MONOIDAL CATAMORPHISMS

BARTOSZ MILEWSKI

I have recently watched a talk by Gabriel Gonzalez about folds, which caught my attention because of my interest in both recursion schemes and optics. A `Fold` is an interesting abstraction. It encapsulates the idea of focusing on a *monoidal contents* of some data structure. Let me explain.

Suppose you have a data structure that contains, among other things, a bunch of values from some monoid. You might want to summarize the data by traversing the structure and accumulating the monoidal values in an accumulator. You may, for instance, concatenate strings, or add integers. Because we are dealing with a monoid, we could even parallelize the accumulation.

In practice, however, data structures are rarely filled with monoidal values. Usually monoidal values have to be extracted from a container. We need a way to convert the contents of the container to monoidal values, perform the accumulation, and convert the result to some output type. This could be done, for instance by applying `fmap` first, and then traversing the container to accumulate monoidal values. For performance reasons, we might prefer the two actions to be done in a single pass.

Here's a data structure that combines two functions, one converting `a` to some monoidal value `m` and the other converting the result to `b`. The traversal itself should not depend on what monoid is being used, so this is an existential type.

```
data Fold a b = forall m. Monoid m => Fold (a -> m) (m -> b)
```

The simplest container to traverse is a list and, indeed, we can use a `Fold` to fold a list. Here's the inefficient, but easy to understand implementation

```
fold :: Fold a b -> [a] -> b
fold (Fold s g) = g . mconcat . fmap s
```

See Gabriel's blog post to see a more efficient implementation.

A `Fold` is a functor

```
instance Functor (Fold a) where
  fmap f (Fold scatter gather) = Fold scatter (f . gather)
```

In fact it's a `Monoidal` functor

```
class Monoidal f where
  init :: f ()
  combine :: f a -> f b -> f (a, b)
```

```
instance Monoidal (Fold a) where
  -- Fold a ()
  init = Fold bang id
```

```
  -- Fold a b -> Fold a c -> Fold a (b, c)
  combine (Fold s g) (Fold s' g') = Fold (tuple s s') (bimap g g')
```

```
bang :: a -> ()
bang _ = ()

tuple :: (c -> a) -> (c -> b) -> (c -> (a, b))
tuple f g = \c -> (f c, g c)
```

A monoidal functor is automatically applicative—a property that can be used to aggregate `Fold`s.

A list is the simplest example of a recursive data structure. More general recursive data structures may be described as initial algebras or fixed points of endofunctors. List folds can be generalized to catamorphisms.

1. Algebras and catamorphisms

Here's a short recap of simple recursion schemes. An algebra for a functor `f` with the carrier `a` is defined as

```
type Algebra f a = f a -> a
```

A fixed point of a functor is the carrier of the initial algebra

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

A catamorphism generalizes a fold

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

1.1. Monoidal algebras.

We would like to use a `Fold` to fold an arbitrary recursive data structure. We are interested in data structures that store values of type `a` that can be converted to monoidal values. Such structures are generated by functors of two arguments (bifunctors). The first argument is the payload and the second is the placeholder for recursion. We define a monoidal algebra for such a functor by assuming that it has a monoidal payload and that the child nodes have already been evaluated to a monoidal value

```
type MAlgebra f = forall m. Monoid m => f m m -> m
```

A monoidal algebra is polymorphic in the monoid reflecting our assumption that the evaluation is only allowed to use monoidal unit and multiplication.

Given a monoidal algebra and a `Fold`, we can use a catamorphism to fold a recursive data structure given by a fixed point of a bifunctor

```
cat :: Bifunctor f => MAlgebra f -> Fold a b -> Fix (f a) -> b
cat malg (Fold s g) = g . cata alg
  where
    alg = malg . bimap s id
```

We can perform this operation because a bifunctor is automatically a functor in its second argument

```
instance Bifunctor f => Functor (f a) where
  fmap g = bimap id g
```

2. Example

Here's a simple example. We define a bifunctor that generates a binary tree with payload stored at the leaves

```
data TreeF a r = Leaf a | Node r r
```

It is indeed a bifunctor

```
instance Bifunctor TreeF where
  bimap f g (Leaf a) = Leaf (f a)
  bimap f g (Node r r') = Node (g r) (g r')
```

The recursive tree is geneerated as its fixed point

```
type Tree a = Fix (TreeF a)
```

We define two smart constructors to simplify the construction of trees

```
leaf :: a -> Tree a
leaf a = Fix (Leaf a)

node :: Tree a -> Tree a -> Tree a
node t t' = Fix (Node t t')
```

We can define a monoidal algebra for our functor. Notice that it only uses monoidal operations (we don't need the monoidal unit in this case, since values are stored in the leaves)

```
myAlg :: MAlgebra TreeF
myAlg (Leaf m) = m
myAlg (Node m m') = m <> m'
```

Separately, we define a `Fold` whose internal monoid is `Sum Int`. It converts `Double` values to this monoid, and converts the result to a `String`

```
myFold :: Fold Double String
myFold = Fold floor' show'
  where
    floor' :: Double -> Sum Int
    floor' = Sum . floor
    show' :: Sum Int -> String
    show' = show . getSum
```

Here's a small tree containing three `Double`s

```
myTree :: Tree Double
myTree = node (node (leaf 2.3) (leaf 10.3)) (leaf 1.1)
```

We can now monoidally fold this tree and display the resulting `String`

```
> cat myAlg myFold myTree
> "13"
```

The following pragmas were used in this program

```
{-# language ExistentialQuantification #-}
{-# language RankNTypes #-}
{-# language FlexibleInstances #-}
{-# language IncoherentInstances #-}
```

3. Relation to Optics

In categorical notation we would write it as a coend over the category of monoids **Mon** in some monoidal category **C**, with U a forgetful functor

$$\int^{m \in Mon} C(s, Um) \times C(Um, t)$$