# COALGEBRAS AND AUTOMATA

(Also, I've since learnt that all of my definitions of fold and unfold below can be wrapped up into a single pair of definitions. See, eg. http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/hodgp.pdf)

An unHTMLified version of this document is available at <http://www.sigfpe.com/Computing/fold.lhs> which you can run in hugs or ghc. We use multiparameter classes so run this with -fglasgow-exts in ghc or -98 in hugs.

> import Maybe

## INTRODUCTION

Many years ago I tried to write some code for a finite state automaton. I wanted to write a function that took as input a state label and then returned a new state label. But then I wondered what the point of the state label was. Couldn't I simply write a function that took as argument some input and then returned another function of the same type. In other words I wanted a type, *X*, such that *X* = *A* → *X*.

Anyway, I decided that maybe I was being a bit perverse and ignored this for many years. But then I tried playing around with generalizations of fold and unfold and I was amazed to find this type reappearing in a very natural way. What's more, it turns out that the function to convert a state machine that uses state labels into one that makes no reference to labels is actually a special case of the unfold operation.

## F-ALGEBRAS

To start at the beginning...

I was looking at problem 2.2.4.2 in Pierce's Basic Category Theory for computer scientists. I constructed the required inverse but the level of abstraction was getting me a bit confused and I was unable to prove that it was the inverse. So I cheated and had a look at Wadler's paper here:

<http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt>

I then decided that I ought to actually work out the details for some examples and was surprised by what I found.

Given a functor *F* in a category C, an *F*-Algebra is an arrow *in* : *F*(*X*)→ *X* for some *X*. We'll also write an F-algebra as (X,fin) to make the dependence on the object of the underlying category clear.

We can draw an F-algebra as:

F(X)

|

| α

V

X

We can define this in Haskell as follows:

> class (Functor f) => Algebra f x where

> fin :: f x -> x

The *F*-Algebras form a category where the arrows are arrows *f* : *X* → *Y* of C so that the following diagram commutes:

α

F X -----------> X

| |

| |

F f | | f

| |

V β V

F Y -----------> Y

Let's consider a simple example: *F* = *Maybe*. Remember that the *Maybe* functor is defined by

*Maybe* *X* = *Nothing* | *Just* *X*.

An object of type *Maybe* *X* is either the distinguished object *Nothing* or an *X*. We can write this as

*F*(*X*) = 1 + *X*

where + is disjoint union and 1 is a terminal object (eg. a singleton set).

An *F*-Algebra is therefore a function *α* : *Maybe* *X* → *X*. So, when we apply *α* to an element of *F X* we need to consider two cases: the possibility that it's acting on *Nothing*, and the possibility that it's acting on *Just* *X*. In the former case *α Nothing* is simply an element *x* : *X*. In the latter case it's essentially an arrow *f* : *X* → *X*. So, choosing a function *α* : *Maybe* *X* → *X* is the same thing as choosing an element of *X* and a function *X* → *X*.

Let's define this example over the naturals:

> data Nat = Zero | Succ Nat

> instance Algebra Maybe Nat where

> fin Nothing = Zero

> fin (Just x) = Succ x

Notice how *α* has been written as the two cases described above.

Now consider another Maybe-algebra (*Y, β*) with *β*(*Nothing*) = *y* and *β*(*Just x*) = *g*(*x*).

Define *pow* : *Nat* → *Y* by *pow*(*n*) = (*g n*) *y*. (*g n* means *g* applied *n* times.) It should be clear that the following diagram comutes:

α

1+N -----------> N

| |

| |

(1+\_) pow | | pow

| |

V β V

1+Y -----------> Y

(Remember that a function *f* : *X* → *Y* lifts to the function *f'* : 1 + *X* →1 + *Y* where

*f* (*Nothing*) = *Nothing*

*f* (*Just x*) = *Just f*(*x*).

See the definition of *fmap* for *Maybe* in the Prelude.)

Let's make another Maybe-algebra:

> instance Algebra Maybe Float where

> fin Nothing = 1.0

> fin (Just x) = 2.0 \* x

In this case we have

> f :: Nat -> Float

> f n = 1.0 \* 2^n

We claim f is a Maybe-algebra arrow so we should compare f . fin and fin . (fmap f) on a few values:

> ex0 = map (fin . fmap f) [Nothing,Just 0,Just 1,Just 16]

> ex1 = map (f . fin) [Nothing,Just 0,Just 1,Just 16]

You should see that ex0 = equals ex1.

So we have exhibited an F-Algebra arrow from the F-Algebra (X,fin) to (Y,gin). What's more, there is only one

possible F-Algebra arrow from (X,fin) to (Y,gin). As our construction was completely general it applies to [any Maybe-algebra] an (Y,gin). This means that for every F-Algebra there is a unique F-Algebra arrow to it from (X,fin). In other words, (N,fin) is an initial object in the category of F-Algebras. Suppose, for some F, we have an initial F-Algebra (X,fin). Then for any F-Algebra (Y,f) the unique map h making the following diagram commute is clearly a function of f.

in

F(X) ----------> X

| |

| |

F(h) | | h

| |

V f V

F(Y) ----------> Y

where h = fold f

This function is called fold. We may now define:

> class (Algebra f x) => Initial f x | x -> f where

> fold :: (f y -> y) -> x -> y

We have an arrow fin : F(N) -> N. As F is a functor we also have an arrow F(fin) : F(F(N)) -> F(N). This is also an F-Algebra. So now consider fold(F(fin)). This must map N -> F(N). It's not hard to show that this is the inverse of fin. In effect it subtracts 1 but if we try to subtract 1 from 0 we get Nothing. It can be shown that in general fold(F(fin)) is the inverse of fin for initial algebras. So initial algebras essentially give a solution to the equation X = F(X) up to isomorphism. That fold(F(fin)) is the inverse to fout is essentially "Lambek's lemma". We may now complete our definition of Initial by writing all this in Haskell as:

> fout :: x -> f x

> fout = fold (fmap fin)

fout is the inverse of fin. Haskell constructs fout for us automatically using fold.

> instance Initial Maybe Integer where

> fold f 0 = f Nothing

> fold f (n + 1) = f $ Just $ fold f n

We can now demonstrate that *in* is the inverse of *out* by applying it to an integer:

> --ex2 = fin $ fout $ (4::Integer)

> ex2 = map (fin . fout) [0,1,16 :: Integer]

> ex3 = map (fout . fin) [Nothing,Just 0,Just 1,Just 16 :: Maybe Integer]

## F-COALGEBRAS

We can now start dualising our constructions to see what new objects appear.

We start by defining F-coalgebras:

> class (Functor f) => Coalgebra f x where

> outf :: x -> f x

This time a F-coalgebra is an arrow f : X -> F(X) and we'll write a coalgebra as (X,f) when we wish to show

the dependence on X. Arrows between F-coalgebras are defined in the obvious way. We now consider, not

the initial objects in the category of F-coalgebras but final objects. This time we have a unique arrow from any

F-coalgebra to the final F-coalgebra. If the F-coalgebra is (Y,f) then the unique map from (Y,f) to the final

F-coalgebra is specified by a function we call 'unfold'. Here is the complete signature for a final F-coalgebra.

> class (Coalgebra f x) => Final f x | x -> f where

> unfold :: (y -> f y) -> (y -> x)

Again we can construct the inverse to outf, this time using unfold:

> inf :: (f x) -> x

> inf = unfold (fmap outf)

Time for an example. Define the following function on the natural numbers.

> collatz :: Integer -> Maybe Integer

> collatz 1 = Nothing

> collatz n

> | even n = Just $ n `div` 2

> | odd n = Just $ 3\*n+1

This defines a Maybe-coalgebra

> instance Coalgebra Maybe Integer where

> outf n = collatz n

The famous Collatz conjecture states that if we repeatedly apply collatz to any natural we eventually hit 1. Suppose

the Collatz conjecture is true. Then we can ask how many times we need to apply collatz to a number before we reach 1. Define:

> count f x = case f x of

> Nothing -> 0

> Just x -> 1 + count f x

This should also give a clue why we used Maybe Natural instead of just Natural. We want to count the number of times we need to iterate the function before we hit a chosen value, not just zero. We can do this by saying our function maps our chosen value to Nothing. So conveniently the function f serves to both describe the iteration and say where it ends.

There is a catch however. If the Collatz conjecture is false then c collatz x might not terminate. So the answer might be 'infinite'. So strictly speaking, c should map into the naturals extended by infinity:

> data Count = Infinity | Natural Integer deriving (Show)

> plusOne (Natural n) = Natural (1+n)

> plusOne Infinity = Infinity

The Count type is a Maybe-coalgebra:

> instance Coalgebra Maybe Count where

> outf (Natural 0) = Nothing

> outf (Natural (x+1)) = Just $ Natural x

> outf Infinity = Just Infinity

So if we adjust count f so it is of type Count, instead of integer,

then it gives a Maybe-coalgebra arrow. It is the unique such arrow to

Count. We used nothing special about the collatz function so we have,

constructed unfold. We can now implement Count as a final Maybe-coalgebra:

> instance Final Maybe Count where

> unfold f x = case f x of

> Nothing -> Natural 0

> (Just x) -> plusOne $ unfold f x

Let's show that inf . outf is the identity:

> ex4 = map (inf . outf) [Natural 0,Natural 1,Natural 17,Infinity]

You may notice a catch: it fails on the last element of the list. unfold returns Infinity if repeated iteration of our function never

hits Nothing. But there's no way we can know this simply by iterating it.

So, mathematically speaking, inf is the inverse of outf. But we can't

always compute it.

## LISTS AND TREES

At this point you're surely wondering why these functions are called fold and unfold.

Well let's define a functor F:

> data F a = One | Pair Integer a deriving (Eq,Show)

F a is essentially Maybe (Integer,a)

We need a function to extract data back out of F:

> pair (Pair a b) = (a,b)

F is a functor because we can define:

> instance Functor F where

> fmap f One = One

> fmap f (Pair a b) = Pair a (f b)

This satisfies the usual rules for functors.

It shouldn't be hard to see that ([Integer], fin), with fin defined below, is actually an F-algebra:

> instance Algebra F [Integer] where

> fin One = []

> fin (Pair a b) = a:b

(One slight proviso, we're really talking about finite lists.)

fin glues together an Integer and a list of Integers to make another list. It also gives a way to make an empty list. So as before, we're using fin to encode a pair of things, a singled out element [] and a function (:).

Suppose we have an element, g, of X, and a function f: X -> Integer -> X. Then we can map any list of integers to X using the standard prelude function foldr. We get foldr f g : [Integer] -> X. In fact, if we view a list like [1,2,3] as being a composition of functions: (:) 1 ((:) 2 ((:) 3 [])) we get fold f g [1,2,3] be replacing (:) with f and [] with g. It should now be easy to see that ([Integer],fin) is an initial F-algebra and that fold and foldr are essentially the same thing:

> instance Initial F [Integer] where

> -- unpack f to be in a suitable form for foldr

> fold f = foldr (cons f) (f One) where

> cons f a b = f $ Pair a b

As usual we can check the isomorphism:

> ex5 = fin $ fout [1::Integer,2,3,4,5]

Many papers discuss unfold. We'll call it unfoldr here because it pairs naturally with foldr.

> unfoldr p f g x = if p x then [] else f x : unfoldr p f g (g x)

g is the handle on a machine. Each time we crank the handle g maps an object to a new object and we collect an output from this machine by applying f to this object. We collect up all of the outputs into a list until the function p tells us to stop. For example:

> ex6 = unfoldr (>20) id (2\*) 1

This returns the list of powers of 2 that don't exceed 20. We crank the (2\*) handle until the condition is satisfied.

unfoldr is essentially the unfold operation for lists (except that this time we \*are\* allowing infinite lists - for example what you get if you have p \_ = False.

> instance Coalgebra F [Integer] where

> outf [] = One

> outf (a:b) = Pair a b

> instance Final F [Integer] where

> -- unpack f to be in a suitable form for unfoldr

> unfold f = unfoldr (isOne . f) (fst . pair . f) (snd . pair . f) where

> isOne One = True

> isOne \_ = False

> ex7 = inf $ outf [1::Integer,2,3,4,5]

We can now carry out a similar analysis for trees instead of lists. Trees are G-algebras for the functor G:

> data G a = One' | Triple Integer a a deriving (Show)

> instance Functor G where

> fmap f One' = One'

> fmap f (Triple a b b') = Triple a (f b) (f b')

> data Tree a = Empty | Tree a (Tree a) (Tree a) deriving (Show)

> instance Algebra G (Tree Integer) where

> fin One' = Empty

> fin (Triple a b b') = Tree a b b'

> instance Initial G (Tree Integer) where

> fold f Empty = f One'

> fold f (Tree a b b') = f (Triple a (fold f b) (fold f b'))

Just as fold can collapse down a list by applying a function between successive elements, fold collapses down a tree in just the same way.

Here's our usual example of the isomorphism at work:

> ex8 = fin $ fout $ Tree

> (6::Integer) (Tree 2 Empty Empty) (Tree 7 Empty (Tree 0 Empty Empty))

Now we get to the interesting example.

## AUTOMATA

Consider the functor H:

> data H x = H (Char -> Maybe x)

> instance Functor H where

> fmap f (H g) = H (fmap f . g)

For convenience we define unH:

> unH (H x) = x

An H-coalgebra is a function X -> H(X). In essence it's a function X -> Char -> Maybe X. Forget about the Maybe for the moment. We have a function f : X -> Char -> X. This is essentially an automaton with X labelling the states and the function f defining the transitions. This is the kind of regular expression you use to recognise strings of characters. But we need one last thing, a way to mark that a string has been recognised. This is why we use Maybe. A value of Nothing, instead of a state, means the string has been matched. (This is known as a partial automaton as it grinds to a halt at this point.)

Here's an example:

> f :: Integer -> Char -> Maybe Integer

> f 0 'c' = Just 1

> f 0 \_ = Just 0

> f 1 'a' = Just 2

> f 1 \_ = Just 0

> f 2 't' = Nothing

> f 2 \_ = Just 0

This function can be used to recognize the string "cat" when it appears

as a substring of another string. In fact, we can define

> runMachine f state "" = False

> runMachine f state (c:cs) = case f state c of

> Nothing -> True

> (Just state') -> runMachine f state' cs

> ex9 = map (runMachine f 0) ["cat","dog","wildcats"]

The Integer labels are now safely locked up inside (runMachine f 0) but the price we pay for that is that we can no longer refer to states. We've managed to hide the labels at the expense of completely hiding away the states. What we want is to manipulate staes without bothering with labels.

So now we construct the H-coalgebra. I call this a Hypertree as it has some similarities to the tree example above. In a sense a Hypertree is a tree with one child for each element of X (though using Maybe means that the child "may be" a dead end with no further children.)

> data Hypertree x = HTree (x -> Maybe (Hypertree x))

Note that I didn't just pluck that definition out of nowhere. I've explicitly defined something that looks like it ought to give a solution to H(X) = X.

> instance Coalgebra H (Hypertree Char) where

> outf (HTree f) = H f

> instance Final H (Hypertree Char) where

> unfold f x = HTree (\c -> case unH (f x) c of

> Nothing -> Nothing

> Just n -> Just (unfold f n)

> )

> run :: Hypertree Char -> [Char] -> Bool

> run \_ "" = False

> run (HTree f) (c:cs) = case f c of

> Nothing -> True

> Just x -> run x cs

This time we can write a version of runMachine that makes no reference to the fact that our states were labelled with integers.

So here goes. We'll take the function f and turn in into a Hypertree using unfold:

> catDetector = unfold (H . f) 0 :: Hypertree Char

No reference to Integer anywhere in the type of catDetector. And yet the catDetector itself represents an automaton state and we can obtain more states from it. We have succeeded in hiding the labels away where

nobody can get at them. (There must be a connection with existential types here.)

Let's see if it works:

> ex10 = map (run catDetector) ["cat","dog","wildcat"]

And now we can test Lambek's lemma:

> catDetector' = inf $ outf $ catDetector

(At this point I have no idea how inf . outf actually works...)

> ex11 = map (run catDetector') ["cat","dog","wildcat"]

So there we have it. A common class for all partial automata (that accept

Char inputs) and a function, unfold, that converts a description of such

an automaton into an actual automaton.

## EPILOGUE

Can you see where this is going? I'll let you fill in the details below...

> data Command = Halt | Push Integer | Pop | Add | Swap | Print

> interpret Halt stack = Nothing

> interpret (Push n) stack = Just (n:stack,"")

> interpret Pop (\_:stack) = Just (stack,"")

> interpret Add (m:n:stack) = Just ((m+n):stack,"")

> interpret Swap (m:n:stack) = Just (n:m:stack,"")

> interpret Print (m:stack) = Just (stack,show m)

> data Interface x = I (Command -> Maybe (x,String))

> data Executable = E (Command -> Maybe (Executable,String))

> instance Functor Interface

> instance Coalgebra Interface Executable

> instance Final Interface Executable

...but see http://www.cs.swan.ac.uk/~csmichel/articles/finco\_revised.pdf

for the paper that inspired me to start on this last example.

I have to add - at first I thought I'd found something new but it turns out that this is the very first baby step in a whole subfield of computer science. The book "Vicious Circles" by Barwise and Moss also discusses this in a slightly less computational context.

## APPENDIX

All of the examples in one big list. (And we may as well exploit extensions while we have them.)

> data AnyExample = forall a. Show a => AnyExample a

> instance Show AnyExample where

> show (AnyExample a) = show a

> ex = [

> AnyExample ex0,

> AnyExample ex1,

> AnyExample ex2,

> AnyExample ex3,

> AnyExample (take 3 ex4), -- because of that Infinity

> AnyExample ex5,

> AnyExample ex6,

> AnyExample ex7,

> AnyExample ex8,

> AnyExample ex9,

> AnyExample ex10,

> AnyExample ex11

> ]

    module Algebra =

        module Option =

            let case z f = function None -> z | Some x -> f x

            //least fixed point of option

            type Nat = In of Nat option

            let out (In x) = x

            let fold f   = fix (fun r -> f << Option.map r << out)

            //greatest fixed point of option

            type CoNat = CoIn of CoNat option Lazy

            let coout (CoIn x) = Lazy.force x

            let unfold f = fix (fun y -> CoIn << (fun x -> lazy Option.map y x) << f)

        module List =

            //type sum\_of\_prod<'b,'a> = ('a \* 'b) option

            let map f = function

                | Some (a, b) -> Some (a, f b)

                | \_           -> None

            type 'a list = In of ('a \* 'a list) option

            let out (In x) = x

            let fold f = fix (fun y -> f << map y << out)

            type 'a colist = CoIn of ('a \* 'a colist) option Lazy

            let coout (CoIn x) = Lazy.force x

            let unfold f = fix (fun y -> CoIn << (fun x -> lazy map y x) << f)

            let uncurry f (x, y) = f x y

            let compose xs = fold (Option.case id (uncurry (<<))) xs

        module Tree =

            type arms<'b,'a> = Branch of ('b \* 'b) | Leaf of 'a

            let map f = function

                | Branch (l,r) -> Branch (f l, f r)

                | Leaf x       -> Leaf x

            type 'a tree = In of arms<'a tree,'a>

            let out (In x) = x

            type arms\_algebra<'b,'a> =  arms<'b,'a> -> 'b

            let fold f = fix (fun y -> f << map y << out)

        module Stream =

            let undermap f (a, b) = (a, f b)

            type 'a stream = CoIn of ('a \* 'a stream) Lazy

            let force (CoIn x) = Lazy.force x

            let head xs = force xs |> fst

            let tail xs = force xs |> snd

            let cons p = CoIn <| lazy p

            let unfold f = fix (fun y -> CoIn << (fun x -> lazy undermap y x) << f)

            let fork x = (x, x)

            let iterate f = unfold (undermap f << fork)

            let repeat x = iterate id x

            let tails xs = iterate tail xs

            let map f = unfold (fun xs -> f (head xs), tail xs)

            let sap xs = unfold (fun (fs, xs) -> head fs (head xs), (tail fs, tail xs)) xs

            let (=>>) xs f = map f (tails xs)

            let skip n xs = Option.fold (Option.case xs tail) n