Recursion Schemes

1 Embedding the Church Encoding

Last time, we saw every Haskell ADT has a church encoding, which provides a sort of fold over its structure. We saw a meta-language process for constructing the thing. First, we unpack recursion via an auxiliary type with an extra parameter. Then we traverse the structure of that type, producing an element of this shape

$$\forall \alpha, (A_1 \to ... \to \alpha) \to ... \to \alpha$$

And then we fill in any extra type parameters with α. This gives us a "final" representation of our datatype. However it's a bit unpleasant to have to do this process ourside the language. As we've seen before, it's convenient if we can do these processes inside the language via some cleverly constructed terms.

The short answer is yes-ish, we can't do this in F_ω directly. But we can do it in haskell. One of the last features of Haskell we have not yet discussed is, so called, ad-hoc polymorphism.

2 Ad Hoc Polymorphism

We have spent a long time talking about parametric polymorphism. The kind of polymorphism you get by using type parameters. It behaves uniformly across all types. It has an impredicative flavor, and gives theorems for free.

Ad-hoc polymorphism, on the other hand, is allowed to behave differently on different values. In Haskell, these are typeclasses. But typeclasses get statically inlined during compilation to F_ω, so we should expect that it is a conservative extension. That is, unlike the jump from STLC to F, the jump from F to "F + adhoc polymorphism" is just syntax sugar.

Nonetheless, it's extremely useful syntax sugar. There are three pieces of syntax sugar. First, an alias for a datatype

```
class Foo m_1 ... m_n where
  f_1 : t_1
  ...
  f_k : t_k
```

which desugars to

```
data FooAsData m_1 ... m_n := FooAsData {f_1 : t_1, ... f_k : t_k}
```

Then we have an abstraction application pair. The constructors look like

```
instance Foo A_1 ... A_n where
  f_1 = e_1
  ...
  f_k = e_k
```

Which desguars to

```
foo_instance_for_A_1_A_n = FooAsData e_1 ... e_k
```

And lastly, we have abstractions, where the type

```
(Foo a_1 ... a_n) => t
```

Becomes

```
FooAsData a_1 ... a_n -> t
```

Then there are rules for synthesizing `FooAsData` given some environment. As long as the rules are setup so that there is only ever zero or one instances of the right type, this translate loslessly into system F.

3 Algebras

I'm assuming you all are familiar enough with this to that I can skip the conventional examples of classes. The translation "adhoc polymorphism ⇒ direct argument passing of datas ⇒ System F" is typical compiler business. The important thing is that everything we are going to notate going forward is still in system F's language.

The class we're going to be interested in is call an "f-algebra".

```
class Functor b => FAlgebra (t :: *) (b :: * -> *) | t -> b  where
  project : t -> b t
```

note the `t -> b` says that there must be an injection from `t` to `b`. In other words, there is a unique `b` for every `t` with an algebra structure. Two interesting things

1. Every recursive ADT is an instance of this, where b is the auxiliary type as above.
2. Any type with this structure supports the church encoding process.

This is pretty neat, so now even our GADTs, existential types, or other crazy haskell types we want have a path forward for embedding into system F nicely. In fact, the church encoding is defined as

```
cata :: (FAlgebra t b) => (b a -> a) -> t -> a
cata f = fix (\k -> f . fmap k . project)

churchEncode :: (FAlgebra t b) => t -> forall a. (Base t a -> a) -> a
churchEncode = flip cata
```

But hold on, this is clearly an unbounded recursive function. Why should this thing even be in System F? In general, it is not. There are two ways to address this.

Firstly, we tried to be clever so that `b` would be a non-recursive ADT. Then when we do enough inlining of fix, we should actually eliminate it. That is, rewriting according to the equation

```
f (fix f) = fix f
```

That "optimized" term will be in system F.

But in general for this typeclass there's no reason to expect this good behavior. And there's no clear way to enforce it. So haskell just works System F, extended by an element

$$\vdash fix : \forall \alpha, (\alpha \to \alpha) \to \alpha$$

A few quirks, note that

```
\vdash fix id : \forall \alpha, \alpha
```

which is a slightly dangerous term. But a neccessary evil for making fix work. So that we can do arbirtary church encodings!

4 CoFAlgebras

But we said what was so special about the church encoding was that there is an isomorphism. So how do we go the other way?

```
class Functor b => CoFAlgebra (t :: *) (b :: * -> *) | t -> b  where
  embed :: b t -> t
```

As before, all our recursive ADTs have one of these. Additionally, we can write

```
ana :: (CoFAlgebra t b) => (b a -> a) -> t -> a
ana g = fix (\k -> embed . fmap k . g)

churchDecode :: (forall a. (Base t a -> a) -> a) -> t
churchDecode ch = ch embed
```

Note we don't actually need the `ana` to observe the isomorphism. However, having both `ana` and `cata` lets us do the somewhat magical. Whenever we have a pair of types which share a auxillary structure, we can write

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo f g = fix (\k -> f . fmap k . g)
```

which entirely bypasses the algebra structure! So if we are first building up a structure via `f b -> b` like an coalgebra, then breaking it down via `a -> f a` like an algebra, we can do both at once!

This way we never actually construct the intermediate structure, effectively fusing the operations. An interesting quirk! Of course by far the most common way to implement `f` and `g` in this setting is via an algebra `f = project . \case ...` and likewise for `g`. But once we fuse, it gets inlined and further optimized.

5 Final Thoughts

In the original Bananas, Lenses, and Barbed Wire paper, (named after the notations they introduce), they take these three operations and build a large family of related ones. They then provide a bunch of evaluation rules (most of which are theorems for free about map-reducing).

To be honest, I find that part a bit tedious. Rather, the important idea is that we can, in vast generality, describe this "structured recursion" for consuming trees and "structured corecursion" for building trees. Sure it saves on code, but it also comes with two varieties of guarantees. The first is the performance guarantee about compatibly chaining construction/destruction operations. So even if write separate modules for building/consuming trees, we know we can combine them efficiently. Furthermore, when we have additional mild assumptions on `b`, we get a guarantee that the computations are expressible in System F. So they terminate, and have the predictable structural recursion flavor.