LECTURE 02

Theory and Design of PL (CS 538)

January 29, 2020

RECURSION

WHY RECURSION?

- Primary way for functional PLs to do iteration
- Natural way to repeat functions, no counters
- May take a bit of getting used to

SIMPLE, "DIRECT" RECURSION

- All functions in Haskell can be recursive

```
plusN :: Int -> Int
plusN 0 = 0
plusN n = n + plusN (n - 1)
```

- Also works in let-definitions

```
myFun :: Int -> Int
myFun n = let prodN 0 = 0
              prodN 1 = 1
              prodN n = n * prodN (n - 1) in
          42 + prodN n
```

BAD EXAMPLE: LOOPING

- Can write non-terminating functions!
- Try not to do this:

```haskell
loopForever :: Int -> Int
loopForever x = loopForever x
-- loopForever 42 = ???
```

“ACCUMULATING” RECURSION

TAIL RECURSION

- If last step is recursive call, optimized to loop (faster)

```
slowSumTo 0 = 0
slowSumTo n = n + slowSumTo (n - 1)
-- After recursive call returns, need to add n

fastSumTo n = helper 0 n
  where helper total 0 = total
        helper total n = helper (total + n) (n - 1)
-- After recursive call returns, can just return
```

- In this example: direct is slower than accumulating
  - Note: this is not always true!

RECURSION EXAMPLES

RECALL: LISTS

- Constructing lists, all elements of same type:

```
myEmptyList    = []                          -- empty list
myNonEmptyList = 1 : 2 : myEmptyList -- [1, 2]

-- singleton list: list with one element
mySingleton = [42]                           -- list with just 42

-- cons operation: add an element to the front of a list
(:) :: a -> [a] -> [a]

-- appending lists: glue two lists together
(++) :: [a] -> [a] -> [a]
myAppList = mySingleton ++ myNonEmptyList -- [42, 1, 2]
```

LENGTH OF A LIST

- With direct recursion:

```
lengthList :: [a] -> Int
lengthList []     = 0
lengthList (x:xs) = 1 + lengthList xs
```

- Accumulating recursion:

```
lengthList :: [a] -> Int
lengthList as = length' 0 as
    where length' acc []     = acc
          length' acc (x:xs) = length' (acc + 1) xs
```

PRODUCT OF A LIST

- With direct recursion:

```
prodList :: [Int] -> Int
prodList []      = 1
prodList (x:xs) = x * prodList xs
```

- Accumulating recursion:

```
prodList :: [Int] -> Int
prodList = prod' 1
    where prod' acc []     = acc
          prod' acc (x:xs) = prod' (x * acc) xs
-- SAME AS: prodList ls = prod' 1 ls ...
```

CHECKING ALL TRUE/FALSE

- Checking if list is all true/false

```
allTrue    :: [Bool] -> Bool
existsTrue :: [Bool] -> Bool

-- Direct recursion
allTrue []     = True
allTrue (x:xs) = x && allTrue xs

-- Accumulating recursion
existsTrue bs = exists' False bs
    where exists' acc []     = acc
          exists' acc (x:xs) = exists' (acc || x) xs
```

PATTERN: MAPPING

- Add 42 to each element of a list of integers (direct)

```
add42 :: [Int] -> [Int]
add42 []     = []
add42 (x:xs) = (x + 42) : add42 xs
```

PATTERN: MAPPING

- Flip all elements of a list of booleans (accumulating)

```haskell
flipBool :: [Bool] -> [Bool]
flipBool bs = flip' [] bs
    where flip' acc []     = acc
          flip' acc (x:xs) = flip' (acc ++ [not x]) xs
```

PATTERN: FILTERING

- Keep only even elements (direct)

```
keepEvens :: [Int] -> [Int]
keepEvens []     = []
keepEvens (x:xs) = if (even x)
                     then x : keepEvens xs
                     else keepEvens xs
```

PATTERN: FILTERING

- Keep only positive numbers (accumulating)

```
keepPos :: [Float] -> [Float]
keepPos fs = keep' [] fs
    where keep' acc []     = acc
          keep' acc (x:xs) = if x > 0
                             then keep' (acc ++ [x]) xs
                             else keep' acc xs
```

EXAMPLE: SORTING

- Sort list of distinct numbers in increasing order

```
sortNums :: [Int] -> [Int]
sortNums []     = []
sortNums (x:xs) = lesser ++ [x] ++ greater
  where lesser  = sortNums (filter (< x) xs)
        greater = sortNums (filter (> x) xs)
```

- Note: branching, hard to write as accumulating

PATTERN: ZIPPING

- Pair up two lists into a list of pairs (direct)

```
zip :: [a] -> [b] -> [(a, b)]
zip [] _            = []
zip _ []            = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys

list  = [1, 2, 3]
list' = [4, 5, 6]

paired = zip list list'
-- paired = [(1, 4), (2, 5), (3, 6)]
```

- Can you define this function in accumulating style?

HIGHER-ORDER FUNCTIONS

"FIRST-CLASS" FUNCTIONS

- Functions are like any other expression
- Can be passed as arguments to functions
- Can be returned from other functions

PATTERN: MAPPING

- Apply function to each element of list:

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

- Earlier examples as special case:

```
add42 = map (\x -> x + 42)
-- SAME AS: add42 list = map (\x -> x + 42) list

flipBool = map (\x -> not x)
-- SAME AS: flipBool list = map (\x -> not x) list
```

PATTERN: FILTERING

- Keep only list elements satisfying some condition

```
filter :: (a -> Bool) -> [a] -> [a]
filter cond []     = []
filter cond (x:xs) = if (cond x)
                       then x : filter cond xs
                       else filter cond xs
```

- Earlier examples as special cases

```
keepEvens = filter (\x -> even x)
keepPos = filter (\x -> x > 0)
```

DIRECT RECURSION, AGAIN

```haskell
lengthList :: [a] -> Int
lengthList []     = 0
lengthList (x:xs) = 1 + lengthList xs

add42 :: [Int] -> [Int]
add42 []     = []
add42 (x:xs) = (x + 42) : add42 xs
```

- Do these look similar? Common pieces:
  1. Combining function (add 1, add 42 and cons)
  2. Initial value (0, empty list)
  3. List to process

PATTERN: FOLDING RIGHT

- Direct recursion is an example of a *right fold*:

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

- `a` is type of item, `b` is type of result

```
lengthList as = foldr (\_ len -> 1 + len) 0 as
add42 ls = foldr (\x acc -> (x + 42) : acc) [] ls
```

ACCUMULATING RECURSION, AGAIN

```
prodList :: [Int] -> Int
prodList ls = prod' 1 ls
    where prod' acc []     = acc
          prod' acc (x:xs) = prod' (x * acc) xs

flipBool :: [Bool] -> [Bool]
flipBool bs = flip' [] bs
    where flip' acc []     = acc
          flip' acc (x:xs) = flip' (acc ++ [not x]) xs
```

- Do these look a bit similar? They should...

PATTERN: FOLDING LEFT

- Accumulating recursion is an example of a *left fold*:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f init []     = init
foldl f init (x:xs) = foldl f (f init x) xs
```

- `a` is type of accumulator/result, `b` is type of item

```
prodList ls = foldl (\acc x -> x * acc) 1 ls
flipBool bs = foldl (\acc x -> acc ++ [not x]) [] bs
```

MORE FOLDING

- Checking membership of an element

```
elemOf :: Int -> [Int] -> Bool
elemOf i []       = False
elemOf i (x:xs) = (i == x) || i `elemOf` xs
```

- Direct recursion, can use a right fold:

```
elemOf' i = foldr (\x acc -> (i == x) || acc) False
```

MORE FOLDING

- Reversing a list of elements

```
reverse :: [a] -> [a]
reverse = helper []
  where helper acc []     = acc
        helper acc (x:xs) = helper (x:acc) xs
```

- Accumulating recursion: left fold

```
reverse' = foldl (\acc x -> x : acc) []
```