Recursion Schemes

by example

Tim Williams

London HUG | 27^{th} March 2013

Introduction

- *Recursion Schemes* are essentially programming patterns
- By structuring our programs in a well-defined way we can:
  - communicate and reason about our programs
  - reuse both code and ideas
  - use a catalogue of theorems to optimise or prove properties
  - identify and exploit opportunities for parallelism
- In this literal Haskell talk, inspired by *Origami progamming* [1], we'll attempt to understand the theory via some practical examples

Overview

- Foldable & Traversable
- Catamorphisms
- Fixed points of Functors
- Composing & Combining Algebras
- Working with fixed data-types
- Anamorphisms & Corecursion
- Hylomorphisms
- Paramorphisms
- Compositional data-types
- Monadic variants
- Apomorphisms
- Memoization
- Zygomorphisms
- Histomorphisms
- Futumorphisms
- Conclusion

Language Pragmas

```haskell
{-# LANGUAGE DeriveFunctor              #-}
{-# LANGUAGE DeriveFoldable             #-}
{-# LANGUAGE DeriveTraversable          #-}
{-# LANGUAGE FlexibleContexts           #-}
{-# LANGUAGE FlexibleInstances          #-}
{-# LANGUAGE StandaloneDeriving         #-}
{-# LANGUAGE UndecidableInstances       #-}
{-# LANGUAGE ScopedTypeVariables        #-}
{-# LANGUAGE ViewPatterns               #-}
{-# LANGUAGE TypeOperators              #-}
{-# LANGUAGE TupleSections              #-}
{-# LANGUAGE RankNTypes                 #-}
{-# LANGUAGE MultiParamTypeClasses      #-}
{-# LANGUAGE FunctionalDependencies     #-}
```

Imports

Haskell platform

```haskell
import Prelude hiding
  (mapM, sequence, replicate, lookup, foldr, length)
import Control.Applicative
  (pure, many, empty, (<$>),(<*>),(<*),(*>),(<|>),(<$))
import Control.Arrow ((&&&),(***),(|||), first, second)
import Control.Monad hiding (mapM, sequence)
import Control.Monad.Reader hiding (mapM, sequence)
import Control.Monad.ST
import Data.Foldable (Foldable)
import qualified Data.Foldable as F
```

```haskell
import Data.List (break)
import Data.Map (Map)
import qualified Data.Map as M
import Data.Set (Set)
import qualified Data.Set as S
import Data.Maybe
import Data.Monoid
import Data.Traversable
import Numeric
```

Third-party Hackage packages

```haskell
import Data.Bool.Extras (bool)
import Data.Hashable
import Data.HashTable.Class (HashTable)
import qualified Data.HashTable.ST.Cuckoo as C
import qualified Data.HashTable.Class as H
import Text.ParserCombinators.Parsec
    hiding (space, many, (<|>))
import Text.PrettyPrint.Leijen
    (Doc, Pretty, (<+>), text, space, pretty)
import qualified Text.PrettyPrint.Leijen as PP
```

Useful functions

- fan-out or *fork* [1]

```
(&&&) :: (b -> c) -> (b -> c') -> b -> (c, c')
(f &&& g) x = (f x, g x)
```

- fan-in [1]

```
(|||) ::: (b -> d) -> (c -> d) -> Either b c -> d
(|||) = either
```

[1] defined more generally in Control.Arrow

- function product [1]

```
(***) :: (b -> c) -> (b' -> c') -> (b, b') -> (c, c')
(f *** g) (x, y) = (f x, g y)
```

- generalised unzip for functors

```
funzip :: Functor f => f (a, b) -> (f a, f b)
funzip = fmap fst &&& fmap snd
```

Foldable

The Foldable class gives you the ability to process the elements of a structure one-at-a-time, discarding the shape.

- Intuitively: list-like fold methods
- Derivable using the `DeriveFoldable` language pragma

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    fold :: Monoid m => t m -> m
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldl :: (a -> b -> a) -> a -> t b -> a
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
```

```haskell
data Tree a = Empty | Leaf a | Node (Tree a) (Tree a)

instance Foldable Tree
  foldMap f Empty        = mempty
  foldMap f (Leaf x)     = f x
  foldMap f (Node l r) = foldMap f l <> foldMap f r

count :: Foldable t => t a -> Int
count = getSum . foldMap (const $ Sum 1)
```

Traversable

Traversable gives you the ability to traverse a structure from left-to-right, performing an effectful action on each element and preserving the shape.

- Intuitively: fmap with effects
- Derivable using the `DeriveTraversable` language pragma
- See *Applicative Programming with Effects*, by McBride and Paterson [2]

```haskell
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f =>
              (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
```

```haskell
instance Traversable Tree where
  traverse f Empty = pure Empty
  traverse f (Leaf x) = Leaf <$> f x
  traverse f (Node k r) =
    Node <$> traverse f l <*> traverse f r
```

Note:

- `mapM` and `sequence` generalize Prelude functions of the same names
- sequence can also be thought of as a generalised matrix transpose!

```haskell
sequence :: Monad m => t (m a) -> m (t a)
sequence = mapM id
```

```haskell
sequence [putStrLn "a", putStrLn "b"] :: IO [()]
```

What if we need to access the structure?

We need to work with a domain of (`f a`) instead of `a`

Catamorphisms

A *catamorphism* (cata meaning “downwards”) is a generalisation of the concept of a fold.

- models the fundamental pattern of (internal) *iteration*
- for a list, it describes processing from the right
- for a tree, it describes a bottom-up traversal, i.e. children first

`foldr` from the Haskell Prelude is a specialised catamorphism:

```
foldr :: (a -> b -> b) -> z -> [a] -> [b]
foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs
```

- We can express the parameters used above in terms of a single *F-algebra* `f b -> b` over a functor `f` and carrier `b`

```haskell
foldr :: (Maybe (a, b) -> b) -> [a] -> b
foldr alg []     = alg $ Nothing
foldr alg (x:xs) = alg $ Just (x, foldr alg xs)
```

- We could also factor out the `List a` to `Maybe (a, [a])` isomorphism

```
foldr :: (Maybe (a, b) -> b) -> [a] -> b
foldr alg = alg . fmap (id *** foldr alg) . unList
  where
    unList []     = Nothing
    unList (x:xs) = Just (x, xs)

length :: [a] -> Int
length = foldr alg where
  alg :: Maybe (a, Int) -> Int
  alg Nothing         = 0
  alg (Just (_, xs)) = xs + 1
```

```
> length "foobar"
6
```

This definition of `foldr` can literally be read from the commutative diagram below.[2]

$$
\begin{tikzcd}
\mathit{Maybe\ (a,\ [a])} \arrow[r, "\text{fmap (id *** foldr alg)}"] & \mathit{Maybe\ (a,\ b)} \arrow[d, "\text{alg}"] \\
\mathit{[a]} \arrow[u, "\text{unList}"] \arrow[r, "\text{foldr alg}"'] & \mathit{b}
\end{tikzcd}
$$

[2]The nodes represent types (objects) and the edges functions (morphisms).

- To demonstrate the expressiveness of foldr, we can even write a left fold using an algebra with a higher-order carrier

```
foldl :: forall a b. (b -> a -> b) -> [a] -> b -> b
foldl f = foldr alg where
  alg :: Maybe (a, b -> b) -> (b -> b)
  alg Nothing        = id
  alg (Just (x,xs)) = \r -> xs (f r x)
```

Fixed points of Functors

An idea from category theory which gives:

- data-type generic functions
- compositional data

Fixed points are represented by the type:

```
-- | the least fixpoint of functor f
newtype Fix f = Fix { unFix :: f (Fix f) }
```

A functor `f` is a data-type of kind `* -> *` together with an `fmap` function.

$$Fix\ f \cong f(f(f(f(f\ldots \text{etc}$$

Data-type generic programming

- allows as to parametrise functions on the structure, or *shape*, of a data-type
- useful for large complex data-types, where boilerplate traversal code often dominates, especially when updating a small subset of constructors
- for recursion schemes, we can capture the pattern as a standalone combinator

Limitations

- The set of data-types that can be represented by means of Fix is limited to *regular* data-types[3]
- Nested data-types and mutually recursive data-types require higher-order approaches[4]

[3]A data-type is regular if it does not contain function spaces and if the type constructor arguments are the same on both sides of the definition.

[4]More specifically, we need to fix higher-order functors.

- In order to work with lists using a data-type generic `cata` combinator, we need a new “unfixed” type representation

```
data ListF a r = C a r | N
```

- `ListF a r` is not an ordinary functor, but we can define a polymorphic functor instance for `ListF a`

```
instance Functor (ListF a) where
  fmap f N        = N
  fmap f (C x xs) = C x (f xs)
```

- we might also want a pattern functor for natural numbers!

```
data NatF r = Succ r | Zero deriving Functor
```

Catamorphisms - revisited

- we would like to write foldr once for all data-types
- category theory shows us how to define it data-type generically for a functor fixed-point

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

Catamorphism

$$
\begin{tikzcd}
f\,(Fix\ f) \arrow[r, "\text{fmap (cata alg)}"] \arrow[d, "\text{Fix}"'] & f\ a \arrow[d, "\text{alg}"] \\
Fix\ f \arrow[r, "\text{cata alg}"'] & a
\end{tikzcd}
$$

The catamorphism-fusion law

The *catamorphism-fusion law* [3], arguably the most important law, can be used to transform the composition of a function with a catamorphism into single catamorphism, eliminating intermediate data structures.

$$h \;.\; f = g \;.\; \mathit{fmap}\; h \implies h \;.\; \mathit{cata}\; f = \mathit{cata}\; g$$

where

```
f :: f a -> a
g :: f b -> b
h :: a -> b
```

Example: a simple expression language

```
data ExprF r = Const Int
             | Var   Id
             | Add   r r
             | Mul   r r
             | IfNeg r r r
              deriving ( Show, Eq, Ord, Functor
                       , Foldable, Traversable )

type Id = String

type Expr = Fix ExprF
```

The *pattern functor* `ExprF` represents the structure of type `Expr`

The isomorphism between a data-type and its pattern functor type is witnessed by the functions `Fix` and `unFix`

We can also conveniently derive instances for fixed functors, although this does require the controversial `UndecidableInstances` extension, amongst others.

```
deriving instance Show (f (Fix f)) => Show (Fix f)
deriving instance Eq (f (Fix f))   => Eq (Fix f)
deriving instance Ord (f (Fix f))  => Ord (Fix f)
```

Example: evaluator with global environment

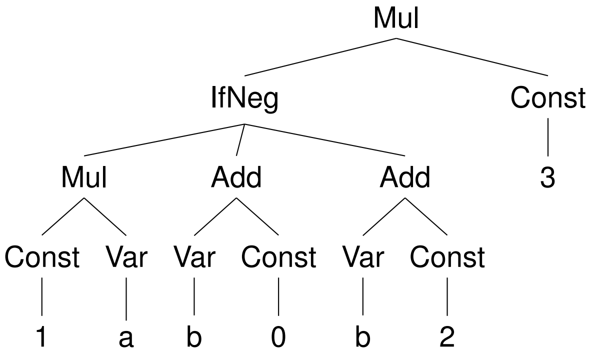
```haskell
type Env = Map Id Int

eval :: Env -> Expr -> Maybe Int
eval env = cata (evalAlg env)

evalAlg :: Env -> ExprF (Maybe Int) -> Maybe Int
evalAlg env = alg where
  alg (Const c)     = pure c
  alg (Var i)       = M.lookup i env
  alg (Add x y)     = (+)  <$> x <*> y
  alg (Mul x y)     = (*)  <$> x <*> y
  alg (IfNeg t x y) = t >>= bool x y . (<0)
```

An example expression

```
e1 = Fix (Mul
           (Fix (IfNeg
                  (Fix (Mul (Fix (Const 1))
                            (Fix (Var "a"))))
                  (Fix (Add (Fix (Var "b"))
                            (Fix (Const 0))))
                  (Fix (Add (Fix (Var "b"))
                            (Fix (Const 2))))))
                (Fix (Const 3)))
```

NB. the `Fix` boilerplate could be removed by defining "smart" constructors.

An example expression



```haskell
testEnv :: Env
testEnv = M.fromList [("a",1),("b",3)]
```

```
> eval testEnv e1
Just 9
```

Example: a pretty printer

```
ppr :: Expr -> Doc
ppr = cata pprAlg

pprAlg :: ExprF Doc -> Doc
pprAlg (Const c)     = text $ show c
pprAlg (Var  i)      = text i
pprAlg (Add x y)     = PP.parens $ x <+> text "+" <+> y
pprAlg (Mul x y)     = PP.parens $ x <+> text "*" <+> y
pprAlg (IfNeg t x y) = PP.parens $ text "ifNeg"   <+> t
                        <+> text "then" <+> x
                        <+> text "else" <+> y
```

```
 > ppr e1
 ((ifNeg (1 * a) then (b + 0) else (b + 2)) * 3)
```

Example: collecting free variables

```
freeVars :: Expr -> Set Id
freeVars = cata alg where
    alg :: ExprF (Set Id) -> Set Id
    alg (Var i) = S.singleton i
    alg e = F.fold e
```

```
> freeVars e1
fromList ["a","b"]
```

Example: substituting variables

```
substitute :: Map Id Expr -> Expr -> Expr
substitute env = cata alg where
  alg :: ExprF Expr -> Expr
  alg e@(Var i) = fromMaybe (Fix e) $ M.lookup i env
  alg e = Fix e
```

```
> let sub = M.fromList [("b",Fix $ Var "a")]
> freeVars $ substitute sub e1
fromList ["a"]
```

Composing Algebras

- It is **not** true in general that catamorphisms compose
- However, there is a very useful special case!

Example: an optimisation pipeline

```
optAdd :: ExprF Expr -> Expr
optAdd (Add (Fix (Const 0)) e) = e
optAdd (Add e (Fix (Const 0))) = e
optAdd e = Fix e

optMul :: ExprF Expr -> Expr
optMul (Mul (Fix (Const 1)) e) = e
optMul (Mul e (Fix (Const 1))) = e
optMul e = Fix e
```

The following composition works, but involves two complete traversals:

```
optimiseSlow :: Expr -> Expr
optimiseSlow = cata optAdd . cata optMul
```

We need an algebra composition operator that gives us *short-cut fusion*:

```
cata f . cata g = cata (f `comp` g)
```

For the special case:

```
f :: f a -> a;  g :: g (Fix f) -> Fix f
```

for arbitrary functors `f` and `g`, this is simply:

```
comp x y = x . unFix . y
```

We can now derive a more efficient optimise pipeline:[5]

```
optimiseFast :: Expr -> Expr
optimiseFast = cata (optMul . unFix . optAdd)
```

We have just applied the *catamorphism compose law* [3], usually stated in the form:

```
f :: f a -> a
h :: g a -> f a

cata f . cata (Fix . h) = cata (f . h)
```

[5] In practice, such a pipeline is likely to be iterated until an equality fixpoint is reached, hence efficiency is important.

Combining Algebras

- Algebras over the same functor but different carrier types can be combined as products, such that two or more catamorphisms are performed as one

Given the following two algebras,

```
f :: f a -> a;  g :: f b -> b
```

we want an algebra of type `f (a, b) -> (a, b)`

- We can use the *banana-split theorem* [3]:

```
cata f &&& cata g =
    cata ( f . fmap fst &&&
           g . fmap snd )
```

- rewrite the product using `funzip`

```
algProd :: Functor f =>
           (f a -> a) -> (f b -> b) ->
           f (a, b) -> (a, b)
algProd f g = (f *** g) . funzip
```

- we can also combine two algebras over different functors but the same carrier type into a coproduct

```
algCoprod :: (f a -> a) -> (g a -> a) ->
             Either (f a) (g a) -> a
algCoprod = (|||)
```

Working with fixed data-types

We can use type classes and functional dependencies to transparently apply the isomorphism between the unfixed representation and the original fixed type, e.g. `[a]` for lists.

```
class Functor f => Fixpoint f t | t -> f where
  inF  :: f t -> t
  outF :: t -> f t

cata :: Fixpoint f t => (f a -> a) -> t -> a
cata alg = alg . fmap (cata alg) . outF
```

Some example `Fixpoint` instances

```
instance Functor f => Fixpoint f (Fix f) where
  inF  = Fix
  outF = unFix

instance Fixpoint (ListF a) [a] where
  inF N        = []
  inF (C x xs) = x : xs
  outF []      = N
  outF (x:xs)  = C x xs

instance Fixpoint NatF Integer where
  inF Zero              = 0
  inF (Succ n)          = n + 1
  outF n | n > 0       = Succ (n - 1)
         | otherwise = Zero
```

Anamorphisms

An *anamorphism* (ana meaning “upwards”) is a generalisation of the concept of an unfold.

- The corecursive dual of catamorphisms
- produces streams and other regular structures from a seed
- `ana` for lists is unfoldr, view patterns help see the duality

```
foldr :: (Maybe (a, b) -> b) -> [a] -> b
foldr f []       = f $ Nothing
foldr f (x : xs) = f $ Just (x, foldr f xs)

unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f (f -> Nothing)                   = []
unfoldr f (f -> Just (x, unfoldr f -> xs)) = x : xs
```

Example: replicate the supplied seed by a given number

```
replicate :: Int -> a -> [a]
replicate n x = unfoldr c n where
  c 0 = Nothing
  c n = Just (x, n-1)
```

```
> replicate 4 '*'
"****"
```

Example: split a list using a predicate

```
linesBy :: (t -> Bool) -> [t] -> [[t]]
linesBy p = unfoldr c where
    c []      = Nothing
    c xs      = Just $ second (drop 1) $ break p xs
```

```
> linesBy (==',') "foo,bar,baz"
["foo","bar","baz"]
```

Example: merging lists

Given two sorted lists, `mergeLists` merges them into one sorted list.

```
mergeLists :: forall a. Ord a => [a] -> [a] -> [a]
mergeLists = curry $ unfoldr c where
  c :: ([a], [a]) -> Maybe (a, ([a], [a]))
  c ([], [])    = Nothing
  c ([], y:ys) = Just (y, ([], ys))
  c (x:xs, []) = Just (x, (xs, []))
  c (x:xs, y:ys) | x <= y = Just (x, (xs, y:ys))
                 | x > y  = Just (y, (x:xs, ys))
```

```
> mergeLists [1,4] [2,3,5]
[1,2,3,4,5]
```

Corecursion

An anamorphism is an example of *corecursion*, the dual of recursion. Corecursion produces (potentially infinite) codata, whereas ordinary recursion consumes (necessarily finite) data.

- Using `cata` or `ana` only, our program is guaranteed to terminate
- However, not every program can be written in terms of just `cata` or `ana`

There is no enforced distinction between data and codata in Haskell, so we can make use of `Fix` again[6]

```haskell
-- | anamorphism
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
```

However, it it often useful to try to enforce this distinction, especially when working with streams.

```haskell
-- | The greatest fixpoint of functor f
newtype Cofix f = Cofix { unCofix :: f (Cofix f) }

-- | an alternative anamorphism typed for codata
ana' :: Functor f => (a -> f a) -> a -> Cofix f
ana' coalg = Cofix . fmap (ana' coalg) . coalg
```

[6] In total functional languages like Agda and Coq, we would be required to make this distinction.

Anamorphism

$$
\begin{tikzcd}
f\ (Cofix\ f) & f\ a \arrow[l, "\text{fmap (ana coalg)}"'] \\
Cofix\ f \arrow[u, "\text{unFix}"] & a \arrow[l, "\text{ana coalg}"'] \arrow[u, "\text{coalg}"']
\end{tikzcd}
$$

Example: coinductive streams

```
data StreamF a r = S a r deriving Show
type Stream a = Cofix (StreamF a)

instance Functor (StreamF a) where
  fmap f (S x xs) = S x (f xs)
```

stream constructor:

```
consS x xs = Cofix (S x xs)
```

stream deconstructors:

```
headS (unCofix -> (S x _ )) = x
tailS (unCofix -> (S _ xs)) = xs
```

- the function `iterateS` generates an infinite stream using the supplied iterator and seed

```
iterateS :: (a -> a) -> a -> Stream a
iterateS f = ana' c where
  c x = S x (f x)

s1 = iterateS (+1) 1
```

```
 > takeS 6 $ s1
 [1,2,3,4,5,6]
```

Hylomorphism

A *hylomorphism* is the composition of a catamorphism and an anamorphism.

- models *general recursion* (!)
- allows us to substitute any recursive control structure with a data structure
- a representation which easily allows us to exploit parallelism

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo g h = cata g . ana h
```

NB. hylomorphisms are **Turing complete**, so we have lost any termination guarantees.

To see the explicit recursion, `cata` and `ana` can be fused together via substitution and the fmap-fusion Functor law:

```
fmap p . fmap q = fmap (p . q)
```

Giving:

```
hylo f g = f . fmap (hylo f g) . g
```

NB. this transformation is the basis for *deforestation*, eliminating intermediate data structures.

- `cata` and `ana` could be defined simply as:

```
cata f = hylo f unFix
ana  g = hylo Fix g
```

Example: Merge sort

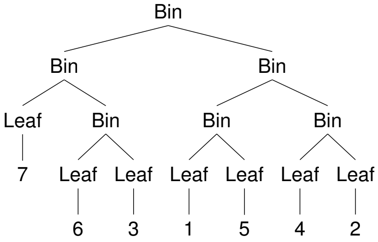
We use a tree data-type to capture the divide-and-conquer pattern of recursion.

```
data LTreeF a r = Leaf a | Bin r r

merge :: Ord a => LTreeF a [a] -> [a]
merge (Leaf x) = [x]
merge (Bin xs ys) = mergeLists xs ys

unflatten [x] = Leaf x
unflatten (half -> (xs, ys)) = Bin xs ys

half xs = splitAt (length xs `div` 2) xs
```

- Finally, we can implement merge-sort as a hylomorphism

```haskell
msort :: Ord a => [a] -> [a]
msort = hylo merge unflatten
```

```
> msort [7,6,3,1,5,4,2]
[1,2,3,4,5,6,7]
```



Paramorphisms

A *paramorphism* (para meaning “beside”) is an extension of the concept of a catamorphism.

- models *primitive recursion* over an inductive type
- a convenient way of getting access to the original input structures
- very useful in practice!

For a pattern functor, a paramorphism is:

```
para :: Fixpoint f t => (f (a, t) -> a ) -> t -> a
para alg = fst . cata (alg &&& Fix . fmap snd)
```

For better efficiency, we can modify the original cata definition:

```
para :: Fixpoint f t => (f (a, t) -> a) -> t -> a
para alg = alg . fmap (para alg &&& id) . outF
```

Example: computing the factorial

- This is the classic example of primitive recursion
- The usual Haskell example `fact n = foldr (*) [1..n]` is actually an unfold followed by a fold

```haskell
fact :: Integer -> Integer
fact = para alg where
  alg Zero          = 1
  alg (Succ (f, n)) = f * (n + 1)
```

$$
\begin{aligned}
0! &= 1 \\
(n+1)! &= n! * (n+1)
\end{aligned}
$$

```
> fact 10
3628800
```

Example: sliding window

```
sliding :: Int -> [a] -> [[a]]
sliding n = para alg where
  alg N               = []
  alg (C x (r, xs)) = take n (x:xs) : r
```

NB. the lookahead via the input argument is left-to-right, whereas the input list is processed from the right.

```
> sliding 3 [1..5]
[[1,2,3],[2,3,4],[3,4,5],[4,5],[5]]
```

Example: collecting all catamorphism sub-results

```
cataTrace :: forall f a.
  (Functor f, Ord (f (Fix f)), Foldable f) =>
  (f a -> a) -> Fix f -> Map (Fix f) a
cataTrace alg = para phi where
  phi :: f (Map (Fix f) a, Fix f) -> Map (Fix f) a
  phi (funzip -> (fm, ft)) = M.insert k v m'
   where
     k  = Fix ft
     v  = alg $ fmap (m' M.!) ft
     m' = F.fold fm
```

```
> let m = cataTrace (evalAlg testEnv) $ optimiseFast e1
> map (first ppr) $ M.toList m
[(2,Just 2),(3,Just 3),(a,Just 1),(b,Just 3),
 ((b + 2),Just 5), ...
```

Compositional Data-types

- “Unfixed” types can be composed in a modular fashion
- explored in the seminar paper *Data types à la carte* [4]

```
-- | The coproduct of pattern functors f and g
data (f :+: g) r = Inl (f r) | Inr (g r)

-- | The product of pattern functors f and g
data (f :*: g) r = (f r) :*: (g r)

-- | The free monad pattern functor
data FreeF f a r = FreeF (f r) | Pure a

-- | The cofree comonad pattern functor
data CofreeF f a r = CofreeF (f r) a
```

Example: Templating

- type-safe templating requires a syntax tree with holes
- ideally we would parse a string template into such a tree, then fill the holes

We use a *free monad* structure `Ctx f a` to represent a node with either a term of type `f` or a hole of type `a`.

```haskell
-- | A Context is a term (f r) which can contain holes a
data CtxF f a r = Term (f r) | Hole a
                  deriving (Show, Functor)

-- | Context fixed-point type. A free monad.
type Ctx f a = Fix (CtxF f a)
```

Fill all the holes of type `a` in the template `Ctx f a` using the supplied function of type `a -> Fix f`

```
fillHoles :: forall f a. Functor f =>
             (a -> Fix f) -> Ctx f a -> Fix f
fillHoles g = cata alg where
  alg :: CtxF f a (Fix f) -> Fix f
  alg (Term t) = Fix t
  alg (Hole a) = g a
```

We will add template variables to JSON by composing data types and parsers.

- we need an "unfixed" JSON datatype and parser (see appendix)

```
pJSValueF :: CharParser () r ->
             CharParser () (JSValueF r)

pJSValue :: CharParser () JSValue
pJSValue = fix $ \p -> Fix <$> pJSValueF p
```

- compose a new `JSTemplate` type

```
type Name = String
type JSTemplate = Ctx JSValueF Name
```

- define a parser for our variable syntax: ${name}

```
pVar :: CharParser () Name
pVar = char '$' *> between (char '{') (char '}')
                           (many alphaNum)
```

- compose the variable parser with the unfixed JSON parser

```
pJSTemplate :: CharParser () (Ctx JSValueF Name)
pJSTemplate = fix $ \p ->
  Fix <$> (Term <$> pJSValueF p <|> Hole <$> pVar)
```

```haskell
temp1 = parse' pJSTemplate "[{\"foo\":${a}}]"
```

```
> temp1
Fix {unFix = Term (
  JSArray [Fix {unFix = Term (
    JSObject [("foo",Fix {unFix = Hole "a"})])}])})
```

```haskell
vlookup :: Ord a => Map a JSValue -> a -> JSValue
vlookup env = fromMaybe (Fix JSNull) . (`M.lookup` env)
```

```
> let env = M.fromList [("a", Fix $ JSNumber 42)]
> fillHoles (vlookup env) temp1
Fix {unFix =
  JSArray [Fix {unFix =
    JSObject [("foo",Fix {unFix = JSNumber 42.0})]}]}
```

Example: Annotating

- useful for storing intermediate values
- inspired by ideas from *attribute grammars*

We use a *cofree comonad* structure `Ann f a` to annotate our nodes of type `f` with attributes of type `a`.

```
-- | Annotate (f r) with attribute a
newtype AnnF f a r = AnnF (f r, a) deriving Functor

-- | Annotated fixed-point type. A cofree comonad.
type Ann f a = Fix (AnnF f a)

-- | Attribute of the root node
attr :: Ann f a -> a
attr (unFix -> AnnF (_, a)) = a
```

```haskell
-- | strip attribute from root
strip :: Ann f a -> f (Ann f a)
strip (unFix -> AnnF (x, _)) = x

-- | strip all attributes
stripAll :: Functor f => Ann f a -> Fix f
stripAll = cata alg where
  alg (AnnF (x, _)) = Fix x

-- | annotation constructor
ann :: (f (Ann f a), a) -> Ann f a
ann = Fix . AnnF

-- | annotation deconstructor
unAnn :: Ann f a -> (f (Ann f a), a)
unAnn (unFix -> AnnF a) = a
```

Synthesized attributes are created in a bottom-up traversal using a catamorphism.

```
synthesize :: forall f a. Functor f =>
              (f a -> a) -> Fix f -> Ann f a
synthesize f = cata alg where
  alg :: f (Ann f a) -> Ann f a
  alg = ann . (id &&& f . fmap attr)
```

For example, annotating each node with the sizes of all subtrees:

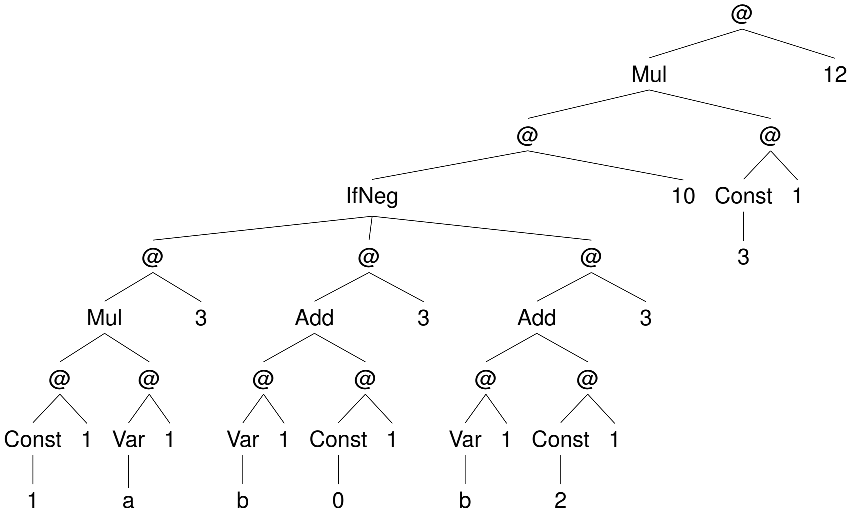
```
sizes :: (Functor f, Foldable f) => Fix f -> Ann f Int
sizes = synthesize $ (+1) . F.sum
```

A pretty-printing catamorphism over such an annotated tree:

```
pprAnn :: Pretty a => Ann ExprF a -> Doc
pprAnn = cata alg where
  alg (AnnF (d, a)) = pprAlg d <+>
                      text "@" <+> pretty a
```

```
> pprAnn $ sizes e1
((ifNeg (1 @ 1 * a @ 1) @ 3
   then (b @ 1 + 0 @ 1) @ 3
   else (b @ 1 + 2 @ 1) @ 3) @ 10
     * 3 @ 1) @ 12
```

annotated with sizes



Inherited attributes are created in a top-down manner from an initial value.

- we can still use a cata/paramorphism by using a higher-order carrier
- the bottom-up traversal happens top-down when the built function is run

```
inherit :: forall f a. Functor f =>
           (Fix f -> a -> a) -> a -> Fix f -> Ann f a
inherit f root n = para alg n root where
  alg :: f (a -> Ann f a, Fix f) -> (a -> Ann f a)
  alg (funzip -> (ff, n)) p = ann (n', a)
    where
      a  = f (Fix n) p
      n' = fmap ($ a) ff
```

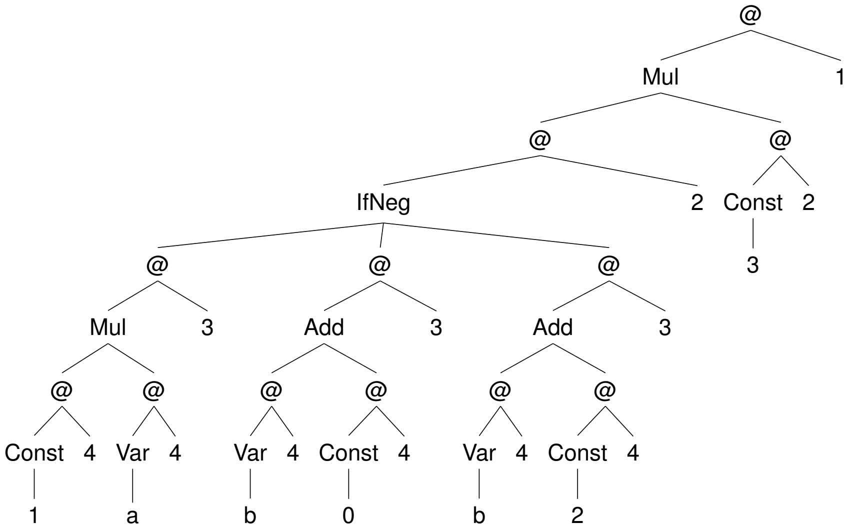
For example, the `depths` function computes the depth of all subtrees:

```
depths :: Functor f => Fix f -> Ann f Int
depths = inherit (const (+1)) 0
```

```
 > pprAnn $ depths e1
((ifNeg (1 @ 4 * a @ 4) @ 3
   then (b @ 4 + 0 @ 4) @ 3
   else (b @ 4 + 2 @ 4) @ 3) @ 2
     * 3 @ 2) @ 1
```

Note that we could combine the `synthesize` and `inherit` algebras and do both in one traversal.

annotated with depths



Monadic variants

A monadic carrier type `m a` gives an algebra `f (m a) -> m a`

This is inconvenient, as we would have to explicitly sequence the embedded monadic values of the argument.

We can define a variant combinator `cataM` that allows us to use an algebra with a monadic codomain only `f a -> m a`

- sequencing is done automatically by using `mapM` instead of `fmap`
- composition with the algebra must now happen in the Kleisli category

```
cataM :: (Monad m, Traversable f) =>
         (f a -> m a) -> Fix f -> m a
cataM algM = algM <=< (mapM (cataM algM) . unFix)
```

Example: eval revisited

- `cataM` simplifies working with a monadic algebra carrier types[7]
- monad transformers can offer much additional functionality, such as error handling

```
eval' :: Env -> Expr -> Maybe Int
eval' env = (`runReaderT` env) . cataM algM where
    algM :: ExprF Int -> ReaderT Env Maybe Int
    algM (Const c)     = return c
    algM (Var i)       = ask >>= lift . M.lookup i
    algM (Add x y)     = return $ x + y
    algM (Mul x y)     = return $ x * y
    algM (IfNeg t x y) = return $ bool x y (t<0)
```

NB. ReaderT would be especially useful for local environments.

[7] compare and contrast the ‘IfNeg‘ clause between eval and eval’

Memoization

- memoization, or caching, lets us trade space for time where necessary
- since we restrict recursion to a library of standard combinators, we can define memoizing variants that can easily be swapped in
- the simplest (pure) memoize function requires some kind of `Enumerable` context

```
memoize :: Enumerable k => (k -> v) -> k -> v
```

- a monadic codomain allows us to use e.g. an underlying State or ST monad

```
memoize :: Memo k v m => (k -> m v) -> k -> m v
memoize f x = lookup x >>= (`maybe` return)
  (f x >>= \r -> insert x r >> return r)

memoFix :: Memo k v m =>
           ((k -> m v) -> k -> m v) -> k -> m v
memoFix f = let mf = memoize (f mf) in mf
```

- runs the memoized computation using a HashTable (see appendix for `Memo` instance)

```
runMemo ::
  (forall s. ReaderT (C.HashTable s k v) (ST s) a) -> a
runMemo m = runST $ H.new >>= runReaderT m
```

- a (transparent) memoizing catamorphism

```
memoCata :: (Eq (f (Fix f)), Traversable f,
            Hashable (Fix f)) =>
            (f a -> a) -> Fix f -> a
memoCata f x = runMemo $
  memoFix (\rec -> fmap f . mapM rec . unFix) x
```

WARNING this could result in a slowdown unless your algebra is significantly more expensive than a hash computation!

Apomorphism

An *apomorphism* (apo meaning “apart”) is the categorical dual of a paramorphism and an extension of the concept of anamorphism (coinduction) [6].

- models *primitive corecursion* over a coinductive type
- allows us to short-circuit the traversal and immediately deliver a result

```
apo :: Fixpoint f t => (a -> f (Either a t)) -> a -> t
apo coa = inF . fmap (apo coa ||| id) . coa
```

- can also be expressed in terms of an anamorphism

```
apo :: Fixpoint f t => (a -> f (Either a t)) -> a -> t
apo coa = ana (coa ||| fmap Right . outF) . Left
```

The function `insertElem` uses an apomorphism to generate a new insertion step when `x>y`, but short-circuits to the final result when `x<=y`

```
insertElem :: forall a. Ord a => ListF a [a] -> [a]
insertElem = apo c where
  c :: ListF a [a] ->
       ListF a (Either (ListF a [a]) [a])
  c N          = N
  c (C x [])  = C x (Left N)
  c (C x (y:xs))
    | x <= y = C x (Right (y:xs))
    | x > y  = C y (Left (C x xs))
```

To implement insertion sort, we simply insert every element of the supplied list into a new list, using `cata`.

```
insertionSort :: Ord a => [a] -> [a]
insertionSort = cata insertElem
```

Zygomorphism

- asymmetric form of mutual iteration, where both a data consumer and an auxiliary function are defined
- a generalisation of paramorphisms

```
algZygo :: Functor f =>
    (f  b      -> b) ->
    (f (a, b) -> a) ->
    f (a, b) -> (a, b)
algZygo f g = g &&& f . fmap snd

zygo :: Functor f =>
        (f b -> b) -> (f (a, b) -> a) -> Fix f -> a
zygo f g = fst . cata (algZygo f g)
```

Example: using evaluation to find discontinuities

The aim is to count the number of live conditionals causing discontinuities due to an arbitrary supplied environment, in a single traversal.

`discontAlg` takes as one of its embedded arguments, the result of evaluating the current term using the environment.

```
discontAlg :: ExprF (Sum Int, Maybe Int) -> Sum Int
discontAlg (IfNeg (t, tv) (x, xv) (y, yv))
  | isJust xv, isJust yv, xv == yv =  t <> x <> y
  | otherwise = maybe (Sum 1 <> t <> x <> y)
                      (bool (t <> y) (t <> x) . (<0)) tv
discontAlg e = F.fold . fmap fst $ e
```

Note that we have to check for redundant live conditionals for which both branches evaluate to the same value.

```haskell
-- | number of live conditionals
disconts :: Env -> Expr -> Int
disconts env = getSum . zygo (evalAlg env) discontAlg
```

- expression `e2` is a function of variables `a` and `b`

```haskell
e2 = Fix (IfNeg (Fix (Var "b")) e1 (Fix (Const 4)))
```

```
>  freeVars e2
fromList ["a","b"]
```

- by supplying `disconts` with a value for `b`, we can look for discontinuities with respect to a new function over just `a`

```
> ppr . optimiseFast $ e2
(ifNeg b
  then
    ((ifNeg a then b else (b + 2)) * 3)
  else
    4)

> disconts (M.fromList [("b",-1)]) e2
1
```

Histomorphism

- introduced by Uustalu & Venu in 1999 [7]
- models *course-of-value recursion* which allows us to use arbitrary previously computed values
- useful for applying dynamic programming techniques to recursive structures

A histomorphism moves bottom-up annotating the tree with results and finally collapses the tree producing the end result.

```
-- | Histomorphism
histo :: Fixpoint f t => (f (Ann f a) -> a) -> t -> a
histo alg = attr . cata (ann . (id &&& alg))
```

Example: computing Fibonacci numbers

```
fib :: Integer -> Integer
fib = histo f where
  f :: NatF (Ann NatF Integer) -> Integer
  f Zero                                          = 0
  f (Succ (unAnn -> (Zero,_)))                    = 1
  f (Succ (unAnn -> (Succ (unAnn -> (_,n)),m)))   = m + n
```

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

```
> fib 100
354224848179261915075
```

Example: filtering by position

The function `evens` takes every second element from the given list.

```
evens :: [a] -> [a]
evens = histo alg where
  alg N                            = []
  alg (C _ (strip -> N     )) = []
  alg (C _ (strip -> C x y)) = x : attr y
```

```
 > evens [1..6]
[2,4,6]
```

Futumorphism

- introduced by Uustalu & Venu in 1999 [7]
- the corecursive dual of the histomorphism
- models *course-of-value* coiteration
- allows us to produce one or more levels

```
futu :: Functor f => (a -> f (Ctx f a)) -> a -> Cofix f
futu coa = ana' ((coa ||| id) . unCtx) . hole

-- | deconstruct values of type Ctx f a
unCtx :: Ctx f a -> Either a (f (Ctx f a))
unCtx c = case unFix c of
  Hole x -> Left x
  Term t -> Right t

term = Fix . Term
hole = Fix . Hole
```

Example: stream processing

The function `exch` pairwise exchanges the elements of any given stream.

```
exch :: Stream a -> Stream a
exch = futu coa where
  coa xs = S (headS $ tailS xs)
             (term $ S (headS xs)
                       (hole $ tailS $ tailS xs))
```

```
> takeS 10 $ exch s1
[2,1,4,3,6,5,8,7,10,9]
```

(1,(2,(3,(4,(5, ... (2,(1,(3,(4,(5, ... (2,(1,(4,(3,(5, ... etc

Conclusion

- catamorphisms, anamorphisms and hylomorphisms (folds, unfolds, and refolds) are fundamental and together capture all recursive computation
- other more exotic recursion schemes are based on the above and just offer more structure
- applying these patterns will help us build more reliable, efficient and parallel programs
- seek to avoid direct explicit recursion wherever possible

Recursion	Corecursion	General
cata	ana	hylo
para	apo	
histo	futu	
zygo		

Table: schemes we discussed in this talk

References

[1] J. Gibbons, “Origami programming.”, The Fun of Programming, Palgrave, 2003.

[2] C. McBride & R. Paterson, “Applicative programming with effects”, Journal of Functional Programming, vol. 18, no. 01, pp. 1-13, 2008.

[3] E. Meijer, “Functional Programming with Bananas , Lenses , Envelopes and Barbed Wire”, 1991.

[4] W. Swierstra, “Data types à la carte”, Journal of Functional Programming, vol. 18, no. 04, pp. 423–436, Mar. 2008.

[5] L. Augusteijn, “Sorting morphisms” pp. 1–23. 3rd International Summer School on Advanced Functional Programming, volume 1608 of LNCS, 1998.

[6] V. Vene, “Functional Programming with Apomorphisms (Corecursion)” pp. 147–161, 1998.

[7] T. Uustalu & V. Venu, “Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically” Informatica, Vol. 10, No. 1, 5–26, 1999.

Appendix

memo monad class and HashTable instance

```
class Monad m => Memo k v m | m -> k, m -> v where
  lookup  :: k -> m (Maybe v)
  insert  :: k -> v -> m ()

-- | HashTable-based Memo monad
instance (Eq k, Hashable k, HashTable h) =>
         Memo k v (ReaderT (h s k v) (ST s)) where
  lookup k   = ask >>= \h -> lift $ H.lookup h k
  insert k v = ask >>= \h -> lift $ H.insert h k v
```

Expr Hashable instance

```haskell
instance Hashable Expr where
  hashWithSalt s = F.foldl hashWithSalt s . unFix

instance Hashable r => Hashable (ExprF r) where
  hashWithSalt s (Const c)     = c
  hashWithSalt s (Var id)      = hashWithSalt s id
  hashWithSalt s (Add x y)     = hashWithSalt s (x, y)
  hashWithSalt s (Mul x y)     = hashWithSalt s (x, y)
  hashWithSalt s (IfNeg t x y) = hashWithSalt s (t, x, y)
```

stream utilities

```
takeS :: Int -> Stream a -> [a]
takeS 0 _                      = []
takeS n (unCofix -> S x xs) = x : takeS (n-1) xs
```

unfixed JSON data-type

```haskell
data JSValueF r
    = JSNull
    | JSBool     Bool
    | JSNumber   Double
    | JSString   String
    | JSArray    [r]
    | JSObject   [(String, r)]
    deriving (Show, Eq, Ord, Functor, Foldable)

type JSValue = Fix JSValueF
```

simple unfixed JSON parser

- Modified from code published in *Real World Haskell*

```
parse' :: CharParser () a -> String -> a
parse' p = either (error . show) id . parse p "(unknown)"

pJSValueF :: CharParser () r ->
             CharParser () (JSValueF r)
pJSValueF r = spaces *> pValue r

pSeries :: Char -> CharParser () r ->
           Char -> CharParser () [r]
pSeries left parser right =
    between (char left <* spaces) (char right) $
            (parser <* spaces) `sepBy`
               (char ',' <* spaces)
```

```
pArray :: CharParser () r -> CharParser () [r]
pArray r = pSeries '[' r ']'

pObject :: CharParser () r -> CharParser () [(String, r)]
pObject r = pSeries '{' pField '}'
    where pField = (,) <$>
              (pString <* char ':' <* spaces) <*> r

pBool :: CharParser () Bool
pBool = True <$ string "true"
     <|> False <$ string "false"
```

```haskell
pValue :: CharParser () r -> CharParser () (JSValueF r)
pValue r = value <* spaces
  where value = choice [ JSString <$> pString
                       , JSNumber <$> pNumber
                       , JSObject <$> pObject r
                       , JSArray <$> pArray r
                       , JSBool <$> pBool
                       , JSNull <$ string "null"
                       ]
                <?> "JSON value"
```

```haskell
pNumber :: CharParser () Double
pNumber = do s <- getInput
             case readSigned readFloat s of
               [(n, s')] -> n <$ setInput s'
               _         -> empty

pString :: CharParser () String
pString = between (char '\"') (char '\"') (many jchar)
    where jchar = char '\\' *> pEscape
                <|> satisfy (`notElem` "\"\\")

pEscape = choice (zipWith decode
                    "bnfrt\\\"/" "\b\n\f\r\t\\\"/")
    where decode c r = r <$ char c
```

LTreeF functor instance

```
instance Functor (LTreeF a) where
  fmap f (Leaf a)     = Leaf a
  fmap f (Bin r1 r2) = Bin (f r1) (f r2)
```

tikz-qtree printer for leaf trees

```
pQtLTree :: Pretty a => Fix (LTreeF a) -> Doc
pQtLTree = (text "\\Tree" <+>) . cata alg where
  alg (Leaf a)  = node ".Leaf"$ pretty a
  alg (Bin l r) = node ".Bin" $ l <+> r
```

a tikz-qtree printer

```
pQt :: Expr -> Doc
pQt = (text "\\Tree" <+>) . cata pQtAlg

pQtAlg :: ExprF Doc -> Doc
pQtAlg (Const c)     = node ".Const" $ text $ show c
pQtAlg (Var id)      = node ".Var"   $ text id
pQtAlg (Add x y)     = node ".Add"   $ x <+> y
pQtAlg (Mul x y)     = node ".Mul"   $ x <+> y
pQtAlg (IfNeg t x y) = node ".IfNeg" $ t <+> x <+> y

node s d = PP.brackets $ text s <+> d PP.<> space
```

tikz-qtree printer for annotated trees

```
pQtAnn :: Pretty a => Ann ExprF a -> Doc
pQtAnn = (text "\\Tree" <+>) . cata alg where
  alg (AnnF (d, a)) = node ".@" $ pQtAlg d <+> pretty a
```