Recursion Schemes for Higher Algebras

Bartosz Milewski

I. Introduction

Recursive schemes [1] are an example of successful application of concepts from category theory to programming. The idea is that recursive data structures can be defined as initial algebras of functors. This allows a separation of concerns: the functor describes the local shape of the data structure, and the fixed point combinator builds the recursion. Operations over data structures can be likewise separated into shallow, non-recursive computations described by algebras, and generic recursive procedures described by catamorphisms. In this way, data structures often replace control structures in driving computations.

Since functors also form a category, it's possible to define functors acting on functors. Such higher order functors show up in a number of free constructions, notably free monads, free applicatives, and cofree comonads. These free constructions have good composability properties and they provide means of separating the creation of effectful computations from their interpretation.

This paper's contribution is to systematize the construction of such interpreters. The idea is that free constructions arise as fixed points of higher order functors, and therefore can be approached with the same algebraic machinery as recursive data structures, only at a higher level. In particular, interpreters can be constructed as catamorphisms or anamorphisms of higher order algebras/coalgebras.

II. Initial Algebras and Catamorphisms

The canonical example of a data structure that can be described as an initial algebra of a functor is a list. In Haskell, a list can be defined recursively:

```
data List a = Nil | Cons a (List a)
```

There is an underlying non-recursive functor:

```
data ListF a x = NilF | ConsF a x
instance Functor (ListF a) where
  fmap f NilF = NilF
  fmap f (ConsF a x) = ConsF a (f x)
```

Once we have a functor, we can define its algebras. An *algebra* consist of a carrier c and a structure map (evaluator). An algebra can be defined for an arbitrary functor f:

```
type Algebra f c = f c -> c
```

Here's an example of a simple list algebra, with Int as its carrier:

```
sum :: Algebra (ListF Int) Int
sum NilF = 0
sum (ConsF a c) = a + c
```

Algebras for a given functor form a category. The initial object in this category (if it exists) is called the initial algebra. In Haskell, we call the carrier of the initial algebra Fix f. Its structure map is a function:

```
f (Fix f) -> Fix f
```

By Lambek's lemma, the structure map of the initial algebra is an isomorphism. In Haskell, this isomorphism is given by a pair of functions: the constructor In and the destructor out of the fixed point combinator:

```
newtype Fix f = In { out :: f (Fix f) }
```

When applied to the list functor, the fixed point gives rise to an alternative definition of a list:

```
type List a = Fix (ListF a)
```

The initiality of the algebra means that there is a unique algebra morphism from it to any other algebra. This morphism is called a *catamorphism* and, in Haskell, can be expressed as:

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

A list catamorphism is known as a fold. Since the list functor is a sum type, its algebra consists of a value—the result of applying the algebra to NilF—and a function of two variables that corresponds to the ConsF constructor. You may recognize those two as the arguments to foldr:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

The list functor is interesting because its fixed point is a free monoid. In category theory, monoids are special objects in monoidal categories—that is categories that define a product of two objects. In Haskell, a pair type plays the role of such a product, with the unit type as its unit (up to isomorphism).

As you can see, the list functor is the sum of a unit and a product. This formula can be generalized to an arbitrary monoidal category with a tensor product $\otimes$ and a unit 1:

$$L\,a\,x = 1 + a \otimes x$$

Its initial algebra is a free monoid [1].

III. Higher Algebras

In category theory, once you performed a construction in one category, it's easy to perform it in another category that shares similar properties. In Haskell, this might require reimplementing the construction.

[1] Strictly speaking, this is true if the tensor product distributes over countable coproducts.

We are interested in the category of endofunctors, where objects are endofunctors and morphisms are natural transformations. Natural transformations are represented in Haskell as polymorphic functions:

```
type f :~> g = forall a.dot f a -> g a
infixr 0 :~>
```

In the category of endofunctors we can define (higher order) functors, which map functors to functors and natural transformations to natural transformations:

```
class HFunctor hf where
 hfmap :: (g :~> h) -> (hf g :~> hf h)
 ffmap :: Functor g => (a->b) -> hf g a -> hf g b
```

The first function lifts a natural transformation; and the second function, ffmap, witnesses the fact that the result of a higher order functor is again a functor.

An algebra for a higher order functor hf consists of a functor f (the carrier object in the functor category) and a natural transformation (the structure map):

```
type HAlgebra hf f = hf f :~> f
```

As with regular functors, we can define an initial algebra using the fixed point combinator for higher order functors:

```
newtype FixH hf a = InH { outH :: hf (FixH hf) a}
```

Similarly, we can define a higher order catamorphism:

```
hcata::HFunctor h => HAlgebra h f -> FixH h :~> f
hcata halg = halg . hfmap (hcata halg) . outH
```

The question is, are there any interesting examples of higher order functors and algebras that could be used to solve real-life programming problems?

IV. Free Monad

We've seen the usefulness of lists, or free monoids, for structuring computations. Let's see if we can generalize this concept to higher order functors.

The definition of a list relies on the cartesian structure of the underlying category. It turns out that there are multiple cartesian structures of interest that can be defined in the category of functors. The simplest one defines a product of two endofunctors as their composition. Any two endofunctors can be composed. The unit of functor composition is the identity functor.

If you picture endofunctors as containers, you can easily imagine a tree of lists, or a list of Maybes.

A monoid based on this particular monoidal structure in the endofunctor category is a monad. It's an endofunctor m equipped with two natural transformations representing unit and multiplication:

```
class Monad m where
  eta :: Identity    :~> m
  mu  :: Compose m m :~> m
```

In Haskell, the components of these natural transformations are known as return and join.

A straightforward generalization of the list functor to the functor category can be written as:

$$L\, f\, g = 1 + f \circ g$$

or, in Haskell,

```
type FunctorList f g = Identity :+: Compose f g
```

where we used the operator :+: to define the coproduct of two functors:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
infixr 7 :+:
```

Using more conventional notation, FunctorList can be written as:

```
data MonadF f g a =
    DoneM a
  | MoreM (f (g a))
```

We'll use it to generate a free monoid in the category of endofunctors. First of all, let's show that it's indeed a higher order functor in the second argument g:

```
instance Functor f => HFunctor (MonadF f) where
  hfmap _   (DoneM a)  = DoneM a
  hfmap nat (MoreM fg) = MoreM $ fmap nat fg
  ffmap h (DoneM a)    = DoneM (h a)
  ffmap h (MoreM fg)   = MoreM $ fmap (fmap h) fg
```

In category theory, because of size issues, this functor doesn't always have a fixed point. For most common choices of f (e.g., for algebraic data types), the initial higher order algebra for this functor exists, and it generates a free monad. In Haskell, this free monad can be defined as:

```
type FreeMonad f = FixH (MonadF f)
```

We can show that FreeMonad is indeed a monad by implementing return and bind:

```
instance Functor f => Monad (FreeMonad f) where
  return = InH . DoneM
  (InH (DoneM a))     >>= k = k a
  (InH (MoreM ffra)) >>= k =
        InH (MoreM (fmap (>>= k) ffra))
```

Free monads have many applications in programming. They can be used to write generic monadic code, which can then be interpreted in different monads. A very useful property of free monads is that they can be composed using coproducts. This follows from the theorem in category theory, which states that left adjoints preserve coproducts (or, more generally, colimits). Free constructions are, by definition, left adjoints to forgetful functors. This property of free monads was explored by Swierstra [4] in his solution to the expression problem. I will use an example based on his paper to show how to construct monadic interpreters using higher order catamorphisms.

A. Free Monad Example

A stack-based calculator can be implemented directly using the state monad. Since this is a very simple example, it will be instructive to re-implement it using the free monad approach.

We start by defining a functor, in which the free parameter k represents the continuation:

```
data StackF k  = Push Int k
               | Top (Int -> k)
               | Pop k
               | Add k
               deriving Functor
```

We use this functor to build a free monad:

```
type FreeStack = FreeMonad StackF
```

You may think of the free monad as a tree with nodes that are defined by the functor StackF. The unary constructors, like Add or Pop, create linear list-like branches; but the Top constructor branches out with one child per integer.

The level of indirection we get by separating recursion from the functor makes constructing free monad trees syntactically challenging, so it makes sense to define a helper function:

```
liftF :: (Functor f) => f r -> FreeMonad f r
liftF fr = InH $ MoreM $ fmap (InH . DoneM) fr
```

With this function, we can define smart constructors that build leaves of the free monad tree:

```
push :: Int -> FreeStack ()
push n = liftF (Push n ())

pop :: FreeStack ()
pop = liftF (Pop ())

top :: FreeStack Int
top = liftF (Top id)

add :: FreeStack ()
add = liftF (Add ())
```

All these preparations finally pay off when we are able to create small programs using do notation:

```
calc :: FreeStack Int
calc = do
  push 3
  push 4
  add
  x <- top
  pop
  return x
```

Of course, this program does nothing but build a tree. We need a separate interpreter to do the calculation. We'll interpret our program in the state monad, with state implemented as a stack (list) of integers:

```
type MemState = State [Int]
```

The trick is to define a higher order algebra for the functor that generates the free monad and then use a catamorphism to apply it to the program. Notice that implementing the algebra is a relatively simple procedure because we don't have to deal with recursion. All we need is to case-analyze the shallow constructors for the free monad functor MonadF, and then case-analyze the shallow constructors for the functor StackF.

```
runAlg :: HAlgebra (MonadF StackF) MemState
runAlg (DoneM a)  = return a
runAlg (MoreM ex) =
  case ex of
    Top  ik  -> get >>= ik  . head
    Pop  k   -> get >>= put . tail   >> k
    Push n k -> get >>= put . (n : ) >> k
    Add  k   -> do (a: b: s) <- get
                   put (a + b : s)
                   k
```

The catamorphism converts the program calc into a state monad action, which can be run over an empty initial stack:

```
runState (hcata runAlg calc) []
```

The real bonus is the freedom to define other interpreters by simply switching the algebras. Here's an algebra whose carrier is the Const functor:

```
showAlg :: HAlgebra (MonadF StackF) (Const String)

showAlg (DoneM a) = Const "Done!"
showAlg (MoreM ex) = Const $
  case ex of
    Push n k ->
      "Push " ++ show n ++ ", " ++ getConst k
    Top ik ->
      "Top, " ++ getConst (ik 42)
    Pop k ->
      "Pop, " ++ getConst k
    Add k ->
      "Add, " ++ getConst k
```

Runing the catamorphism over this algebra will produce a listing of our program:

```
getConst $ hcata showAlg calc
```

V. Free Applicative

There is another monoidal structure that exists in the category of functors. In general, this structure will work for functors from an arbitrary monoidal category C to Set. Here, we'll restrict ourselves to endofunctors on Set. The product of two functors is given by Day convolution, which can be implemented in Haskell using an existential type:

```
data Day f g c where
  Day :: f a -> g b -> ((a, b) -> c) -> Day f g c
```

The intuition is that a Day convolution contains a container of some as, and another container of some bs, together with a function that can convert any pair (a, b) to c.

Day convolution is a higher order functor:

```
instance HFunctor (Day f) where
  hfmap nat (Day fx gy xyt) = Day fx (nat gy) xyt
  ffmap h   (Day fx gy xyt) = Day fx gy (h . xyt)
```

In fact, because Day convolution is symmetric up to isomorphism, it is automatically functorial in both arguments.

To complete the monoidal structure, we also need a functor that could serve as a unit with respect to Day convolution. In general, this would be the the hom-functor from the monoidal unit:

$$C(1, -)$$

In our case, since 1 is the singleton set, this functor reduces to the identity functor.

We can now define monoids in the category of functors with the monoidal structure given by Day convolution. These monoids are equivalent to lax monoidal functors which, in Haskell, form the class:

```
class Functor f => Monoidal f where
  unit  :: f ()
  (>*<) :: f x -> f y -> f (x, y)
```

Lax monoidal functors are equivalent to applicative functors [6], as seen in this implementation of pure and <*>:

```
  pure  :: a -> f a
  pure a = fmap (const a) unit
  (<*>) :: f (a -> b) -> f a -> f b
  fs <*> as = fmap (uncurry ($)) (fs >*< as)
```

We can now use the same general formula, but with Day convolution as the product:

$$L\, f\, g = 1 + f \star g$$

to generate a free monoidal (applicative) functor:

```
data FreeF f g t =
      DoneF t
    | MoreF (Day f g t)
```

This is indeed a higher order functor:

```
instance HFunctor (FreeF f) where
    hfmap _ (DoneF x)      = DoneF x
    hfmap nat (MoreF day) = MoreF (hfmap nat day)
    ffmap f (DoneF x)      = DoneF (f x)
    ffmap f (MoreF day)    = MoreF (ffmap f day)
```

and it generates a free applicative functor as its initial algebra:

```
type FreeA f = FixH (FreeF f)
```

A. Free Applicative Example

The following example is taken from the paper by Capriotti and Kaposi [5]. It's an option parser for a command line tool, whose result is a user record of the following form:

```
data User = User
  { username :: String
  , fullname :: String
  , uid      :: Int
  } deriving Show
```

A parser for an individual option is described by a functor that contains the name of the option, an optional default value for it, and a reader from string:

```
data Option a = Option
  { optName    :: String
  , optDefault :: Maybe a
  , optReader  :: String -> Maybe a
  } deriving Functor
```

Since we don't want to commit to a particular parser, we'll create a parsing action using a free applicative functor:

```
userP :: FreeA Option User
userP  = pure User
  <*> one (Option "username" (Just "John")  Just)
  <*> one (Option "fullname" (Just "Doe")   Just)
  <*> one (Option "uid"      (Just 0)    readInt)
```

where readInt is a reader of integers:

```
readInt :: String -> Maybe Int
readInt s = readMaybe s
```

and we used the following smart constructors:

```
one :: f a -> FreeA f a
one fa = InH $ MoreF $ Day fa (done ()) fst

done :: a -> FreeA f a
done a = InH $ DoneF a
```

We are now free to define different algebras to evaluate the free applicative expressions. Here's one that collects all the defaults:

```
alg :: HAlgebra (FreeF Option) Maybe
alg (DoneF a) = Just a
alg (MoreF (Day oa mb f)) =
  fmap f (optDefault oa >*< mb)
```

I used the monoidal instance for Maybe:

```
instance Monoidal Maybe where
  unit = Just ()
  Just x >*< Just y = Just (x, y)
  _ >*< _ = Nothing
```

This algebra can be run over our little program using a catamorphism:

```
parserDef :: FreeA Option a -> Maybe a
parserDef = hcata alg
```

And here's an algebra that collects the names of all the options:

```
alg2 :: HAlgebra (FreeF Option) (Const String)
alg2 (DoneF a) = Const ".dot"
alg2 (MoreF (Day oa bs f)) =
  fmap f (Const (optName oa) >*< bs)
```

Again, this uses a monoidal instance for Const:

```
instance Monoid m => Monoidal (Const m) where
  unit = Const mempty
  Const a >*< Const b = Const (a <> b)
```

We can also define the Monoidal instance for IO:

```
instance Monoidal IO where
  unit = return ()
  ax >*< ay = do a <- ax
                 b <- ay
                 return (a, b)
```

This allows us to interpret the parser in the IO monad:

```
alg3 :: HAlgebra (FreeF Option) IO
alg3 (DoneF a) = return a
alg3 (MoreF (Day oa bs f)) = do
  putStrLn $ optName oa
  s <- getLine
  let ma = optReader oa s
      a = fromMaybe (fromJust (optDefault oa)) ma
  fmap f $ return a >*< bs
```

VI. Cofree Comonad

Every construction in category theory has its dual—the result of reversing all the arrows. The dual of a product is

a coproduct, the dual of an algebra is a coalgebra, and the dual of a monad is a comonad.

Let's start by defining a higher order coalgebra consisting of a carrier f, which is a functor, and a natural transformation:

```
type HCoalgebra hf f = f :~> hf f
```

An initial algebra is dualized to a terminal coalgebra. In Haskell, both are the results of applying the same fixed point combinator, reflecting the fact that the Lambek's lemma is self-dual. The dual to a catamorphism is an anamorphism. Here is its higher order version:

```
hana :: HFunctor hf
     => HCoalgebra hf f -> (f :~> FixH hf)
hana hcoa = InH . hfmap (hana hcoa) . hcoa
```

The formula we used to generate free monoids:

$$1 + a \otimes x$$

dualizes to:

$$1 \times a \otimes x$$

and can be used to generate cofree comonoids [2].

A cofree functor is the right adjoint to the forgetful functor. Just like the left adjoint preserved coproducts, the right adjoint preserves products. One can therefore easily combine comonads using products (if the need arises to solve the coexpression problem).

Just like the monad is a monoid in the category of endofunctors, a comonad is a comonoid in the same category. The functor that generates a cofree comonad has the form:

```
type ComonadF f g = Identity :*: Compose f g
```

where the product of functors is defined as:

```
data (f :*: g) e = Both (f e) (g e)
infixr 6 :*:
```

Here's the more familiar form of this functor:

```
data ComonadF f g e = e :< f (g e)
```

It is indeed a higher order functor, as witnessed by this instance:

```
instance Functor f => HFunctor (ComonadF f) where
  hfmap nat (e :< fge) = e :< fmap nat fge
  ffmap h (e :< fge) = h e :< fmap (fmap h) fge
```

A cofree comonad is the terminal coalgebra for this functor and can be written as a fixed point:

```
type Cofree f = FixH (ComonadF f)
```

Indeed, for any functor f, Cofree f is a comonad:

```
instance Functor f => Comonad (Cofree f) where
  extract (InH (e :< fge)) = e
  duplicate fr@(InH (e :< fge)) =
                InH (fr :< fmap duplicate fge)
```

[2]Not in every category. The actual requirements are quite involved, but they do work for this particular example.

A. Cofree Comonad Example

The canonical example of a cofree comonad is an infinite stream:

```
type Stream = Cofree Identity
```

We can use this stream to sample a function. We'll encapsulate this function inside the following functor (in fact, itself a comonad):

```
data Store a x = Store a (a -> x)
    deriving Functor
```

We can use a higher order coalgebra to unpack the Store into a stream:

```
streamCoa ::
    HCoalgebra (ComonadF Identity)(Store Int)
streamCoa (Store n f) =
    f n :< (Identity $ Store (n + 1) f)
```

The actual unpacking is a higher order anamorphism:

```
stream :: Store Int a -> Stream a
stream = hana streamCoa
```

We can use it, for instance, to generate a list of squares of natural numbers:

```
stream (Store 0 (^2))
```

Since, in Haskell, the same fixed point defines a terminal coalgebra as well as an initial algebra, we are free to construct algebras and catamorphisms for streams. Here's an algebra that converts a stream to an infinite list:

```
listAlg :: HAlgebra (ComonadF Identity) []
listAlg(a :< Identity as) = a : as

toList :: Stream a -> [a]
toList = hcata listAlg
```

VII. Future Directions

In this paper I concentrated on one type of higher order functor:

$$1 + a \otimes x$$

and its dual. This would be equivalent to studying folds for lists and unfolds for streams. But the structure of the functor category is richer than that. Just like basic data types can be combined into algebraic data types, so can functors. Moreover, besides the usual sums and products, the functor category admits at least two additional monoidal structures generated by functor composition and Day convolution.

Another potentially fruitful area of exploration is the profunctor category, which is also equipped with two monoidal structures, one defined by profunctor composition, and another with Day convolution. A free monoid with respect to profunctor composition is the basis of Haskell Arrow library [2]. Profunctors also play an important role in the Haskell lens library [3].

REFERENCES

[1] Meijer, Erik and Fokkinga, Maarten and Paterson, Ross "Functional programming with bananas, lenses, envelopes and barbed wire" Functional Programming Languages and Computer Architecture Lecture Notes in Computer Science, 1991, p. 124-144.

[2] Rivas, Exequiel and Jaskelioff, Mauro "Notions of computation as monoids" Journal of Functional Programming, 2017.

[3] Kmett, Edward A "lens: Lenses, Folds and Traversals" url=https://hackage.haskell.org/package/lens, 2012

[4] Swierstra, Wouter "Data types à la carte" Journal of Functional Programming, vol 18, No 04, 2008.

[5] Capriotti, Paolo and Kaposi, Ambrus, "Free Applicative Functors", Electronic Proceedings in Theoretical Computer Science, May 2014, p. 2-30.

[6] McBride, Conor and Paterson, Ross "Applicative programming with effects" Journal of Functional Programming, Vol 18, No 01, 2007.