Peeling the Banana

Recursion schemes from first principles

Zainab Ali

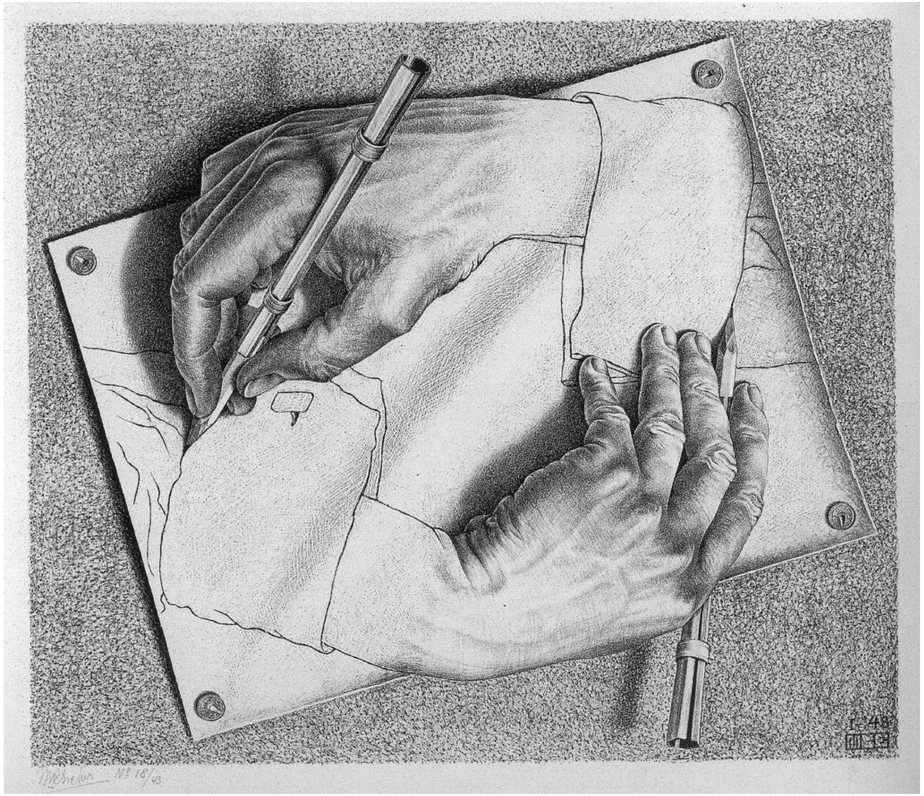
The kind of banana

Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Erik Meijer * Maarten Fokkinga † Ross Paterson ‡

Abstract

We develop a calculus for lazy functional programming based on recursion operators associated with data type definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. We shall show that all



The journey

- Explore recursive data types
- Category theory
- Derive recursion schemes
- Don’t panic!

Recursive data types

```haskell
data List = Nil | Cons Int List

xs = Cons 1 $ Cons 2 $ Cons 3 Nil
```

Recursive collapse

```haskell
multiply :: List -> Int
multiply Nil = 1
multiply (Cons h t) = h * multiply t

length :: List -> Int
length Nil = 0
length (Cons _ t) = 1 + length t
```

Recursive collapse

```
foldList :: a -> (Int -> a -> a) -> List -> a
foldList onNil _ Nil = onNil
foldList onNil onCons (Cons h t) = onCons h $ foldList onNil
onCons t

multiply = foldList 1 (*)
length = foldList 0 (const (+1))
```

multiply and length are **catamorphisms**

Recursive collapse

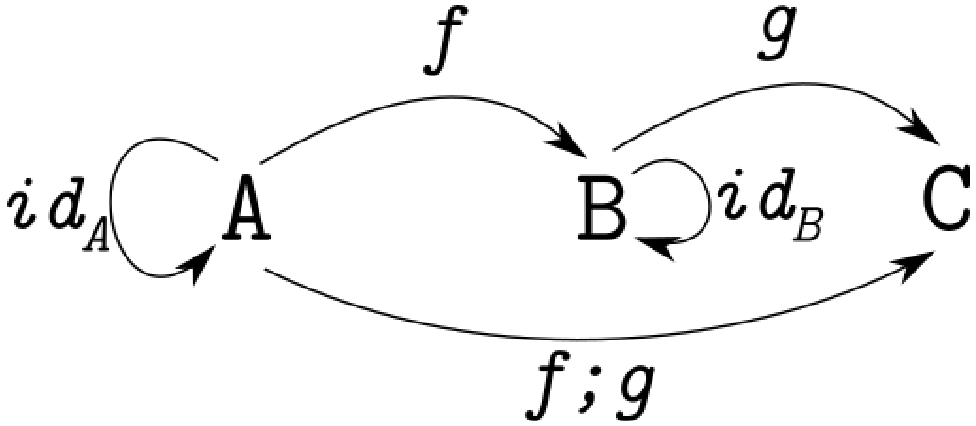
```haskell
data Tree = Leaf Int | Node Tree Tree

foldTree :: (Int -> a) -> (a -> a -> a) -> Tree -> a
foldTree onLeaf _ (Leaf i) = onLeaf i
foldTree onLeaf onNode (Node l r) = onNode (f l) (f r)
  where f = foldTree onLeaf onNode

sum :: Tree -> Int
sum = foldTree id (+)

countLeaves :: Tree -> Int
countLeaves = foldTree (const 1) (+)
```

Generalized collapse

```haskell
data List = Nil | Cons Int List

foldList :: a -> (Int -> a -> a) ->
                  List -> a
foldList onNil _ Nil = onNil
foldList onNil onCons (Cons h t) =
  onCons h $ foldList onNil onCons t
```

```haskell
data Tree = Leaf Int | Node Tree Tree

foldTree :: (Int -> a) -> (a -> a -> a) ->
                              Tree -> a
foldTree onLeaf _ (Leaf i) = onLeaf i
foldTree onLeaf onNode (Node l r) =
  onNode (f l) (f r)
  where f = foldTree onLeaf onNode
```

Category Theory

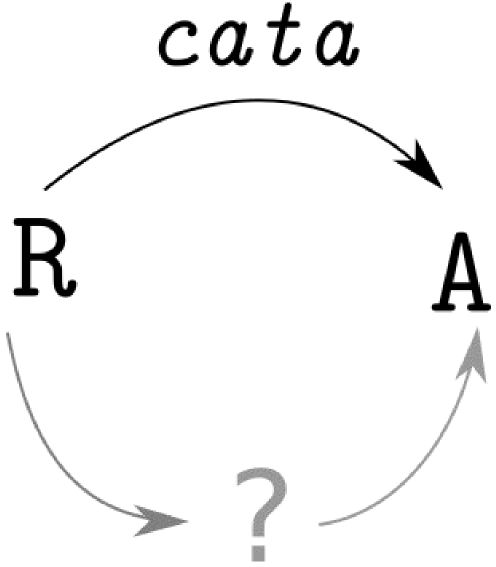


$$id_A ; f = f ; id_B = f$$

$$(f;g);h = f;(g;h)$$

Category Theory

Collapse a recursive data type **R** to a value **A**



Higher Kinded Types

```
data ListF a = NilF | ConsF Int a

foo = ConsF 1 "foo"
xfs = ConsF 1 $ ConsF 2 $ ConsF 3 NilF
xs = Cons 1 $ Cons 2 $ Cons 3 Nil
```

Higher Kinded Types

```
in' :: ListF List -> List
in' NilF = Nil
in' (ConsF h t) = Cons h t


out :: List -> ListF List
out Nil = NilF
out (Cons h t) = ConsF h t
```

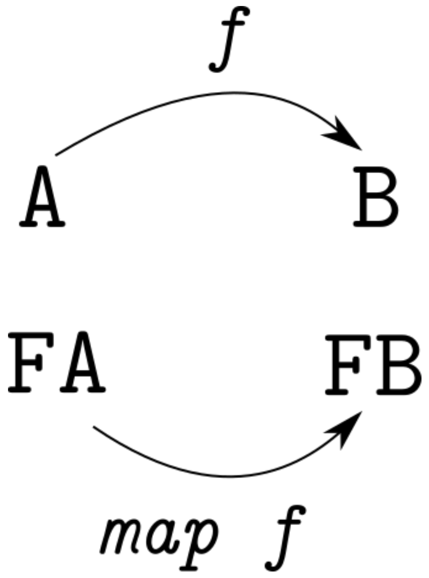
Isomorphism

$$
\begin{tikzcd}
R \arrow[loop left, "id_R"] \arrow[r, bend right, "out"'] & FR \arrow[l, bend right, "in"'] \arrow[loop right, "id_{FR}"]
\end{tikzcd}
$$

$$
\begin{aligned}
in;out &= id_{FR} \\
out;in &= id_R
\end{aligned}
$$

Functors

- Takes a object **A** into an object **FA**
- Takes a morphism **A -> B** into a morphism **FA -> FB**



Functors

```
class Functor f where
  map :: (a -> b) -> f a -> f b
```

```
instance Functor ListF where
  map f NilF = NilF
  map f (ConsF h a) = ConsF h $ f a
```

F-Algebras

$\mathbf{F_A}$ **->** **A** for a functor **F**

$$\mathtt{FA} \xrightarrow{alg} \mathtt{A}$$

F-Algebras

FA -> A for a functor **F**

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

in' :: ListF List -> List
in' NilF = Nil
in' (ConsF h t) = Cons h t

multiplyAlgebra :: Algebra ListF Int
multiplyAlgebra NilF = 1
multiplyAlgebra (ConsF h a) = h * a
```

Catamorphisms

$$
\begin{tikzcd}
\mathtt{R} \arrow[rr, bend left, "cata"] \arrow[d, bend left, "out"] & & \mathtt{A} \\
\mathtt{FR} \arrow[u, bend left, "in"] \arrow[rr, bend right, "map\ cata"'] & & \mathtt{FA} \arrow[u, bend right, "alg"']
\end{tikzcd}
$$

Catamorphisms

$$
\begin{tikzcd}
R \arrow[rr, bend left, "cata"] \arrow[d, bend left, "out"] & & A \\
FR \arrow[u, bend left, "in"] \arrow[rr, bend right, "map\ cata"'] & & FA \arrow[u, bend right, "alg"']
\end{tikzcd}
$$

$$cata = out;map\ cata;alg$$

Catamorphisms

```haskell
cata :: Functor f => (Algebra f a) -> (r -> f r) -> r -> a
cata alg out = alg . (map (cata alg out)) . out

multiply = cata multiplyAlgebra out
```

Generalized building

```
range :: Int -> List
range n = if n > 0 then Cons n (range (n - 1)) else Nil
```

`range` is an **anamorphism**

Anamorphism

$$\begin{tikzcd}
R \arrow[d, bend left, "out"] & A \arrow[l, bend right, "ana"'] \\
FR \arrow[u, bend left, "in"] & FA \arrow[u, bend right, "alg"'] \arrow[l, bend left, "map\ ana"]
\end{tikzcd}$$

Anamorphism

$$
\begin{tikzcd}
R \arrow[d, bend left, "out"] & A \arrow[l, bend right, "ana"'] \arrow[d, bend left, "coalg"] \\
FR \arrow[u, bend left, "in"] & FA \arrow[l, bend left, "map\ ana"]
\end{tikzcd}
$$

Anamorphism

$$\begin{tikzcd}
R \arrow[d, shift left, "out"] & A \arrow[l, bend right, "ana"'] \arrow[d, bend left, "coalg"] \\
FR \arrow[u, shift left, bend left, "in"] & FA \arrow[l, bend left, "map\ \ ana"]
\end{tikzcd}$$

$$ana = coalg;map\ \ ana;in$$

Anamorphism

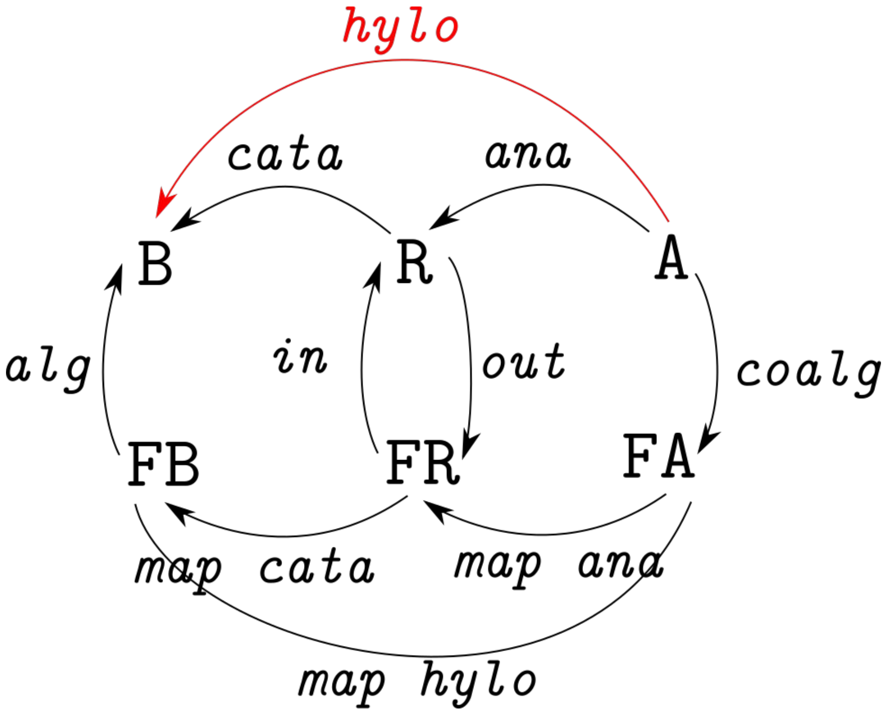
```haskell
type Coalgebra f a = a -> f a

ana :: Functor f => Coalgebra f a -> (f r -> r) -> a -> r
ana coalg in' = in' . (map (ana coalg in')) . coalg

rangeCoalgebra :: Coalgebra ListF Int
rangeCoalgebra n = if n > 0 then ConsF n (n - 1) else NilF

range = ana rangeCoalgebra in'
```

Generalized recursion

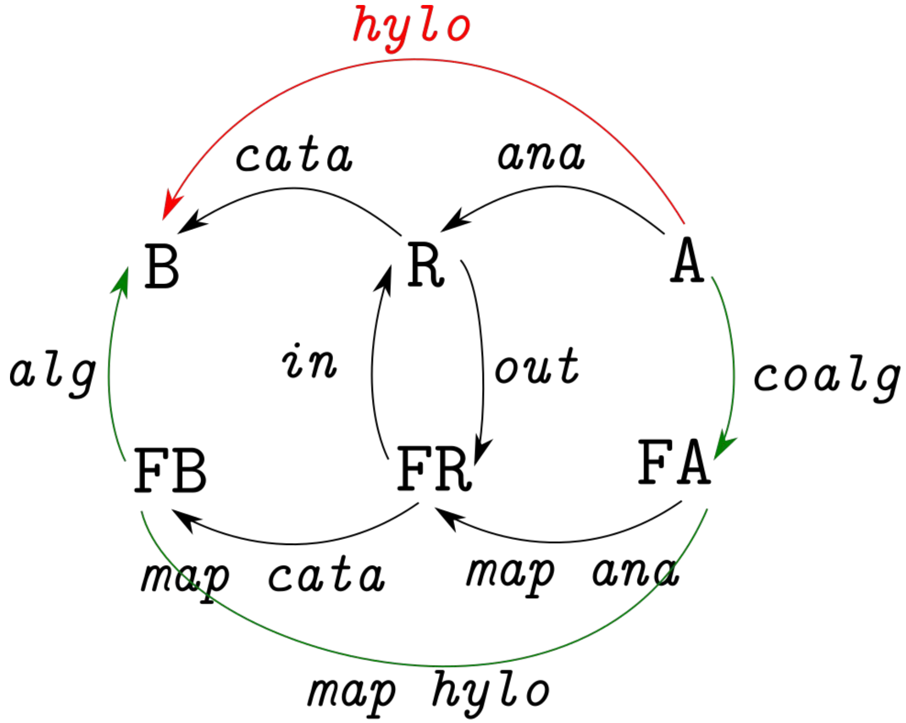
```haskell
factorial :: Int -> Int
factorial n =
  if n > 0
  then n * factorial (n - 1)
  else 1
```

factorial is a **hylomorphism**

Hylomorphism



Hylomorphism



$$hylo = coalg;map\ hylo;alg$$

Hylomorphism

```
hylo :: Functor f => Coalgebra f a -> Algebra f b -> a -> b
hylo coalg alg = alg . (map (hylo coalg alg)) . coalg

factorial = hylo rangeCoalgebra multiplyAlgebra
```

Boilerplate!

```
data List = Nil | Cons Int List
data ListF a = NilF | ConsF Int a

in' :: ListF List -> List
in' NilF = Nil
in' (ConsF h t) = Cons h t

out :: List -> ListF List
out Nil = NilF
out (Cons h t) = ConsF h t
```

Removing boilerplate

```
data Foo = ???

in' :: Algebra ListF Foo
in' = ???

out :: Coalgebra ListF Foo
out = ???
```

Removing boilerplate

```
data Fix f = Fix { unfix :: f (Fix f)}

in' :: Algebra ListF (Fix ListF)
in' = Fix

out :: Coalgebra ListF (Fix ListF)
out = unfix

xfs = Fix $ ConsF 1 $ Fix $ ConsF 2 $ Fix $ ConsF 3 $ Fix NilF
```

There’s more!

- Fusion
- Comonads
- para / meta / zygo ...

Takeaways

- Recursion schemes!
  - Catamorphism
  - Anamorphism
  - Hylomorphism
- Fixed points
- **Category theory is awesome!**

In the wild

- **matryoshka** in Scala
  - https://github.com/slamdata/matryoshka
- **recursion-schemes** in Haskell
  - https://github.com/ekmett/recursion-schemes
- **recursion_schemes** in Idris
  - https://github.com/vmchale/recursion_schemes

Some resources

Meijer, E., Fokkinga M. and Paterson R. **Functional programming with bananas, lenses, envelopes and barbed wire**
https://maartenfokkinga.github.io/utwente/mmf91m.pdf

Milewski, B. **Understanding F-Algebras**
https://bartoszmilewski.com/2013/06/10/understanding-f-algebras/

Wadler, P. **Recursive types for free!**
http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt

Gibbons, J. **Datatype-Generic Programming**
http://www.cs.ox.ac.uk/jeremy.gibbons/publications/dgp.pdf

Thank you!

Questions