Algebra of Programming **OPLSS 2022**

Lecture 2: Catamorphisms and Anamorphisms

Lecturer: Jeremy Gibbons Scribes: Ke Du, Jessica Li, Slim Lim, Cody Rivera, Michelle Thalakottur

In this lecture, we revisit catamorphisms and least fixed points from the previous lecture, and implement them in Haskell. We dualize these constructions into anamorphisms and greatest fixed points, which represent *unfolding* into codata.

1 Catamorphisms

All polynomial functors in **Set** have least fixed points. Recall these shape functors from last time:

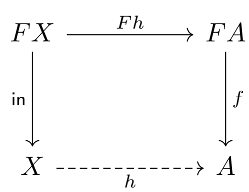
$$
\begin{array}{lrl}
N(X) = 1 + X & \mu N = \mathbb{N} & \text{(Maybe)} \\
L_A(X) = 1 + A \times X & \mu L = \text{finite lists of } A & \text{(List)} \\
T_A(X) = A + X \times X & \mu T_A = \text{finite trees of } A & \text{(Tree)}
\end{array}
$$

Remark 1. *The least fixed point μF of a functor F is also known as an **initial algebra** for F.*

Recall that the least fixed point of a functor F is a pair $(X, \mathsf{in} : FX \to X)$ such that for all other fixed points $(A, f : FA \to A)$, there exists a unique $h : X \to A$ satisfying the universal property

$$h \circ \mathsf{in} = f \circ Fh$$

that makes (A, f) factor through $(\mu F, \mathsf{in})$ in the commutative diagram below:



We write μF for X, and $\mathtt{cata}\, f$ for h. The function $\mathtt{cata}\, f$ is a **catamorphism** and represents a fold of the function f over the structure μF.

Remark 2. *The isomorphism $\mathsf{in} : F(\mu F) \to \mu F$ is what makes μF a fixed point. The universal property $h \circ in = f \circ Fh$ is what makes μF a least fixed point.*

Example 1 (Least fixed point of L_A)**.** *Consider the shape functor $L_A(X) = 1 + A \times X$, corresponding to the coproduct of unit and a pair (A, X) for fixed A.*

Then the least fixed point μL corresponds to finite lists of type A, which we will write as `List A`.

We can write the type $L_A(X)$ in Haskell as a coproduct parameterized by type variables `a` and `x`.

```
data L a x = Nil | Cons a x
          -- 1   +      A*X
```

How can we turn this generic shape functor into a monomorphized list type? If we were writing the type directly, we could say

```
data ListInt = Nil | Cons Int ListInt
```

but if we want to use our `L a x` functor (NB: no relation to lax functors), we have a problem: we cannot have a cyclical type synonym.

```haskell
type ListInt = L Int ListInt  -- Doesn't work.
          -- = L Int (L Int (L Int ...))
```

The solution is to use the least fixed point $\mu L_A(X)$. We define μ in Haskell as follows:

```haskell
data Mu f = In (f (Mu f))
```

On the left, we are defining a *type* `Mu` parameterized by a *type variable* `f`. On the right, our type `Mu` defines a single term-level *constructor* called `In`, which is a function of type `f (Mu f) -> Mu f`.

We can now use this `Mu f` to take the least fixed point of our functor `L a x`:

```haskell
data Mu f = In (f (Mu f))
data ListInt = In (L ListInt)
```

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

1.1 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:

```haskell
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:

```haskell
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:

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

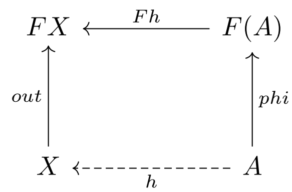
The argument `f` to `bimap` maps the elements, and the argument `fmap f` recursively maps the child structures. We could alternatively define this in terms of a catamorphism — this is left as an exercise.

1.2 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:



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))
```

1.3 Colists

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

1.4 Cotrees

Another example with a different tree shape is given below.

```
-- 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]
```

1.5 Conaturals

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.

```
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'
```