Algebra of Programming **OPLSS 2022**

Lecture 3: Extending Catamorphisms

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

In this lecture, we introduce additional recursive patterns, all of which generalize catamorphisms (and eventually anamorphisms) in some way.

1 Paramorphisms

We start by recalling the definition of `cata`:

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

All of the catamorphisms we've seen (such as `sum = cata add`) follow the same general structure: each invocation may use the current element of the data structure, as well as the result of the recursive call on the child data.

However, not all functions obey this structure. Here is an example of a function that is not a catamorphism:

```
-- Returns the suffix of the list, beginning with the first element that does NOT
-- satisfy the given predicate.
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p []     = []
dropWhile p (x:xs) = if p x then dropWhile p xs
                     else x:xs

dropWhile even [2, 4, 5, 6, 8] = [5, 6, 8]
```

Observe that `dropWhile` is not a catamorphism because it is not enough to know the *result* of recurring on the tail `xs` — we also need to return `xs` itself, if we enter the `else` branch. This property ("needing the whole list") indicates that `dropWhile` is a **paramorphism**, or a generalization of a catamorphism that adds more information to the accumulator type.

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

We can compare the type signatures of `para` and `cata`:

```
cata :: Bifunctor f => (f a b           -> b) -> Mu f a -> b
para :: Bifunctor f => (f a (b, Mu f a) -> b) -> Mu f a -> b
```

In `cata phi`, we only keep track of the result type `b`. But in `para phi`, we keep track of `(b, Mu f a)` — not only `b` but also the entire substructure `Mu f a` upon which the recursive call is performed. For `dropWhile`, this substructure is the tail of the list.

The implementation changes similarly. Instead of mapping `cata phi` directly over the data structure, a paramorphism uses `fork` to lift `para phi` into the newly-introduced tuple layer.

```
cata phi (In x) = phi (bimap id (cata phi)             x)
para phi (In x) = phi (bimap id ((para phi) `fork` id) x)
```

Paramorphisms can be said to provide a categorical representation of primitive recursion, while catamorphisms represent primitive iteration. Just as we can write a program using iteration or recursion, paramorphisms are not inherently more expressive than catamorphisms in any formal sense. However, paramorphisms can be more *efficient* than catamorphisms, by accumulating results that would otherwise need to be recomputed for each successive iteration.

In this sense, we say that `para` generalizes `cata`. More specifically, we can make the following observations:

Remark 1. *Every catamorphism is a paramorphism.*

Intuitively, this is because you "don't have to use the tail." A paramorphism keeps track of more information than a catamorphism – the remaining substructure, in addition to the result of the computation over the substructure. A catamorphism is just a paramorphism that ignores this additional information.

```
cata phi = para psi
    where phi = ... psi ...  -- Exercise: Define phi in terms of psi.
```

Remark 2. *Every paramorphism is a catamorphism with additional post-processing.*

As we have seen with `dropWhile`, not every paramorphism is a catamorphism. However, we can define a paramorphism in terms of a catamorphism by post-processing the result of the catamorphism, often using tuple projections to extract the definitive result.

```
para phi = post . cata phi
    where phi = ... psi ...
          post = fst  -- For example
```

In order to motivate the next concept, think of a paramorphism as "forking" `phi` with the identity function. The logical next question to ask is: what happens if we fork `phi` with something other than the identity?

1.1 Zygomorphisms

The Greek root *zygo-* apparently refers to two oxen yoked together. Accordingly, zygomorphisms describe paramorphisms in which the "main" function h depends on some auxiliary function h':

```
h (x:xs) = ... x ... h xs ... h' xs ...
```

Under this interpretation, paramorphisms are simply zygomorphisms with $h' = \mathrm{id}$.

The intuition for `zygo` is that we extend `para` in two ways:

1. First, we generalize the accumulator from `(b, Mu f a)` to `(b, c)` where the second field `c` can be anything (including `c = Mu f a` if we are nostalgic for paramorphisms)
2. Second, we pass `zygo` another function `psi` to transform this new field `c`, just as `phi` transforms `b` in `para`.

We can observe the progression of generality from `cata` to `para` to `zygo` in the type signatures for each function:

```
cata :: Bifunctor f => (f a b          -> b)                -> Mu f a -> b
para :: Bifunctor f => (f a (b, Mu f a) -> b)               -> Mu f a -> b
zygo :: Bifunctor f => (f a (b, c)      -> b) -> (f a c -> c) -> Mu f a -> b
```

To implement `zygo` efficiently, we use a helper `zygo'`:

```
zygo  :: Bifunctor f => (f a (b, c) -> b) -> (f a c -> c) -> Mu f a -> b
zygo                    phi                  psi           = fst . zygo' phi psi

zygo' :: Bifunctor f =>
```

```
        (f a (b, c)  -> b) -> (f a c -> c) -> Mu f a -> (b, c)
zygo'   phi                   psi             (In x)  =
    let x' = bimap id (zygo' phi psi) x  -- (zygo' phi psi) is what we need for phi
        in (phi x', psi (bimap id snd x'))
```

Zygomorphisms are useful whenever our paramorphism calls some auxiliary function. For example, suppose we are writing a function to test whether a binary tree is *perfect*, meaning all tips have the same heights. In this case, the function `height` is auxiliary:

```
data Tree a = Tip a | Bin (Tree a) (Tree a)

perfect :: Tree a -> Bool
perfect (Tip x) = True
perfect (Bin t u) = perfect t and perfect u and (height t == height u)
```

The above implementation of `perfect` recomputes `height t` and `height u` during each iteration. Instead, we can rewrite `perfect` as a zygomorphism, using a catamorphism `ph`:

```
ph :: Tree a -> (Bool, Int)
ph (Tip x) = (True, 1)
ph (Bin t u) = let (b, m) = ph t
                   (c, n) = ph u
               in (b and c and m == n, 1 + max m n)
```

Then we can rewrite `perfect' = fst . ph`, which is more efficient than our original implementation of `perfect`.

By extension of the reasoning in the previous section,

Remark 3. *Every paramorphism is a zygomorphism.*

Remark 4. *Every zygomorphism is a post-processed catamorphism.*

1.2 Mutumorphisms

Zygomorphisms have two oxen, but they are not evenly yoked. In the previous example, the primary function `perfect` depends on the auxiliary function `height`, but not vice-versa. Another way to see the uneven relationship is from the type signature, where the main function takes `(b, c)` but the auxiliary function only takes `c`:

```
zygo :: Bifunctor f
        => (f a (b, c) -> b)  -- (b, c) -> b
        -> (f a c      -> c)  --      c -> c
        -> Mu f a -> b
```

Mutumorphisms extend zygomorphisms to mutually-recursive functions. (The root *mutu-* (as in "mutual") describes morphisms that depends on each other.) We can see that `mutu` extends `zygo` by giving the second function the same information as the first:

```
cata :: Bifunctor f => (f a b          -> b)                    -> Mu f a -> b
para :: Bifunctor f => (f a (b, Mu f a) -> b)                   -> Mu f a -> b
zygo :: Bifunctor f => (f a (b, c)      -> b) -> (f a c      -> c) -> Mu f a -> b
mutu :: Bifunctor f => (f a (b, c)      -> b) -> (f a (b, c) -> c) -> Mu f a -> b
```

One example of a mutumorphism is the minimax algorithm, in which players take turns, and each turn depends on the opponent's previous move.

2 Histomorphisms

All of the previous recursive patterns were progressive generalizations of `cata` in the same direction. **Histomorphisms** also extend `cata`, but in an orthogonal direction. Histomorphisms keep track of the *history* of the computation so far, which is useful for dynamic programming (building a bottom-up memoization table of results) or any time we need to accumulate the result of `h xs` for all previous values of `xs`.

We define the type of `histo` using the *cofree comonad* on the functor `f a`:

```
cata  :: Bifunctor f => (f a b                -> b) -> Mu f a -> b
histo :: Bifunctor f => (f a (Cofree (f a) b) -> b) -> Mu f a -> b
```

The definitions of "cofree" and "comonad" are beyond the scope of this lecture. We can think of `Cofree (f a) b` as the type of a labeled data structure representing the "whole table" of results on predecessors. The type is defined as follows (note the similarity to `Mu`):

```
data Cofree f b = Node b (f (Cofree f b))
data Mu     f   = In     (f (Mu     f  ))
```

The added type parameter `b` represents the label `Cofree` adds to each level of the comonad, compared to `Mu`.

Here are a few more ways to think about the cofree comonad:

- Given a shape functor `f`, `Cofree f` is the easiest way to make a comonad of that shape.
- `Cofree f b` represents destruction on the label `b`.

We can define the following operations on `Cofree f b`, then use those to implement `histo`:

```
-- Note duality to monadic `return`.
root :: Cofree f b -> b
root (Node x t) = x

-- Note duality to monadic `join`.
subs :: Cofree f b -> f (Cofree f b)
subs (Node x t) = t

histo :: Bifunctor f =>
         (f a (Cofree (f a) b) -> b) -> Mu f a -> b
histo    phi                         = root . cata (\x -> Node (phi x) x)
```

The "labelling" action can be illustrated through a concrete example:

```
data L a b = Nil | Cons a b

-- What is Cofree (L a) c? Effectively, a list with a label at each node.
data LabeledList a c = LabeledNil c | LabeledCons c a (LabeledList a c)
```

For a list of length n, `Cofree (L a) c` contains $n+1$ labels of type c in total, because `Nil` is also labeled.

3 Duality

Here are all of the recursion patterns introduced today:

```
cata  :: Bifunctor f => (f a b                -> b)                    -> Mu f a -> b
para  :: Bifunctor f => (f a (b, Mu f a)      -> b)                    -> Mu f a -> b
```

```
zygo  :: Bifunctor f => (f a (b, c)           -> b) -> (f a c      -> c) -> Mu f a -> b
mutu  :: Bifunctor f => (f a (b, c)           -> b) -> (f a (b, c) -> c) -> Mu f a -> b
histo :: Bifunctor f => (f a (Cofree (f a) b) -> b)                      -> Mu f a -> b
```

What about anamorphisms, which we discussed in the previous lecture? Just as `ana` is the categorical dual of `cata`, we can define duals for each of the new patterns introduced today:

cata	**ana**
para	apo
zygo	co-zygo
muto	co-mutu
histo	futu

The first row, which relates paramorphisms to apomorphisms, is particularly interesting to consider. Notice the duality between the bifunctors under consideration:

cata	para	$f\ a\ (b \times \mathrm{Mu}\ f\ a)$
ana	apo	$f\ a\ (b + \mathrm{Nu}\ f\ a)$

Insertion sort is an example of this row.