Chapter 4

Datatypes

The types we have seen so far, that is functions ($\_\to\_$), products ($\_\times\_$,$\top$) and sums ($\_+\_$, $\bot$) are finite or preserve finiteness. In computation but also in Mathematics we often use infinite types, such as the natural numbers ($\mathbb{N}$), which I have already used for illustration. In this chapter we will look at such *datatypes*, starting with the natural numbers as the canonical example. We discuss primitive recursion first in a schematic way and then given by combinators: the recursor which can be reduced to the iterator (also called fold). We then look at various examples of inductive types such as lists, syntax trees and infinite trees representing ordinal notations. We investigate the limits of what constitutes a reasonable inductive type and what not — this is related to the notion of positivity. Using the notion of categorical mirror we also investigate the dual of inductive types: coinductive types. Examples here are streams and coinductive natural numbers, which can also be infinite.

4.1 The natural numbers

One of the most versatile datatypes we consider is the type of natural numbers, i.e. the counting numbers $0, 1, 2, 3, 4, \ldots$. Indeed, because we are computer scientists we are starting counting with 0, which is useful when calculating off-sets in datastructures but it is also the answer to the question *How many elephants do you have in your fridge* (I hope).

It was Guiseppe Peano from Torino who wrote down the precise laws of the natural numbers in predicate logic at the end of the 19th century, this is nowadays known as Peano Arithmetic. We aren't going to study these laws in detail but use one important idea: according to Peano a natural number is either zero or it is the successor of another natural number. We can implement this in Agda:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

This is the definition of a sum, but the novelty is that it is recursive, that is we are using $\mathbb{N}$ which is just being defined in the argument of the constructor suc.

Given this we can derive the usual notation: [1]

$$
\begin{array}{lll}
0 = \mathsf{zero} & & \\
1 = \mathsf{suc}\ 0 & = \mathsf{suc}\ \mathsf{zero} & \\
2 = \mathsf{suc}\ 1 & = \mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}) & \\
3 = \mathsf{suc}\ 2 & = \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) & \\
\vdots & \vdots &
\end{array}
$$

Let's define a function on the natural numbers, for example the predecessor:

```
pred : ℕ → ℕ
pred zero    = zero
pred (suc n) = n
```

Here we use pattern matching to analyse the constructor as we have seen already. Actually this function is a bit of a cheat because there isn't really a predecessor of 0. A better version uses Maybe where Maybe A is just a shorthand for $\top \uplus$ A:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just :    A → Maybe A
```

Using Maybe we can now define a version of pred that signals an error on 0 by returning nothing:

```
pred : ℕ → Maybe ℕ
pred zero    = nothing
pred (suc n) = just n
```

This version of pred indicates an error on 0 by returning nothing but for any other number returns just the predecessor.

This predecessor is the exact inverse of the constructors which can be combined into one function:

```
zerosuc : Maybe ℕ → ℕ
zerosuc nothing  = zero
zerosuc (just n) = suc n
```

It is not hard to see that the functions pred and zerosuc are inverse to each other.

[1] Luckily we can tell Agda to use the standard notation for numbers simply by invoking a *pragma*:

```
{-# BUILTIN NATURAL ℕ #-}
```

4.1.1 Recursion

To define more interesting functions on the natural numbers we need to use recursion, for example let's define the doubling function:

```
double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

That is the double of 0 is 0 and the double of a number of the form suc n is 2 + double n, which as we haven't yet defined _+_ is just suc (suc (double n).

For example double 3 = 6. Let's go through this step by step:

```
double 3 = double (suc (suc (suc zero)))
         = suc (suc (double (suc (suc zero))))
         = suc (suc (suc (suc (double (suc zero)))))
         = suc (suc (suc (suc (suc (double zero))))))
         = suc (suc (suc (suc (suc zero))))
         = 6
```

We can justify the use of recursion: all natural numbers are constructed via zero and suc. To define a function f over the natural numbers we just have to give a case for zero and one for suc n. Since the number n has to be constructed *before* we suc n we can construct the answer to the function f in the same way and compute the answer to f n before we compute the answer to f (suc n) hence we can use f n when constructing f (suc n).

Not all functions are exactly of this form, already the inverse of double, that is half which forgets the remainder, uses a slightly more general scheme

```
half : ℕ → ℕ
half zero          = zero
half (suc zero)    = zero
half (suc (suc n)) = suc (half n)
```

That is half of 0 and 1 is zero and half orf 2 + n is 1 + half n. The recurrence from suc (suc n) can be justified in the same way as before because certainly n is constructed before suc (suc n).

4.1.2 Arithmetic operations

Let's define the standard functions of arithmetic starting with addition:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

The idea of this definition of addition is that for example 3 + 5 is the 3rd successor of 5 that suc (suc (suc 5)). This means that 0 + n is just n and suc m + n is one more than m + n.

Next we define multiplication _*_ which is repeated addition in the same way as _+_ is repeated suc.

```
_*_ : ℕ → ℕ → ℕ
zero  * n = 0
suc m * n = n + m * n
```

That is for example 3 * 5 is 5 + 5 + 5 + 0. Recursively we express this by saying that 0 * n is just 0 and suc n * m is n + (m * n).

We can go further and define exponentiation as repeated multiplication. In the previous cases it didn't matter over which argument we recursed because they were commutative. But exponentiation certainly isn't, e.g. $2^3 = 8$ but $3^2 = 9$. If we want to mimic the usual order of arguments for exponentiation we need to recur on the 2nd argument:

```
_^_ : ℕ → ℕ → ℕ
m ↑ zero  = 1
m ↑ suc n = m * m ↑ n
```

For example 3 ↑ 5 is 3 * 3 * 3 * 3 * 3 * 1. Hence m ↑ 0 = 1 and m ↑ suc n = m * m ↑ n.

4.1.3 The Ackermann function

Each instance of recursion correspond to using a for-loop in conventional programming. While addition is just defined using one for-loop, multiplication calls addition inside its loop, hence we have a nested for-loop. And exponentiation uses a double nested for-loop. Historically functions that only use for-loops are called primitive recursive, and all the examples we have seen are examples of primitive recursion. Hilbert, a famous german Mathematician, asked wether all total recursive functions can be defined using primitive recursion. His student Ackermann showed that this is not the case by exhibiting a function that is not primitive recursive because it grows faster than any primitive recursive function. The idea of the Ackermann function is to have one extra parameter which determines the nesting of for loops, that is he defined a function ack : ℕ → ℕ → ℕ such that

```
ack 0 m n = suc n
ack 1 m n = m + n
ack 2 m n = m * n
ack 3 m n = m ↑ n
    ⋮   ⋮
```

The next function, hyperexponentiation that is repeated exponentiation doesn't really have a standard notation anymore. We can define ack using recursion over the natural numbers:

```
ack : ℕ → ℕ → ℕ → ℕ
ack zero                    m n        = suc n
ack (suc zero)              m zero     = m
ack (suc (suc zero))        m zero     = 0
ack (suc (suc (suc zero)))  m zero     = 1
ack (suc (suc (suc (suc l)))) m zero   = m
ack (suc l)                 m (suc n)  = ack l m (ack (suc l) m n)
```

To stay consistent with exponentiation we define the functions by recursion over the 2nd argument. The first line defines ack 0 to be just the successor of the 2nd argument because then we can obtain addition by repeating it. The next three lines define the 0 cases for addition m + 0 = m, multiplication m * 0 = 0 and exponentiation m ↑ 0 = 1. The idea for multiplication and exponentiation is to use the neutral element of the previous operation, corresponding to 0 iterations. However, since the is no neutral element for exponentiation, that is no number such that $x^n = n$, we just use the first argument as for addition. The last line uniformly expresses that the function for ack (suc l) is defined by iterating the previous function ack l n times.

Actually the definition above is a bit complicated due to all the different cases for 0, hence often a simplified version of the Ackermann function is used where the levels don't exactly correspond to the usual definitions of the arithmetic operations but have the same nesting of for loops:

```
acks : ℕ → ℕ → ℕ → ℕ
acks zero    m n       = suc n
acks (suc l) m zero    = m
acks (suc l) m (suc n) = acks l m (acks (suc l) m n)
```

Using either ack or acks we can define the function fast:

```
fast : ℕ → ℕ
fast n = ack n n n
```

We can only evaluate the first few instances of fast

```
fast 1 = 1 + 1                      = 2
fast 2 = 2 * 2                      = 4
fast 3 = 3 ↑ 3                      = 27
fast 4 = 4 ↑ (4 ↑ (4 ↑ 4))          = ?
```

Due to the fact that our implementation of numbers using a unary representation we can't even compute fast 4 in a reasonable time. But even using a very efficient implementation of numbers will not save us much, the incredible fast

growing behaviour of fast will spoil any attempt to compute many more results. And indeed, fast grows faster than any primitive recursive function that is any function definable only with for-loops.

4.2 Iterator and recursor

We intuitively explained and justified recursive definitions over the natural numbers but now we shall be more precise and say exactly what recursive definitions are permissable.

In general we define a function $f : \mathbb{N} \to M$ [2] by recursion over the natural numbers

```
f 0 = z
f (suc n) = s (f n)
```

where

```
z : M
s : M → M
```

We can turn this into a higher order function, the iterator [3]:

```
It𝕟 : M → (M → M) → 𝕟 → M
It𝕟 z s zero    = z
It𝕟 z s (suc n) = s (It𝕟 z s n)
```

[4] We call M the *motive* and z,s methods. Our claim is that we can reduce recursion over the natural number to the iterator. Ok, we haven't actually made precise what we mean by recursion over the natural numbers and introducing the iterator is exactly one way to do this.

We can derive dbl using only the iterator:

```
double-it : 𝕟 → 𝕟
double-it = It𝕟 0 (λ double-n → suc (suc double-n))
```

The instances for the methods z and s can be read off the recursive definition of double:

```
double : 𝕟 → 𝕟
double zero    = zero
double (suc n) = suc (suc (double n))
```

The reduction of half is less obvious - here again the recursive definition:

[2] In the following I use A B M : Set as variables as in chapter 2.

[3] In category theory It𝕟 z s : 𝕟 → M is the universal morphism from the initial algebra. It is also called a *fold* in functional programming, or a *catamorphism*.

[4] Remember that M is one of the names we have declared to be used as implicit parameters for Set.

```
half : ℕ → ℕ
half zero          = zero
half (suc zero)    = zero
half (suc (suc n)) = suc (half n)
```

half doesn't follow exactly the pattern we have stated before but it is reducible to it. The idea is that we define an auxiliary function half-aux which computes the results of half n and half (suc n) simultaneously using a product.

```
half-aux : ℕ → ℕ × ℕ
half-aux zero = zero , zero
half-aux (suc n) = proj₂ (half-aux n) , suc (proj₁ (half-aux n))
```

[5] The first line is justified by the observation that half 0 , half 1 = 0 , 0. The recursive call follows form the fact that we have already half (suc n) as the 2nd component of half-aux n and by the recursive defintion half (suc (suc n)) = half n which explains the 2nd component. Now half-aux is an instance of our scheme and we can define

```
half-aux-it : ℕ → ℕ × ℕ
half-aux-it = It ℕ (zero , zero) (λ p → proj₂ p , suc (proj₁ p))
half-it : ℕ → ℕ
half-it n = proj₁ (half-aux-it n)
```

The definition of _+_ is straightforward, here the recursive version:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

We can transform it into an instance of our scheme using higher order functions:

```
zero  +_ = λ n → n
suc m +_ = λ n → suc (m + n)
```

And hence we define

```
_+it_ : ℕ → ℕ → ℕ
_+it_ = It ℕ (λ n → n) (λ m+ n → suc (m+ n))
```

[5] The 2nd line can be more nicely written using a local definition:

```
half-aux (suc n) = proj₂ hn, suc (proj₁ hn)
  where hn = half-aux n
```

or even better using **with** which enables pattern matching on an intermediate result:

```
half-aux (suc n) with half-aux n
half-aux (suc n) | half-n , half-sn = half-sn , (suc half-n)
```

The same technique works for multiplication:

```
_*-it_ : ℕ → ℕ → ℕ
_*-it_ = Itℕ (λ n → 0) λ m* n → n + (m* n)
```

A more interesting case is a function which doesn't only use the recursive result but also the input. An example is the factorial function which can be recursively defined we follows:

```
_! : ℕ → ℕ
zero ! = 1
suc n ! = suc n * n !
```

n ! is the number of ways we can put n people on n chairs. If there are no people and no chairs we have to do nothing hence the answer is 1 (this seems to confuse some people). Otherwise in the case (suc n) ! there are suc n ways to place the first person and then n ! to distribute n people on n chairs.

The factorial function is an instance of a slightly modified scheme, to define f : ℕ → A we use:

```
f 0 = z
f (suc n) = s n (f n)
```

where

```
z : A
s : ℕ → A → A
```

As above we can turn this into a general higher order function, called the recursor [6]:

```
Rℕ : M → (ℕ → M → M) → ℕ → M
Rℕ z s zero = z
Rℕ z s (suc n) = s n (Rℕ z s n)
```

We can define the factorial function using only the recursor:

```
_!-r : ℕ → ℕ
_!-r = Rℕ 1 (λ n n! → suc n * n!)
```

We can actually derive Rℕ from Itℕ by (re-)computing the input and the output together:

```
Rℕ-it : M → (ℕ → M → M) → ℕ → M
Rℕ-it {M} z s n = proj₂ (Itℕ z' s' n)
  where
    z' : ℕ × M
```

[6] Called *paramorphism* by Meertens.

```
z' = 0 , z
s' : ℕ × M → ℕ × M
s' (n , m) = suc n , s n m
```

That is we use $\mathbb{N} \times \mathsf{M}$ as the motive, the first component is identical to the input. Hence in the 0-case we use 0 , z and in the successor case given n , m we compute suc n , s n m, hence we are able to supply the extra argument. In the end we have only to extract the 2nd component to obtain the result.

Finally let's have a look at the simplified Ackermann function:

```
acks : ℕ → ℕ → ℕ → ℕ
acks zero    m n        = suc n
acks (suc l) m zero     = m
acks (suc l) m (suc n)  = acks l m (acks (suc l) m n)
```

Can we compute the Ackermann function using the iterator or the recursor? It seems that our scheme exactly captures primitive recursion. However, the traditional notion of primitive recursion doesn't include the use of higher order functions which leads to a much more powerful scheme. And indeed the Ackermann function is definable:

```
acks-it : ℕ → ℕ → ℕ → ℕ
acks-it = Itℕ (λ m n → suc n)
  (λ ack-l m → Itℕ m λ ack-sl-m-n → ack-l m ack-sl-m-n)
```

The main idea here is to translate the pattern using higher order functions as in the case for addition and multiplication, and then use the iterator again to translate the nested pattern.

Indeed, the power of primitive recursion with higher order function is much more than ordinary primitive recursion. We can define all recursive functions that can be proven total in Peano Arithmetic. Indeed a lambda calculus with a type of natural numbers and the recursor is called *System T* and was introduced by Gödel as a functional system corresponding to Peano Arithmetic.

4.2.1 Non primitive recursive functions

There are recursive functions over the natural numbers that cannot be defined directly using primitive recursion. An example is the computation fo the *greatest common divisor* using subtraction. First we define *cut-off subtraction* which stops at 0:

```
_-_ : ℕ → ℕ → ℕ
zero - n = zero
suc m - zero = suc m
suc m - suc n = m - n
```

That is 3 - 2 = 1 while 1 - 2 = 0. Another ingredient is a comparison function comp which gets two numbers m,n and returns different values depending on wether m < n , m = n or m > n:

```
comp : ℕ → ℕ → A → A → A → A
comp zero zero m<n m=n m>n        = m=n
comp zero (suc n) m<n m=n m>n     = m<n
comp (suc m) zero m<n m=n m>n     = m>n
comp (suc m) (suc n) m<n m=n m>n  = comp m n m<n m=n m>n
```

Now we can define gcd where we always subtract the smaller number from the larger one until they are equal:

```
gcd : ℕ → ℕ → ℕ
gcd m n  =  comp m n (gcd m (n - m)) m (gcd (m - n) m)
```

e.g. gcd 56 84 = 27.

However, Agda complains *Termination checking failed* because it cannot see that the arguments were reduced during recursion. [7] And indeed, there is no obvious way to translate this function into one using the iterator or the recursor.

I this case we can save the day by noticing that the recursion is bounded by the sum of the numbers and define:

```
gcd-aux : ℕ → ℕ → ℕ → ℕ
gcd-aux zero m n  =  zero
gcd-aux (suc fuel) m n  =
  comp m n (gcd-aux fuel m (n - m)) m (gcd-aux fuel (m - n) m)
gcd' : ℕ → ℕ → ℕ
gcd' m n  =  gcd-aux (m + n) m n
```

The first argument is often called fuel because it acts like the fuel for a car, we have to pay one unit for each recursive call. However, we have set up our definition so that we can never run out of fuel. Hence the first line in the definition of gcd-aux will never be executed and we could have put in any vaule instead of 0. There are better ways to deal with this situation, especially once we have dependent types.

However, there are recursive and terminating functions which we cannot define using higher order primitive recursion.

4.3 Inductive Types

4.3.1 Lists

The type of natural numbers

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

[7] We can override this warning by using the pragma

{-# TERMINATING #-}

before the declaration of the function.

of an *inductive datatype* which we can view as a recursively defined sum. Another example of an *inductive datatype* is the type of lists, representing finite sequences of a given type:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

Constructors in this case are the empty list [] (pronounced *nil*) and given a : A and another list l : Lis A we can construct a new list a :: l using the constructor _::_ [8] (pronounced *cons*). All lists are made from [] and _::_, for example 1 :: 2 :: 3 :: [] : List ℕ.

Recursive definitions on lists follow the same idea as for natural numbers. Examples are the length function:

```
length : List A → ℕ
length [] = 0
length (x :: l) = 1 + length l
```

which computes the length of a list; a function that sums the numbers in a list:

```
sumList : List ℕ → ℕ
sumList [] = 0
sumList (n :: ns) = n + sumList ns
```

and the append function which concatenates two lists:

```
_++_ : List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

So for example

```
length (1 :: 2 :: 3 :: []) = 3
sumList (1 :: 2 :: 3 :: []) = 6
(1 :: 2 :: 3 :: []) ++ (4 :: 5 :: []) = 1 :: 2 :: 3 :: 4 :: 5 :: []
```

Append is very similar to addition, indeed addition can be viewed as a special case of append if we identify ℕ with List ⊤. However, unlike _+_, _++_ is not commutative in general, since the order of elements matter.

Lists are a functor, that is an operation that can not only applied to elements but also to functions. This is witnessed by the map-function, that applies a function to every element of a list, producing a new list:

[8] In the language Haskell a single : is used for cons, and a double :: for typing. Type Theoreticians put more emphasis on typing hence they prefer to have the shorter notation for this.

```
map : (A → B) → List A → List B
map f [] = []
map f (x :: xs) = f x :: map f xs
```

So for example

```
map suc (1 :: 2 :: 3 :: []) = 2 :: 3 :: 4 :: []
```

As for natural numbers recursive definitions over lists can be reduced to one combinator, which is usually called fold but which we consequently call ItList:

```
ItList : M → (A → M → M) → List A → M
ItList n c [] = n
ItList n c (a :: l) = c a (ItList n c l)
```

All the recursive functions over lists we have defined can be reduced to ItList using the ideas we have established in the previous section:

```
length-it : List A → ℕ
length-it = ItList 0 (λ a length-l → suc length-l)
sumList-it : List ℕ → ℕ
sumList-it = ItList 0 _+_
_++it_ : {A : Set} → List A → List A → List A
_++it_ = ItList (λ l → l) (λ a m++ l → a :: (m++ l))
map-it : {A B : Set} → (A → B) → List A → List B
map-it f = ItList [] (λ a map-f-l → f a :: map-f-l)
```

We can also derive a recursor for lists and other inductive datatypes which allows to use the input list in the recursive call but we leave the details as an exercise.

4.3.2 Syntax trees

A common application of inductive datatypes is the representation of syntax. An expression like 2 * (3 + 4) can be viewed as a tree generated by constructors for constants and for each of the operators:

```
data Expr : Set where
  const : ℕ → Expr
  _[+]_ : Expr → Expr → Expr
  _[*]_ : Expr → Expr → Expr
```

So our example expression can be represented as follows:

```
const 2 [*] (const 3 [+] const 4) : Expr
```

An obvious function we want to define is the evaluator which computes the numeric value of an expression:

```
eval : Expr → ℕ
eval (const x) = x
eval (e₁ [+] e₂) = eval e₁ * eval e₂
eval (e₁ [*] e₂) = eval e₁ + eval e₂
```

So for example

```
eval (const 2 [*] (const 3 [+] const 4)) = 14
```

As for other inductive types we introduce an iterator for Expr:

```
ItExpr : (ℕ → M) → (M → M → M)
  → (M → M → M)
  → Expr → M
ItExpr c p m (const x) = c x
ItExpr c p m (e₁ [+] e₂) = p (ItExpr c p m e₁) (ItExpr c p m e₂)
ItExpr c p m (e₁ [*] e₂) = m (ItExpr c p m e₁) (ItExpr c p m e₂)
```

I hope that the general scheme for deriving iterators is now evident. The methods correspond to the constructors where every occurence of the datatype is replaced with the motive M. The function which is computed by the iterator recursively replaces each constructor with the method.

We can define the evaluator using the iterator:

```
eval-it : Expr → ℕ
eval-it = ItExpr
  (λ n → n)
  (λ eval-e₁ eval-e₂ → eval-e₁ + eval-e₂)
  (λ eval-e₁ eval-e₂ → eval-e₁ * eval-e₂)
```

4.3.3 Rose trees

The syntax trees from the previous section were binary trees, the nodes created by _[+]_ and _[*]_ each have two subtrees. Using lists of subtrees we can define trees with an arbitrary number of subtrees which are called *rose trees*.

```
data RoseTree : Set where
  node : List RoseTree → RoseTree
```

We don't need a special constructor for leaves, because we can just have a node with no subtrees node [].

As an example we define the following rose tree:

```
rt : RoseTree
rt = node (node []
  :: node (node (node [] :: []) :: node [] :: [])
```

```
  :: node (node [] :: [])
  :: [])
```

As an example for a recursive function on rose trees we count the number of nodes:

```
countNodes : RoseTree → ℕ
countNodes (node ts) = suc (countNodesList ts)
  where countNodesList : List RoseTree → ℕ
    countNodesList [] = 0
    countNodesList (t :: ts) = countNodes t + countNodesList ts
```

Here we use an auxiliary function which counts the nodes in a list of rose trees. We use **where** to do this. [9]

So for example countNodes rt evaluates to 8.

What is the iterator for Rose Trees? We just apply the scheme and obtain the type of methods by replacing RoseTree with the motive. We end up with

```
ItRoseTree : ((List M) → M) → RoseTree → M
```

But how can we recursively apply the iterator to the arguments which are inside a list. A good way to address this is to use map:

```
ItRoseTree m (node ts) = m (map (ItRoseTree m) ts)
```

However, Agda's termination checker isn't clever enough to see that recursion through map will terminate and we have to use the following more elaborate version where we implement an instance of map locally to the iterator:

```
ItRoseTree {M} m (node ts) = m (mapIt ts)
  where mapIt : List RoseTree → List M
    mapIt [] = []
    mapIt (t :: ts) = ItRoseTree m t :: mapIt ts
```

Now we can derive our count function from the iterator. The method is the function sumList which, as we have seen, can be defined using only the iterator for lists:

```
countNodes-it : RoseTree → ℕ
countNodes-it = ItRoseTree (λ ns → suc (sumList ns))
```

4.3.4 Ordinals

All the trees we have seen so far were *finitely branching* that is any node only had finitely many subtrees. We can also consider infinitely branching trees and a nice application of those is a notation for ordinal numbers.

[9] Agda has both **let** and where, but for reasons I don't understand only **where** supports the definition of functions by pattern matching.

The idea of ordinal numbers is that we also consider infinite numbers. That is we have the sequence

$$0, 1, 2, 3, 4, \ldots$$

but now we add an element on the top which we call ω:

$$0, 1, 2, 3, 4, \ldots \omega$$

The idea is that ω is greater than any of the numbers in the sequence $0, 1, 2, 3, 4, \ldots$ and it is the smallest such ordinal. We can continue this and create a new infinite sequence:

$$0, 1, 2, 3, 4, \ldots \omega, \omega + 1, \omega + 2, \omega + 3, \omega + 4, \ldots$$

And again we have an element on the top:

$$0, 1, 2, 3, 4, \ldots \omega, \omega + 1, \omega + 2, \omega + 3, \omega + 4, \ldots \omega + \omega$$

We can continue this game infinitely and produce yet another ordinal $\omega * \omega$ and so on. The main idea is that every ordinal number is associated with the corresponding _<_ relation. In each case this relation is well-founded, that is you can only go down a finite number of steps before reaching 0. Note that this also holds for ω, in the first step we can go down to any finite number and after this it is clear that we will end up at 0. However, there is no bound to the length of this sequence.

Here is the definition of our notation for ordinals. It starts like the definition for natural numbers but then we add another constructor lim which allows us to construct ordinals from infinite sequences:

```
data Ord : Set where
  zero : Ord
  suc : Ord → Ord
  lim : (ℕ → Ord) → Ord
```

We can embed the natural numbers into the ordinals:

```
emb : ℕ → Ord
emb zero = zero
emb (suc x) = suc (emb x)
```

Using emb we can derive ω:

```
ω : Ord
ω = lim emb
```

We can lift the arithmetic operations to ordinals to derive ordinals like $\omega + \omega$ and ω * ω. Before we define addition it is a good idea to note that ordinal addition is not commutative, that is $\alpha + \beta$ is not the same as $\beta + \alpha$. We can understand addition by just putting the ordinals next to each other. So for example $1 + \omega$ looks like this:

$$*, 0, 1, 2, 3, 4, \ldots$$

Here we write $*$ for the 0 in the ordinal 1. On the other hand $\omega + 1$ looks like this

$$0, 1, 2, 3, 4, \cdots *$$

In the first case we have just shifted the ordinals by 1, but the order we obtain is equivalent to the one of ω, that is in the sense of orders the ordinals $1 + \omega$ and ω are equivalent. This is not the case for $\omega + 1$ because this time there is an infinite gap between the finite elements and the new one, which is not equivalent to ω.

While when defining _+_ for natural numbers it didn't matter over which argument we did the recursion for the ordinals it does. And the correct choice is to do recursion over the 2nd argument:

```
_+ord_ : Ord → Ord → Ord
m +ord zero = m
m +ord suc n = suc (m +ord n)
m +ord lim f = lim (λ i → m +ord f i)
```

We define addition for a limit ordinal, that is an ordinal of the form lim f by adding the same element to each element of the sequence and then taking the limit of the sequence. So for example $\omega + \omega$ is obtained by taking the limit of the sequence $\omega, \omega + 1, \omega + 2, \ldots$ matching our intuitive explanation.

Ordinals can be used to quantify the recursive strength of a recursion scheme but also the strength of a logic. A famous example is related to the fight of Hercules against the Hydra. The Hydra is a monsters with many heads which we view as a rose tree. Hercules can chop of one head of the Hydra (that is remove a subtree of the form node []) but as a response the Hydra can grow an arbitrary number of heads at lower levels. However, Hercules will always win after a finite number of steps and be able to chop off all the heads.

The fact that Hercules will win is an instance of a particular ordinal called ϵ_0 which we can obtain by defining multiplication and exponentiation and then take the limit of ω-towers of the form

$$\omega, \omega^{\omega}, \omega^{(\omega^{\omega})}, \ldots$$

Any Hydra configuration can be viewed as an ordinal in this sequence and any move by Hercules chooses a smaller ordinal. However, one can show that the ordinal ϵ_0 corresponds to the logical strength of Arithmetic and hence we cannot prove that the game terminate using only induction and natural numbers.

Before we finish let's derive the iterator for Ord:

```
ItOrd : {M : Set} → M → (M → M) → ((ℕ → M) → M)
  → Ord → M
ItOrd z s l zero = z
ItOrd z s l (suc x) = s (ItOrd z s l x)
ItOrd z s l (lim f) = l (λ i → ItOrd z s l (f i))
```

and implement ordinal addition as an instance:

```
_+ord-it_ : Ord → Ord → Ord
α +ord-it β = ItOrd α suc lim β
```

Note that we explicitly do recursion over the 2nd argument.

4.4 Positivity

Introductions to λ-calculus often start with the untyped λ-calculus which presents the syntax of the λ-calculus without types. We can try to simulate this idea by defining a strange datatype Λ:

```
data Λ : Set where
  lam : (Λ → Λ) → Λ
```

Agda won't accept this definition (for good reasons) but for the moment we can switch this off by saying `NO_POSITIVITY_CHECK`. The idea is that any function $\Lambda \to \Lambda$ gives rise to another untyped λ-term which corresponds to its λ-abstraction. The simplest example is the identity function $\lambda\, x \to x$ which is represented as lam $(\lambda\, x \to x) : \Lambda$. To be able to form more interesting terms we need to use application. However, this is easy to define using pattern matching:

```
app : Λ → Λ → Λ
app (lam f) x = f x
```

Since every untyped λ-term is produced by an abstraction, we can just use this to obtain a function and hence we have application. We can obtain some interesting terms such as self-apply which is

```
λ x → x x
```

Hence using Λ we define:

```
self-apply : Λ
self-apply = lam (λ x → app x x)
```

What happens if we apply self-apply to itself? This term is called Ω:

```
Ω = (λ x → x x) (λ x → x x)
```

we translate this to

```
Ω : Λ
Ω = app self-apply self-apply
```

Now the strange thing about Ω is that when we β-reduce it we get back to Ω:

$$\begin{aligned}\Omega &= (\lambda\, x \to x\, x)\ (\lambda\, x \to x\, x)\\ &= (x\, x)\ [\, x \mapsto (\lambda\, x \to x\, x)\,]\\ &= (\lambda\, x \to x\, x)\ (\lambda\, x \to x\, x)\\ &= \Omega\end{aligned}$$

This means that we have constructed an element of Λ which is not generated by lam. And indeed if we try to evaluate Ω Agda will hang.

It is the negative occurence of the type we are defining which causes this. Another example of a type which only has a negative occurence is the the following (and again we have to switch off the positivity checker).

```
data Weird₁ : Set where
  foo : (Weird₁ → ⊥) → Weird₁
```

It is hard to explain Weird_1 but that is the point. We can show that Weird_1 is empty using pattern matching. The idea is that if there is an element of Weird_1 then we have a function $\mathsf{Wierd}_1 \to \bot$ and hence we can derive $\bot$:

```
¬weird₁ : Weird₁ → ⊥
¬weird₁ (foo x) = x (foo x)
```

However, now using $\neg\mathsf{weird}_1$ we can actually construct an element of Wierd_1, namely foo ¬weird. But since we have shown that Weird_1 is empty we can derive an element of the empty type:

```
bad : ⊥
bad = ¬weird₁ (foo ¬weird₁)
```

This is certainly bad, because the empty type was supposed to be empty.

These examples illustrate that we need to have some restrictions on the types of constructors when defining an inductive datatype. The question to answer first is what do we actually mean by an inductive datatype? All the elements of an inductive datatype are generated from the constructors. In the case that an argument of the constructor is the type we are defining means that this should be an element which we know already. That is we create the elements in stages, e.g. with the natural numbers in the first step we can only create zero : $\mathbb{N}$, or for the expressions we can only create the constants const 1 : Expr. But in the next step we can create suc n : $\mathbb{N}$ or in the case of Expr we can create for example const 1 [+] const 2 and so on. This also justifies the recursion principles expressed via the iterators. This line of thought explains inductive datatypes where the type appears as an argument to a constructor.

This also works for rose trees but here we have to create a whole list of trees before we can go on creating a new tree.

An interesting example is

```
data T₁ : Set where
  foo : T₁ → T₁
```

What are the elements of this type? Since we have no leaves we can't even get started to create elements. Hence the type is empty. We can actually show this by defining a function into the empty type by recursion:

```
¬t₁ : T₁ → ⊥
¬t₁ (foo x) = ¬t₁ x
```

Understanding infinitely branching trees like Ord is a bit more involved. We first create all the trees we can build in finitely many stages. These include some of the lim-trees but only the one which only use the trees we have constructed before, hence which have only a finite number of different subtrees. Once we have created all of them we have reached a stage ω. Now we can continue to create trees using all the infinitely many we have created up to ω. And so on, we can repeat this process to get more and more trees.

Hence we can allow constructor arguments which are of the form $A \to X$ where X is the type we are defining just now and A is a type we already know. This is called a *strictly positive type* and this is the condition Agda is using to decide wether a datatype declaration is acceptable.

But what about the following type?

```
data Weird₂ : Set where
  foo : ((Weird₂ → Bool) → Bool) → Weird₂
```

In this example the occurence of Weird_2 in the argument is not strictly positive but still positive, i.e. double negative.

Should we allow such types? Agda doesn't think so, but why? First of all our semantic explanation is out of the window, we cannot generate Weird_2 in stages. But is it inconsistent? This doesn't seem to be the case even though the proof needs quite strong logical principles. However, this type is inconsistent with classical logic.

If we assume TND or RAA then the type Bool is the type of propositions and Weird_2 constructs a fixpoint of the double power set and we can use well known constructions (Cantor's diagonalisation) to derive a contradiction.

Hence, while positive datatypes on their own seem to be consistent, they rule out certain assumptions like the assumption of classical logic. Moreover they are not justified semantically. These seem to be good reasons to rule them out.

4.5 Coinductive types

One of the nice features of category theory is the availability of a mirror: for any construction X there is a dual one, which is usually called co-X. For example from the point of category products are dual to sums and indeed sums are sometimes called coproducts. We have seen that recursive sums aka inductive types are very useful, hence now we wonder what can we do with recursive products or coinductive types.

4.5.1 Streams

An example of a coinductive type is the type of streams, or infinite sequences. Given a stream over A we can obtain its head which is an element of A and its tail which is another stream. Hence we define:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

Unlike in the case for sums we have to indicate that we want to define a recursive record by invoking the keyword coinductive.

To construct a stream we just have to present a head and a tail. We can derive a constructor for streams using copattern matching which we have already seen for products:

```
_::_ : A → Stream A → Stream A
head (a :: as) = a
tail (a :: as) = as
```

Indeed we can automatically derive the constructor by saying:

```
record Stream (A : Set) : Set where
  constructor _::_
  coinductive
  field
    head : A
    tail : Stream A
```

We can use copatternmatching which we have already seen for products to define the function

```
from : ℕ → Stream ℕ
```

which for any number n returns the stream of numbers staring with n that is n, n+1, n+2, ...

```
head (from n) = n
tail (from n) = from (suc n)
```

Clearly the head of from n is just n, while the tail is the stream starting with the next number, hence from (suc n).

While we understood inductive types by construction, that is all elements are constructed using the constructors, we understand coinductive types by the way they are destructed. E.g. a stream is anything of which we can take a head and a tail.

The recursive definition of from n is reasonable, because we can always compute the head and the tail of the result of from n, we never get stuck.

Clearly we cannot define from for lists

```
fromList : ℕ → List ℕ
fromList n = n :: fromList (suc n)
```

because the recursive definition is not justified by descending along an inductively defined type eg $\mathbb{N}$.

On the other hand while we can define a function that filters out elements from a list given a boolean function:

```
filterL : {A : Set} → (A → Bool) → List A → List A
filterL f [] = []
filterL f (x :: xs) with f x
filterL f (x :: xs) | false = filterL f xs
filterL f (x :: xs) | true = x :: filterL f xs
```

However, Agda will complain if we try to do the same for streams:

```
filterS : {A : Set} → (A → Bool) → Stream A → Stream A
filterS f xs with f (head xs)
filterS f xs | false = filterS f (tail xs)
filterS f xs | true = (head xs) :: filterS f (tail xs)
```

Indeed, we cannot answer the question what is the head of filterS f xs. And indeed there may be none, we can define a stream of false:

```
falses : Stream Bool
head falses = false
tail falses = falses
```

but what is head (filterS (λ x → x) falses) : Bool?

4.5.2 Conatural numbers

Actually streams are not the dual of lists because they do not allow empty streams. We can define a type of colists which contain both finite and infinite sequences but instead we are going to study the dual of natural numbers, the conatural numbers. As we have observed previously the destructor for natural numbers is pred : $\mathbb{N}$ → Maybe $\mathbb{N}$. Hence we define conatural numbers as objects that have a predecessor:

```
record ℕ∞ : Set where
  coinductive
  field
    pred∞ : Maybe ℕ∞
```

As we have derived the predecssor for the natural numbers we can now derive 0 and suc for conatural numbers:

```
zero∞ : ℕ∞
pred∞ zero∞ = nothing
suc∞ : ℕ∞ → ℕ∞
pred∞ (suc∞ n) = just n
```

The rationale behind these definitions should be clear: zero∞ is the conatural number on which pred∞ returns an error, while the sucessor of a conatural number is the one whose predecessor is just that number.

Apart from those finite numbers we can also create an infinite number:

```
∞ : ℕ∞
pred∞ ∞ = just ∞
```

The definition is straightforward: what is the predecessor of ∞? It is certainly not 0 but it is just ∞.

Let's define addition for conatural numbers. It is a good idea to have the definition of addition of natural numbers in mind:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

In contrast to define _+∞_ we have to determine what is the predeccessor of m +∞ n?

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
pred∞ (m +∞ n) with pred∞ m
pred∞ (m +∞ n) | nothing = pred∞ n
pred∞ (m +∞ n) | just m' = just (m' +∞ n)
```

To find this out we query the first argument. If it was 0, that is pred∞ m returns nothing then it is the predecessor of the 2nd argument. However, if the predecessor is just m' then the result is just m' +∞ n.

_+∞_ is the infinite extension of _+_. To make this statement a bit more precise, we can define an embedding

```
ℕ→ℕ∞ : ℕ → ℕ∞
ℕ→ℕ∞ zero = zero∞
ℕ→ℕ∞ (suc n) = suc∞ (ℕ→ℕ∞ n)
```

just using recursion. Now the definng property of _+∞_ wrt _+_ is

```
ℕ→ℕ∞ (m + n) ≡ (ℕ→∞ m) +∞ (ℕ→∞ n)
```

We haven't yet developed the machinery to prove this and we havent explained when two coinductive numbers are equal.

4.5.3 Coiterator and corecursor

As for recursion over an inductive type we can derive a general combinator to do corecursion over a coinductive type. To define a function f : W → Stream A corecursively we are using copattern patching:

```
head (f x) = h x
tail (f x) = f (t x)
```

where

```
h : A → W
t : W → W
```

This scheme of *coiteration* can be expressed as one higher order function:

```
ColtStream : {A W : Set} → (W → A) → (W → W)
                         → W → Stream A
head (ColtStream h t x) = h x
tail (ColtStream h t x) = ColtStream h t (t x)
```

As an example the from function can be defined using only the coiterator:

```
from-coit : ℕ → Stream ℕ
from-coit = ColtStream (λ n → n) (λ n → suc n)
```

Similarily in the case of the conatural numbers we can define:

```
Colt ℕ∞ : {W : Set} → (W → Maybe W) → W → ℕ∞
pred∞ (Colt ℕ∞ p x) with p x
pred∞ (Colt ℕ∞ p x) | nothing = nothing
pred∞ (Colt ℕ∞ p x) | just x' = just (Colt ℕ∞ p x')
```

As an example we can derive

```
∞-coit : ℕ∞
∞-coit = Colt ℕ∞ just tt
```

However, it turns deriving _+∞_ is a bit more difficult. Here is the code we have written using copatternmatching:

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
pred∞ (m +∞ n) with pred∞ m
pred∞ (m +∞ n) | nothing = pred∞ n
pred∞ (m +∞ n) | just m' = just (m' +∞ n)
```

We clearly do corecursion on the first argument. However, if the first argument is 0, that is of pred∞ m is nothing we immediately return pred∞ n. This sort of behaviour cannot be directly achieved with the coiterator, but requires a *corecursion*. The corecursor can stop corecursion and just return an answer, hence the method returns a sum:

```
CoRℕ∞ : {W : Set} → (W → ℕ∞ ⊎ Maybe W) → W → ℕ∞
pred∞ (CoRℕ∞ p x) with p x
pred∞ (CoRℕ∞ p x) | inj₁ n = pred∞ n
```

```
pred∞ (CoRℕ∞ p x) | inj₂ nothing = nothing
pred∞ (CoRℕ∞ p x) | inj₂ (just x') = just (CoRℕ∞ p x')
```

Using the corecursor we can define _+∞_. We are also using a case combinator for Maybe

```
caseMaybe : {A M : Set} → M → (A → M) → Maybe A → M
caseMaybe n j nothing = n
caseMaybe n j (just x) = j x


_+∞-corec_ : ℕ∞ → ℕ∞ → ℕ∞
m +∞-corec n =
  CoRℕ∞ (λ x → caseMaybe (inj₁ n) (λ m' → inj₂ (just m')) (pred∞ x)) m
```

The corecursor is dual to the recursor, which can be also formulated using a product but this is curried in the usual formulation. And as for the recursor, the corecursor can be derived from the coiterator.

4.6 Functorial semantics

Ok, we have seen a number of examples of inductive and coinductive definitions but what is the general scheme. Let's start with inductive definitions, for example once again the natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Instead of using two constructors we could have used a sum and combine them into one:

```
data ℕ' : Set where
  zerosuc : Maybe ℕ' → ℕ'
```

The function zerosuc which we defined previously now is the constructor. We recover zero and suc as

```
zero' : ℕ'
zero' = zerosuc nothing
suc' : ℕ' → ℕ'
suc' n = zerosuc (just n)
```

What about the iterator? Its type should be the following:

```
Itℕ' : (Maybe M → M) → ℕ' → M
```

Now we want to define Itℕ' m (zerosuc n) by recursively using m and Itℕ' on n. However, the problem is that n : Maybe ℕ so we cannot directly apply it.

However, luckily Maybe is a *functor* that is it comes with a map function so that we can apply it to functions:

```
map-Maybe : (A → B) → Maybe A → Maybe B
map-Maybe f nothing = nothing
map-Maybe f (just x) = just (f x)
```

Now using map-Maybe we can complete the definition of It-$\mathbb{N}$': [10]

```
Itℕ' m (zerosuc n) = m (map-Maybe (Itℕ' m) n)
```

This can be turned into a general pattern: an inductive datatype is given by a functor (often called the signature functor), i.e. an operation

```
T : Set → Set
```

together with a map function:

```
map-T : (A → B) → T A → T B
```

The datatype definition μT then is given by just one constructor:

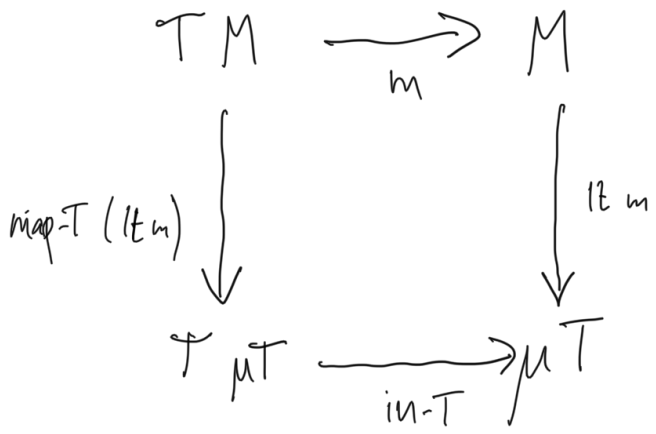
```
data μT : Set where
  in-T : T μT → μT
```

and the iterator is:

```
It-T : (T M → M) → μT → M
It-T m (in-T x) = m (map-T (It-T m) x)
```

This is a good opportunity to draw a categorical diagram:

[10] Actually Agda rejects this definition because the recursion goes through map-Maybe which is defined separately. We could have fixed this by defining map-Maybe-Itℕ' mutually but here I just said {-# TERMINATING #-} .



This diagram expresses that

$$\mathsf{in\text{-}T} \circ (\mathsf{mapT}\ (\mathsf{lt\text{-}M}\ \mathsf{m})) \;=\; (\mathsf{lt}\ \mathsf{m}) \circ \mathsf{m}$$

In general categorical diagrams express that different paths through the diagram yield the same result, where sequencing of arrows is interpreted as composition.

We can also describe the recursor in this framework:

$$\begin{array}{l}\mathsf{R\text{-}T} : (\mathsf{T}\ (\mu\mathrm{T} \times \mathsf{M}) \to \mathsf{M}) \to \mu\mathrm{T} \to \mathsf{M}\\ \mathsf{R\text{-}T}\ \mathsf{m}\ (\mathsf{in\text{-}T}\ \mathsf{x}) \;=\; \mathsf{m}\ (\mathsf{map\text{-}T}\ (\lambda\ \mathsf{y} \to \mathsf{y}\ ,\ \mathsf{R\text{-}T}\ \mathsf{m}\ \mathsf{y})\ \mathsf{x})\end{array}$$

And indeed we can generically derive it from the iterator:

$$\begin{array}{l}\mathsf{R\text{-}T\text{-}it} : (\mathsf{T}\ (\mu\mathrm{T} \times \mathsf{M}) \to \mathsf{M}) \to \mu\mathrm{T} \to \mathsf{M}\\ \mathsf{R\text{-}T\text{-}it}\ \{\mathsf{M}\}\ \mathsf{f}\ \mathsf{x} \;=\; \mathsf{proj}_2\ (\mathsf{lt\text{-}T}\ \{\mu\mathrm{T} \times \mathsf{M}\}\ (\lambda\ \mathsf{p} \to \mathsf{in\text{-}T}\ (\mathsf{map\text{-}T}\ \mathsf{proj}_1\ \mathsf{p})\ ,\ \mathsf{f}\ \mathsf{p})\ \mathsf{x})\end{array}$$

We can define functors for all the examples we have considered so far:

$$\begin{array}{l}\mathsf{T\text{-}List} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\\ \mathsf{T\text{-}List}\ \mathsf{C}\ \mathsf{X} \;=\; \mathsf{Maybe}\ (\mathsf{C} \times \mathsf{X})\\ \\ \mathsf{T\text{-}Expr} : \mathsf{Set} \to \mathsf{Set}\\ \mathsf{T\text{-}Expr}\ \mathsf{X} \;=\; \mathbb{N} \uplus \mathsf{X} \times \mathsf{X} \uplus \mathsf{X} \times \mathsf{X}\\ \\ \mathsf{T\text{-}RoseTree} : \mathsf{Set} \to \mathsf{Set}\\ \mathsf{T\text{-}RoseTree}\ \mathsf{X} \;=\; \mathsf{List}\ \mathsf{X}\\ \\ \mathsf{T\text{-}Ord} : \mathsf{Set} \to \mathsf{Set}\\ \mathsf{T\text{-}Ord}\ \mathsf{X} \;=\; \mathsf{Maybe}\ (\mathsf{X} \uplus (\mathbb{N} \to \mathsf{X}))\end{array}$$

As already indicated we can view codatatypes as the mirror image. Codatatypes are also given by functors, in the case of streams this is $\mathsf{T\text{-}stream}\ \mathsf{X} \;=$

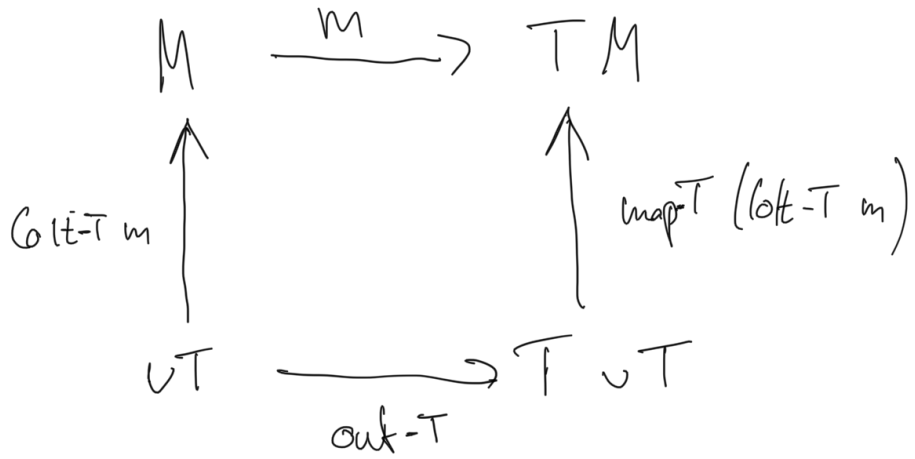
$\mathsf{A} \times \mathsf{X}$ and the conatural numbers use Maybe, ie. the same as the natural numbers. The coinductive type corresponding to T is:

```
record νT : Set where
  coinductive
  field
    out-T : T νT
```

and the coiterator is given by copattern matching:

```
Colt-T : (M → T M) → M → νT
out-T (Colt-T f x) = map-T (Colt-T f) (f x)
```

This calls for another diagram:



Comparing this with the previous diagram we can see that the directions of all the arrows have been reversed. I leave it as an exercise to define the corecursor and derive it from the Coiterator.

You may have noticed that I have cheated in a number of ways. None of the definitions I have given are acceptable in Agda as they stand. The definitions of the μT and νT are rejected because they go through T and Agda doesn't know anything about T. Ok, we have assumed that it is a functor but actually this is not enough because Wierd_2 is given by a functor:

```
T-Weird₂ : Set → Set
T-Weird₂ X = (X → Bool) → Bool

T-Wierd₂-map : (A → B) → T-Weird₂ A → T-Weird₂ B
T-Wierd₂-map f g h = g (h ∘ f)
```

We will get back to this later and give a definition which only applies to strictly positive types but we need dependent types for this.

I would also like to add that it is not enough to have map-T but to be a proper functor also needs to preserve identity and composition, i.e. map-T id = id and map-T (f ∘ g) = map-T f ∘ map-T g but we need equality types to specify this in Type Theory.

4.7 History

Guiseppe Peano lived in Turino in the late 19th century. He wrote down the rules of reasoning for the natural numbers which is now in his honour called *Peano Arithmetic*. A proper account of Peano Arithmetic uses predicate logic which we haven't yet introduced. Peano only used addition and multiplication in his presentation but it can be shown that all primitive recursive functions can be derived. Interestingly, this ceases to be the case if you leave out multiplication which leads to something called Pressburger Arithmetic.

By primitive recursive functions one means traditionally functions which can be derived using no function types, that is with RN but only products and $\mathbb{N}$ can be used. This intuitively corresponds to for-loops in programming. Ackermann formulated his function to show that there are computable functions which cannot be defined with primitive recursion.

It was Gödel who formulated a system which can express all functions that are definable in Peano arithmetic. This is called System T [11] and it is basically what we have presented, i.e. you have RN and function types. There are still functions which are not definable in this system, corresponding to the problem of Archilles and the hydra which we have mentioned already in the section on ordinals.

Our presentation of inductive types goes back to algebraic datatyes and pattern matching invented by my PhD supervisor Rod Burstall which were first implemented in a language called Hope [**?**]. Subsequently algebraic datatypes and pattern matching was used in many functional programming languages including Miranda and Haskell. Coinductive types were investigated by Coquand [**?**] who also formulated the guarded corecursion principle we are using. Copatternmatching was introduced much later in the context of Type Theory by Andreas Abel and and Brigitte Pientka [**?**]. The functorial understanding of datatypes and codatatypes can be found in the thesis of Tatsuyo Hagino [**?**]. [12]

We only briefly touched upon the untyped λ-calculus which, together with untyped combinatory logic, is a basic computational mechanism indeed an alternative to Turing machines. A good overview over the untyped λ-calculus is Barendregt's book [**?**].

[11] Gödel introduced System T in [**?**] (in german).

[12] Hagino was also supervised by Rod Burstall, but he finished before I arrived in Edinburgh.

4.8 Exercises

1. Define the following functions using pattern matching and structural recursion on the natural numbers.

   (a) Define a function even that determines wether its input is even.

   $$\mathsf{even?} : \mathbb{N} \to \mathsf{Bool}$$

   Examples :

   $$\begin{aligned} \mathsf{even?}\ 3 &= \mathsf{false} \\ \mathsf{even?}\ 6 &= \mathsf{true} \end{aligned}$$

   (b) Define a function sum that sums the numbers from 0 until n-1.

   $$\mathsf{sum} : \mathbb{N} \to \mathbb{N}$$

   Examples:

   $$\begin{aligned} \mathsf{sum}\ 2 &= 3 \\ \mathsf{sum}\ 10 &= 55 \end{aligned}$$

   (c) Define a function max that calculates the maximum of 2 numbers.

   $$\mathsf{max} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

   Examples:

   $$\begin{aligned} \mathsf{max}\ 2\ 3 &= 3 \\ \mathsf{max}\ 10\ 1 &= 10 \end{aligned}$$

   (d) Define a function fib which calculates the nth item of the Fibonacci sequence: $1, 1, 2, 3, 5, 8, 13, \ldots$ (each number is the sum of the two previous ones).

   $$\mathsf{fib} : \mathbb{N} \to \mathbb{N}$$

   Examples:

   $$\begin{aligned} \mathsf{fib}\ 0 &= 1 \\ \mathsf{fib}\ 6 &= 13 \end{aligned}$$

   (e) Define a function eq that determines wether two numbers are equal.

   $$\mathsf{eq?} : \mathbb{N} \to \mathbb{N} \to \mathsf{Bool}$$

   Examples:

   $$\begin{aligned} \mathsf{eq?}\ 4\ 3 &= \mathsf{false} \\ \mathsf{eq?}\ 6\ 6 &= \mathsf{true} \end{aligned}$$

(f) Define a function rem such that rem m n returns the remainder when dividing m by suc n (this way we avoid division by 0).

```
rem : ℕ → ℕ → ℕ
```

Examples:

```
rem 5 1  = 1
rem 11 2 = 2
```

(g) Define a function div such that div m n returns the result of dividing m by suc n (ignoring the remainder).

```
div : ℕ → ℕ → ℕ
```

Examples:

```
div 5 1  = 2
div 11 2 = 3
```

2. Define all the functions of the previous question but this time only use the iterator It$\mathbb{N}$ and/or the recursor R$\mathbb{N}$.

   Use the following naming convention: if the function was called f then call it f-i if you only use the iterator, but f-r if you use the recusor (and possibly the iterator).

3. In this exercise we implement tree-sort to sort a list of natural numbers.

   We define the type of trees with labels on the nodes.

```
data Tree (A : Set) : Set where
  leaf : Tree A
  node : Tree A → A → Tree A → Tree A
```

   Here is an example tree:

```
node (node leaf 1 (node leaf 2 leaf)) 5 leaf
```

   Define a function tree2list which collects all the leaves from left to right in a list.

```
tree2list : {A : Set} → Tree A → List A
```

   Here is a simple unit test

```
tree2list t = 1 :: 2 :: 5 :: []
```

We are going to produce sorted trees. A sorted tree is one where for every node the leaves on the left have smaller values and the leaves on the right have larger values.

I give you a comparison function on $\mathbb{N}$.

```
_⩽?_ : ℕ → ℕ → Bool
zero ⩽? m = true
suc n ⩽? zero = false
suc n ⩽? suc m = n ⩽? m
```

Write a function that transforms a list of natural number into a sorted tree.

```
list2tree : List ℕ → Tree ℕ
```

Here is a test:

```
list2tree (10 :: 2 :: 1 :: 5 :: [])
  = node (node leaf 1 (node leaf 2 leaf)) 5 (node leaf 10 leaf)
```

Hint: it may be a good idea to write a function

```
insert : ℕ → Tree ℕ → Tree ℕ
```

that inserts one value into a sorted tree maintaining its sortedness.

Using the previous function it is now easy to define tree-sort a function that sorts lists.

```
tree-sort : List ℕ → List ℕ
```

Here is a unit test for treesort.

```
tree-sort (10 :: 2 :: 1 :: 5 :: [])        = 1 :: 2 :: 5 :: 10 :: []
```

4. Reimplement tree-sort using only the iterators for lists and trees (also called folds).

   The functions should **not** use pattern matching on lists or trees or any recursion but only use the combinators defined here:

```
foldList : {A : Set} {M : Set} →
  M → (A → M → M) → List A → M
foldList mnil mcons [] = mnil
foldList mnil mcons (x :: xs) = mcons x (foldList mnil mcons xs)
foldTree : {A : Set} {M : Set} →
  M → (M → A → M → M) → Tree A → M
foldTree mleaf mnode leaf = mleaf
```

```
foldTree mleaf mnode (node l x r) =
  mnode (foldTree mleaf mnode l) x (foldTree mleaf mnode r)
```

If the original function is called x then the function with folds only is called x-f.

5. We have already defined addition for conatural numbers:

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
prd∞ (m +∞ n) with prd∞ m
prd∞ (m +∞ n) | just m' = just (m' +∞ n)
prd∞ (m +∞ n) | nothing = prd∞ n
```

Our goal is now to define multiplication:

```
_*∞_ : ℕ∞ → ℕ∞ → ℕ∞
```

This is not so easy because we need to make sure that the recursion is syntactically guarded, i.e. the recursive call has to appear directly at the right hand side not inside another function.

Hint: Define an auxilliary function first which solves a more general problem.

Now here is a simple unit test:

$$\mathbb{N}\infty\to\mathbb{N}\ (\mathbb{N}\to\mathbb{N}\infty\ 3\ *\infty\ \mathbb{N}\to\mathbb{N}\infty\ 5) = 15$$

6. Define the map operations for all the examples of functors:

```
T-List-map : (A → B) → T-List C A → T-List C B
T-Expr-map : (A → B) → T-Expr A → T-Expr B
T-RoseTree-map : (A → B) → T-RoseTree A → T-RoseTree B
T-Ord-map : (A → B) → T-Ord A → T-Ord B
```

7. Dualize the derivation of the generic recursor in section 4.6

That is define the corecursor:

```
CoR-T : (M → T (νT ⊎ M)) → M → νT
```

using copattern matching, map-T and guarded recursion.

The show that it can be derived directly from CoIt without recursion:

```
CoR-T-coit : (M → T (νT ⊎ M)) → M → νT
```