INITIAL ALGEBRA AS A DIRECTED COLIMIT

BARTOSZ MILEWSKI

There is a bit of folklore about algebras in Haskell, which says that both the initial algebra and the terminal coalgebra for a given functor are defined by the same fixed point formula. This works for most common cases, but is not true in general. What is definitely true is that they are both fixed points–this result is called the Lambek's lemma–but there may be many fixed points. The initial algebra is the *least fixed point*, and the terminal coalgebra is the *greatest fixed point*.

In this series of blog posts I will explore the ways one can construct these (co-)algebras using category theory and illustrate it with Haskell examples.

In this first installment, I'll go over the construction of the initial algebra.

1. A functor

Let's start with a simple functor that generates binary trees. Normally, we would store some additional data in a tree (meaning, the functor would take another argument), either in nodes or in leaves, but here we're just interested in pure shapes.

```
data F a = Leaf
         | Node a a
  deriving Show
```

Categorically, this functor can be written as a *coproduct* (sum) of the terminal object 1 (singleton) and the *product* of a with itself, here written simply as a^2

$$Fa = 1 + a^2$$

The lifting of functions is given by this implementation of `fmap`

```
instance Functor F where
  fmap _ Leaf       = Leaf
  fmap f (Node x y) = Node (f x) (f y)
```

We can use this functor to build arbitrary level trees. Let's consider, for instance, terms of type `F Int`. We can either build a `Leaf`, or a `Node` with two numbers in it

```
x1, y1 :: F Int
x1 = Leaf
y1 = Node 1 2
```

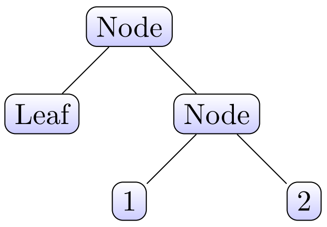
With those, we can build next-level values of the type F^2a or, in our case, `F (F Int)`

```
x2, y2 :: F (F Int)
x2 = Leaf
y2 = Node x1 y1
```

We can display `y2` directly using `show`

```
> Node Leaf (Node 1 2)
```

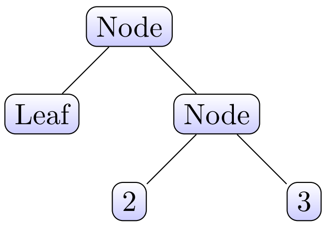
or draw the corresponding tree



Since F is an endofunctor, so is F^2. Lifting a function $f\colon a \to b$ to F^2 can be implemented by applying `fmap` twice. Here's the action of the function `(+1)` on our test values

```
fmap (fmap (+1)) x2
> Leaf
fmap (fmap (+1)) y2
> Node Leaf (Node 2 3)
```

or, graphically,



You can see that `Leaf`s at any level remain untouched; only the contents of bottom `Node`s in the tree are transformed.

2. The colimit construction

The carrier of the initial algebra can be constructed as a colimit of an infinite sequence. This sequence is constructed by applying powers of F to the initial object which we'll denote as 0. We'll first see how this works in our example.

The initial object in Haskell is defined as a type with no data constructor (we are ignoring the question of non-termination in Haskell).

```
data Void
  deriving Show
```

In Set, this is just an empty set.

The `Show` instance for `Void` requires the pragma

```
{-# language EmptyDataDeriving #-}
```

Even though there are no values of the type `Void`, we can still construct a value of the type `F Void`

```
z1 :: F Void
z1 = Leaf
```

This degenerate version of a tree can be drawn as

This illustrates a very important property of our F: Its action on an empty set does not produce an empty set. This is what allows us to generate a non-trivial sequence of powers of F starting with the empty set.

Not every functor has this property. For instance, the construction of the initial algebra for the functor

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

will produce an uninhabited type (empty set). Notice that this is different from its terminal coalgebra, which is the infinite stream

```
data Stream a = Cons a (Stream a)
```

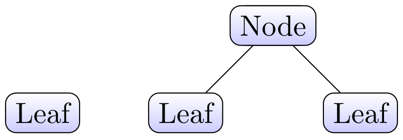
This is an example of a functor whose initial algebra is not the same as the terminal coalgebra.

Double application of our `F` to `Void` produces, again, a `Leaf`, as well as a `Node` that contains two `Leaf`s.

```
z2, v2 :: F (F Void)
z2 = Leaf

v2 = Node z1 z1
> Node Leaf Leaf
```

Graphically,



In general, powers of F acting on `Void` generate trees which terminate with `Leaf`s, but there is no possibility of having terminal `Node`s). Higher and higher powers of F acting on `Void` will eventually produce any tree we can think of. But for any given power, there will exist even larger trees that are not generated by it.

In order to get all the trees, we could try to take a sum (a coproduct) of infinitely many powers. Something like this

$$\sum_{n=0}^{\infty} F^n 0$$

The problem is that we'd also get a lot of duplication. For instance, we saw that `z1` was the same tree as `z2`. In general, a single `Leaf` is produced at all non-zero powers of F acting on `Void`. Similarly, all powers of F greater than one produce a single node with two leaves, and so on. Once a particular tree is produced at some power of F, all higher powers of F will also produce it.

We have to have a way of identifying multiply generated trees. This is why we need a *colimit* rather than a simple coproduct.

As a reminder, a coproduct is defined as a universal cocone. Here, the base of the cocone is the set of all powers of F acting on 0 (Haskell `Void`).

$$\begin{tikzcd}
0 \arrow[dr, "\iota_0"'] & F0 \arrow[d, "\iota_{(F0)}"] & F^2 0 \arrow[dl, "\iota_{(F^2 0)}"] & \dots \\
& \sum_{n=0}^{\infty} F^n 0 & &
\end{tikzcd}$$

In a more general colimit, the objects in the base of the cocone may be connected by morphisms.

Coming from the initial object, there can be only one morphism. We'll call this morphism ! or, in Haskell, `absurd`

```
absurd :: Void -> a
absurd a = case a of {}
```

This definition requires another pragma

```
{-# language EmptyCase #-}
```

We can construct a morphism from $F0$ to $F^2 0$ as a lifting of !, $F!$. In Haskell, the lifting of `absurd` doesn't change the shape of trees. Here it is acting on a leaf

```
z1' :: F (F Void)
z1' = fmap absurd z1
> Leaf
```

We can continue this process of lifting `absurd` to higher and higher powers of F

```
z2', v2' :: F (F (F Void))

z2' = fmap (fmap absurd) z2
> Leaf

v2' = fmap (fmap absurd) v2
> Node Leaf Leaf
```

We can construct an infinite chain (this kind of directed chain indexed by natural numbers is called an ω-chain)

$$\begin{tikzcd}
0 \arrow[r, "!"] & F0 \arrow[r, "F!"] & F^2 0 \arrow[r, dashed] & \dots
\end{tikzcd}$$

We can use this chain as the base of our cocone. The colimit of this chain is defined as the universal cocone. We will call the apex of this cocone μF

$$\begin{tikzcd}
0 \arrow[r, "!"] \arrow[d, "\iota_0"] & F0 \arrow[r, "F!"] \arrow[dl, "\iota_{(F0)}"] & F^2 0 \arrow[r, dashed] \arrow[dll, bend left, "\iota_{(F^2 0)}"] & \ldots \\
\mu F & & &
\end{tikzcd}$$

In *Set* these constructions have simple interpretations. A coproduct is a discriminated union. A colimit is a discriminated union in which we identify all those injections that are connected by morphisms in the base of the cocone. For instance

$$\iota_0 = \iota_{(F0)} \circ !$$

$$\iota_{(F0)} = \iota_{(F^2 0)} \circ F!$$

and so on.

Here we use the lifted `absurd` (or ! in the picture above) as the morphisms that connect the powers of F acting of `Void` (or 0 in the picture).

These are exactly the identifications that we were looking for. For instance, $F!$ maps the leaf generated by $F0$ to the leaf which is the element of $F^2 0$. Or, translating it to Haskell, `(fmap absurd)` maps the leaf generated by `F Void` to the leaf generated by `F (F Void)`, and so on.

All trees generated by the n'th power of F are injected into the $n+1$'st power of F by `absurd` lifted by the nth power of F.

The colimit is formed by equivalence classes with respect to these identifications. In particular, there is a class for a degenerate tree consisting of a single leaf whose representative can be taken from `F Void`, or from `F (F Void)`, or from `F (F (F Void))` and so on.

3. Initiality

The colimit μF is exactly the initial algebra for the functor F. This follows from the universal property of the colimit. First we will show that for any algebra $(A, \alpha \colon FA \to A)$ there is a unique morphism from μF to A. Indeed, we can build a cocone with A at its apex and the injections given by

$$!$$

$$\alpha \circ F!$$

$$\alpha \circ F\alpha \circ F^2!$$

and so on...

$$\begin{tikzcd}
0 \arrow[r, "!"] \arrow[d, "!"] & F0 \arrow[r, "F!"] \arrow[d, "F!"] & F^2 0 \arrow[r, dashed] \arrow[d, "F^2!"] & \ldots \\
A & FA \arrow[l, "\alpha"'] & F^2 A \arrow[l, "F\alpha"'] & \ldots \arrow[l, dashed]
\end{tikzcd}$$

Since the colimit μF is defined by the universal cocone, there is a unique morphism from it to A. It can be shown that this morphism is in fact an algebra morphism. This morphism is called a *catamorphism*.

4. Fixed Point

Lambek's lemma states that the initial algebra is a fixed point of the functor that defines it

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

This can also be seen directly, by applying the functor to every object and morphism in the ω-chain that defines the colimit. We get a new chain that starts at $F0$

$$\begin{tikzcd} F0 \arrow[r, "F!"] & F^2 0 \arrow[r, "F^2!"] & F^3 0 \arrow[r, dashed] & ... \end{tikzcd}$$

But the colimit of this chain is the same as the colimit μF of the original chain. This is becuase we can always add back the initial object to the chain, and define its injection ι_0 as the composite

$$\iota_0 = \iota_{(F0)} \circ !$$

On the other hand, if we apply F to the whole universal cocone, we'll get a new cocone with the apex $F(\mu F)$. In principle, this cocone doesn't have to be universal, so we cannot be sure that $F(\mu F)$ is a colimit. If it is, we say that F *preserves* the particular type of colimit—here, the ω-colimit.

Remember: the image of a cocone under a functor is always a cocone (this follows from functoriality). Preservation of colimits is an additional requirement that the image of a *universal* cocone be *universal.*

The result is that, if F preserves ω-colimits, then the initial algebra μF is a fixed point of F

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

because both sides can be obtained as a colimit of the same ω-chain.

5. Bibliography

- Adámek, Milius, Moss, Initial Algebras, Terminal Coalgebras, and the Theory of Fixed Points of Functors