

Refactoring Recursion

Harold Carr

Oracle Labs

introduction

- recursion is a pattern
- there are different patterns of recursion
- “factoring” recursion : benefits
 - code/idea reuse
 - use “proven loops” — less bugs

overview

- explicit recursive functions
- factor recursion out of **functions** with `fold`
- library functions to do recursion
 - folds : apply function to every element
 - unfolds : create structure from seed
 - unfolds followed by folds
 - (un)fold with early exit
 - ...
- factor recursion out of **data** with
 - Foldable, Traversable, Fix

recursion / data

both

corecursion / codata

cata

ana

hylo

para (cata++)

apo (ana++)

histo

futu

zygo/mutu (para++)

**refactoring recursion
out of functions**

explicit recursion

note the pattern

`sumE [] = 0`

`sumE (x:xs) = x + sumE xs`

`andE [] = True`

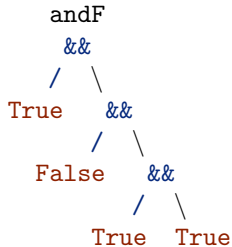
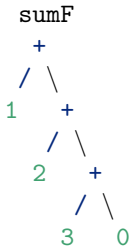
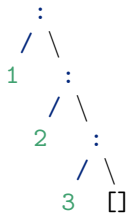
`andE (x:xs) = x && andE xs`

same recursive structure, except

- 0 or True : base case (i.e., empty list)
- + or && : operator in inductive case

factor recursion out of functions with ‘fold’

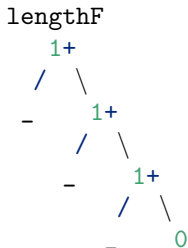
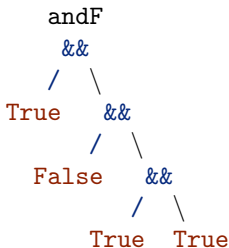
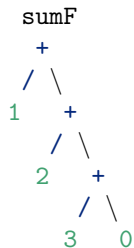
```
sumF  = foldr (+) 0
andF  = foldr (&&) True
```



lengthE [] = 0

lengthE (_:xs) = 1 + lengthE xs

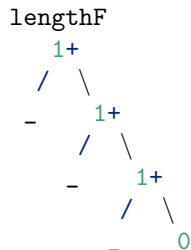
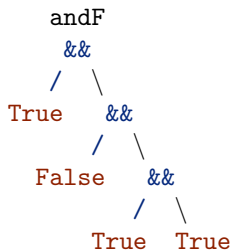
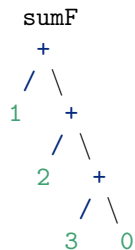
lengthF = foldr (_ n -> 1 + n) 0




```

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

```



cata	catamorphism	fold
ana	anamorphism	unfold
hylo	hylomorphism	ana then cata
para	paramorphism	cata with access to cursor
apo	apomorphism	ana with early exit
histo	histomorphism	cata with access to prev values
futu	futumorphism	ana with access to future values
zygo	zygomorphism	cata with helper function
mutu	mutumorphism	cata with helper function

catamorphism

cata meaning *downwards* : aka fold

- iteration

```
cataL :: (a -> b -> b) -> b -> [a] -> b
cataL f b (a : as) = f a (cataL f b as)
cataL _ b [] = b
```

```
c1 = U.t "c1"
    (cataL (+) 0 [1,2,3])
6
```

```
cd = U.t "cd"  
      (cataL (\a b -> read a + b)  
             0.0  
             ["1.1", "2.2", "3.3"])  
6.6
```

```
filterL  :: (a -> Bool) -> [a] -> [a]
filterL p =
  cataL (\a as -> if p a then a : as
                    else      as)

      []
```

```
filterL' p = cataL alg [] where
  alg a | p a      = (a :)
        | otherwise = id
```

```
c2 = U.tt "c2" [ filterL  odd [1,2,3]
                  , filterL' odd [1,2,3] ]
              [1,3]
```

anamorphism

ana meaning *upwards* : aka unfold

- corecursive dual of catamorphism
- produces structures from a seed
- corecursion produces (infinite?) *codata*
 - (recursion consumes finite *data*)

```
anaL  :: (b -> (a, b)) -> b -> [a]
anaL  f b = let (a, b') = f b in a:anaL f b'
```

```
anaL' :: (b -> Maybe (a, b)) -> b -> [a]
anaL' f b = case f b of
    Just (a, b') -> a:anaL' f b'
    Nothing      -> []
```

```
replicate :: Int -> a -> [a]
replicate n0 x = anaL' coalg n0 where
    coalg 0 = Nothing
    coalg n = Just (x, n-1)

rep = U.t "rep" (replicate 4 '*') "****"
```

```
fibs :: [Integer]
fibs = anaL ff (0, 1)

ff (a, b) = (a, (b, a + b))
```

```
fib = U.tt "fib"
[ fibs
,
    anaL ff ( 0, 1)
, (let (a, b) = ff (0, 1) in a : anaL ff b)
, (0 :
    anaL ff ( 1, 1))
, (0 : 1 :
    anaL ff ( 1, 2))
, (0 : 1 : 2 :
    anaL ff ( 2, 3))
, (0 : 1 : 2 : 3 :
    anaL ff ( 3, 5))
, (0 : 1 : 2 : 3 : 5 :
    anaL ff ( 5, 8))
, (0 : 1 : 2 : 3 : 5 : 8 :
    anaL ff ( 8,13))
, (0 : 1 : 2 : 3 : 5 : 8 : 13 :
    anaL ff (13,21))
]
13
```



```
linesBy :: (t -> Bool) -> [t] -> [[t]]
linesBy p = anaL' c where
  c []    = Nothing
  c xs    = Just (second (drop 1) (break p xs))
```

```
lb = U.t "lb"
    (linesBy (==',') "foo,bar,baz")
    ["foo","bar","baz"]
```

```
break (==',') "foo,bar,baz"
=> ("foo",",bar,baz")
```

```
second (drop 1) ("foo",",bar,baz")
=> ("foo","bar,baz")
```

example: merging lists

given 2 sorted, produce 1 sorted list

```
mergeLists :: Ord a => [a] -> [a] -> [a]
```

```
mergeLists = curry (anaL' c) where
```

```
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 ))
```

```
ml = U.t "ml"
    (mergeLists [1,4] [2,3,5])
    [1,2,3,4,5]
```

ex: coinductive streams

```
-- generates infinite stream
iterateS :: (a -> a) -> a -> [a]
iterateS f = anaL c where
    c x = (x, f x)

sFrom1 :: [Integer]
sFrom1 = iterateS (+1) 1

tsf = U.t "tsf"
      (take 6 sFrom1)
      [1,2,3,4,5,6]
```

hylomorphism

composition of catamorphism and anamorphism

- corecursive codata production
- followed by recursive data consumption

```
hyloL :: (a -> c -> c)           -- cata algebra
      -> c                         -- cata zero
      -> (b -> Maybe (a, b))      -- ana coalgebra
      -> b                         -- ana seed
      -> c                         -- result

hyloL alg c coalg = cataL alg c . anaL' coalg
```

```
fact :: Integer -> Integer
fact b0 = hyloL alg 1 coalg b0 where
    coalg 0 = Nothing
    coalg b = Just (b, b - 1)
    alg      = (*)

hf = U.t "hf" (fact 5) 120
```

fusion / deforestation

```
hyloL alg c coalg = cataL alg c . anaL' coalg
```

```
hyloL' alg c coalg b0 = h b0 where
```

```
  h b = case coalg b of
```

```
    Nothing      -> c
```

```
    Just (a', b') -> alg a' (h b')
```

paramorphism

para meaning *beside* (or “parallel with”)

extension of catamorphism

- given each element, and
- current cursor in iteration (e.g., current tail)

```
paraL  :: (a -> [a] -> b -> b) -- f
        -> b                    -- zero
        -> [a]                 -- input
        -> b                    -- output

paraL  f b  (a : as) = f a as (paraL f b as)
paraL  _ b  []      = b
```

```
tails :: [a] -> [[a]]
tails = paraL (\_ as b -> as:b) []

p1 = U.t "p1"
    (tails [1,2,3,4])
    [[2,3,4],[3,4],[4],[]]
```



```
slide :: Int -> [a] -> [[a]]
slide n = paraL alg [] where
    alg _ [] b = b
    alg a as b | length (a:as) < n = b
                | otherwise = take n (a:as) : b
```

```
sl = U.t "sl"
    (slide 3 [1..5])
    [[1,2,3],[2,3,4],[3,4,5]]
```

apomorphism

apo meaning *apart*

- dual of paramorphism
- extension of anamorphism
- enables short-circuiting traversal

```
apoL :: ([b] -> Maybe (a, Either [b] [a]))  
      -> [b]  
      -> [a]
```

```
apoL f bs = case f bs of  
  Nothing -> []  
  Just (a, Left  bs') -> a : apoL f bs'  
  Just (a, Right as)  -> a : as
```

```
anaL' :: (b -> Maybe (a, b)) -> b -> [a]
anaL' f b = case f b of
    Just (a, b')      -> a : anaL' f b'
    Nothing            -> []
```

```
apoL :: ([b] -> Maybe (a, Either [b] [a]))
      -> [b]
      -> [a]
apoL f bs = case f bs of
    Nothing -> []
    Just (a, Left  bs') -> a : apoL f bs'
    Just (a, Right as)  -> a : as
```

- short-circuits to final result when $x \leq y$

```
insertElemL :: Ord a => a -> [a] -> [a]
insertElemL a as = apoL c (a:as) where
    c (x:y:xs) | x<=y = Just (x, Right (y:xs))
                | x> y  = Just (y, Left  (x:xs))
    c      [x]         = Just (x, Left  [])
    c      []          = Nothing
```

```
iel = U.t "iel"
      (insertElemL 3 [1,2,5])
      [1,2,3,5]
```

zygomorphism

- generalisation of paramorphism
- fold that depends on result of another fold
 - on each iteration of fold
 - f sees its answer from previous iteration
 - g sees both answers from previous iteration
 - fused into one traversal

```
zygoL :: (a -> b -> b)           -- f
        -> (a -> b -> c -> c)    -- g depends on f result
        -> b -> c                -- zeroes
        -> [a]                  -- input
        -> c                    -- result

zygoL f g b0 c0 =
    snd . cataL (\a (b, c) -> (f a b, g a b c))
            (b0, c0)
```

```
pmL :: [Int] -> [Int]
pmL = zygoL (\_ b    -> not b)
          (\a b c -> pm b a c)
          False
          [45]
  where pm b a c = (if b then -a else a) : c

zpm = U.t "zpm"
      (pmL [ 1,2, 3,4, 5,6])
      [-1,2,-3,4,-5,6,45]
```

histomorphism

- gives access to previously computed values
- moves bottom-up annotating stack with results

```

data History a b
  = End b | Step a b (History a b)
  deriving (Eq, Read, Show)

history :: (a -> History a b -> b)
        -> b -> [a] -> History a b
history f b = cataL (\a h -> Step a (f a h)h)
                  (End b)

thistory = U.t "thistory"
  (history (\a h -> (a, show h)) (0, "") [1, 2, 3])
  (Step 1 (1, "Step 2 (2, \"Step 3 (3, \\\\\"End (0
    (Step 2 (2, "Step 3 (3, \"End (0, \\\\\"\\\\\\")\\
      (Step 3 (3, "End (0, \"\\")")
        (End (0, ""))))))

```



```
histoL :: (a -> History a b -> b)
         -> b -> [a] -> b
histoL f b = valH . history f b
```

```
valH :: History a b -> b
valH (Step _ b _) = b
valH (End      b)  = b
```

```
prevH :: History a b -> History a b
prevH (Step _ _ h) = h
prevH h@(End      _) = h
```

```
fibHL :: Integer -> History Integer Integer
fibHL n = history f 1 [3..n] where
    f _ h = valH h + valH (prevH h)
```

```
tfibHL = U.t "tfibHL"
    (fibHL 8)
    (Step 3 21
      (Step 4 13
        (Step 5 8
          (Step 6 5
            (Step 7 3
              (Step 8 2
                (End 1))))))))))
```

futumorphism

- corecursive dual of histomorphism
 - histo : access to previously-computed values
 - futu : access to future values

```
futuL :: (a -> Maybe (b, ([b], Maybe a)))  
      -> a  
      -> [b]
```

```
futuL f a =  
  case f a of  
    Nothing          -> []  
    Just (b, (bs, ma)) -> b : (bs ++ futuBs)  
    where futuBs = case ma of  
                      Nothing -> []  
                      Just a'  -> futuL f a'
```

```
exchL = futuL coa where
  coa xs = Just ( head (tail xs)
                  , ( [head xs]
                      , Just (tail (tail xs))
                      )
                )
    )
```

```
exs1 = U.t "exs1"
      (take 10 (exchL sFrom1))
      [2,1,4,3,6,5,8,7,10,9]
```

```
exs2 = U.t "exs2"
      (take 9 (exchL sFrom1))
      [2,1,4,3,6,5,8,7,10]
```

**refactoring recursion
out of data**

```
data Tree1 a
  = Leaf a
  | Bin (Tree1 a) (Tree1 a)
  deriving (Foldable)

ext1 = Bin (Bin (Leaf "1") (Leaf "2"))
         (Bin (Leaf "3") (Leaf "4"))

et1 = U.t "et1"
     (foldr (++) "" ext1)
     "1234"
```

```
data LTreeF a r
  = LeafF a
  | BinF r r
  deriving (Functor)
```

```
type Tree a      = Fix (LTreeF a)
```

```
leaf a  = Fix (LeafF a)
```

```
bin l r = Fix (BinF l r)
```

```
ext = bin (bin (leaf "1") (leaf "2"))
         (bin (leaf "3") (leaf "4"))
```

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

```
sumT = cata alg where
  alg (LeafF a) = a
  alg (BinF l r) = l ++ r
```

```
et = U.t "et"
    (sumT ext)
    "1234"
```


references

Tim Williams' recursion schemes presentation

- <http://www.timphilipwilliams.com/slides.html>
- <https://www.youtube.com/watch?v=Zw9KeP30zpU>

Functional Programming with Bananas, Lenses,
Envelopes and Barbed Wire

- <https://maartenfokkinga.github.io/utwente/mmf91m.pdf>

These slides

- <https://github.com/haroldcarr/presentations/>