Chapter 5

Dependent types

We are now getting to one of the main innovations of Type Theory: dependent types. In general a dependent type is a function with codomain Set. Examples are vectors Vec A : ℕ → Set where Vec A n is a sequence elements of A of length n, and Fin : ℕ → Set where Fin n corresponds to numbers less then n.

Actually we have already seen dependent types. For example List : Set → Set is a dependent type indexed by Set. Hence *Polymorphism* is actually subsumed by dependent types! Dependent types can be indexed by several indices, and indeed Vec : Set → ℕ → Set if we spell out all the dependencies.

We will discuss Π-types which generalize function types and Σ-types which generalize products. There are a number of interesting relations between dependent type formers and simply typed ones and we will also extend the *type arithmetic* from 2.5.4 to dependent types.

5.1 Vectors and dependent functions

We are going to look at some examples for dependent functions using vectors. I will postpone the actual definition of vectors until I have explained some syntactic conventions. Vec A n is the type of sequences of elements of A of length n, it is a refinement of the type of lists and we use the same constructors. For example 1 :: 2 :: 3 :: [] : Vec ℕ 3.

We can define a function that produces a list of 0s of length n a list:

```
zeroes-l : ℕ → List ℕ
zeroes-l zero    = []
zeroes-l (suc n) = 0 :: zeroes-l n
```

E.g. zeroes-l 3 = 0 :: 0 :: 0 :: []. Using vectors we can define an analogous function with a more precise type because we know that zeroes-l n will produce a list of length n. So instead we output a vector of the length given by the input.

```
zeroes-v : (n : ℕ) → Vec ℕ n
```

```
zeroes-v zero = []
zeroes-v (suc n) = 0 :: (zeroes-v n)
```

We see that now the function type is *dependent* and the function type constructor actually introduces a variable.

In this example we can also see that we can often turn non-dependent functions into dependent ones with a more precise type. Let's look at another example: append. We have already seen the append function on lists which was defined infix but for the purpose of the current discussion I am going to redefine it in prefix form:

```
append-l : List A → List A → List A
append-l []       ys = ys
append-l (x :: xs) ys = x :: (append-l xs ys)
```

We can define a very similar operation on vectors, which we call append-v which appends two vectors and records in its type that the length of the result is the sum of its inputs:

```
append-v : (i j : ℕ) → Vec A i → Vec A j → Vec A (i + j)
append-v zero    j [] w       = w
append-v (suc i) j (x :: v) w = x :: (append-v i j v w)
```

Here the type (i j : ℕ) → ... is just a shorthand for (i : ℕ) → (j : ℕ) → .... Let's define some vectors

```
v1 : Vec ℕ 2
v1 = 1 :: 2 :: []
v2 : Vec ℕ 3
v2 = 3 :: 4 :: 5 :: []
```

and append them using append-v:

```
append-v 2 3 v1 v2 : Vec ℕ 5
```

using that 2 + 3 = 5. And this term will evaluate to

```
1 :: 2 :: 3 :: 4 :: 5 :: [] : Vec ℕ 5
```

Already at this point we note that always having to supply the indices for an operation like append-v will clutter up the code and potentially make it unreadable. However, it is also completely unnecessary because we can deduce them from the subsequent arguments.

To avoid this clutter dependently typed languages have introduced the notion of *implicit arguments* which are arguments which are filled in automatically by the type checker. To make an argument implicit we replace (x : A) → ... by {x : A} → .... Hence we can write

```
append-v : {i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
```

and when applying append-v we just write

```
append-v v1 v2 : Vec ℕ 5
```

and the type checker will insert the hidden arguments 2 and 3 which it can infer from the later arguments. We can also make implicit arguments explicit again by using curly brackets again:

```
append-v {2} {3} v1 v2 : Vec ℕ 5
```

Actually using implicit parameters we can also switch to the more readable infix definition of append for vectors resembling our original definition for lists:

```
_++v_ : {i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
[]       ++v w = w
(x :: v) ++v w = x :: (v ++v w)
```

We have already seen implicit arguments when looking at polymorphic functions like length : List A → ℕ which exploited our convention that A is a set parameter. Agda will expand this to

```
length : {A : Set} → List A → ℕ
```

and in an application like length (1 :: 2 :: 3 :: []) the implicit parameter ℕ is inserted, i.e. this expression *elaborates* to length {ℕ} (1 :: 2 :: 3 :: []).

We see that not only are datatypes like List : Set → Set are actually dependent types but also the mystery of polymorphic functions can be subsumed by dependent function types without much notational overhead.

And indeed the constructor _::_ for lists already has an implicit parameter namely:

```
_::_ : {A : Set} → A → List A → List A
```

and hiding it enables us to use the customary infix notation.

After having explained this syntactic convention, I can reveal the actual definition of vectors:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

This is an inductive definition, very similar to the inductive definition of lists. As before A : Set is a parameter but we define not a Set but a dependent type ℕ → Set. The constructors create elements for different instances of Vec A.

We can also see that the overloaded [1] _::_ for vectors has actually two implicit arguments:

[1] Agda allows the overloading of constructors since it can determine which actual constructor is meant from the type.

```
_::_ : {A : Set} {n : ℕ} → A → Vec A n → Vec A (suc n)
```

You may notice that the dependency on Set and on the ℕ in the declaration of Vec looks different. This is partly convenience since A is the same in the whole declaration but in this case it also has an impact on keeping Vec A n in Set but this is something which we will discuss later in more detail.

5.2 The family of finite sets

Using dependent types we can avoid run-time errors such as the typical *index out of range error* we get when accessing an array. As a simple example lets look at the indexing operation for lists. We would like to implement an operation:

```
_!!_ : List A → ℕ → A
```

but while we can fill in the cases where the list is a cons we get stuck on the case where it is empty:

```
_!!_ : List A → ℕ → A
[] !! n       = ?
(x :: xs) !! zero  = x
(x :: xs) !! suc n = xs !! n
```

Indeed there is no way to complete this program since A could be empty.

We can handle the error explicitly by using Maybe:

```
_!!_ : {A : Set} → List A → ℕ → Maybe A
[] !! n       = nothing
(x :: as) !! zero  = just x
(x :: as) !! suc n = as !! n
```

This is ok and we can use the fact that Maybe is a monad to write programs with error handling. However, we still may get an error anywhere in the middle of a computation.

Wouldn't it better, if we could show statically that no such error can occur? And indeed we can use vectors for this purpose. The other ingredient is Fin which provides us with the allowed index-range of a vector. As Vec was a refinement of List, Fin is a refinement of ℕ:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} → Fin n → Fin (suc n)
```

The idea is that every non-empty finite type has a zero and that the elements of Fin n get embedded into Fin (suc n) using suc for Fin. That is the first levels

of Fin look like this

$$\begin{aligned}
&\mathsf{Fin}\ 0 = \{\}\\
&\mathsf{Fin}\ 1 = \{\mathsf{zero}\ \{0\}\}\\
&\mathsf{Fin}\ 2 = \{\mathsf{zero}\ \{1\}, \mathsf{suc}\ \{1\}\ (\mathsf{zero}\ \{0\})\}\\
&\mathsf{Fin}\ 3 = \{\mathsf{zero}\ \{2\}, \mathsf{suc}\ \{2\}\ (\mathsf{zero}\ \{1\}), \mathsf{suc}\ \{2\}\ (\mathsf{suc}\ \{1\}\ (\mathsf{zero}\ \{0\}))\}\\
&\quad\vdots\ \ \vdots
\end{aligned}$$

Here it is actually helpful to use the convention to make the implicit arguments explicit.

Now we can use Vec and Fin to write a safe version of the lookup function:

```
_!!v_ : {A : Set} {n : ℕ} → Vec A n → Fin n → A
(x :: v) !!v zero = x
(x :: v) !!v suc i = v !!v i
```

Using _!!v_ we can access

```
v1 !!v (suc zero) = 2
```

but

```
v1 !!v (suc (suc (suc zero)))
```

isn't well-typed.

5.3 Π-types and Σ-types

In the previous section we have encountered dependent function types which are also called Π-types. [2] In general we have a type A : Set and a dependent type B : A → Set and we can form (x : A) → B x : Set. We can understand the simple function type A → B as a special case where the 2nd type doesn't actually depends on the first and hence we omit the binding.

```
A → B = (_ : A) → B
```

In all other aspects dependent function types behave very much like simple function types. The application rule now reads that given f : (x : A) → B and a : A we obtain f a : B a. We can define dependent function either by an explicit definition or using λ-abstraction.

Previously, after introducing _→_ we defined products _×_. And indeed as Π types generalize → we also have a dependent generalisation of ×, the Σ-type. As for _×_ this is just an instance of a record type:

[2] I will explain later in this chapter where the Π comes from.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

Elements of Σ A B are pairs a , b where the 1st component is a : A and the 2nd component is b : B a, that is the type of the 2nd component depends on the 1st. Correspondingly we have the projections

```
proj₁ : Σ A B → A
proj₂ : (a : Σ A B) → B (proj₁ a)
```

Note that the type of the second projection is a dependent function type. To improve readability I am introducing a special syntax for Σ-types and write Σ[x ∈ A] P for Σ A (λ x → P). [3]

A simple example would be a type of flexible vectors:

```
FlexVec : Set → Set
FlexVec A = Σ[ n ∈ ℕ ] Vec A n
```

An element of FlexVec A is a pair (n , v) where n : ℕ and v : Vec A n. It isn't hard to see that FlexVec A is equivalent to List A and indeed we can implement the standard functions on lists:

```
[]fv : FlexVec A
[]fv = 0 , []
_::fv_ : A → FlexVec A → FlexVec A
a ::fv (n , v) = suc n , (a :: v)
```

I leave it as an exercise to derive ItList for FlexVec.

As before for function types products are a special case of Σ-types:

```
A × B = Σ[ _ ∈ A ] B
```

However, there is also a different relation we shall explore.

5.4 Relating simple and dependent type formers

You may have noticed that we haven't got a dependent version of sums _⊎_. However, doesn't the use of the symbol Σ indicate that Σ-types should be related to sums and not products?

And indeed we can derive a binary sums from Σ-types and if_then_else_. [4]

[3] I am using ∈ here because Agda doesn't allow me to use : in abbrevations.

[4] if_then_else_ was defined in section 3.6, but only for Set. Here I need a *level polymorphic* one to return sets. I will explain this later.

```
_⊎'_ : Set → Set → Set
A ⊎' B = Σ[ x ∈ Bool ] if x then A else B
```

The idea is that an element of $A \uplus' B$ is a pair (b , x) consisting of a boolean b representing the choice between the two alternatives and then depending on this choice an element of either A or B. Given this idea it is straightforward to derive the injections:

```
inj₁' : A → A ⊎' B
inj₁' a = true , a
inj₂' : B → A ⊎' B
inj₂' b = false , b
```

We can also derive case using pattern matching:

```
case' : (A → C) → (B → C) → A ⊎' B → C
case' f g (false , b) = g b
case' f g (true , a)  = f a
```

In the same vain we can derive _×_ from Π:

```
_×'_ : Set → Set → Set
A ×' B = (x : Bool) → if x then A else B
```

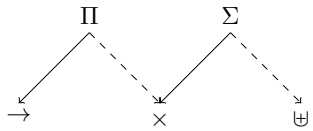
An element of A ×' B is simply a dependent function from booleans to either A or B depending on the input. Hence, the projections are realised by application:

```
proj₁' : A ×' B → A
proj₁' f = f true
proj₂' : A ×' B → B
proj₂' f = f false
```

The pairing constructor introduces a dependent function by pattern matching:

```
_,'_ : A → B → A ×' B
(a ,' b) false = b
(a ,' b) true  = a
```

We can relate all the basic type formers in one picture:



Here → goes from the dependent to the non-dependent version while -‣ goes from the indexed version to the binary one as discussed above.

You may notice that there are two ways to derive ×: either is the non-dependent version of Σ or as the binary case of Π. This corresponds to the

fact that $\times$ can be viewed inductively – if derived from Σ – or coinductively, if derived from Π.

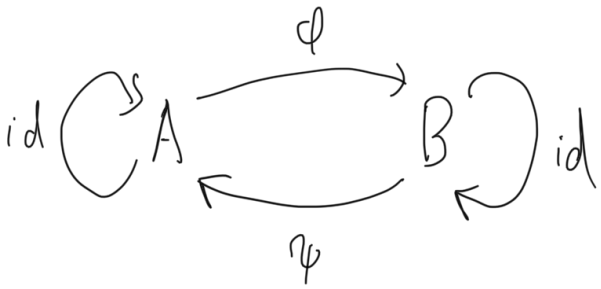
This also explains the use of symbols: Π-types are really infinite products and Σ-types are infinite sums.

5.5 Arithmetic of types

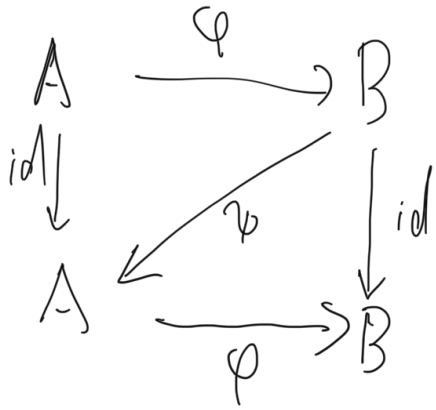
In section 2.5.4 we observed that there is a nice connection between operation on finite types and operations on numbers. Using Fin we can make this relation more precise:

$$\begin{aligned} \mathsf{Fin\ m} \uplus \mathsf{Fin\ n} &\cong \mathsf{Fin\ (m + n)} \\ \mathsf{Fin\ m} \times \mathsf{Fin\ n} &\cong \mathsf{Fin\ (m * n)} \\ \mathsf{Fin\ m} \to \mathsf{Fin\ n} &\cong \mathsf{Fin\ (n \uparrow m)} \end{aligned}$$

By $\mathsf{A} \cong \mathsf{B}$ we mean *A and B have the same number of elements*. This can be made precises by saying that we have functions $\phi : \mathsf{A} \to \mathsf{B}$ and $\psi : \mathsf{B} \to \mathsf{A}$ which are inverse to each other, that is $\phi \circ \psi = \mathsf{id}$ and $\phi \circ \psi = \mathsf{id}$. I usually draw the following categorical diagram to depict this situation:



But I realized that it is not clear what compositions should commute. A better diagram is:



In any case since I still haven't introduced equality types [5] we will limit ourselves to just talking about the finctions in both direction. Hence I define:

```
record _~_ (A B : Set) : Set where
  field
    φ : A → B
    ψ : B → A
```

As an example we can give translations for Bool ∼ Fin 2.

```
bool~2 : Bool ∼ Fin 2
φ bool~2 false = zero
φ bool~2 true = suc zero
ψ bool~2 zero = false
ψ bool~2 (suc zero) = true
```

It should be clear that these two functions are indeed inverse to each other.

As an example I construct the first equivalence:

```
sum-eq : {m n : ℕ} → (Fin m ⊎ Fin n) ∼ Fin (m + n)
```

The idea is that we implement the operations for Fin m ⊎ Fin n on Fin (m + n) and then use this for the translation.

The first injection is an embedding which keeps the value the same but changes the index:

```
inj₁f : {m n : ℕ} → Fin m → Fin (m + n)
inj₁f zero = zero
inj₁f (suc i) = suc (inj₁f i)
```

[5] They are coming soon, but the equivalences involving functions realy on functional extensionality which is only provable in cubical type theory.

The second injection has to shift the elements over the first m that is it applies m successors:

```
inj₂f : {m n : ℕ} → Fin n → Fin (m + n)
inj₂f {zero} i = i
inj₂f {suc m} x = suc (inj₂f {m} x)
```

This is also an example where we make a hidden parameter explicit. We didn't really have to supply {m} in the recursive call but writing it here actually increases readability.

We can now use the defined injections as the translations of the actual injections and hence implement the first translation:

```
φ sum-eq (inj₁ x) = inj₁f x
φ sum-eq (inj₂ y) = inj₂f y
ψ sum-eq = ?
```

To implement the inverse direction we need to implement case on Fin (m + n) and then use the original injections depending on the interval we are in.

```
casef : {m n : ℕ} {C : Set} → (Fin m → C) → (Fin n → C)
  → Fin (m + n) → C
casef {zero} f g x = g x
casef {suc m} f g zero = f zero
casef {suc m} f g (suc i) =
  casef {m} (λ j → f (suc j)) g i
```

To achieve this we recur both over the first index m and the input x : Fin (m + n). If m = 0 we know that we are in the 2nd case and we can just apply g. Otherwise suc m we check the input: if it is zero we are certainly in the first case and we just use f zero. Otherwise we recur on m and i but we need to adjust f by composing it with the successor.

Using casef we can complete our translation:

```
φ sum-eq (inj₁ x) = inj₁f x
φ sum-eq (inj₂ y) = inj₂f y
ψ sum-eq = casef inj₁ inj₂
```

We can now extend the equivalence to Π- and Σ-types. In Mathematics we use them to represent products and sums over a finite series (actually also infinite but we won't cover this here):

$$\Sigma_{i=0}^{i<n} a_i = a_0 + a_1 + \cdots + a_{n-1}$$
$$\Pi_{i=0}^{i<n} a_i = a_0 * a_1 * \cdots * a_{n-1}$$

Using finite types we can implement these operations. The sequence a is simply represented as a function Fin n → ℕ. We define

```
Σℕ : (n : ℕ) (a : Fin n → ℕ) → ℕ
Σℕ zero vs    = 0
Σℕ (suc n) a = a zero + Σℕ n (λ i → a (suc i))
```

and using the same pattern:

```
Πℕ : (n : ℕ) (a : Fin n → ℕ) → ℕ
Πℕ zero a = 1
Πℕ (suc n) a = a zero * Πℕ n (λ i → a (suc i))
```

We now have the following equivalences:

$$\begin{array}{lcl} \Sigma[\, i \in \mathsf{Fin}\ n\ ]\ \mathsf{Fin}\ (a\ i)) & \sim & \mathsf{Fin}\ (\Sigma\mathbb{N}\ n\ a) \\ ((i : \mathsf{Fin}\ n) \to \mathsf{Fin}\ (f\ i)) & \sim & \mathsf{Fin}\ (\Pi\mathbb{N}\ n\ f) \end{array}$$

which is one explanation of the use of Π and Σ for the operations on types.

Here is an example to illustrate the equivalence for Σ-types. Let

```
i : ℕ
i = 3
a : Fin i → ℕ
a zero = 3
a (suc zero) = 1
a (suc (suc n)) = 2
```

Now clearly

```
Σℕ i a = 3 + 1 + 2
       = 6
```

The following table shows the equivalence:

0 ↦ (0 , 0)	1 ↦ (0 , 1)	2 ↦ (0 , 2)
3 ↦ (1 , 0)		
4 ↦ (2 , 0)	5 ↦ (2 , 1)	

I leave it as an exercise to construct the equivalences. The idea is the same as I have shown in the example of for _⊎_ we just have to implement the basic operations on the type for the finite representation.

5.6 History

Dependent types were the main innovation which Per Martin-Löf introduced when developing his Type Theory in the early 1970ies [**?**] . Hence it is sometimes called *dependent type theory*. One motivation for dependent types is to extend the proposition as types explanation to predicate logic, which we will study in the next chapter.

Martin-Löf already mentioned the possibility of implementing his theory and this challenge was taken up in the 1980ies by Bob Constable and his group who implemented the NuPRL system, where NuPRL is short for *Nearly ultimate Proof Representation Language* [**?**].

A comprehensive overview over Martin-Löf's Type Theory is given in [**?**] which was coauthored with Giovanni Sambin. This book presents what we now call *extensional type theory* a presentation of *intensional type theory* which is closer to what is implemented in Agda can be found in [**?**].

My own first encounter with dependent types was Randy Pollack's LEGO system [**?**] which was based on Thierry Coquand's calculus of constructions [**?**] which combined dependent types and Girard's System F [**?**]. The calculus of constructions was also the basis of the Coq system [**?**] which has been in development since the 1990ies and which is still very popular.

Apart from using it for proof assistants dependent types are very useful for programming. A relatively early language with dependent types is Cayenne by Lennart Augustsson [**?**]. The Agda system we are using was developed by Ulf Norell [**?**] at Gothenburg. It was influenced by earlier implementations of dependently typed languages in Gothenburg (like the ALF system [**?**]) and got its name from a language developed by Catarina Coquand. It was also influenced by Conor's McBride's Epigram system which famously featured a 2-dimensional syntax [**?**]. Edwin Brady developed Idris emphasising the programming aspect of dependent types [**?**] using type-driven program development.

5.7 Exercises

1. We implement some basic operations on vectors and matrices of natural numbers.

   We define vectors using Vec:

   ```
   Vector : ℕ → Set
   Vector m = Vec ℕ m
   ```

   Here Vector n is an n-dimensional vector. Here are some examples:

   ```
   v1 : Vector 3
   v1 = 1 :: 2 :: 3 :: []
   v2 : Vector 3
   v2 = 2 :: 3 :: 0 :: []
   ```

   Implement the multiplication of a skalar with a vector:

   ```
   _*v_ : {n : ℕ} → ℕ → Vector n → Vector n
   ```

   Here is an example:

   ```
   2 *v v1 = 2 :: 4 :: 6 :: []
   ```

Next we implement vector addition:

```
_+v_ : {n : ℕ} → Vector n → Vector n → Vector n
```

Here is an example:

```
v1 +v v2 = 3 :: 5 :: 3 :: []
```

Next we introduce matrices. A matrix is simply a Vec of row vectors:

```
Matrix : ℕ → ℕ → Set
Matrix m n = Vec (Vector n) m
```

Hence Matrix m n is an m×n-Matrix.

Here are some examples:

```
id3 : Matrix 3 3
id3 = (1 :: 0 :: 0 :: [])
  :: (0 :: 1 :: 0 :: [])
  :: (0 :: 0 :: 1 :: [])
  :: []
inv3 : Matrix 3 3
inv3 = (0 :: 0 :: 1 :: [])
  :: (0 :: 1 :: 0 :: [])
  :: (1 :: 0 :: 0 :: [])
  :: []
m3 : Matrix 3 3
m3 = (1 :: 2 :: 3 :: [])
  :: (4 :: 5 :: 6 :: [])
  :: (7 :: 8 :: 9 :: [])
  :: []
m4 : Matrix 3 2
m4 = (1 :: 2 :: [])
  :: (4 :: 5 :: [])
  :: (7 :: 8 :: [])
  :: []
```

The next task is to define the multiplication of a vector with a matrix:

```
_*vm_ : {m n : ℕ} → Vector m → Matrix m n → Vector n
```

The vector is now viewed as a column vector. Here are some examples:

```
v1 *vm m4 = 30 :: 36 :: []
v1 *vm m3 = 30 :: 36 :: 42 :: []
```

If the input vector is empty you have to produce a vector of 0s with the size given by the matrix. This calls for an axilliary function.

Finally, the task is to implement matrix multiplication. It is recommended to make use of the function we have just defined previously.

```
_*mm_ : {l m n : ℕ} → Matrix l m → Matrix m n → Matrix l n
```

Here are several examples:

```
inv3 *mm m3 = (7 :: 8 :: 9 :: []) ::
              (4 :: 5 :: 6 :: []) ::
              (1 :: 2 :: 3 :: []) :: []
m3 *mm m4 = (30 :: 36 :: []) ::
            (66 :: 81 :: []) ::
            (102 :: 126 :: []) :: []
m3 *mm m3 = (30 :: 36 :: 42 :: []) ::
            (66 :: 81 :: 96 :: []) ::
            (102 :: 126 :: 150 :: []) :: []
```

2. We continue the topic of matrices. The task is now to define the transpose of a matrix:

   ```
   transpose : {m n : ℕ} → Matrix m n → Matrix n m
   ```

   There are certainly several ways to solve this exercise but a nice way is to first establish combinators to establish that `F A = Vec A n` is a *applicative* functor. That is we have operations

   ```
   return : A → F A
   app : F (A → B) → F A → F B
   ```

   which are subject to some laws which we will ignore just now.

3. Derive the translations for `_×_` and `_→_` and their numerical counterparts. That is derive:

   ```
   prod-eq : {m n : ℕ} → (Fin m × Fin n) ∼ Fin (m * n)
   exp-eq  : {m n : ℕ} → (Fin m → Fin n) ∼ Fin (n ↑ m)
   ```

4. Derive the translations for Σ- and Π-types and their numerical counterparts. That is derive:

   ```
   Σ−eq : {m : ℕ} {f : Fin m → ℕ}
          → (Σ[ i ∈ Fin m ] Fin (f i))   ∼ Fin (Σℕ m f)
   Σ−eq : {m : ℕ} {f : Fin m → ℕ}
          → ((i : Fin m) → Fin (f i)) ∼ Fin (Πℕ m f)
   ```