

Different recursion schemas and their properties

I.Zhirkov

2015

Morphisms

- Catamorphisms (destruction)
- Anamorphisms (construction)
- Hylomorphisms (combination of two)
- Paramorphisms (saves more information than hylomorphisms)

Catamorphism

- “generalized fold”

$b :: B$

$h :: \text{List } A \rightarrow B$

$h \text{ Nil} = b$

$h (\text{Cons } a \text{ as}) = f a (h \text{ as})$

- Notation: $h = (b, f)$
- Arises from algebra $(f a \rightarrow a)$.

Anamorphism

- “generalized unfold”

$p :: B \rightarrow \text{Bool}$

$g :: B \rightarrow (A, B)$

$h :: B \rightarrow \text{List } A$

$h\ b = \text{Nil},$

$h\ b = \text{Consa } (h\ b'),$

$p\ b$

otherwise

where $(a, b') = g\ b$

- Notation $h = \llbracket g, p \rrbracket$
- Arises from coalgebra $(a \rightarrow f\ a)$.

Hylomorphism

- “call-tree looks like data structure”

$$c :: C$$
$$f :: B \rightarrow C \rightarrow C$$
$$g :: A \rightarrow (B, A)$$
$$p : A \rightarrow \text{Bool}$$
$$h :: \rightarrow \text{List } A$$
$$h\ a = c,$$
$$h\ b = f\ b(h\ a'),$$
$$p\ a$$
$$\text{otherwise}$$

where $(b, a') = g\ a$

- Notation $h = \llbracket (c, f), (g, p) \rrbracket$

Hylomorphism-2

- Is a composition of anamorphism and catamorphism
 $\llbracket (c, f), (g, p) \rrbracket = \llbracket c, f \rrbracket \circ \llbracket g, p \rrbracket$
- *Look at the whiteboard for a fancy call-tree image*
- Example: factorial is $\llbracket (1, \times), (g, p) \rrbracket$, where:

$$p\ n = n == 0$$

$$g\ n = (n, n - 1)$$

Paramorphism

- Hylomorphism for *fac* is not inductively defined on *nat*.
- A nat paramorphism example:

$$h\ 0 = b$$

$$h\ (\text{Succ } n) = f\ n\ (h\ n)$$

- A list example:

$$h\ \text{Nil} = b$$

$$h\ (\text{Cons } x\ xs) = f\ x\ xs\ (h\ xs)$$

- Notation: $\langle [b, f] \rangle$

Category

$$C = (obj, hom, \circ)$$

Objects and morphisms between objects with their compositions.

Laws:

- Identity morphisms for each object
- Composition is associative
- Path equality

“Point” is synonymous to “Object”

Categories: *Set*

- $ob(Set)$ – all sets
- $hom(E, F)$ – functions between sets E and F
- \circ – composition

Categories: *Set*

- $ob(Set)$ – all sets
- $hom(E, F)$ – functions between sets E and F
- \circ – composition
- Note: $ob(Set)$ is not a set itself (is a class).

Categories

- *Mon*: (monoids, morphisms, composition)
- *Grp*: (groups, morphisms, composition)
- *Hask*: (haskell types, functions, $(.)$)
- ...

Functor

- Functor is a morphism between categories (preserves structures)

Functor in FP

- an (endo-)functor is an operation from types to types (from Hask to Hask)
- preserves identity and composition.
- functions can be 'mapped over' functors
- Basic ones: identity, product, sum (tagged), arrow ...

A usual recursive datatype

```
data Expr = Const Int  
          | Add Expr Expr  
          | Mul Expr Expr
```

is same as:

```
newtype Fix f = Fix (f (Fix f))
```

```
data ExprF a = Const Int | Add a a | Mul a a
```

```
type Expr = Fix ExprF
```

Test

```
testExpr = Fx $ (Fx $ (Fx $ Const 2) 'Add' (Fx $ Const 3))  
          'Mul' (Fx $ Const 4)
```

It is a functor!

```
instance Functor ExprF where
    fmap eval (Const i) = Const i
    fmap eval (Add x y) = Add (eval x) (eval y)
    fmap eval (Mul x y) = Mul (eval x) (eval y)
```

It is possible to construct an algebra on top of any functor.

```
type Algebra f a = f a → a
```

We expect to be able to 'evaluate' children of `Expr`. Example:

```
alg':: ExprF Int → Int
alg' (Const i) = i
alg' (Add x y) = x + y
alg' (Mul x y) = x * y
```


What is an algebra?

(C, F, A, m)

- Category C (*Hask*, points are types of Haskell)
- Endofunctor F (maps *Hask* points into other *Hask* points)
- Point A (**carrier**) of category C (some type).
- Function m mapping $F A \rightarrow A$

Hence our definition:

```
type Algebra f a = f a → a
```

C is implied (*Hask*), F and A are type-level, the rest is m , the function itself (of type `Algebra f a`).

Look at the whiteboard for a fancy image!

Initial algebra

- There are infinitely many algebras based on same functor
- One is particular **Initial algebra**,
- Does not 'forget' anything, preserves all information about input.
- \exists a homomorphism from initial algebra to any other algebra.
- Assume: carrier is `Fix f`, algebra is `Fx` (its ctor).

```
type Algebra f a = f a → a
```

```
type ExprInitAlgebra = Algebra ExprF (Fix ExprF)  
ex_init_alg :: ExprF (Fix ExprF) → Fix ExprF  
ex_init_alg = Fx
```

Functor f is fixed. Let a be the carrier object for a new algebra.

	Carrier	Evaluator
Initial algebra	Fix f	Fx
Some algebra	a	alg

Constructing any algebra-1

We want to get a homomorphism from initial algebra to some other algebra. Carrier mapper:

$g :: \text{Fix } f \rightarrow a$

Remember:

$\text{newtype Fix } f = \text{Fx } (f (\text{Fix } f))$

Thanks to the fact that f is a functor, fmap is at our disposal to map :

$\text{fmap } g :: f (\text{Fix } f) \rightarrow f a$

$$\begin{array}{ccc} f(\text{Fix } f) & \xrightarrow{\text{fmap } g} & f a \\ \downarrow Fx & & \downarrow alg \\ \text{Fix } f & \xrightarrow{g} & a \end{array}$$

Constructing any algebra-2

`Fix` is lossless, thus invertible.

`unFix :: Fix f → f (Fix f)`

`unFix (Fix x) = x`

$$\begin{array}{ccc} f(\text{Fix } f) & \xrightarrow{\text{fmap } g} & f \ a \\ \downarrow \text{Fix} & & \downarrow \text{alg} \\ \text{Fix } f & \xrightarrow{g} & a \end{array} \quad \text{becomes} \quad \begin{array}{ccc} f(\text{Fix } f) & \xrightarrow{\text{fmap } g} & f \ a \\ \uparrow \text{unFix} & & \downarrow \text{alg} \\ \text{Fix } f & \xrightarrow{g} & a \end{array}$$

`g` can be defined with `unfix`, `fmap` and evaluator:

`g = alg. (fmap g) . unFix`

Meet catamorphism

```
g = alg. (fmap g) . unFix
```

```
cata :: Functor f => (f a -> a) -> (Fix f -> a)
```

```
cata alg = alg . fmap (cata alg) . unFix
```

Quick check:

```
alg :: ExprF String -> String
```

```
alg (Const i) = [ chr (ord 'a' + i)]
```

```
alg (Add x y) = x ++ y
```

```
alg (Mul x y) = concat [[a,b] | a <- x, b <- y]
```

```
*Main> :t cata alg
```

```
cata alg :: Fix ExprF -> String
```

Coalgebra

Algebra: $f\ a \rightarrow a$

Coalgebra: $a \rightarrow f\ a$

$$\begin{array}{ccc} f(\text{Fix } f) & \xrightarrow{\text{fmap } g} & f\ a \\ \downarrow Fx & & \downarrow alg \\ \text{Fix } f & \xrightarrow{g} & a \end{array} \quad \text{becomes} \quad \begin{array}{ccc} f(\text{Fix } f) & \xleftarrow{\text{fmap } g} & f\ a \\ \uparrow unFix & & \uparrow coalg \\ \text{Fix } f & \xleftarrow{g} & a \end{array}$$

The same reasoning about initial coalgebra applies.

Coalgebra-2

μ combines `Fx` and `unFix`.

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

```
type CoAlgebra f a = a → f a
```

```
type Algebra f a = f a → a
```

```
catam :: Functor f ⇒ Algebra f a → (Mu f → a)  
catam alg = alg . fmap (catam alg) . out
```

```
anam :: Functor f ⇒ CoAlgebra f a → (a → Mu f)  
anam coalg = In. fmap (anam coalg) . coalg
```


Coalgebra-3

As *out* is an initial algebra, *in* is a terminal coalgebra (there exists a unique homomorphism from any coalgebra to *in*).

$$\begin{array}{ccc} f(\mu f) & \xleftarrow{\text{fmap } g} & f a \\ \downarrow in & & \uparrow coalg \\ \mu f & \xleftarrow{g} & a \end{array}$$

Notation summary

Expanding notation to arbitrary types:

- Catamorphism:

$$(\llbracket \phi \rrbracket)_F = \mu(\phi \xleftarrow{F} out)$$

- Anamorphism:

$$[\![\psi]\!]_F = \mu(in \xleftarrow{F} \psi)$$

- Anamorphism:

$$[\![\psi]\!]_F = \mu(in \xleftarrow{F} \psi)$$

“Actually, even in Haskell recursion is not completely first class because the compiler does a terrible job of optimizing recursive code. This is why F-algebras and F-coalgebras are pervasive in high-performance Haskell libraries like vector, because they transform recursive code to non-recursive code, and the compiler does an amazing job of optimizing non-recursive code.”

Further reading:

- `Control.Functor.Algebra`