Algebra of Programming **OPLSS 2022**

Lecture 4: Hylomorphisms and Metamorphisms

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

Recall that we are interested in two broad classes of programming patterns: *catamorphisms*, which generalize folds, and *anamorphisms*, which generalize unfolds. These patterns are categorical duals:

```
cata :: Bifunctor f => (f a b -> b) -> Mu f a -> b
ana  :: Bifunctor f => (b -> f a b) -> b -> Nu f a
```

Today, we consider how to compose catamorphisms and anamorphisms in both directions. While the basis is intuitive (many problems require us to construct and then destruct some intermediate data, or vice-versa), the details are surprisingly subtle. We begin with the first direction: unfolding, then folding.

1 Hylomorphisms

A **hylomorphism** describes an anamorphism followed by a catamorphism: first we unfold the input into an intermediate structure, then we fold that structure back into our final result.

Crucially, it is not always necessary to actually materialize the intermediate data structure. By *fusing* `ana` and `cata` together, hylomorphisms can operate over the *idea* of an intermediate data structure without actually performing allocations.

Hylo- means "dust" in Greek, because hylomorphisms can be thought of as dealing with the "spirit" of a data structure that is not physically present.

Example: Tree sort

A tree sorting algorithm takes a list of numbers, inserts all of the elements into a binary search tree, then traverses the BST to emit the results in order.

We can implement both steps in Haskell:

```
build :: Ord a => [a] -> Tree a
build [] = Empty
build (x:xs) = Fork (build ys) x (build zs)
    where (ys, zs) = filter (< x) xs, filter (>= x) xs

flatten :: Tree a -> [a]
flatten Empty = []
flatten (Fork t x u) = flatten t ++ [x] ++ flatten u
```

Note that `build` is an anamorphism, while `flatten` is a catamorphism. It is extremely tempting to define `sort` as their composition

```
sort = flatten . build
```

Unfortunately, this does not typecheck for a subtle reason not obvious in the above definition: as a catamorphism, `flatten` expects a `Tree` defined using `Mu`, but the anamorphism `build` produces a *cotree* based on `Nu`. Although these types coincide in Haskell, they are morally different: `Mu` computes the greatest fixed point of a functor, and `Nu` computes the least.

There are two ways to reconcile our types:

1. Translate our work to a new category that, like Haskell, does not distinguish between initial and final (co)algebras.
2. Modify `build` to show that given a finite list of type `[a]`, the resulting `Cotree` is also finite.

1.1 Unified `Fix`

The first option is to move to a different category that better matches our computational model. So far we have been working in **Set**, where objects are finite sets and morphisms are total functions between objects. Instead we will consider the category where objects are complete partial orders (cpos) and morphisms are continuous functions between cpos. Since our cpos all include $\bot$, they are *pointed* cpos. We will call this category $\mathbf{cpo}_\bot$.

In $\mathbf{cpo}_\bot$, the least and greatest fixed points coincide. This means that instead of having separate definitions for `Mu` and `Nu`, we can define a single type called `Fix`:

```
data Fix f a = In { out :: f a (Fix f a)}

-- Old definitions, for reference.
data Mu f a = In    { unIn :: f a (Mu f a) }
data Nu f a = UnOut { out  :: f a (Nu f a) }
```

We can redefine `cata` and `ana` by replacing `Mu` (resp. `Nu`) with `Fix`:

```
cata :: Bifunctor f => (f a b -> b) -> Fix f a -> b
ana  :: Bifunctor f => (b -> f a b) -> b -> Fix f a
```

Now that `cata` and `ana` operate over the same fixed point, we can define `hylo` as the composition of `cata` and `ana`:

```
hylo :: Bifunctor f =>
        (f b c -> c) -> (a -> f b a) -> a -> c
hylo    phi             psi           = cata phi . ana psi
```

What happens if we expand the definitions of `cata phi` and `ana psi`?

```
hylo :: Bifunctor f => (f b c -> c) -> (a -> f b a) -> a -> c
hylo phi psi = cata phi . ana psi
             = phi . bimap id (cata phi) . out . In . bimap id (ana phi) . psi
             --                            ^^^^^^^^ out and In cancel
             = phi . bimap id (cata phi) . bimap id (ana phi) . psi
             --     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             -- Functors respect composition, according to the functor laws, i.e.
             -- bimap f g . bimap h k = bimap (f . h) (g . k)
             = phi . bimap (id . id) (cata phi . ana psi) . psi
             = phi . bimap id (cata phi . ana psi) . psi
             --                ^^^^^^^^^^^^^^^^^^^ original definition of hylo
             = phi . bimap id (hylo phi psi) . psi
```

Note that our final definition of `hylo` is recursive, and can be understood as follows:

$$\mathrm{hylo}(\phi, \psi) = \underbrace{\phi}_{\text{extract result}} \circ \underbrace{\mathrm{bimap}(\mathrm{id}, \mathrm{hylo}(\phi, \psi))}_{\text{recursive part}} \circ \underbrace{\psi}_{\text{handle seed}}$$

1.1.1 Deforestation

Recall that tree sort works by constructing a binary search tree from the input list, then deconstructing the tree to produce the output list. The binary search tree is never observed outside of the function, so it represents potential cost savings if we can avoid allocating the tree at all.

By rewriting tree sort as a hylomorphism, we perform *deforestation* on the intermediate tree. This is most clearly visible at the cancellation of `In` and `out` in the derivation of `hylo` above.

While hylomorphic tree sort does not actually build a tree, it still recursively sorts on the structure of the tree. In this sense, the "spirit" of the tree remains, hence the name "hylo".

Remark 1. *Hylomorphisms are well-suited to divide-and-conquer algorithms.*

Examples of divide-and-conquer algorithms include:

- mergesort, which recursively splits the input list and sorts each sublist, then merges the pieces back together,
- any problem that can be expressed as a recurrence relation (e.g. naïve solutions to dynamic programming problems)

Remark 2. *A hylomorphism is the composition of a catamorphism and anamorphism, and therefore generalizes both.*

Recall the implementation of `hylo`:

```
hylo :: Bifunctor f => (f b c -> c) -> (a -> f b a) -> a -> c
hylo phi psi = phi . bimap id (hylo phi psi) . psi
```

We can understand `cata` as the special case where `psi` is `out`, which decomposes the seed. Likewise, `ana` is the special case where `psi` is `In`, which puts together the seeds.

1.1.2 Uniqueness of solutions

The main drawback of moving to $\mathbf{cpo}_\perp$ is that we must impose a strictness condition in order to guarantee that our fixed point admits a unique solution.

Remark 3 (Fokkinga and Meijer 1991)**.** *In a pointed cpo, if F is a polynomial functor, then there exists a fixed point μF and algebra (resp. coalgebra) constructors* in *(resp.* out*). In this setting, anamorphisms are guaranteed to be unique, but catamorphisms are only unique under strict evaluation.*

This idea is discussed further in Fokkinga and Meijer's paper "Program Calculation Properties of Continuous Algebras." In the diagram below, morphisms requiring strictness to preserve uniqueness have been labeled with a superscript *:

$$\begin{tikzcd}
F(C) \arrow[d, "\phi^*"'] & F(T) \arrow[l, "F(\text{cata }\phi)"'] \arrow[d, bend left, "\text{in}"] & F(A) \arrow[l, "F(\text{ana }\psi)"'] \\
C & T \arrow[l, "(\text{cata }\phi)^*"] \arrow[u, bend left, "\text{out}"] & A \arrow[l, "\text{ana }\psi"] \arrow[u, "\psi"']
\end{tikzcd}$$

1.2 Recursive coalgebras

Instead of moving our constructions to $\mathbf{cpo}_\perp$, we could stay in **Set** and define tree sort hylomorphically by proving that when $\text{xs} : [a]$ is a finite list, then `build xs` is guaranteed to produce a finite cotree. We build upon *recursive coalgebras* as defined by Uustalu, Vene, and Capretta.

The idea hinges on what we call the *hylo equation*:

$$h = \phi \circ \text{bimap id } h \circ \psi \tag{1}$$

In general, the hylo equation is not guaranteed to have a unique solution. There are a few special cases where we do get uniqueness, though:

- When $\psi = \text{in}^\circ$
- When $\phi = \text{out}^\circ$ (which is just `ana`)
- When ψ is a recursive coalgebra

Tree sorting by partition is an example of a recursive coalgebra, since we can show that our list grows structurally smaller each time we recur. More formally,

Remark 4. *ψ is a recursive coalgebra if the hylo equation has a unique solution for each ϕ.*

If ψ is a recursive coalgebra, then we are guaranteed to have a unique solution to the hylo equation which satisfies the universal property:

$$h = \text{hylo } \phi\ \psi \iff h \text{ is a solution to the hylo equation}$$

Remark 5. *In* **Set***, a finitary functor coalgebra on a carrier with a strictly decreasing measure is always a recursive coalgebra.*

$$\begin{tikzcd}
D \arrow[d, "\phi'"'] & C \arrow[l, "k"'] & A \arrow[l, "h"'] \\
F(BD) & F(BC) \arrow[l, "F \text{ id } k"] \arrow[u, "\phi"'] & F(BA) \arrow[l, "F \text{ id } h"] \arrow[u, "\psi"']
\end{tikzcd}$$

Now that we have discussed hylomorphisms, which compose a catamorphism after an anamorphism, we will consider the dual construction achieved by reversing the order of composition.

2 Metamorphisms

Metamorphisms, developed by Jeremy Gibbons, represent the composition of an anamorphism after a catamorphism. They are so named because they *metamorphize* representations of data.

Informally, metamorphisms work by reading a small part of the stream, processing that data, and writing an output stream. They are partially motivated by work done by Michael Jackson with Jackson Structured Programming, which is intended to work with programs that work on a large stream of input data/punch cards using very little memory.

Here are some other examples of tasks well-suited to metamorphisms:

- Reformatting lines of text in a paragraph
- Heapsort, which takes an unsorted list to a sorted list by constructing an intermediate heap
- Numeric base conversion on infinite sequences of digits

One caveat is that metamorphisms are still relatively new, and do not (yet?) satisfy the categorical semantics for arbitrary F-algebras. As a result, we will only consider operations on lists, as opposed to generic functors.

2.1 Folds and unfolds

Metamorphizing tasks can be viewed as first combining input data and then re-expanding parts of data. The combination of data is done by the `foldl` and `foldr` functions, and the expansion of data is done by the `unfoldr` function.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f r (x:xs) = f x (foldr f e xs)
```

One source of awkwardness is that we actually need the fold to be tail-recursive, so we use `foldl` instead, which accumulates brackets to the left of the expression.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e [] = e
foldl f e (x:xs) =  foldl f (f e x) xs
```

Now we can define `unfoldr`. It's not as obvious from the type signature that it is the dual of `foldr`, but this is a limitation of the Haskell standard library as opposed to a fundamental problem with the construction.

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f z = case f z of Nothing -> []
                          Just (x, z') -> x : unfoldr f z'
```

2.2 Streams

The idea is to perform a `foldl` followed by `unfoldr`. This composition is captured by the notion of a *stream*. We can define a specification stream as follows:

```
stream0 :: (b -> a -> b) -> b -> (b -> Maybe (c, b)) -> [a] -> [c]
stream0    f                y    g    = unfoldr g . foldl f y
--                                     ^^^^^^^^^   ^^^^^^^^^
--                                      consume     produce
```

The above definition of `stream0` has a problem: it will not produce on infinite input. To remedy this, we define a better version that behaves as follows:

1. If there is data in the buffer, produce that data.
2. Otherwise, try to consume data into the buffer.

We will model these concepts with three variables: `y` is a buffer that stores intermediate data processed by the function, `f` is a consumer function that inputs data into the buffer `y`, and `g` is a producer function that takes data from `y` if possible.

```
stream :: (b -> a -> b) -> b -> (b -> Maybe (c, b)) -> [a] -> [c]
stream f y g xs = case g y of
                    Just (z, y') -> z:(stream f y' g xs)  -- Produce z.
                    Nothing      -> case xs of
                                        x:xs' -> stream f (f y x) g xs'  -- Consume x.
                                       [] -> []
```

2.2.1 Streaming condition

Intuitively, in order for an algorithm to be a streaming algorithm, no set of input data can overwhelm the output stream. Furthermore, on an infinite data stream, some data should eventually be outputted (a stronger condition than what is shown here).

The **streaming condition** for finite inputs is represented in the following commutative diagram, relating consuming states to producing states. Effectively, it says that the order of consuming and producing should not affect the behavior of the program.

More specifically, if the stream can produce and chooses not to, the resulting state must:

- also be productive
- produce the same result
- end in the same state

as if the stream had chosen to produce.

$$\begin{tikzcd} y \arrow[r, "\text{produce } z"] \arrow[d, "\text{consume } x"'] & y' \arrow[d, "\text{consume } x"] \\ f\ y\ x \arrow[r, "\text{produce } z"'] & f\ y'\ x \end{tikzcd}$$

Figure 1: Streaming Condition

If the streaming condition holds, then the implementation `stream` is equivalent to `stream0` on finite inputs.

2.2.2 Flushable streams

Sometimes it is not safe for a stream to produce, as the production result might depend on input we have yet to consume. We can add a safety check to the production function `f` to confirm that no additional input can change the result of the next production. (Consumption carries no such risks, so we typically do not need to change `g`.)

However, in some circumstances the input stream is depleted before the entire buffer is flushed, so our safety check might hold up production forever. In order to use this buffered data in computation, we can *flush* the buffer to force the final production.

We call this flushable stream an `fstream`, and extend the definition of `stream` by passing an additional function `h :  b -> [c]`, the “flusher”. Every recursive call to `fstream` must additionally pass `h`.

```
fstream :: (b -> a -> b) -> b -> (b -> Maybe (c, b)) -> (b -> [c]) -> [a] -> [c]
fstream f y g h xs = case g y of
                        Just (z, y') -> z:(fstream f y' g h xs)
                        Nothing -> case xs of
                                      x:xs' -> fstream f (f y x) g h xs'
                                      []    -> h y  -- This is the only real difference from stream.
```

Note that the implementation of `fstream` only differs substantively from `stream` in the final case when the stream can neither produce nor consume data. In this case, `fstream` calls `h y` to flush the buffer with the provided flusher function.

Remark 6. *If the streaming condition holds, then for finite input, we can describe* `fstream` *using an apomorphism:*

```
fstream f y g h = apo ... g ... h ... . foldl f y
```

As an exercise, define the apomorphism in question.

2.3 Example: Converting from base 3 to base 7

As another application of metamorphisms, consider the process of converting a number from base 3 to base 7. The input number may have an arbitrarily many digits.

Consider a number $x \leq 1$ with a fractional expansion of the form

$$0.d_1 d_2 d_3 d_4 d_5 \ldots$$

where $x = \sum_{i=1} d_n b^{-i}$, and b is the base of this number. (Note that since the number is not base 10, we call it a fractional expansion rather than a decimal expansion.)

Our goal is to convert such a number from base 3 to base 7. Below are two functions to do this. We take in a base 3 fraction as a list of integers, convert the fraction to a rational number, and then convert that number to a base 7 fraction.

```
-- Input: a finite list of digits of a decimal in base 3
fromBase3 :: [Int] -> Rat
fromBase3 = foldr stepr 0
    where stepr d x = (d + x) / 3

fromBase3 [0, 1, 2] = 5/27 -- because 0/3^1 + 1/3^2 + 2/3^3 = 5/27

toBase7 :: Rat -> [Int]
toBase7 = unfoldr split
    where split x = let y = 7 * x in
        Just (floor y, y - floor y)
```

The problem is that our definition of `fromBase3` uses `foldr` rather than the tail-recursive `foldl`, which is necessary for the metamorphism to work. Unfortunately there is not a clear way to redefine `fromBase3` using both `foldl` and the same concise formulation. Instead, we can write it as follows:

```
fromBase3' = extract . foldl stepl (0 1)
    where stepl (u, v) d = (d + u * 3 , v / 3)
```

This definition uses (u, v) as a defunctionalized representation of `(v *) (u +)`. In other words, `extract (u, v) = v * u`.

Our second definition of `fromBase3'` using `foldl` is correct. Further means of modifying this radix conversion in order for it to meet the streaming condition (at least for finite input) are detailed in the exercises.

Note that this radix conversion algorithm will only work correctly on finite lists. It is an exercise to see which inputs do not produce data.

The homework assignment for this lecture can be found here.