FIXED POINTS OF ENDOFUNCTORS

BARTOSZ MILEWSKI

Previously, we've been discussing the construction of initial algebras and terminal coalgebras in terms of colimits and limits. By Lambek's lemma, both constructions lead to fixed points of the functor in question (the least one and the greatest one, respectively). In this installment we'll dig into alternative definitions of these (co-)algebras, first in Haskell, to build some intuition, and then using (co-)end calculus in category theory.

1. Initial algebra and catamorphisms

The initial algebra for a given functor `f` is defined by its universal property. Take any other algebra with the carrier `a` and the structure map `f a -> a`. There must exist a unique algebra morphism from the initial algebra to it. Let's call the carrier for the initial algebra `Mu f`. The unique morphism is given by the catamorphism, with the following signature

```
cata :: Functor f => (f a -> a) -> (Mu f -> a)
```

or, after rearranging the arguments,

```
cata :: Functor f => Mu f -> (f a -> a) -> a
```

This universal property can be used in the definition of `Mu f`

```
newtype Mu f = Mu (forall a. (f a -> a) -> a)
```

A value of this type is a polymorphic function. This function, for any type `a` and any function `f a -> a`, will produce a value of type `a`. In other words, it's a *gigantic product* of all possible catamorphisms.

Notice that this definition requires the following language pragma

```
{-# language RankNTypes #-}
```

This is an example of a definition in terms of the *mapping out* property. An object is uniquely defined by the outgoing morphisms to all other objects. This is the consequence of the Yoneda lemma.

It's easy to define a catamorphism in terms of `Mu`

```
cataMu :: Functor f => Algebra f a -> Mu f -> a
cataMu alg (Mu cata) = cata alg
```

This is because `Mu` is a polymorphic catamorphism, and the compiler will automatically instantiate it for the proper type.

The definition of `Mu f` is workable, if a bit unwieldy—the challenge is to construct terms of type `Mu f`. I'll show you how to do it using a list as an example. We'll convert a list of `a` to a term of type `Mu (ListF a)`.

As a reminder, the functor `ListF a` is defined as

```
data ListF a x = NilF | ConsF a x
```

In essence, if we can provide the implementation of all possible folds for a given list, we have defined the list. This is a little like defining a function by specifying its integrals with all possible test functions (that's how distributions are defined in calculus).

```
fromList :: [a] -> Mu (ListF a)
fromList as = Mu (\alg -> go alg as)
  where go alg [] = alg NilF
        go alg (n: ns) = alg (ConsF n (go alg ns))
```

You can now verify that

```
cataMu sumAlg (fromList [1..10])
```

produces the correct result for the following algebra

```
sumAlg :: Algebra (ListF Int) Int
sumAlg NilF = 0
sumAlg (ConsF a x) = a + x
```

2. Terminal coalgebra and anamorphisms

The terminal coalgebra, on the other hand, is defined by its *mapping in* property. We start by looking at the signature of the anamorphism. `Nu f` is the terminal coalgebra

```
ana :: Functor f => (a -> f a) -> (a -> Nu f)
```

We can uncurry it to get

```
ana :: Functor f => (a -> f a, a) -> Nu f
```

We want to define `Nu` as a gigantic product of anamorphisms, one anamorphism for every type. To do that, we have to universally quantify these anamorphisms over all types `a`

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

We can now apply the standard trick: a function from a sum type is equivalent to a product of functions. Here we start with a gigantic product of functions, so we need a gigantic sum. A gigantic sum is known as the existential type. So, symbolically, in pseudo Haskell, we would define `Nu` as

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

This is how we read it: the data constructor `Nu` is a function that takes an existential type and produces a value of the type `Nu f`

```
Nu :: (exists a. (a -> f a, a)) -> Nu f
```

Indeed, this is equivalent to a polymorphic function (parenthesis for emphasis)

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

which can be curried to give us back all the anamorphisms

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

Existential types can be encoded in Haskell using Generalized Algebraic Data Types or GADTs

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

In a GADT, any type parameter that doesn't occur in the header (here, `a`) is automatically existentially quantified. The use of GADTs requires the language pragma

```
{-# language GADTs #-}
```

Let's analyze this definition. Since an existential type provides no access to the hidden type, it has to contain both the "producer" and the "consumer" for this type. Here we have the existential

```
exists a. (a -> f a, a)
```

When we are given a value of this type, we have some value of the unknown type `a` on the producer side, and the function `g :: a -> f a` on the consumer side. Since we don't know the type of `a`, all we can do is to apply `g` to it, to obtain the term of the type `f a`.

Since `f` is a functor, we can lift `g` and apply it again, to get something of the type `f (f a)`. Continuing this process, we can obtain arbitrary powers of the functor `f` acting on `a`. This is how this definition is equivalent to the recursive data type. In fact, we can write a recursive function that converts `Nu f` to `Fix f`

```
toFix :: Functor f => Nu f -> Fix f
toFix (Nu coa a) = ana coa a
  where ana coa = Fix . fmap (ana coa) . coa
```

An anamorphism in terms of `Nu` is simply given by

```
anaNu :: Functor f => Coalgebra f a -> a -> Nu f
anaNu coa a = Nu coa a
```

This literally does nothing, just stores a coalgebra and a seed. But that's how existential types work: their construction is easy. The client provides one particular value of one particular type. The whole logic is in the deconstruction, or the usage of this value.

Let's apply this method to our earlier example of an infinite stream generated by the functor

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

An infinite stream of `a` is the terminal coalgebra for this functor

```
type StreamNu a = Nu (StreamF a)
```

Here's a coalgebra we used before to generate arithmetic sequences

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

We can use it to generate a stream of natural numbers, but this time we'll express it using `Nu`

```
intStream :: Nu (StreamF Int)
intStream = Nu coaInt 0
```

We can either define accessors for this stream directly, as in

```
hd :: Nu (StreamF a) -> a
hd (Nu coa a) =
  let StreamF a' _ = coa a in a'
```

```
tl :: Nu (StreamF a) -> Nu (StreamF a)
tl (Nu coa a) =
  let StreamF a' x = coa a in Nu coa x
```

or use `toFix` to convert it to the more familiar form.

The fact that we can convert `Nu f`, which is the greatest fixed point of `f`, to `Fix f` is not a surprise, since we have already established that, in Haskell, `Fix f` is the *greatest* fixed point. Interestingly enough, we can also squeeze `Fix f` into `Mu f`, which is the *least fixed point* of `f`.

```
fromFix :: Functor f => Fix f -> Mu f
fromFix fx = Mu (flip cata fx)
```

This is only possible at the cost of `cata` occasionally hitting the bottom $\perp$, or never terminating, which is okay in Haskell.

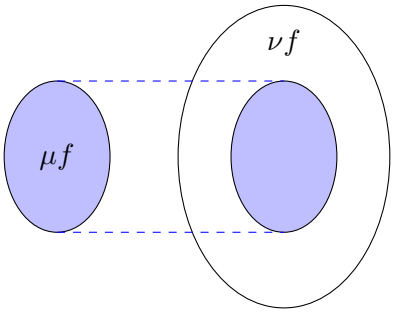
We can even compose `fromFix` with `toFix` to construct a direct mapping from `Nu f` to `Mu f`

```
fromNuToMu :: Functor f => Nu f -> Mu f
fromNuToMu (Nu coa a) = Mu $ flip cata (ana coa a)
  where ana  coa = Fix. fmap (ana coa) . coa
        cata alg = alg . fmap (cata alg) . unFix
```

We can further simplify this by cutting out the `Fix` in the middle

```
fromNuToMu :: Functor f => Nu f -> Mu f
fromNuToMu (Nu coa a) = Mu $ \alg -> hylo alg coa a
  where hylo alg coa = alg . fmap (hylo alg coa) . coa
```

We know from the previous installment that there is a canonical mapping (an injection) from `Mu f` to `Nu f`.



It turns out that `fromNuToMu` is its inverse, proving that, in Haskell (but *not* in *Set*), these two fixed points are isomorphic.

The fact that, in Haskell, we can map a terminal coalgebra to an initial algebra is the reason why we have hylomorphisms. A hylomorphism uses a coalgebra to build a recursive data structure from a seed, and then applies an algebra to fold it. It applies a catamorphism to the result of an anamorphism, which is only possible because they have the same type in the middle.

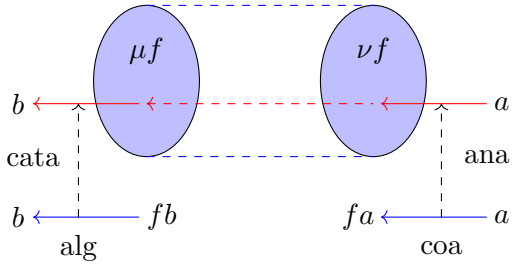


FIGURE 1. A hylomorphism (in red), assuming that μf is isomorphic to νf

Here's a quick exercise that was hinted at in the previous installment. This is a tree functor

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

We define a coalgebra that uses a list of `a` as a seed, and partitions this list between its two children, storing the pivot at the node

```
split :: Ord a => Coalgebra (TreeF a) [a]
split [] = LeafF
split (a : as) = NodeF a l r
  where (l, r) = partition (<a) as
```

Here's the algebra that concatenates child lists with the pivot in between

```
combine :: Algebra (TreeF Int) [Int]
combine LeafF = []
combine (NodeF a b c) = b ++ [a] ++ c
```

And this is quicksort implemented using functions defined in this section

```
qsort = cataMu combine . fromNuToMu . anaNu split
```

This is an example of a hylomorphism. The actual implementation of a hylomorphism in the Haskell library is more efficient, as it fuses the the recursive functions

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

3. End/Coend formulation

Let's rewrite `Mu` using GADTs

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

In category theory, this is called an end

$$\mu f = \int_a a^{C(fa,a)}$$

In general an end over a profunctor `p a b` is defined in Haskell as

```
data End p where
  End :: (forall a. p a a) -> End p
```

Here, the profunctor is

$$pab = b^{C(fb,a)}$$

where a and b are objects in the category C and $C(fb, a)$ is the hom-set (the set of morphisms) from fb to a. In Haskell, we would write it as

```
data P f a b = P ((f b -> a) -> b)
```

A profunctor is covariant in its second argument and contravariant in the first

```
instance Functor f => Profunctor (P f) where
  dimap g g' (P h) = P (\j -> g'( h (g . j . fmap g')))
```

The Haskell function type from `(f b -> a)` to `b` is represented in category theory as a *power*

$$b^{C(fb,a)}$$

We have to use a power because b is an object whereas $C(fb, a)$ is a set. The power is defined by the isomorphism

$$C(x, a^s) \cong Set(s, C(x, a))$$

The end behaves like a gigantic product in the sense that it defines projections π_a for every a. In our case, these projections are the catamorphisms

$$\pi_a \colon \mu f \to a^{C(fa,a)}$$

Indeed, such a projection is a member of the hom-set

$$C(\mu f, a^{C(fa,a)})$$

By the definition of the power, it corresponds to an element of the set

$$Set(C(fa, a), C(\mu f, a))$$

which maps algebras, members of $C(fa, a)$, to mappings out of the initial algebra, $C(\mu f, a)$. These are the catamorphisms.

Similarly, we can rewrite Nu

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

as a coend

$$\nu f = \int^a C(a, fa) \cdot a$$

over the profunctor

$$qab = C(a, fb) \cdot b$$

where the dot is the copower, defined by the isomorphism

$$C(s \cdot a, x) \cong Set(s, C(a, x))$$

In Haskell, this profunctor would be defined as

```
data Q f a b = Q (a -> f b) b
```

```
instance Functor f => Profunctor (Q f) where
  dimap g g' (Q h b) = Q (fmap g' . h . g) (g' b)
```

and the coend is an existential data type defined using a GADT

```
data Coend p where
  Coend :: p a a -> Coend p
```

A coend behaves like a gigantic sum, in the sense that it defines injections ι_a for every a. In our case, these injections are anamorphisms

$$\iota_a \colon C(a, fa) \cdot a \to \nu f$$

Indeed ι_a is a member of the hom-set

$$C(C(a, fa) \cdot a, \nu f)$$

or, using the definition of the copower, an element of the set

$$Set(C(a, fa), C(a, \nu f))$$

which we recognize as an anamorphism.

4. Hylomorphism categorically

If the mapping from from the terminal coalgebra ν to the initial algebra μ exists in a particular category C, it is an element of the following hom-set

$$C\Big(\int^a C(a, fa) \cdot a, \int_b b^{C(fb,b)}\Big)$$

We'll apply some coend-fu to transform it into a more familiar form.

We use the co-continuity of the hom-functor to move the coend out of the left side of the hom-set. It becomes an end in the process

$$\int_a C\Big(C(a, fa) \cdot a, \int_b b^{C(fb,b)}\Big)$$

Using continuity we move the end out of the right side of the hom-set

$$\int_{a,b} C\Big(C(a, fa) \cdot a, b^{C(fb,b)}\Big)$$

Using the definition of the copower

$$C(s \cdot a, x) \cong Set(s, C(a, x))$$

we get

$$\int_{a,b} Set\Big(C(a, fa), C(a, b^{C(fb,b)})\Big)$$

And using the definition of the power

$$C(x, a^s) \cong Set(s, C(x, a))$$

we get

$$\int_{a,b} Set\Big(C(a, fa), Set\big(C(fb, b), C(a, b)\big)\Big)$$

Finally, applying the currying adjunction in Set we get

$$\int_{a,b} Set\Big(C(a, fa) \times C(fb, b), C(a, b)\Big)$$

which has the form reminiscent of the Haskell definition of a hylomorphism

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

This makes sense in Haskell even though it won't work in Set, because Haskell is okay with non-terminating recursion.

I had a discussion with Derek Elkins, who pointed me to the paper by Fokkinga and Meijer, in which they discuss the existence of hylomorphisms in CPOs.

5. BIBLIOGRAPHY

- Fosco Loregian, Coend calculus
- Fokkinga, Meijer, Program Calculation Properties of Continuous Algebras