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"
  $b :: B$

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

$$h\ Nil = b$$
$$h\ (Cons\ a\ as) = f\ a\ (h\ as)$$

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

Anamorphism

- "generalized unfold"
  $p: \ B \to Bool$
  $g : B \to (A, B)$

$$
\begin{aligned}
& h :: B \to List\ A \\
& h\ b = Nil, && p\ b \\
& h\ b = Consa\ (h\ b'), && otherwise
\end{aligned}
$$

  where $(a, b') = g\ b$
- Notation $h = [\![(g, p)]\!]$
- Arises from coalgebra $(a \to f\ a)$.

Hylomorphism

- “call-tree looks like data structure”
  $c :: \ C$
  $f :: B \to C \to C$
  $g :: A \to (B, A)$
  $p : \ A \to Bool$

$$
\begin{aligned}
& h :: \ \to List\ A \\
& h\ a = c, && p\ a \\
& h\ b = fb(h\ a'), && otherwise
\end{aligned}
$$

  where $(b, a') = g\ a$
- Notation $h = [\![(c, f), (g, p)]\!]$

Hylomorphism-2

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

$$p\ n = n == 0$$
$$g\ n = (n, n - 1)$$

Paramorphism

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

$$
\begin{aligned}
h\,0 &= b \\
h\,(\mathrm{Succ}\;n) &= f\;n\,(h\;n)
\end{aligned}
$$

- A list example:

$$
\begin{aligned}
h\,\mathrm{Nil} &= b \\
h\,(\mathrm{Cons}\;x\;xs) &= f\;x\;xs\,(h\;xs)
\end{aligned}
$$

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

Functor

- an (endo-)functor is an operation from types to types
- 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 = Fx (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 $(FA) \to A$

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`, 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:: Fix f → a`

Remember:

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

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

`fmap g :: f (Fix f) → 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

`Fx` is lossless, thus invertible.

```
unFix :: Fix f → f (Fix f)
unFix (Fx x) = x
```

$$\begin{tikzcd} f(\mathrm{Fix}\ f) \arrow[r, "\mathrm{fmap}\ g"] \arrow[d, "Fx"] & f\ a \arrow[d, "alg"] \\ \mathrm{Fix}\ f \arrow[r, "g"] & a \end{tikzcd} \quad \text{becomes} \quad \begin{tikzcd} f(\mathrm{Fix}\ f) \arrow[r, "\mathrm{fmap}\ g"] & f\ a \arrow[d, "alg"] \\ \mathrm{Fix}\ f \arrow[r, "g"] \arrow[u, "unFix"] & a \end{tikzcd}$$

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: $f\ a \to a$
Coalgebra: $a \to f\ a$

$$
\begin{tikzcd}
f(\mathrm{Fix}\ f) \arrow[r, "\mathrm{fmap}\ g"] \arrow[d, "Fx"] & f\ a \arrow[d, "alg"] \\
\mathrm{Fix}\ f \arrow[r, "g"] & a
\end{tikzcd}
\quad \text{becomes} \quad
\begin{tikzcd}
f(\mathrm{Fix}\ f) \arrow[r, "\mathrm{fmap}\ g"] & f\ a \\
\mathrm{Fix}\ f \arrow[r, "g"] \arrow[u, "unFix"] & a \arrow[u, "coalg"]
\end{tikzcd}
$$

The same reasoning about initial coalgebra applies.
Fx and unFix are inverses.
g can be defined with Fx, fmap and evaluator.
We will call **in** the coalgebra evaluator and **out** the algebra one.

Notation summary

Expanding notation to arbitrary types:

- Catamorphism:
  $(\!|\phi|\!)_F = \mu(\phi \overset{F}{\leftarrow} \mathit{out})$
- Anamorphism:
  $[\!(\psi)\!]_F = \mu(\mathit{in} \overset{F}{\leftarrow} \psi)$

“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