

# **Refactoring Recursion**

Harold Carr

# introduction

- Recursion is a pattern
- There are different patterns of recursion
- “factoring” recursion : benefits
  - code/idea reuse
  - use “proven loops” — less bugs
  - use catalogue of theorems to optimise/prove properties

# overview

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

**refactoring recursion  
out of functions**



**recursion / data**

**both**

**corecursion / codata**

---

cata

ana

hylo

para (cata++)

apo (ana++)

histo

futu

zygo/mutu (para++)

# explicit recursion

note the pattern

`sumE [] = 0`

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

`andE [] = True`

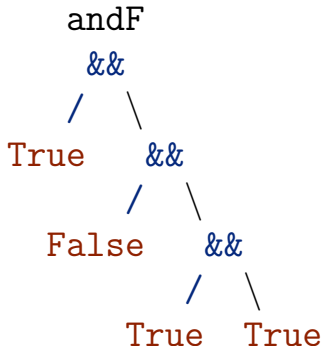
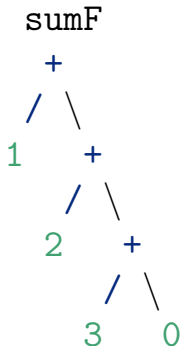
`andE (x:xs) = x && andE xs`

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
```

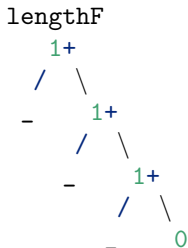
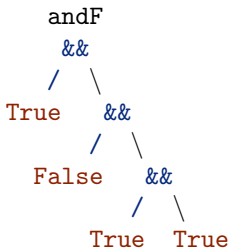
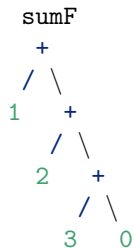




lengthE [] = 0

lengthE (\_:xs) = 1 + lengthE xs

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

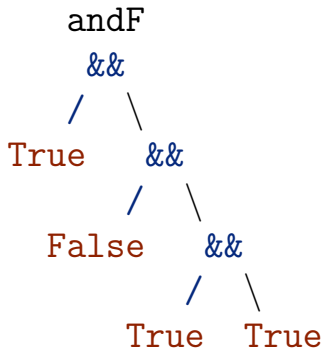
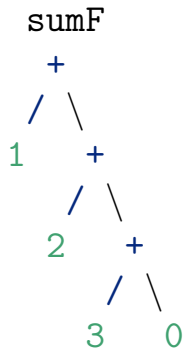


## 'foldr' operation

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr f z [] = z`

`foldr f z (x:xs) = f x (foldr f z xs)`



cata	catamorphism	folds
ana	anamorphisms	unfolds
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

# catamorphisms

*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 (\x acc -> if p x then x : acc
                      else acc)
      []
```

```
filterL' :: (a -> Bool) -> [a] -> [a]
filterL' p = cataL alg [] where
  alg x | p x      = (x :)
        | 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 catamorphisms
- corecursion produces (infinite?) *codata*
- recursion consumes (finite) *data*
- produces structures from a seed

```
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 (\(a, b) -> (a, (b, a + b)))
        (0, 1)

fib = U.t "fib" (fibs !! 7) 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]
```

# example: 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 f
      -> c                         -- cata zero
      -> (b -> Maybe (a, b))      -- ana g
      -> b                         -- ana seed
      -> c                         -- result

hyloL f c g = cataL f c . anaL' g
```

```
fact :: Integer -> Integer
fact n0 = hyloL' a 1 c n0 where
  c 0 = Nothing
  c n = Just (n, n - 1)
  a    = (*)

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

## fusion/deforestation

```
hyloL f c g b = cataL f c . anaL' g b
```

```
hyloL' f a g b0 = h b0 where
```

```
  h b = case g b of
```

```
    Nothing      -> a
```

```
    Just (a', b') -> f 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
```

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

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

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  $f$ 
        -> 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)
          True
          []
  where pm b a c = (if b then -a else a) : c

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

```

pmL' :: [Int] -> [Int]
pmL' = zygoL (\_ b -> b + 1)
          (\a b c -> pm (b `mod` 3 == 0) a c)
          (-1)
          []
  where pm b a c = (if b then -a else a) : c

zpm' = U.t "zpm'"
      (pmL' [1,2,3,4,5,6,7])
      [1,2,-3,4,5,-6,7]

```



# 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)
```

```
valH (End b) = b
```

```
valH (Step _ b _) = b
```

```
histoL :: (a -> History a b -> b)
        -> b -> [a] -> b
```

```
histoL f b = valH . history f b
```

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

```
prevH (Step _ _ h) = h
prevH z               = z
```

```
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=Zw9KeP3OzpU>

Functional Programming with Bananas, Lenses,  
Envelopes and Barbed Wire -

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

These slides -

<https://github.com/haroldcarr/presentations/blob/master/2017-05-27-lambdaconf-recursion-schemes.pdf> -

<https://github.com/haroldcarr/presentations/blob/master/2017-05-27-lambdaconf-recursion-schemes.pdf>

---

---