Chapter 7

Recursion

When you step between two mirrors, you see your reflection, the reflection of your reflection, the reflection of that reflection, and so on. Each reflection is defined in terms of the previous reflection, but together they produce infinity.

Recursion is a decomposition pattern that splits a single task into many steps, the number of which is potentially unbounded.

Recursion is based on suspension of disbelief. You are faced with a task that may take arbitrarily many steps. You tentatively assume that you know how to solve it. Then you ask yourself the question: ”How would I make the last step if I had the solution to everything *but* the last step?”

7.1 Natural Numbers

An object of natural numbers N does not contain numbers. Objects have no internal structure. Structure is defined by arrows.

We can use an arrow from the terminal object to define one special element. By convention, we'll call this arrow Z for "zero."

$$Z\colon 1 \to N$$

But we have to be able to define infinitely many arrows to account for the fact that, for every natural number, there is another number that is one larger than it.

We can formalize this statement by saying: Suppose that we know how to create a natural number $n\colon 1 \to N$. How do we make the next step, the step that will point us to the next number—its successor?

This next step doesn't have to be any more complex than just post-composing n with an arrow that loops back from N to N. This arrow should not be the identity, because we want the successor of a number to be different from that number. But a single such arrow, which we'll call S for "successor" will suffice.

The element corresponding to the successor of n is given by the composition:

$$1 \xrightarrow{n} N \xrightarrow{S} N$$

(We sometimes draw the same object multiple times in a single diagram, if we want to straighten the looping arrows.)

In particular, we can define One as the successor of Z:

$$\begin{tikzcd}
1 \arrow[r, "Z"] \arrow[rr, bend left, "One"] & N \arrow[r, "S"] & N
\end{tikzcd}$$

and Two as the successor of the successor of Z

$$\begin{tikzcd}
1 \arrow[r, "Z"] \arrow[rrr, bend left, "Two"] & N \arrow[r, "S"] & N \arrow[r, "S"] & N
\end{tikzcd}$$

and so on.

Introduction Rules

The two arrows, Z and S, serve as the introduction rules for the natural number object N. The twist is that one of them is recursive: S uses N as its source as well as its target.

$$\begin{tikzcd}
1 \arrow[r, "Z"] & N \arrow[loop above, "S"]
\end{tikzcd}$$

The two introduction rules translate directly to Haskell

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

They can be used to define arbitrary natural numbers; for instance:

```
zero, one, two :: Nat
zero = Z
one  = S zero
two  = S one
```

This definition of natural number type is not very useful in practice. However, it's often used in defining type-level naturals, where each number is its own type.

You may encounter this construction under the name of *Peano arithmetic*.

Elimination Rules

The fact that the introduction rules are recursive complicates the matters slightly when it comes to defining elimination rules. We will follow the pattern from previous chapters of first assuming that we are given a mapping out of N:

$$h \colon N \to A$$

and see what we can deduce from there.

Previously, we were able to decompose such an h into simpler mappings (pairs of mappings for sum and product; a mapping out of a product for the exponential).

The introduction rules for N look similar to those for the sum, so we would expect that h could be split into two arrows corresponding to two introduction rules. And, indeed, we can easily get the first one by composing $h \circ Z$. This is an arrow that picks an element of A. We call it *init*:

$$init \colon 1 \to A$$

But there is no obvious way to find the second one.

Let's try to plug h into the definition of N.

$$\begin{tikzcd}
1 \arrow[r, "Z"] \arrow[dr, "init"'] & N \arrow[r, "S"] \arrow[d, dashed, "h"] & N \arrow[r, "S"] \arrow[d, dashed, "h"] & N \arrow[d, dashed, "h"] & \ldots \\
& A & A & A &
\end{tikzcd}$$

The intuition is that an arrow from N to A represents a *sequence* a_n of elements of A. The zeroth element is given by $a_0 = init$. The next element is

$$a_1 = h \circ S \circ Z$$

followed by

$$a_2 = h \circ S \circ S \circ Z$$

and so on.

We have thus replaced one arrow h with infinitely many arrows a_n. Granted, the new arrows are simpler, since they represent elements of A, but there are infinitely many of them.

The problem is that, no matter how you look at it, an arbitrary mapping out of N contains infinite amount of information.

We have to drastically simplify the problem. Since we used a single arrow S to generate all natural numbers, we can try to use a single arrow $A \to A$ to generate all the elements a_n. We'll call this arrow *step*:

$$\begin{tikzcd}
1 \arrow[r, "Z"] \arrow[dr, "init"'] & N \arrow[r, "S"] \arrow[d, dashed, "h"] & N \arrow[d, dashed, "h"] \\
& A \arrow[r, "step"] & A
\end{tikzcd}$$

The mappings out of N that are generated by such pairs, *init* and *step*, are called *recursive*. Not all mappings out of N are recursive. In fact very few are; but recursive mappings are enough to define the object of natural numbers.

We use the above diagram as the elimination rule. Every recursive mapping out of N is in one-to-one correspondence with a pair *init* and *step*.

This means that the *evaluation rule* (extracting $(init, step)$ for a given h) cannot be formulated for an arbitrary arrow $h \colon N \to A$, only for recursive arrows that have been defined using a pair $(init, step)$.

The arrow *init* can be always recovered by composing $h \circ Z$. The arrow *step* is a solution to the equation:

$$step \circ h = h \circ S$$

If h was defined using some *init* and *step*, then this equation obviously has a solution.

The important part is that we demand that this solution be *unique*.

Intuitively, the pair *init* and *step* generate the sequence of elements a_0, a_1, a_2, ... If two arrows h and h' are given by the same pair $(init, step)$, it means that the sequences they generate are the same.

So if h were different from h', it would mean that N contains more than just the sequence of elements Z, SZ, $S(SZ)$, ... For instance, if we added -1 to N (that is, made Z somebody's successor), we could have h and h' differ at -1 and yet be generated by the same *init* and *step*. Uniqueness means there are no natural number before, after, or in between the numbers generated by Z and S.

The elimination rule we've discussed here corresponds to *primitive recursion*. We'll see a more advanced version of this rule, corresponding to the induction principle, in the chapter on dependent types.

In Programming

The elimination rule can be implemented as a recursive function in Haskell:

```haskell
rec :: a -> (a -> a) -> (Nat -> a)
rec init step = \n ->
  case n of
    Z     -> init
    (S m) -> step (rec init step m)
```

This single function, which is called a *recursor*, is enough to implement all recursive functions of natural numbers. For instance, this is how we could implement addition:

```haskell
plus :: Nat -> Nat -> Nat
plus n = rec init step
  where
    init = n
    step = S
```

This function takes `n` as an argument and produces a function (a closure) that takes another number and adds `n` to it.

In practice, programmers prefer to implement recursion directly—an approach that is equivalent to inlining the recursor `rec`. The following implementation is arguably easier to understand:

```haskell
plus n m = case m of
  Z -> n
  (S k) -> S (plus k n)
```

It can be read as: If `m` is zero then the result is `n`. Otherwise, if `m` is a successor of some `k`, then the result is the successor of `k + n`. This is exactly the same as saying that `init = n` and `step = S`.

In imperative languages recursion is often replaced by iteration. Conceptually, iteration seems to be easier to understand, as it corresponds to sequential decomposition. The steps in the sequence usually follow some natural order. This is in contrast with recursive decomposition, where we assume that we have done all the work up to the n'th step, and we combine that result with the next consecutive step.

On the other hand, recursion is more natural when processing recursively defined data structures, such as lists or trees.

The two approaches are equivalent, and compilers often convert recursive functions to loops in what is called *tail recursion optimization*.

Exercise 7.1.1. *Implement a curried version of addition as a mapping out of N into the function object N^N. Hint: use these types in the recursor:*

```haskell
init :: Nat -> Nat
step :: (Nat -> Nat) -> (Nat -> Nat)
```

7.2 Lists

A list of things is either empty or a thing followed by a list of things.

This recursive definition translates into two introduction rules for the type L_A, the list of A:

$$\text{Nil}\colon 1 \to L_A$$

$$\text{Cons}\colon A \times L_A \to L_A$$

The Nil element describes an empty list, and $Cons$ constructs a list from a head and a tail.

The following diagram depicts the relationship between projections and list constructors. The projections extract the head and the tail of the list that was constructed using $Cons$.

$$\begin{tikzcd}
& A \times L_A \arrow[dl, "fst"'] \arrow[dr, bend right, "snd"'] \arrow[dr, bend left, red, "\text{Cons}"] & & 1 \arrow[dl, red, "\text{Nil}"] \\
A & & L_A &
\end{tikzcd}$$

This description can be immediately translated to Haskell:

```haskell
data List a where
  Nil  :: List a
  Cons :: (a, List a) -> List a
```

Elimination Rule

Given a mapping out, $h\colon L_A \to C$, from a list of A to some arbitrary type C, this is how we can plug it into the definition of the list:

$$\begin{tikzcd}
1 \arrow[r, "Nil"] \arrow[dr, "init"'] & L_A \arrow[d, dashed, "h"] & A \times L_A \arrow[l, "Cons"'] \arrow[d, dashed, "id_A \times h"] \\
& C & A \times C \arrow[l, "step"]
\end{tikzcd}$$

We used the functoriality of the product to apply the pair (id_A, h) to the product $A \times L_A$.

Similar to the natural number object, we can try to define two arrows, $init = h \circ Nil$ and $step$. The arrow $step$ is a solution to:

$$step \circ (id_A \times h) = h \circ Cons$$

Again, not every h can be reduced to such a pair of arrows.

However, given $init$ and $step$, we can define an h. Such a function is called a *fold*, or a list catamorphism.

This is the list recursor in Haskell:

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
recList init step = \as ->
  case as of
    Nil          -> init
    Cons (a, as) -> step (a, recList init step as)
```

Given `init` and `step`, it produces a mapping out of a list.

A list is such a basic data type that Haskell has a built-in syntax for it. The type `List a` is written as `[a]`. The `Nil` constructor is an empty pair of square brackets, `[]`, and the `Cons` constructor is an infix colon `:`.

We can pattern match on these constructors. A generic mapping out of a list has the form:

```
h :: [a] -> c
h []      = -- empty-list case
h (a: as) = -- case for the head and the tail of a non-empty list
```

Corresponding to the recursor, here's the type signature of the function `foldr` (fold *right*), which you can find in the standard library:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

Here's a possible implementation:

```
foldr step init = \as ->
  case as of
    [] -> init
    a : as -> step a (foldr step init as)
```

As an example, we can use `foldr` to calculate the sum of the elements of a list of natural numbers:

```
sum :: [Nat] -> Nat
sum = foldr plus Z
```

Exercise 7.2.1. *Consider what happens when you replace A in the definition of a list with the terminal object. Hint: What is base-one encoding of natural numbers?*

Exercise 7.2.2. *How many mappings $h \colon L_A \to 1 + A$ are there? Can we get all of them using a list recursor? How about Haskell functions of the signature:*

```
h :: [a] -> Maybe a
```

Exercise 7.2.3. *Implement a function that extracts the third element from a list, if the list is long enough. Hint: Use `Maybe a` for the result type.*

7.3 Functoriality

Functoriality means, roughly, the ability to transform the "contents" of a data structure. The contents of a list L_A is of the type A. Given an arrow $f \colon A \to B$, we need to define a mapping of lists $h \colon L_A \to L_B$.

Lists are defined by the mapping out property, so let's replace the target C of the elimination rule by L_B. We get:

$$\begin{tikzcd}
1 \arrow[r, "Nil_A"] \arrow[dr, "init"'] & L_A \arrow[d, dashed, "h"] & A \times L_A \arrow[l, "Cons_A"'] \arrow[d, dashed, "id_A \times h"] \\
& L_B & A \times L_B \arrow[l, "step"]
\end{tikzcd}$$

Since we are dealing with two different lists here, we have to distinguish between their constructors. For instance, we have:

$$\text{Nil}_A \colon 1 \to L_A$$

$$\text{Nil}_B \colon 1 \to L_B$$

and similarly for $Cons$.

The only candidate for $init$ is Nil_B, which is to say that h acting on an empty list of As produces an empty list of Bs:

$$h \circ Nil_A = Nil_B$$

What remains is to define the arrow:

$$step \colon A \times L_B \to L_B$$

We can take:

$$step = \text{Cons}_B \circ (f \times id_{L_B})$$

This corresponds to the Haskell function:

```
mapList :: (a -> b) -> List a -> List b
mapList f = recList init step
  where
    init = Nil
    step (a, bs) = Cons (f a, bs)
```

or, using the built-in list syntax and inlining the recursor,

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a : as) = f a : map f as
```

You might wonder what prevents us from choosing $step = snd$, resulting in:

```
badMap :: (a -> b) -> [a] -> [b]
badMap f [] = []
badMap f (a : as) = badMap f as
```

We'll see, in the next chapter, why this is a bad choice. (Hint: What happens when we apply `badMap` to id?)