Previously, we talked about the construction of initial algebras. The dual construction is that of terminal coalgebras. Just like an algebra can be used to fold a recursive data structure into a single value, a coalgebra can do the reverse: it lets us build a recursive data structure from a single seed.

Here's a simple example. We define a tree that stores values in its nodes

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

We can build such a tree from a single list. We can choose the algorithm in such a way that the tree is ordered in a particular way

```
split :: Ord a => [a] -> Tree a
split [] = Leaf
split (a : as) = Node a (split l) (split r)
  where (l, r) = partition (<a) as
```

A traversal of this tree will produce a sorted list. We'll get back to this example after working out some theory behind it.

1. The functor

The tree in our example can be derived from the functor

```
data TreeF a x = LeafF | NodeF a x x
```

```
instance Functor (TreeF a) where
  fmap _ LeafF = LeafF
  fmap f (NodeF a x y) = NodeF a (f x) (f y)
```

Let's simplify the problem and forget about the payload of the tree. We're interested in the functor

```
data F x = LeafF | NodeF x x
```

Remember that, in the construction of the initial algebra, we were applying consecutive powers of the functor to the initial object. The dual construction of the terminal coalgebra involves applying powers of the functor to the terminal object: the unit type `()` in Haskell, or the singleton set 1 in *Set*.

Let's build a few such trees. Here are a some terms generated by the single power of `F`

```
w1, u1 :: F ()
w1 = LeafF
u1 = NodeF () ()
```

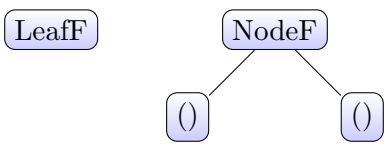


Figure 1. Trees generated by `F ()`

And here are some generated by the square of `F` acting on `()`

```
w2, u2, t2, s2 :: F (F ())

w2 = LeafF
u2 = NodeF w1 w1
t2 = NodeF w1 u1
s2 = NodeF u1 u1
```

Or, graphically

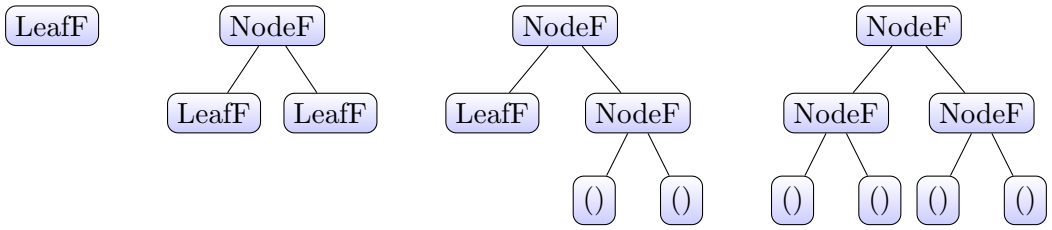


FIGURE 2. Examples of trees generated by $F^2 1$ or `F (F ())`

Notice that we are getting two kinds of trees, ones that have units in their leaves and ones that don't. Units may appear only at the $(n+1)$-st layer (root being the first layer) of F^n.

We are also getting some duplication between different powers of F. For instance, we get a single `LeafF` at the F level and another one at the F^2 level (in fact, at every consecutive level after that as well). The node with two `LeafF` leaves appears at every level starting with F^2, and so on. The trees without unit leaves are the ones we will be interested in, as they contribute to the terminal coalgebra. We'll construct it as a limit of an ω-chain.

2. Terminal coalgebra as a limit

As was the case with initial algebras, we'll construct a chain of powers of F, except that we'll start with the terminal rather than the initial object, and we'll use a different morphism to link them together. By definition, there is only one morphism from any object to the terminal object. In category theory, we'll call this morphism $¡ \colon a \to 1$ (upside-down exclamation mark) and implement it in Haskell as

```
unit :: a -> ()
unit _ = ()
```

First, we'll use ¡ to connect $F1$ to 1, then lift ¡ to connect $F^2 1$ to $F1$, and so on, using $F^n ¡$ to transform $F^{n+1}1$ to $F^n 1$.

$$1 \xleftarrow[¡]{} F1 \xleftarrow[F¡]{} F^2 1 \xleftarrow[F^2¡]{} F^3 1 \leftarrow\!\!\text{-----} \ \ldots$$

Let's see how it works in Haskell. Applying `unit` directly to `F ()` turns it into `()`.

Values of the type `F (F ())` are mapped to values of the type `F ()`

```
w2' = fmap unit w2
> LeafF
u2' = fmap unit u2
```

```
> NodeF () ()
t2' = fmap unit t2
> NodeF () ()
s2' = fmap unit s2
> NodeF () ()
```

and so on.

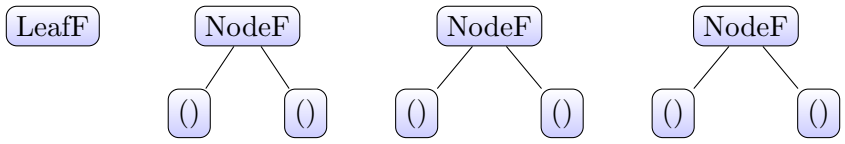


FIGURE 3. Examples of `fmap unit` acting on trees from the previous figure. Notic

The following pattern emerges. $F^n 1$ contains trees that end with either leaves (at any level) or values of the unit type (only at the lowest, $(n+1)$-st layer). The lifted morphism $F^{n-1}¡$ (the $(n-1)$st power of `fmap` acting on `unit`) operates on the nth level of a tree. It turns leaves (at that level) and two-unit-nodes into single units.

Alternatively, we can look at the inverse mapping. Observe that all trees at the F^2 level can be generated from the trees at the F level by replacing every unit `()` with either a leaf `LeafF` or a node `NodeF () ()`. It's as if a unit were a universal seed that can either sprout a leaf or a two-unit-node. We'll see later that this process of growing recursive data structures from seeds corresponds to anamorphisms. Here, the terminal object plays the role of a universal seed that may give rise to many parallel futures. These correspond to the inverse image of the lifted `unit`.

Now that we have an ω-chain, we can define its limit. It's easier to understand a limit in the category of sets. A limit is a set of cones whose apex is the singleton set.

The simplest limit is a product of sets. A cone with a singleton apex corresponds to a selection of elements, one per set. This agrees with our understanding of a product as a set of tuples.

A limit of a directed finite chain is just the starting set of the chain. This is because all projections, except for the starting one, are determined by commuting triangles. In the example below, π_b is given by:

$$\pi_b = f_1 \circ \pi_a$$

and so on. So every cone from 1 is fully determined by a particular function $1 \to a$, and the set of such functions is isomorphic to a.

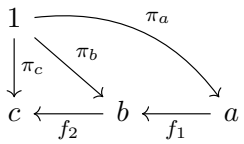
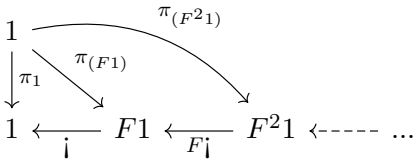


FIGURE 4. The limit of this chain is isomorphic to its starting set a

Things are more interesting when the chain is infinite, and there is no starting object, as in the case of our ω-chain. The limit of this chain (if it exists) is the terminal coalgebra for the functor F.



In this case, the inverted view is very helpful. We can look at a particular power of F as a set of trees generated by expanding the universal seeds embedded in the trees from the lower power of F. Those trees that had no seeds, only $LeafF$ leaves, are just propagated without change. So the limit will definitely contain all these trees. But it will also contain infinite trees. These correspond to cones that select ever growing trees in which there are always some seeds that are replaced with double-seed-nodes rather than `LeafF` leaves.

Compare this with the initial algebra construction which only generated finite trees. The terminal coalgebra for the functor `TreeF` is larger than the initial algebra for the same functor.

We have also seen a functor whose initial algebra was an empty set

```
data StreamF a x = ConsF a x
```

This functor has a well-defined terminal coalgebra. The n-th power of `(StreamF a)` acting on `()` consists of lists

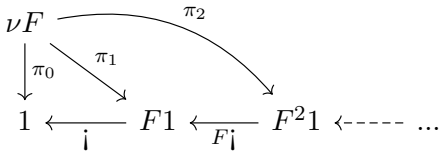
```
ConsF a1 (ConsF a2 (... (ConsF an ())...))
```

The appropriate lifting of `unit` acting on such a list replaces `(ConsF an ())` with `()` thus shortening the list by one item. Its "inverse" replaces the seed `()` with any value of type `a` (so it's a *multi-valued* inverse). The limit is isomorphic to an infinite stream of `a`. In Haskell it can be written as a recursive data structure

```
data Stream a = ConsF a (Stream a)
```

3. Anamorphism

The limit of a diagram is defined as a universal cone. In our case this would be the cone consisting of the object we'll call νF with a set of projections π_n



such that any other cone factors through νF. We want to show that νF (if it exists) is a terminal coalgebra.

First, we have to show that νF is indeed a coalgebra, that is, there exists a morphism

$$k \colon \nu F \to F(\nu F)$$

We can apply F to the whole diagram. If F preserves ω-limits, then we get a universal cone with the apex $F(\nu F)$ and the ω-chain with $F1$ on the left. But our original object νF forms a cone over the same chain (with the projection π_0 to spare). Therefore there must be a unique mapping from it to $F(\nu F)$.

The coalgebra $(\nu F, k)$ is terminal if there is a unique morphism from any other coalgebra to it. Consider, for instance, a coalgebra $(A, \kappa\colon A \to FA)$. With this coalgebra, we can construct an ω-chain

$$A \xrightarrow{\kappa} FA \xrightarrow{F\kappa} F^2A \xrightarrow{F^2\kappa} F^3A \dashrightarrow \ldots$$

We can connect the two omega chains using the terminal morphism from A to 1 and its liftings by powers of F.

$$\begin{array}{ccccccccc} A & \xrightarrow{\kappa} & FA & \xrightarrow{F\kappa} & F^2A & \xrightarrow{F^2\kappa} & F^3A & \dashrightarrow & \ldots \\ \downarrow{\scriptstyle ¡} & & \downarrow{\scriptstyle F¡} & & \downarrow{\scriptstyle F^2¡} & & \downarrow{\scriptstyle F^3¡} & & \\ 1 & \xleftarrow[¡]{} & F1 & \xleftarrow[F¡]{} & F^21 & \xleftarrow[F^2¡]{} & F^31 & \dashleftarrow & \ldots \end{array}$$

Notice that all squares in this diagram commute. The leftmost one commutes because 1 is the terminal object, therefore the mapping from A to it is unique, and the composite $¡ \circ F¡ \circ \kappa$ must be the same as ¡. A is therefore an apex of a cone over our original ω-chain. There must be a unique morphism from A to the limit of this ω-chain, νF. This morphism is in fact a coalgebra morphism and is called the *anamorphism*.

4. Adjunction

The constructions of initial algebras and terminal coalgebras can be compactly described using adjunctions.

There is an obvious forgetful functor U from the category of F-algebras C^F to Set. Under certain conditions, the left adjoint free functor Φ exists

$$C^F(\Phi x, a) \cong C(x, Ua)$$

This adjunction can be evaluated at the initial object (the empty set in Set).

$$C^F(\Phi 0, a) \cong C(0, Ua)$$

This shows that there is a unique algebra morphism—the catamorphism— from $\Phi 0$ to any algebra a. This is because the hom-set $C(0, Ua)$ is a singleton for every a. This shows that $\Phi 0$ is the initial algebra νF.

Conversely, there is a cofree functor Ψ

$$C_F(c, \Psi x) \cong C(Uc, x)$$

It can be evaluated at a terminal object

$$C_F(c, \Psi 1) \cong C(Uc, 1)$$

showing that there is a unique coalgebra morphism—the anamorphism—from any coalgebra c to $\Psi 1$. This shows that $\Psi 1$ is the terminal coalgebra νF.

5. Fixed point

Lambek's lemma works for both, initial algebras and terminal coalgebras. It states that their structure maps are isomorphisms, therefore their carriers are fixed points of the functor F

$$\mu F \cong F(\mu F)$$
$$\nu F \cong F(\nu F)$$

The difference is that μF is the least fixed point, and νF is the greatest fixed point of F. They are, in principle, different. And yet, in a programming language like Haskell, we only have one recursively defined data structure defining the fixed point

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

So which one is it?

We can define both the catamorphisms from-, and anamorphisms to-, the fixed point:

```
type Algebra f a = f a -> a

cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . unfix
```

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

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

so it seems like `Fix f` is both initial as the carrier of an algebra and terminal as the carrier of a coalgebra. But we know that there are elements of νF that are not in μF—namely infinite trees and infinite streams—so the two fixed points are not isomorphic and cannot be both described by the same `Fix f`.

However, they are not unrelated. Because of the Lambek's lemma, the initial algebra $(\mu F, j)$ gives rise to a coalgebra $(\mu F, j^{-1})$, and the terminal coalgebra $(\nu F, k)$ generates an algebra $(\nu F, k^{-1})$.

Because of universality, there must be a (unique) algebra morphism from the initial algebra $(\mu F, j)$ to $(\nu F, k^{-1})$, and a unique coalgebra morphism from $(\mu F, j^{-1})$ to the terminal coalgebra $(\nu F, k)$. It turns out that these two are given by the same morphism $f \colon \mu F \to \nu F$ between the carriers. This morphism satisfies the equation

$$k \circ f \circ j = Ff$$

which makes it both an algebra and a coalgebra morphism

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{Ff} & F(\nu F) \\ \downarrow{\scriptstyle j} & & {\scriptstyle k}\uparrow \\ \mu F & \xrightarrow{f} & \nu F \end{array}$$

Furthermore, it can be show that, in *Set*, f is injective: it embeds μF in νF. This corresponds to our observation that νF contains μF plus some infinite data structures.

The question is, can `Fix f` describe infinite data? The answer depends on the nature of the programming language: infinite data structures can only exist in a lazy language. Since Haskell is lazy, `Fix f` corresponds to the *greatest fixed point*. The least fixed point forms a subset of `Fix f` (in fact, one can define a metric in which it's a dense subset).

This is particularly obvious in the case of a functor that has no terminating leaves, like the stream functor.

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

We've seen before that the initial algebra for `StreamF a` is empty, essentially because its action on `Void` is uninhabited. It does, however have a terminal coalgebra. And the fixed point of the stream functor, in Haskell, generates infinite streams

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

How do we know that? Because we can construct an infinite stream using an anamorphism. Notice that, unlike in the case of a catamorphism, the recursion in an anamorphism doesn't have to be well founded and, indeed, in the case of a stream, it never terminates. This is why this won't work in an eager language. But it works in Haskell. Here's a coalgebra whose carrier is `Int`

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

It generates an infinite stream of integers

```
ints = ana coaInt 0
```

Of course, in Haskell, the work is not done until we demand some values. Here's the function that extracts the head of the stream:

```
hd :: Stream a -> a
hd (Fix (StreamF x _)) = x
```

And here's one that advances the stream

```
tl :: Stream a -> Stream a
tl (Fix (StreamF _ s)) = s
```

6. Bibliography

- Adámek, Introduction to coalgebra