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). In this installment we'll dig into alternative definitions of these (co-)algebras, first in Haskell, to build some intuition; then in terms of the (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, which the following signature

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

This 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 catamorphisms.

Notice that this definition requires the following language pragma

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

This is a definition in terms of the *mapping out* property. A data type is uniquely defined by outgoing functions to all other types. This is the consequence of the Yoneda lemma.

It's easy to define a catamorphism in terms of `Mu`, since `Mu` *is a* catamorphism

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

The definition of `Mu f` is workable, if a bit unwieldy. The challenge is to construct terms of type `Mu f`. For instance, let's 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

```
mkList :: forall a. [a] -> Mu (ListF a)
mkList as = Mu cata
  where cata :: forall x. (ListF a x -> x) -> x
        cata unf = go as
          where
            go [] = unf NilF
            go (n: ns) = unf (ConsF n (go ns))
```

Notice that we want the type in the type signature of the helper function `cata` to be the same type `a` defined in the type signature of `mkList`. For the compiler to make this connection, we have to use the pragma

```
{-# language ScopedTypeVariables #-}
```

You can now verify that

```
cataMu sumAlg (mkList [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 in terms of the terminal coalgebra `Nu f`

```
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. In other words, 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 have a gigantic product of functions, so we need a gigantic sum. A gigantic sum is known as the existential type. Symbolically, in pseudo Haskell, we would define `Nu` as

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

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
```

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 universally 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 `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 now 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 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 given by

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

This literally does nothing, just stores a single coalgebra and a seed. But that's how existential types work: they can be instantiated with a single value. 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` has the type

```
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)
```

Its terminal coalgebra is a stream of natural numbers. We'll express it using `Nu`

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

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

```haskell
fromFix :: forall f. 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`

```haskell
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 know from the previous installment that there is a mapping (an injection) from `Mu f` to `Nu f`. It turns out that `fromNuToMu` is its inverse, proving that these two fixed points are indeed 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 `Fix f` in common.

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

```haskell
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

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

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

```haskell
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

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

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

This is 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 $\pi_a$ 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 morphism from 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 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)
```

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

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

Indeed $\iota_a$ is a morphism from the hom-set

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

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

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

## 4. Hylomorphism

If the mapping from from the terminal coalgebra $\nu$ to the initial algebra $\mu$ 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.

By co-continuity of the hom-functor, this is isomorphic to

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

Using continuity we get

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

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 we get

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

in which you may recognize the hylomorphism

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