \Sigma_{x=0}^{x<\overline{m}} f\, x \\ \overline{\Pi x : m.f\, x} &= \Pi_{x=0}^{x<\overline{m}} f\, x \end{aligned}$$

## 4.2 Induction and recursion

Following Peano the natural numbers are introduced by saying that 0 is a natural number ($0 : \mathbb{N}$), and if $n$ is a natural number ($n : \mathbb{N}$) then $\mathrm{suc}\, n$ is a natural number ($\mathrm{suc}\, n$), which is equivalent to saying $\mathrm{suc} : \mathbb{N} \to \mathbb{N}$. When defining a function out of the natural numbers, we allow ourselves to recursively use the function value on $n$ to compute it for $\mathrm{suc}\, n$. An example is the doubling function:

$$\begin{aligned} &\mathrm{double} : \mathbb{N} \to \mathbb{N} \\ &\mathrm{double}\, 0 :\equiv 0 \\ &\mathrm{double}\, (\mathrm{suc}\, n) :\equiv \mathrm{suc}\, (\mathrm{suc}\, (\mathrm{double}\, n)) \end{aligned}$$

We can distill this idea into a non-dependent eliminator which is now rightfully called the recursor:

$$\begin{aligned}
&\mathrm{R}^{\mathbb{N}} : C \to (C \to C) \to \mathbb{N} \to C \\
&\mathrm{R}^{\mathbb{N}}\, z\, s\, 0 :\equiv z \\
&\mathrm{R}^{\mathbb{N}}\, z\, s\, (\mathrm{suc}\, n) :\equiv s\, (\mathrm{R}\, z\, s\, n)
\end{aligned}$$

**Exercise 8** *We define addition recursively:*

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

*Define addition using only the recursor* $\mathrm{R}^{\mathbb{N}}$*.*

**Exercise 9** *Not all recursive functions exactly fit into this scheme. For example consider the function that halves a number forgetting the remainder:*

$$\begin{aligned}
&\mathrm{half} : \mathbb{N} \to \mathbb{N} \\
&\mathrm{half}\, 0 :\equiv 0 \\
&\mathrm{half}\, (\mathrm{suc}\, 0) :\equiv 0 \\
&\mathrm{half}\, (\mathrm{suc}\, (\mathrm{suc}\, n)) :\equiv \mathrm{suc}\, (\mathrm{half}\, n)
\end{aligned}$$

*Try to derive* half *only using the recursor* $\mathrm{R}^{\mathbb{N}}$*.*

When we want to prove a statement about natural numbers we have to construct a dependent function. An example is the proof that half is the left inverse of double: $\forall n : \mathbb{N}.\mathrm{half}\, (\mathrm{double}\, n) = n$. I haven't introduced equality yet but we only need two ingredients to carry out this construction, given $A, B$ : **Type**:

$$\begin{aligned}
&\mathrm{refl} : \Pi x : A.x = x \\
&\mathrm{resp} : \Pi f : A \to B.\Pi_{m,n:\mathbb{N}} m = n \to f\, m = f\, n
\end{aligned}$$

Using those we can define a dependent function verifying the statement:

$$\begin{aligned}
&h : \Pi n : \mathbb{N}.\mathrm{half}\, (\mathrm{double}\, n) = n \\
&h\, 0 :\equiv \mathrm{refl}\, 0 \\
&h\, (\mathrm{suc}\, n) :\equiv \mathrm{resp}\, suc\, (h\, n)
\end{aligned}$$

As for $\Sigma$-types we can derive dependent functions out of the natural numbers using a dependent recursor or eliminator. Assume that we have a dependent type $C : \mathbb{N} \to$ **Type**:

$$\begin{aligned}
&\mathrm{E}^{\mathbb{N}} : C\, 0 \to (\Pi n : \mathbb{N}.C\, n \to C\, (\mathrm{suc}\, n)) \to \Pi n : \mathbb{N}.C\, n \\
&\mathrm{E}^{\mathbb{N}}\, z\, s\, 0 :\equiv z \\
&\mathrm{E}^{\mathbb{N}}\, z\, s\, (\mathrm{suc}\, n) :\equiv s\, (\mathrm{E}\, z\, s\, n)
\end{aligned}$$

**Exercise 10** *Derive $h$ using only the dependent eliminator* $\mathrm{E}^{\mathbb{N}}$.

The type of $\mathrm{E}^{\mathbb{N}}$ precisely corresponds to the principle of induction - indeed from the propositions as types point of view induction is just dependent recursion.

**Exercise 11** *Show that the natural numbers with* $+$ *and* $0$ *form a commutative monoid:*

1. $\forall x : \mathbb{N}.0 + x = x$
2. $\forall x : \mathbb{N}.x + 0 = x$
3. $\forall x, y, z : \mathbb{N}.x + (y + z) = (x + y) + z$
4. $\forall x, y : \mathbb{N}.x + y = y + x$

Not all dependent functions out of the natural numbers arise from the propositions as types translation. An example is the function zeroes : $\Pi n : \mathbb{N}.\mathbb{N}^n$ which we only introduced informally. We can make this precise by inductively defining tuples:

$$\begin{aligned} &\mathrm{nil} : A^0 \\ &\mathrm{cons} : \Pi_{n:\mathbb{N}} A^n \to A^{\mathrm{suc}\, n} \end{aligned}$$

Using this we can define zeroes by recursion

$$\begin{aligned} &\mathrm{zeroes}\, 0 :\equiv \mathrm{nil} \\ &\mathrm{zeroes}\, (\mathrm{suc}\, n) :\equiv \mathrm{cons}\, 0\, (\mathrm{zeroes}\, n) \end{aligned}$$

This can be easily translated into an application of the eliminator

$$\mathrm{zeroes} :\equiv \mathrm{E}^{\mathbb{N}}\, \mathrm{nil}\, (\mathrm{cons}\, 0)$$

**Exercise 12** *We can also define the finite types in an inductive way, overloading* $0$ *and* suc*:*

$$\begin{aligned} &0 : \Pi_{n:\mathbb{N}} \overline{\mathrm{suc}\, n} \\ &\mathrm{suc} : \Pi_{n:\mathbb{N}} \overline{m} \to \overline{\mathrm{suc}\, n} \end{aligned}$$

*Using this and the inductive definition of* $A^n$ *derive a general projection operator*

$$\mathrm{nth} : \Pi_{n:\mathbb{N}} A^n \to \overline{n} \to A$$

*that extracts an arbitrary component of a tuple.*

**Exercise 13** *Suggest definitions of eliminators for tuples and finite types. Can you derive all the examples using them?*

## 4.3 The equality type

Given $a, b : A$ the equality type $a =_A b : \mathbf{Type}$ is generated from one constructor $\mathrm{refl} : \Pi_{x:A} x =_A x$. That is we are saying that two things which are identical are equal and this is the only way to construct an equality. Using this idea we can establish some basic properties of equality, namely that it is an equivalence relation, that is a relation that is reflexive, symmetric and transitive. Moreover, it is also a congruence, it is preserved by all functions. Since we already have reflexivity from the definition, lets look at symmetry first. We can define symmetry by just saying how it acts on reflexivity:

$$\begin{aligned} &\mathrm{sym} : \Pi_{x,y:A} x = y \to y = x \\ &\mathrm{sym}\,\mathrm{refl}_a :\equiv \mathrm{refl}_a \end{aligned}$$

The main idea here is that once we apply sym to refl we also know that tht wto points $x, y : A$ must be identical and hence we can prove the result using refl again.

**Exercise 14** *Provide proofs of*

$$\begin{aligned} &\mathrm{trans} : \Pi_{x,y,z:A} x = y \to y = z \to x = z \\ &\mathrm{resp} : \Pi f : A \to B.\Pi_{m,n:\mathbb{N}} m = n \to f\, m = f\, n \end{aligned}$$

*using the same idea.*

For equality there is a recursor and an eliminator, and for the examples above we only need the recursor because we have no dependency on the actual proofs of equality. However, there is some dependency because equality itself is a dependent type. We assume a family that depends on the indices of equality but not on the equality proofs themselves: $C : A \to A \to \mathbf{Type}$ then the recursor is:

$$\begin{aligned} \mathrm{R}^{=} : (\Pi x : A.C\, x\, x) \to \Pi_{x,y:A} x = y \to C\, x\, y \\ \mathrm{R}^{=}\, f\, (\mathrm{refl}_a) :\equiv f\, a \end{aligned}$$

**Exercise 15** *Derive* $\mathrm{sym}, \mathrm{trans}, \mathrm{resp}$ *using the recursor* $\mathrm{R}^{=}$.

What would be a statement that actually depends on equality proofs? It seems that equality is rather trivial since there s at most one proof of it and we should be able to prove this. This is called uniqueness of equality proofs and states that any two proofs of equality are equal and it has an easy direct definition exploiting exactly the fact that the only proof of equality is reflexivity:

$$\begin{aligned} &\mathrm{uep} : \Pi_{x,y:A} \Pi p, q : x = y \\ &\mathrm{uep}\,\mathrm{refl}_a\,\mathrm{refl}_a :\equiv \mathrm{refl}_{\mathrm{refl}_a} \end{aligned}$$

We now define the dependent eliminator for equality which $\mathrm{E}^{=}$ which is also called $J$ but we stick to our terminology. This time we use a family that does

indeed depend on the equality proof $C : \Pi_{x,y:A} x = y \to \mathbf{Type}$

$$\begin{aligned} &\mathrm{E}^{=} : (\Pi_{x:A} C\,(\mathrm{refl}_x)) \to \Pi_{x,y:A} \Pi p : x = y . C\,p \\ &\mathrm{E}^{=}\, f\, \mathrm{refl}_x :\equiv f_x \end{aligned}$$

Now we should be able to perform the usual exercise and reduce the direct definition of uep to one using only the eliminator. The first step is clear, by eliminating one argument we can reduce the problem to:

$$\Pi_{x:A} \Pi q : x = x . \mathrm{refl}_x = q$$

but now we are stuck. We cannot apply the eliminator because we need a family were both indices are arbitrary. Indeed, Hofmann and Streicher were able to show that uep is unprovable from the eliminator. In the next section we will discuss reasons why this is actually not a bad thing. However, at least based on our current understanding of equality it seems that this is an unwanted incompleteness. One which can actually fixed by introducing a special eliminator which works exactly in the case when we want to prove something about equality proofs where both indices are equal. That is we assume as given a family $C : \Pi_{x:A} x = x \to \mathbf{Type}$ and introduce

$$\begin{aligned} &K : (\Pi_{x:A} C\,\mathrm{refl}_x) \to \Pi_{x:A} \Pi p : x = x . C\,p \\ &K\, f\, \mathrm{refl}_x :\equiv f_x \end{aligned}$$

This eliminator is called $K$ because K is the next letter after J.

**Exercise 16** *Derive uep using only* $\mathrm{E}^{=}$ *and* $K$*.*

**Exercise 17** *Instead of viewing equality as a relation generated by* refl*, we can also fix one index* $a : A$ *and now define the predicate of being equal to* $a$*:* $a = - : A \to \mathbf{Type}$*. This predicate is generated by* $\mathrm{refl}_a : a = a$ *so no change here. However, the eliminator looks different. Let's fix* $C : \Pi x : A . a = x \to \mathbf{Type}$

$$\begin{aligned} &J'_a : (C\,\mathrm{refl}_a) \to \Pi_{x:A} \Pi p : a = x . C\,p \\ &J'_a\, f\, (\mathrm{refl}_a) :\equiv f \end{aligned}$$

*Show that* $\mathrm{E}^{=}$ *and* $J'$ *are interderivable, that is both views of equality are equivalent.*