Chapter 13

Coalgebras

Coalgebras are just algebras in the opposite category. End of chapter!

Well, maybe not... As we've seen before, the category in which we're working is not symmetric with respect to duality. In particular, if we compare the terminal and the initial objects, their properties are not symmetric. Our initial object has no incoming arrows, whereas the terminal one, besides having unique incoming arrows, has lots of outgoing arrows.

Since initial algebras were constructed starting from the initial object, we might expect terminal coalgebras—their duals, generated from the terminal object—not to be just their mirror images, but to add their own interesting twists.

We've seen that the main application of algebras was in processing recursive data structures: in folding them. Dually, the main application of coalgebras is in generating, or unfolding, of recursive, tree-like, data structures. The unfolding is done using an anamorphism.

We use catamorphisms to chop trees, we use anamorphisms to grow them.

We cannot produce information from nothing so, in general, both a catamorphism and an anamorphism reduce the amount of information that's contained in their input.

After you sum a list of integers, it's impossible to recover the original list.

By the same token, if you grow a recursive data structure using an anamorphism, the seed must contain all the information that ends up in the tree. The advantage is that the information is now stored in a form that's more convenient for further processing.

13.1 Coalgebras from Endofunctors

A coalgebra for an endofunctor F is a pair consisting of a carrier A and a structure map: an arrow $A \to FA$.

In Haskell, we define:

```haskell
type Coalgebra f a = a -> f a
```

We often think of the carrier as the type of a seed from which we'll grow the data structure, be it a list or a tree.

For instance, here's a functor that can be used to create a binary tree, with integers stored at the nodes:

```
data TreeF x = LeafF | NodeF Int x x
  deriving (Show, Functor)
```

We don't even have to define the instance of `Functor` for it—the `deriving` clause tells the compiler to generate the canonical one for us (together with the `Show` instance to allow conversion to `String`, if we want to display it).

A coalgebra is a function that takes a seed of the carrier type and produces a functor-ful of new seeds. These new seeds can then be used to generate the subtrees, recursively.

Here's a coalgebra for the functor `TreeF` that takes a list of integers as a seed:

```
split :: Coalgebra TreeF [Int]
split [] = LeafF
split (n : ns) = NodeF n left right
  where
    (left, right) = partition (<= n) ns
```

If the seed is empty, it generates a leaf; otherwise it creates a new node. This node stores the head of the list and fills the node with two new seeds. The library function `partition` splits a list using a user-defined predicate, here `(<= n)`, less-than-or-equal to `n`. The result is a pair of lists: the first one satisfying the predicate; and the second, not.

You can convince yourself that a recursive application of this coalgebra creates a binary sorted tree. We'll use this coalgebra later to implement a sort.

13.2 Category of Coalgebras

By analogy with algebra morphisms, we can define coalgebra morphisms as the arrows between carriers that satisfy a commuting condition.

Given two coalgebras (A, α) and (B, β), the arrow $f \colon A \to B$ is a coalgebra morphism if the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "\alpha"] & B \arrow[d, "\beta"] \\
FA \arrow[r, "Ff"] & FB
\end{tikzcd}$$

The interpretation is that it doesn't matter if we first map the carriers and then apply the coalgebra β, or first apply the coalgebra α and then apply the arrow to its contents, using the lifting Ff.

Coalgebra morphisms can be composed, and the identity arrow is automaticaly a coalgebra morphism. It's easy to see that coalgebras, just like algebras, form a category.

This time, however, we are interested in the terminal object in this category—a *terminal coalgebra*. If a terminal coalgebra (T, j) exists, it satisfies the dual of the Lambek's lemma.

Exercise 13.2.1. *Lambek's lemma: Show that the structure map j of the terminal coalgebra (T, j) is an isomorphism. Hint: The proof is dual to the one for the initial algebra.*

As a consequence of the Lambek's lemma, the carrier of the terminal algebra is a fixed point of the endofunctor in question.

$$FT \cong T$$

with j and j^{-1} serving as the witnesses of this isomorphism.

It also follows that (T, j^{-1}) is an algebra; just as (I, i^{-1}) is a coalgebra, assuming that (I, i) is the initial algebra.

We've seen before that the carrier of the initial algebra is a fixed point. In principle, there may be many fixed points for the same endofunctor. The initial algebra is the least fixed point and the terminal coalgebra the greatest fixed point.

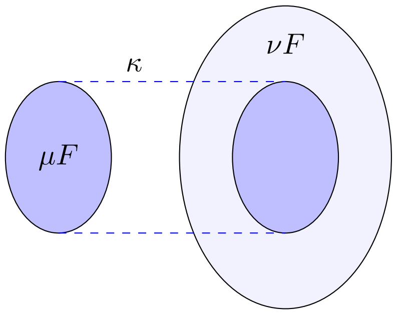
The greatest fixed point of an endofunctor F is denoted by νF, so we have:

$$T = \nu F$$

We can also see that there must be a unique algebra morphism (a catamorphism) from the initial algebra to the terminal coalgebra. That's because the terminal coalgebra is an algebra.

Similarly, there is a unique coalgebra morphism from the initial algebra (which is also a coalgebra) to the terminal coalgebra. In fact, it can be shown that it's the same underlying morphism $\kappa \colon \mu F \to \nu F$ in both cases.

In the category of sets, the carrier set of the initial algebra is a subset of the carrier set of the terminal coalgebra, with the function κ embedding the former in the latter.



We'll see later that in Haskell the situation is more subtle, because of lazy evaluation. But, at least for functors that have the leaf component—that is, their action on the initial object is non-trivial—Haskell's fixed point type works as a carrier for both the initial algebra and the terminal coalgebra.

```haskell
data Fix f where
  In :: f (Fix f) -> Fix f
```

Exercise 13.2.2. *Show that, for the identity functor in* **Set**, *every object is a fixed point, the empty set is the least fixed point, and the singleton set is the greatest fixed point. Hint: The least fixed point must have arrows going to all other fixed points, and the greatest fixed point must have arrows coming from all other fixed points.*

Exercise 13.2.3. *Show that the empty set is the carrier of the initial algebra for the identity functor in* **Set**. *Dually, show that the singleton set is this functor's terminal coalgebra. Hint: Show that the unique arrows are indeed (co-) algebra morphisms.*

13.3 Anamorphisms

The terminal coalgebra (T, j) is defined by its universal property: there is a unique coalgebra morphism h from any coalgebra (A, α) to (T, j). This morphism is called the *anamorphism*. Being a coalgebra morphism, it makes the following diagram commute:

$$\begin{tikzcd} A \arrow[r, dashed, "h"] \arrow[d, "\alpha"] & T \arrow[d, "j"] \\ FA \arrow[r, "Fh"] & FT \end{tikzcd}$$

Just like with algebras, we can use the Lambek's lemma to "solve" for `h`:

$$h = j^{-1} \circ Fh \circ \alpha$$

Since the terminal coalgebra (just like the initial algebra) is a fixed point of a functor, the above recursive formula can be translated directly to Haskell as:

```
ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coa = In . fmap (ana coa) . coa
```

Here's the interpretation of this formula: Given a seed of type `a`, we first act on it with the coalgebra `coa`. This gives us a functorful of seeds. We expand these seeds by recursively applying the anamorphism using `fmap`. We then apply the constructor `In` to get the final result.

As an example, we can apply the anamorphism to the `split` coalgebra we defined earlier: `ana split` takes a list of integers and creates a sorted tree.

We can then use a catamorphsims to fold this tree into a sorted list. We define an algebra:

```
toList :: Algebra TreeF [Int]
toList LeafF = []
toList (NodeF n ns ms) = ns ++ [n] ++ ms
```

that concatenates the left list with the singleton pivot and the right list. To sort a list we combine the anamorphism with the catamorphism:

```
qsort = cata toList . ana split
```

This gives us a (very inefficient) implementation of quicksort. We'll come back to it in the next section.

Infinite data structures

When studying algebras we relied on data structures that had a leaf component—or functors that, when acting on the initial object, would produce a result different from the initial object. When constructing recursive data structures we had to start somewhere, and that meant constructing the leaves first.

With coalgebras, we are free to drop this requirement. We no longer have to construct recursive data structures "by hand"—we have anamorphisms to do that for us. An endofunctor that has no leaves is perfectly acceptable: its coalgebras are going to generate infinite data structures.

Infinite data structures are representable in Haskell because of its laziness. Things are evaluated on the need-to-know basis. Only those parts of an infinite data structure that are explicitly demanded are calculated; the evaluation of the rest is kept in suspended animation.

To implement infinite data structures in strict languages, one must resort to representing values as functions—something Haskell does behind the scenes (these functions are called *thunks*).

Let's look at a simple example: an infinite stream of values. To generate it, we first define a functor that looks very much like the one we used to generate lists, except that it lacks the leaf component—the empty list constructor. You may recognize it as a product functor, with the first component fixed—it describes the stream's payload:

```
data StreamF a x = StreamF a x
  deriving Functor
```

An infinite stream is the fixed point of this functor.

```
type Stream a = Fix (StreamF a)
```

Here's a simple coalgebra that uses a single integer `n` as a seed:

```
step :: Coalgebra (StreamF Int) Int
step n = StreamF n (n+1)
```

It stores the seed as a payload, and seeds the next budding stream with `n + 1`.

The anamorphism for this coalgebra, seeded with zero, generates the stream of all natural numbers.

```
allNats :: Stream Int
allNats = ana step 0
```

In a non-lazy language this anamorphism would run forever, but in Haskell it's instantaneous. The incremental price is paid only when we want to retrive some of the data, for instance, using these accessors:

```
head :: Stream a -> a
head (In (StreamF a _)) = a

tail :: Stream a -> Stream a
tail (In (StreamF _ s)) = s
```

13.4 Hylomorphisms

The type of the output of an anamorphism is a fixed point of a functor, which is the same type as the input to a catamorphism. In Haskell, they are both described by the same data type, `Fix f`. Therefore it's possible to compose them together, as we've done when implementing quicksort. In fact, we can combine a coalgebra with an algebra in one recursive function called a *hylomorphism*:

```
hylo :: Functor f => Algebra f b -> Coalgebra f a -> a -> b
hylo alg coa = alg . fmap (hylo alg coa) . coa
```

We can rewrite quicksort as a hylomorphism:

```haskell
qsort = hylo toList split
```

Notice that there is no trace of the fixed point in the definition of the hylomorphism. Conceptually, the coalgebra is used to build (unfold) the recursive data structure from the seed, and the algebra is used to fold it into a value of type `b`. But because of Haskell's laziness, the intermediate data structure doesn't have to be materialized in full in memory. This is particularly important when dealing with very large intermediate trees. Only the branches that are currently being traversed are evaluated and, as soon as they have been processed, they are passed to the garbage collector.

Hylomorphisms in Haskell are a convenient replacement for recursive backtracking algorithms, which are very hard to implement correctly in imperative languages. We take advantage of the fact that designing a data structure is easier than following complicated flow of control and keeping track of our place in a recursive algorithm.

This way, data structures can be used to visualize complex flows of control.

The impedance mismatch

We've seen that, in the category of sets, the initial algebras don't necessarily coincide with terminal coalgebras. The identity functor, for instance, has the empty set as the carrier of the initial algebra and the singleton set as the carrier of its terminal coalgebra.

We have other functors that have no leaf components, such as the stream functor. The initial algebra for such a functor is the empty set as well.

In **Set**, the initial algebra is the subset of the terminal coalgebra, and hylomorphisms can only be defined for this subset. It means that we can use a hylomorphism only if the anamorphism for a particular coalgebra lands us in this subset. In that case, because the embedding of initial algebras in terminal coalgebras is injective, we can find the corresponding element in the initial algebra and apply the catamorphism to it.

In Haskell, however, we have one type, `Fix f`, combining both, the initial algebra and the terminal coalgebra. This is where the simplistic interpretation of Haskell types as sets of values breaks down.

Let's consider this simple stream algebra:

```haskell
add :: Algebra (StreamF Int) Int
add (StreamF n sum) = n + sum
```

Nothing prevents us from using a hylomorphism to calculate the sum of all natural numbers:

```haskell
sumAllNats :: Int
sumAllNats = hylo add step 1
```

It's a perfectly well-formed Haskell program that passes the type checker. So what value does it produce when we run it? (Hint: It's not $-1/12$.) The answer is: we don't know, because this program never terminates. It runs into infinite recursion and eventually exhausts the computer's resources.

This is the aspect of real-life computations that mere functions between sets cannot model. Some computer function may never terminate.

Recursive functions are formally described by *domain theory* as limits of partially defined functions. If a function is not defined for a particular value of the argument, it is said to return a bottom value $\bot$. If we include bottoms as special elements of every type (these are then called *lifted* types), we can say that our function `sumAllNats` returns a bottom of the type Int. In general, catamorphisms for infinite types don't terminate, so we can treat them as returning bottoms.

It should be noted, however, that the inclusion of bottoms complicates the categorical interpretation of Haskell. In particular, many of the universal constructions that rely on uniqueness of mappings no longer work as advertised.

The "bottom" line is that Haskell code should be treated as an illustration of categorical concepts rather than a source of rigorous proofs.

13.5 Terminal Coalgebra from Universality

The definition of an anamorphism can be seen as an expression of the universal property of the terminal coalgebra. Here it is, with the universal quantification made explicit:

```
ana :: Functor f => forall a. Coalgebra f a -> (a -> Fix f)
ana coa = In . fmap (ana coa) . coa
```

It says that, given any coalgebra, there is a mapping from its carrier to the carrier of the terminal coalgebra, Fix f. We know, from the Lambek's lemma, that this mapping is in fact a coalgebra morphism.

Let's uncurry this definition:

```
ana :: Functor f => forall a. (a -> f a, a) -> Fix f
ana (coa, x) = In (fmap (curry ana coa) (coa x))
```

In this form, we can take it as the definition of the carrier for the terminal coalgebra. We can replace Fix f with the type we are defining—let's call it Nu f. The type signature:

```
forall a. (a -> f a, a) -> Nu f
```

tells us that we can construct an element of Nu f from a pair (a -> f a, a). It looks just like a data constructor, except that it's polymorphic in a.

Data types with a polymorphic constructor are called *existential* types. To construct an element of an existential type, we have the option of picking the most convenient type—the type for which we have the data required by the constructor.

For instance, we can construct a term of the type Nu (StreamF Int) by picking Int as the convenient type, and providing the pair:

```
nuArgs :: (Int -> StreamF Int Int, Int)
nuArgs =  (\n -> StreamF n (n+1) , 0)
```

The clients of an existential data type have no idea what type was used in its construction. All they know is that such a type *exists*—hence the name. If they want to use an existential type, they have to do it in a way that is not sensitive to the choice that was made in its construction. In practice, it means that an existential type must carry with itself both the producer and the consumer of the hidden value.

This is indeed the case in our example: the producer is just the value of type a, and the consumer is the function a -> f a.

Naively, all that the clients could do with this pair, without any knowledge of what the type a was, is to apply the function to the value. But if f is a functor, they can do much more. They can repeat the process by applying the lifted function to the contents of f a, and so on. They end up with all the information that's contained in the infinite stream.

There are several ways of defining existential data types in Haskell. We can use the uncurried version of the anamorphism directly as the data constructor:

```
data Nu f where
  Nu :: forall a f. (a -> f a, a) -> Nu f
```

Notice that, in Haskell, if we explicitly quantify one type, all other type variables must also be quantified: here, it's the type constructor `f` (however, `Nu f` is not existential in `f`, since it's an explicit parameter).

We can also omit the quantification altogether:

```
data Nu f where
  Nu :: (a -> f a, a) -> Nu f
```

This is because type variables that are not arguments to the type constructor are automatically treated as existentials.

We can also use the more traditional form:

```
data Nu f = forall a. Nu (a -> f a, a)
```

(This one requires the quantification of `a`.)

Finally, although this syntax is not allowed in Haskell, it's often more intuitive to write it like this:

```
data Nu f = Nu (exists a. (a -> f a, a))
```

(Later we'll see that existential data types correspond to coends in category theory.)

The constructor of `Nu f` is literally the (uncurried) anamorphism:

```
anaNu :: Coalgebra f a -> a -> Nu f
anaNu coa a = Nu (coa, a)
```

If we are given a stream in the form of `Nu (Stream a)`, we can access its element using accessors functions. This one extracts the first element:

```
head :: Nu (StreamF a) -> a
head (Nu (unf, s)) =
  let (StreamF a _) = unf s
  in a
```

and this one advances the stream:

```
tail :: Nu (StreamF a) -> Nu (StreamF a)
tail (Nu (unf, s)) =
  let (StreamF _ s') = unf s
  in Nu (unf, s')
```

You can test them on an infinite stream of integers:

```
allNats = Nu nuArgs
```

13.6 Terminal Coalgebra as a Limit

In category theory we are not afraid of infinitities—we make sense of them.

At face value, the idea that we could construct a terminal coalgebra by applying the functor F infinitely many times to some object, let's say the terminal object 1, makes no sense. But the idea is very convincing: Applying F one more time is like adding one to infinity—it's still infinity. So, naively, this is a fixed point of F:

$$F(F^{\infty}1) \cong F^{\infty+1}1 \cong F^{\infty}1$$

To turn this loose reasoning into a rigorous proof, we have to tame the infinity, which means we have to define some kind of a limiting procedure.

As an example, let's consider the product functor:

$$F_A X = A \times X$$

Its terminal coalgebra is an infinite stream. We'll approximate it it by starting with the terminal object 1. The next step is:

$$F_A 1 = A \times 1 \cong A$$

which we could imagine is a stream of length one. We can continue with:

$$F_A(F_A 1) = A \times (A \times 1) \cong A \times A$$

a stream of length two, and so on.

This looks promising, but what we need is one object that would combine all these approximations. We need a way to glue the next approximation to the previous one.

Recall, from an earlier exercise, the limit of the "walking arrow" diagram. This limit has the same elements as the starting object in the diagram. In particular, consider the limit in this diagram:

$$\begin{tikzcd}
 & LimD_1 \arrow[dl, "\pi_0"'] \arrow[d, "\pi_1"] \\
1 & F1 \arrow[l, "!"]
\end{tikzcd}$$

(! is the unique morphism targeting the terminal object 1). This limit has the same elements as $F1$. Similarly, this limit:

$$\begin{tikzcd}
 & & LimD_2 \arrow[dll, "\pi_0"'] \arrow[dl, "\pi_1"] \arrow[d, "\pi_2"] \\
1 & F1 \arrow[l, "!"] & F(F1) \arrow[l, "F!"]
\end{tikzcd}$$

has the same elements as $F(F1)$.

We can continue extending this diagram to infinity. The limit of the infinite chain is the fixed point carrier of the terminal coalgebra.

$$\begin{tikzcd}
 & & T \arrow[dll, "\pi_0"'] \arrow[dl, "\pi_1"] \arrow[d, "\pi_2"] \arrow[drr, "\pi_n"] & & & \\
1 & F1 \arrow[l, "!"] & F(F1) \arrow[l, "F!"] & ... \arrow[l, "F(F!)"] & F^n 1 \arrow[l] & ... \arrow[l, "F^n!"]
\end{tikzcd}$$

The proof of this fact can be obtained from the analogous proof for initial algebras by reversing the arrows.