Algebra of Programming, Lecture 2

Jeremy Gibbons

Recap

We start by reviewing catamorphisms and the initial algebras on which they are based.

Recall these polynomial functors:

- $NX = 1 + X$.
- $L_A X = 1 + A \times X$.
  (This is a generalization of yesterday's L where L_A is the shape of lists whose elements are of type A.)
- $T_A X = A + X \times X$
  (This is the shape of a tree which has either an element or a pair of children.)

A special property of these functors is that they have least fixed points.

The least fixed point (also called the "initial algebra") of a functor F is $(X,\ in : FX \to X)$ such that for all $(A,\ f : FA \to A)$, there exists a unique $h : X \to A$ such that $h \circ in = f \circ Fh$.

We can visualize this definition with the following diagram:

$$\begin{tikzcd}
FX \arrow[r, "Fh"] \arrow[d, "in"'] & F(A) \arrow[d, "f"] \\
X \arrow[r, dashed, "h"'] & A
\end{tikzcd}$$

We write μF for X and $cata\ f$ for h.

For example, $List\ A = \mu(L_A)$. The justification for saying this is a fixed point is that there is an isomorphism between $F(\mu F)$ and μF. The leastness comes from the universal property, $h \circ in = f \circ Fh$.

In Haskell, we have

```
data L a b = Nil | Cons a b
data Mu f a = In (f a (Mu f a))
type List a = Mu L a
```

In the second line, for example, we are defining `Mu` at the type-level and `In` as a constructor at the term-level.

Bifunctors

One way to think of a functor is as a container of some kind of thing — for example, lists are containers of elements. A **bifunctor** is a container of two different kinds of things. Hence in Haskell we also have the following typeclasses:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d

-- Example for lists.
instance Bifunctor L where
```

```
    bimap f g Nil = Nil
    bimap f g (Cons x y) = Cons (f x) (g y)
```

The definition of *cata* can be rewritten in terms of bifunctors:

```
cata :: Bifunctor f => (f a b -> b) -> Mu f a -> b
cata phi (In x) = phi (bimap id (cata phi) x)
```

Said in words, we use `bimap` and a recursive call to `cata` to turn the children (of type `Mu f a`) into terms of type `b`, and then we apply `phi` to the result.

Observe that we can also construct the following instance:

```
instance Bifunctor f => Functor (Mu f) where
    fmap f (In x) = In (bimap f (fmap f) x)
```

The argument `f` to `bimap` maps the elements, and the argument `fmap f` recursively maps the child structures. We could alternatively define this in terms of a catamorphism — this is left as an exercise.

Anamorphisms

We can dualize the definition of the least fixed point by "turning the arrows around."

In particular, we have that the **greatest fixed point** (also called the "final coalgebra") is $(X,\ out : X \to FX)$ such that for all $(A,\ phi : A \to FA)$, there exists a unique $h : A \to X$ such that $out \circ h = Fh \circ phi$.

This is depicted in the following diagram:

$$\begin{tikzcd}
FX & F(A) \arrow[l, "Fh"'] \\
X \arrow[u, "out"] & A \arrow[u, "phi"'] \arrow[l, dashed, "h"]
\end{tikzcd}$$

Before, we wrote μF for X and *cata phi* for h. Here, we instead write νF for X and *ana phi* for h.

Whereas the least fixed point of L_A is finite lists of type A, the greatest fixed point also includes infinite lists.

Next, we can see what this looks like in Haskell, using the previously defined bifunctors.

```
data Nu f a = Out (f a (Nu f a))

out :: Nu f a -> f a (Nu f a)
out (Out x) = x

-- An idiomatic alternative:
data Nu f a = UnOut {out :: f a (Nu f a)}

-- cf. cata :: Bifunctor f => (f a b -> b) -> Mu f a -> b
ana :: Bifunctor f => (b -> f a b) -> b -> Nu f a
ana phi z = UnOut (bimap id (ana phi) (phi z))
```

Here is an example of an anamorphism for lists.

```
data L a b = Nil | Cons a b
type Colist a = Nu L a

-- range (0, 3) = [0, 1, 2]
-- range (0, 0) = []
-- range (3, 2) = [3, 4, 5, ...]
range :: (Int, Int) -> Colist Int
range = ana next where
    next :: (Int, Int) -> L Int (Int, Int)
```

```haskell
    next (m, n)
        | m == n = Nil
        | otherwise = Cons m (m + 1, n)
        -- This allows for infinite lists in the case where m > n.
```

Another example with a different tree shape is given below.

```haskell
-- This is an externally labeled tree instead of internally labeled as seen before.
data U a b = Empty | Fork b a b
type Cotree a = Nu U a

-- Build a binary search tree using anamorphisms.
build :: [a] -> Cotree a
build = ana next where
    next :: [a] -> U a [a]
    next [] = Empty
    next (x:xs) = fork ys x zs where
        ys = [y | y <- xs, y < x]
        zs = [z | z <- xs, z >= x]
```

Consider one final example. We redefine `Nu'` and `ana'` since `Maybe` is only parameterized on one type variable but the previous definitions using bifunctors expect two.

```haskell
data Nu' f = UnOut {out' :: f (Nu' f)}

type Conat = Nu' Maybe

-- Anamorphism for functors rather than bifunctors:
-- ana' :: Functor f => (b -> f b) -> b -> N u' f

-- When specialized,
ana' :: (b -> Maybe b) -> b -> Conat
ana' phi z = UnOut (fmap (ana' phi) (phi z))

unfoldConat :: (b -> Maybe b) -> b -> Conat
unfoldConat phi z = case phi z of
                        Nothing -> 0
                        Just z' -> 1 + unfoldConat z'
```