An Introduction to fold

Pavel Paroulek

October 19, 2015

About explicit recursion

- Explicit recursion in FP is harder to read and reason about
- Some consider it a *goto* statement in the FP world
- One can avoid it with more structured aproaches to recursion
- Usually higher order functions that 'abstract-away' recursion statements
- Possibility to take advantage of laws and properties which these abstraction conform to

xkcd 292 - goto

Example of higher order function

- Some useful functions in FP: map, filter, zipWith, **fold**, unfold ...
- zipWith :: $(a \to b \to c) \to [a] \to [b] \to [c]$
- Example - sum of elements at the same position in $[1, 1, 1]$ and $[1, 2, 3]$
- Result should be $[1 + 1, 1 + 2, 1 + 3]$ (i.e. $[2, 3, 4]$)

```
count [] [] = 0
count (x:xs) (y:ys) = x + y + (count xs ys)
```

- We can use the function *zipWith*

```
let l1 = [1,1,1]
let l2 = [1,2,3]
count l1 l2 == [2,3,4] // True
zipWith (+) l1 l2 == count l1 l2 // True
```

FOLD AS A POWERFUL HIGHER ORDER FUNCTION

- *fold* is a structured way to go over lists
- It reduces/processes the input list applying a given function
- The signature is $(a \to b \to a) \to a \to [b] \to a$ where the first argument is a function, second argument is a initial value of the accumulator and the third value the list being processed
- Through fold one can define other higher order functions like length, map, filter, reverse, sum ...

```
map :: (a -> b) -> [a] -> [b]
map f = foldl (\xs x -> xs ++ [f x]) []
map (+1) [1,2,3] == [2,3,4] //True
```

Properties of fold

- There are certain properties, that have been derived for fold
- The properties apply generally for folds on lists
- The 'universal property' implies that if you have the following idiom

```
h []     = v
h (x:xs) = f x (h xs)
```

- it is possible to replace it with

```
h = fold f v
```

Example of the Universal property

- Let's rewrite the filter function as a fold using the universal property
- Based on a given function filter removes certain elements from the input list
- 'h (x:xs) = f x (h xs)'

```
filter :: (a -> Bool) -> [a] -> [a]
filter g  [] = []
filter g (x:xs)
    | g x       = x : filter g xs
    | otherwise = filter g xs
```

- in this case f could have the form f x xs = IF condition THEN recursive_op1 ELSE recursive_op2

Example of the Universal property

- 'h (x:xs) = f x (h xs)'
- We can extract the recursion and get

```
filter :: (a -> Bool) -> [a] -> [a]
filter g  [] = [] // h[] = v
filter g (x:xs) = f x (filter g xs) // h (x:xs) =
                                    // f x (h xs)
  where
    f x xs = if g x then x : xs else xs
```

- now we can use the 'f' function (remember 'h = fold f v') in fold and we get

```
filter g = foldr (\x xs -> if g x then x : xs else xs) []
```

The fusion law

- The fusion law implies that under some conditions you can combine the composition of a function and fold into a single fold

```
h . fold g w = fold f v
```

- This holds only under some conditions, which can be derived from the universal property
- A complete description of the fusion law can be found in [1]
- Example - the composition of a fold and map can be reduced to a fold

```
foldr f e . map g = foldr (f . g) e
```

The banana-split law

- This law states that that if you have 2 folds operating on the same argument, they can be merged into a single fold.
- Only a single traversal of the list is necessary

```
fork (foldr f a, foldr g b) = foldr h (a, b)

fork (f , g) x = (f x, g x)
h x (y, z) = (f x y, g x z)
```

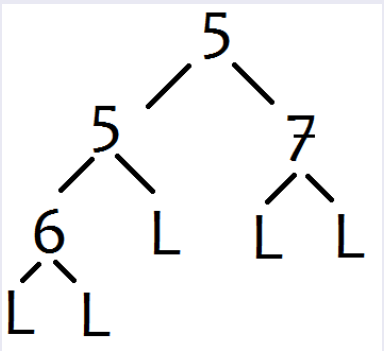
- Described in the article Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire [7]
- The article is very hard to read because it uses the Bird — Meertens formalism

Towards a more general fold

- fold operates on lists.
- Can we make a fold over other types of datatypes?
- Let's define a *recursive* datatype for a tree with 3 different nodes

```
Tree v = Op1 v (Tree v) (Tree v)
       | Op2 v (Tree v) (Tree v)
       | Leaf

someTree = Op1 5 (Op2 5 (Op1 6 Leaf Leaf) Leaf) (Op2 7
    Leaf Leaf)
```

someTree



Towards a more general fold

- The recursive structure (in this case Op1 and Op2 child nodes) is specific for each recursive datatype
- We can define this datatype without recursion as follows

```
TreeF v c = Op1 v c c
          | Op2 v c c
          | Leaf

someTree :: TreeF Integer (TreeF Integer (TreeF Integer (
    TreeF v c)))
someTree = Op1 4 (Op2 3 (Op1 4 Leaf Leaf) Leaf) Leaf
```

- This form is also called the 'pattern functor' of Tree
- The return type will grow with the depth of the tree
- The goal is to express any tree with a *single type*

Fixed point combinator

- For the purpose of expressing the TreeF with a single type we can use a datatype fixed point combinator
- The definition of this combinator is

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

out :: Fix f -> f (Fix f)
out (In x) = x
```

- The combinator closes the type under repeated applications
- It is analogous to the fixed point equation $x = f(x)$
- The In and out functions have a structure of a function, that is

```
*Main> :t out
out :: Fix f -> f (Fix f)

*Main> :t In
In :: f (Fix f) -> Fix f
```

Using the fixed point combinator in Tree

- Now to get rid of the repeated applications of TreeF to itself we can apply in to In in the type Fix f

```
data TreeF v c = Op1 v c c | Op2 v c c | Leaf

newtype Fix f = In (f (Fix f))
type Tree v = Fix (TreeF v)

exampleTree :: Tree Int
exampleTree = In $ Op1 5
                   (In $ Op2 5
                      (In $ Op1 6 (In Leaf) (In Leaf))
                      (In Leaf))
                   (In $ Op2 7 (In Leaf) (In Leaf))
```

- We are wrapping the TreeF constructors with an additional In constructor of the Fix datatype
- We get a single type signature in a function dealing with recursive datatypes
- For the datatype to be useful we need a way to traverse and evaluate it (fold-like style)

TreeF as a functor

- fold traverses each element in a list and gives back a single value
- In order to evaluate TreeF node we need evaluate it's children
- We decouple the evaluation of the node itself and the recursive traversal of it's children

```
instance Functor (TreeF v) where
  fmap f Leaf = Leaf
  fmap f (Op1 v l r) = Op1 v (f l) (f r)
  fmap f (Op2 v l r) = Op2 v (f l) (f r)
```

- We make a functor out of TreeF where fmap :: Functor f $\Rightarrow (a \rightarrow b) \rightarrow$ f a $\rightarrow$ f b
- If we fmap a function on a TreeF value it will propagate this function through the whole tree

Defining an algebra

- Now we need to define a function for evaluation the nodes in the tree
- We define this function on the TreeF datatype

```
data TreeF v c = Op1 v c c | Op2 v c c | Leaf

intAlg :: TreeF Int Int -> Int
intAlg Leaf = 1
intAlg (Op1 v l r) = v * (l + r)
intAlg (Op2 v l r) = v * (l - r)
```

- In the literature this function is called an algebra
- An algebra describes how to map a functor where the recursive positions have already been evaluated to a result
- Notice we have specialized the recursive positions to an Int datatype. This is called a carrier.
- We can use any type as carrier, thus redefining the purpose and calculations in the tree.

Algebra in more detail

- What exactly is an algebra?
- Let C be a category from the category theory. An algebra of a functor F is a tuple containing an object A (the carrier) in C and a morphism alg:$F(X) \to X$.
- Why was intAlg called an algebra?
- The type signature was intAlg :: TreeF Int Int $\to$ Int
- In plain english it was intAlg :: functor type_used_in_functor carrier_type $\to$ carrier_type
- In other words it has all the parts an algebra needs: a functor, a carrier and it itself defines a morphism $F(X) \to X$

Toward a general fold-like function

- So far we have a way to traverse the tree and a way to evaluate each node of the tree.
- We could already formulate a fold-like function which would collapse the tree into a single value (of the carrier datatype)
- The goal is to define a general higher order fold-like function, which would be able to operate on any algebra and functor
- In order to derive this function, we have to go slightly deeper into the theoretical underpinings of algebra

Toward a general fold-like function

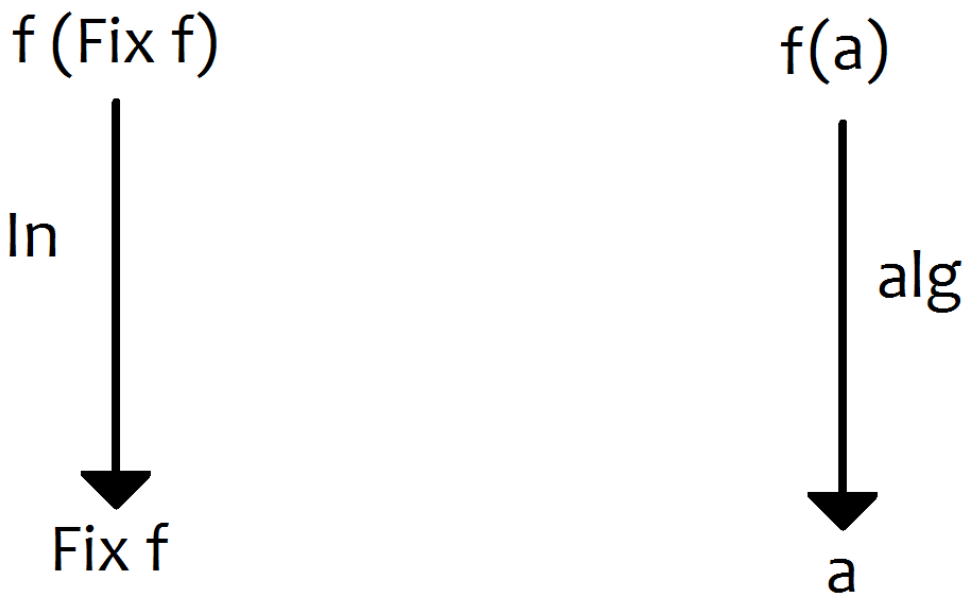
- If we specify the generalised fold through the algebra and carrier that we defined so far, we would get a function specialised for just that one case
- we have to use a more 'generic' algebra, one that would also encompass this one
- How could we find such an algebra?
- Recap: Let C be a category from the category theory. An algebra of a functor F is a tuple containing an object A (the carrier) in C and a morphism alg:$F(X) \to X$.
- Hint: we already can hide **any** functor in the Fix datatype, so it's 'generic' enough
- Could we somehow use the Fix to create this generalised algebra? What would be the alg morphism and what would be the carrier?
- If we remember the contructor of Fix (the In function):

```
*Main> :t In
In :: f (Fix f) -> Fix f
```

- We see the resamblance to the morphism $F(X) \to X$. As Fix is a 'fix point' we have that the carrier would be the Fix datatype

Pictorial representation of algebras

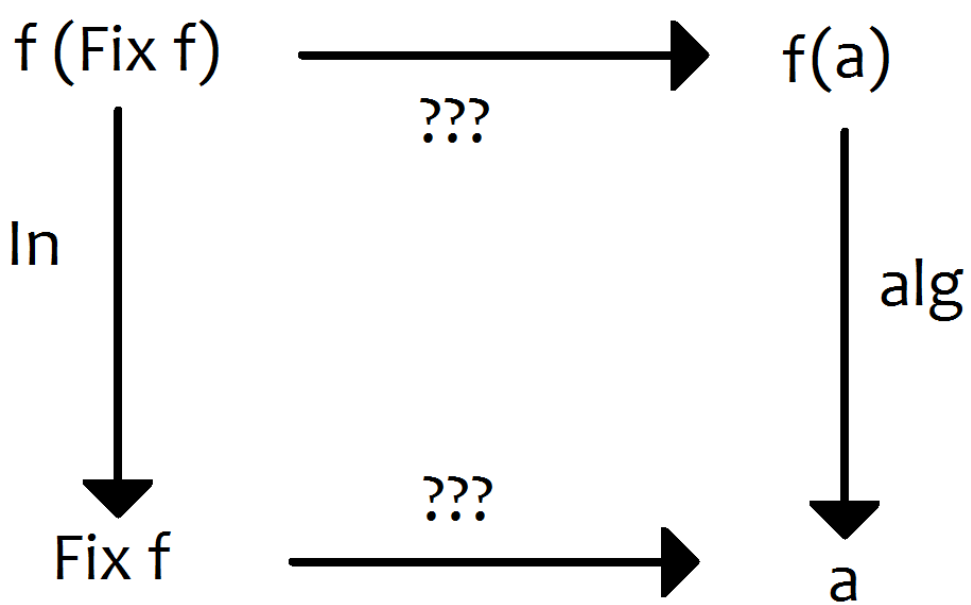
Two algebras



- We have two separate algebras, the one with Fix and some another which represents any other algebra
- if we define our general fold on the Fix-Algebra we need a way to map from it to any other algebra.

Commutative diagram

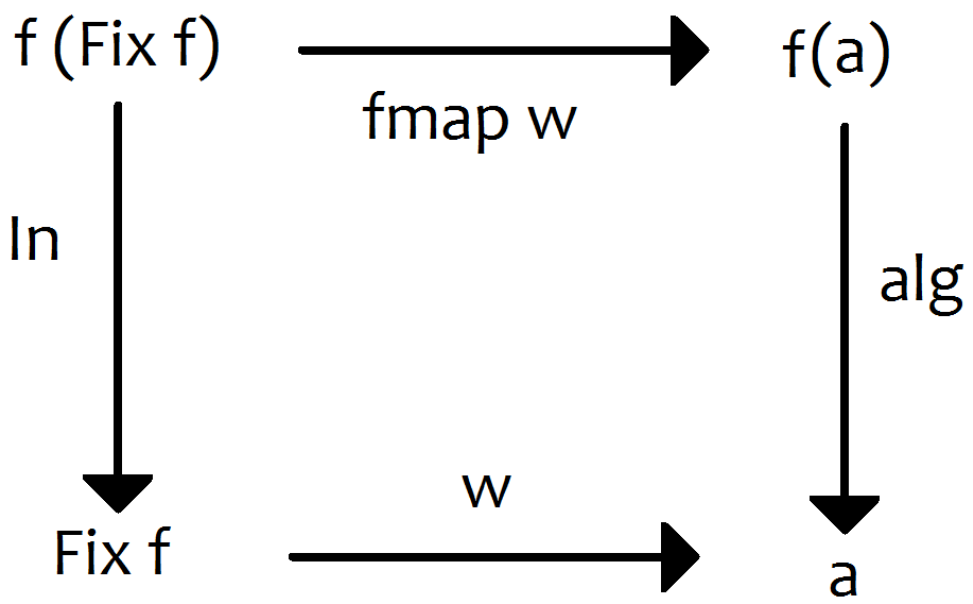
Unknown mapping



- We somehow need to define the mapping between these datatype

Commutative diagram

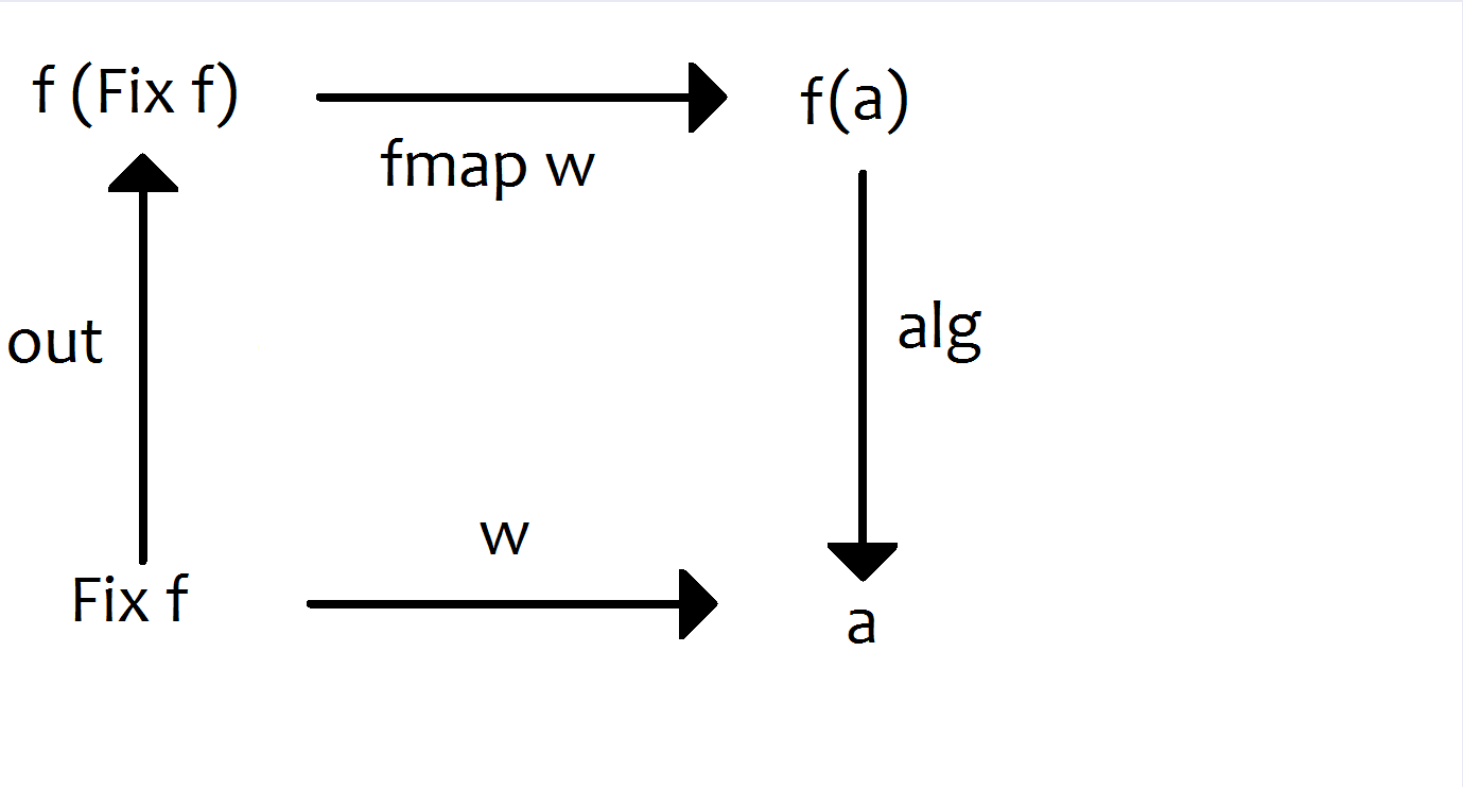
With mapping



- We just replaced the questionmarks with w
- Because f is a functor we can use fmap to apply w to f (Fix f)
- How can we find w and somehow incorporate alg so that we can evaluate any algebra?

Commutative diagram

Inverting the arrow



- There are (at least) 2 tricks! The first is that we invert an arrow and use the out function
- The second trick is that we define w recursively using itself in the definition

```
w = alg . (fmap w) . out
```

Defining cata

- Why wouldn't the recursive definition loop forever? w calls w in it's body, so it could loop forever.
- But since we are using fmap, it peels only a finite sequence of layers in Fix, at some point it will stop
- The generalised fold is called a catamorphism
- In order to supply in with a custom alg function, we can turn in into a higher order function

```
cata' = alg . (fmap cata') . out

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

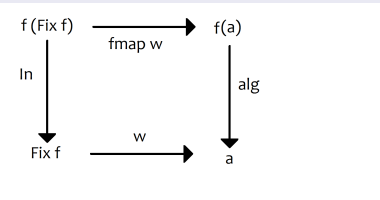
A practical example

- We can try the catamorphism on our tree

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

exampleTree :: Tree Int
exampleTree = In $
    Op1 5
      (In $ Op2 5
         (In $ Op1 6 (In Leaf) (In Leaf))
         (In Leaf))
      (In $ Op2 7 (In Leaf) (In Leaf))

intAlg :: TreeF Int Int -> Int
intAlg Leaf = 1
intAlg (Op1 v l r) = v * (l + r)
intAlg (Op2 v l r) = v * (l - r)

print $ cata intAlg exampleTree //265
```

Properties

Catamorphism



- Catamorphisms have the same laws as folds. As to avoid introducing new mathematical concepts I only mention the spirit of the laws without regard to the exactness of the formulation
- The universality property translated to the diagram above would state that

```
alg . (fmap w) = w . In
```

- The fusion law states that the composition of a function and a catamorphism can be rewritten as a catamorphism (no need for intermediate data representations).

```
h . w = w'
```

- When we have 2 algebras in the same functor with a different carrier then using the Banana Split Law 2 catamorphisms can be expressed as a single catamorphism (single traversal of the datatype)

Limitations

- Catamorphisms are part of the datatype generic approch to programming
- Generic functions can be used to write serialisers [2], evaluators, pretty-printers ...
- The approach deals with datatypes with regular structure.
- Different methods have to be used to deal with non-regular datatypes (nested or mutually recursive datatypes)
- Example of a nested datatype:

```
data Nest a = Nil | Cons (a, Nest (a, a))

Cons
   (10,
      Cons
        ((10,10),
          Cons
             (((10,10),(10,10)),
                  Nil)))
```

Related topics

- Paramorphism is an extension of catamorphism
- Paramorphism operate on the results of the evaluation of substructures as well as on substructures themselves
- Usefull when you need to further examine the data from which the result was computed
- We could define a monadic catamorphism where the algebra is alg : f a $\rightarrow$ m a where m is a monad
- The benefit is that the we can use monads in the algebra but don't have to specify the monads in the pattern functor
- We can use memoization in catamorphism (define a memoizing catamorphism) to speed up computations

Size Matters

- Smaller code bases have fewer bugs
- Therefore it is advantageous to strive for less code
- There are to trends related to project size
- 1. Concision (syntax, convention over configuration ..), which leads to small improvements
- 2. Generality (leads to big improvements)
- The difference of solving problems more **efficiently** vs. solving problems **differently**

References

1. A tutorial on the universality and expressiveness of fold, Graham Hutton, 1999
2. Generic Storage in Haskell, Sebastiaan Visser, Andres Löh, 2010
3. When is a function a fold or an unfold?, Jeremy Gibbons, Graham Hutton, Thorsten Altenkirch, 2001
4. Understanding F-Algebras, blog post on www.fpcomplete.com, Bartosz Milewski, 2013
5. Catamorphisms, blog post on www.fpcomplete.com, Edward Kmett, 2014
6. Initial Algebra Semantics is Enough!, Patricia Johann, Neil Ghani, 2007
7. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, Erik Meijer Maarten Fokkinga, Ross Paterson, 1991

- Questions?

Recursive giraffe, Farley Katz cartoon