Introduction to Functional Programming with Haskell - RCOS Presentation

Avi Weinstock

March 6, 2015

What characterizes Functional Programming?

- Abstraction of common patterns via higher order functions
- A mathematical use of the term "function" (as a pure mapping from input to output), as opposed to the concept of procedures (e.g. in C)

What is Haskell?

A general-purpose programming language that:

- Encourages a functional style of programming
- Enables very short, readable code
- Has static typechecking, with type inference
- Compiles to native code, in the same efficiency class as Java/C# (can be made as performant as Assembly/C/C++, with some effort)

Hello world

- ```
  main = putStrLn "Hello, world!"
  ```
- ```
  main = do
      putStr "Enter a string: "
      str <- getLine
      putStrLn ("Echo: " ++ str)
  ```

Examples with numbers

- ```
  factorial n = product [1..n]
  ```
- ```
  dotProduct xs ys = sum $ zipWith (*) xs ys
  dotProduct' xs ys = sum (zipWith (*) xs ys) -- equivalent definition
  magnitude xs = sqrt $ dotProduct xs xs
  ```
- ```
  divides n i = (n `mod` i) == 0
  isPrime n = not $ any (divides n) [1..n]
  isPrime' n = not (any (divides n) [1..n]) -- equivalent definition
  ```
  ```

Examples with numbers (continued)

- ```
  fibonaccis = 1 : 1 : zipWith (+) fibonaccis (tail fibonaccis)
  ```
- ```
  approxDerivative epsilon f x = (f (x+epsilon) - f x) / epsilon
  ```
- ```
  stars n = replicate (floor n) '*'
  showLines = putStr . unlines
  prepend = zipWith ((++) . show)
  plot f xs = showLines $ prepend xs (map (stars . f) xs)
  ```
  ```

Execution of examples

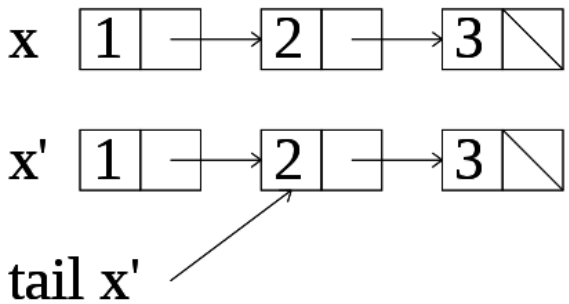
```
ghci> :load haskell_lecture_numberslide.hs
[1 of 1] Compiling Main             ( haskell_lecture_numberslide.hs, interpreted )
Ok, modules loaded: Main.
ghci> :browse
factorial :: (Enum a, Num a) => a -> a
dotProduct :: Num a => [a] -> [a] -> a
magnitude :: Floating a => [a] -> a
divides :: Integral a => a -> a -> Bool
isPrime :: Integral a => a -> Bool
fibonaccis :: Num a => [a]
approxDerivative :: Fractional a => a -> (a -> a) -> a -> a
stars :: RealFrac a => a -> [Char]
showLines :: [String] -> IO ()
prepend :: Show a => [a] -> [[Char]] -> [[Char]]
plot :: (RealFrac b, Show a) => (a -> b) -> [a] -> IO ()
ghci> map factorial [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]
ghci> magnitude [1,1]
1.4142135623730951
ghci> take 10 $ filter isPrime [2..]
[2,3,5,7,11,13,17,19,23,29]
ghci> take 10 fibonaccis
[1,1,2,3,5,8,13,21,34,55]
ghci> plot (^2) [0..7]
0.0
1.0*
2.0****
3.0*********
4.0*****************
5.0*************************
6.0*********************************
7.0*****************************************
ghci> plot (approxDerivative 0.01 (^2)) [0..7]
0.0
1.0**
2.0****
3.0******
4.0********
5.0**********
6.0************
7.0**************
ghci>
```

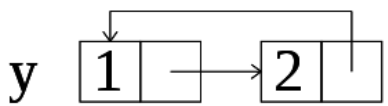
Functions used in previous examples

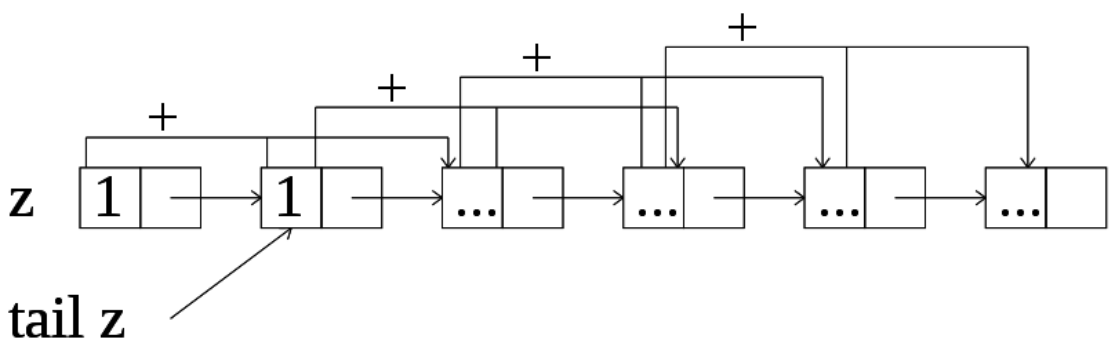
```
($) :: (a -> b) -> a -> b
f $ x = f x

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

not :: Bool -> Bool
any, all :: (a -> Bool) -> [a] -> Bool

zip :: [a] -> [b] -> [(a, b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

sum, product :: Num a => [a] -> a

replicate :: Int -> a -> [a]

putStr, putStrLn :: String -> IO ()
```

Linked lists - Boxes and Pointers!

```
x = [1,2,3]
x' = 1:2:3:[]
y = 1 : 2 : y
z = 1 : 1 : zipWith (+) z (tail z)
```

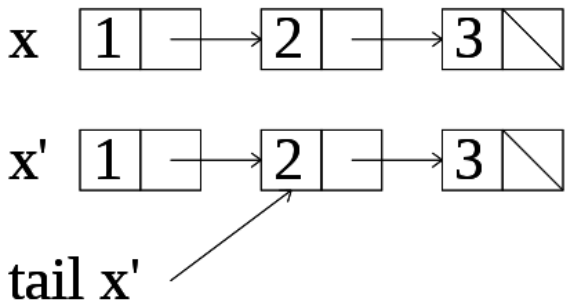


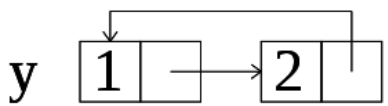


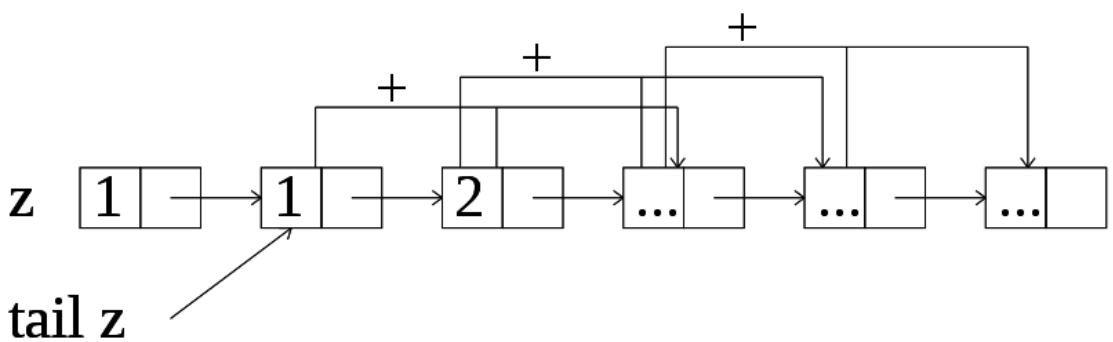


Linked lists - Boxes and Pointers!

```
x = [1,2,3]
x' = 1:2:3:[]
y = 1 : 2 : y
z = 1 : 1 : zipWith (+) z (tail z)
```

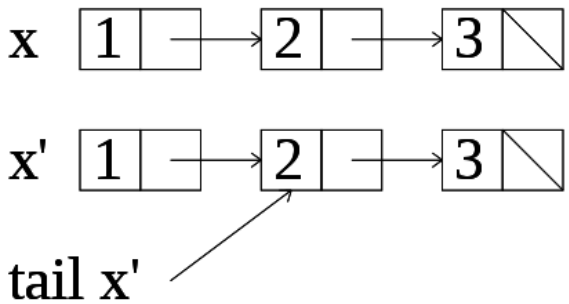


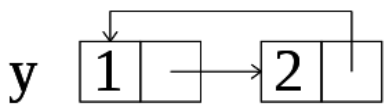


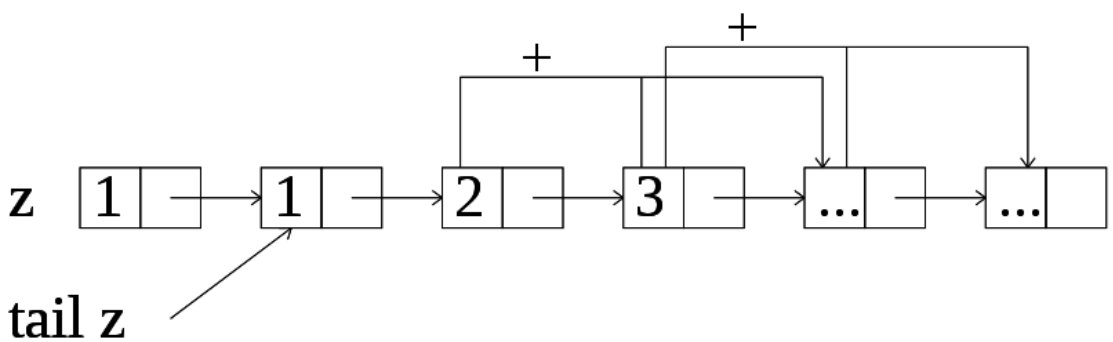


Linked lists - Boxes and Pointers!

```
x = [1,2,3]
x' = 1:2:3:[]
y = 1 : 2 : y
z = 1 : 1 : zipWith (+) z (tail z)
```

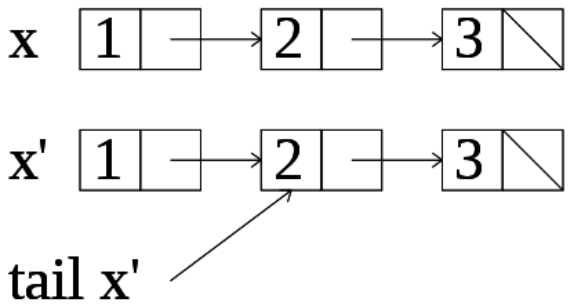


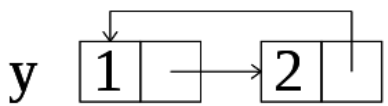


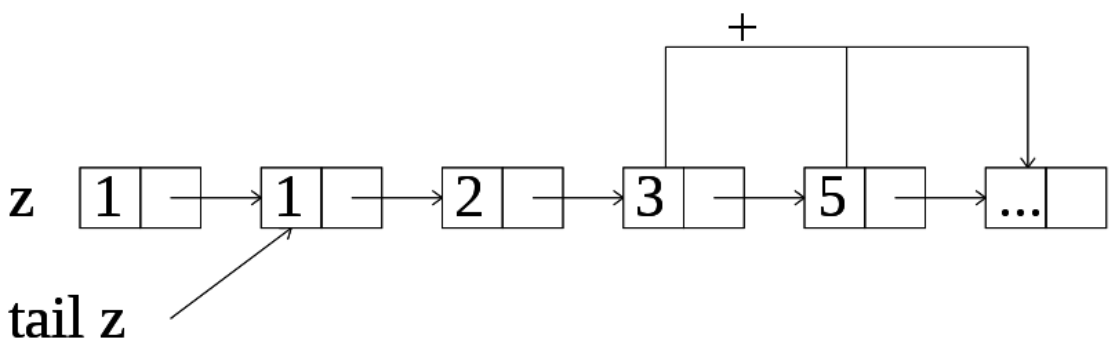


Linked lists - Boxes and Pointers!

```
x = [1,2,3]
x' = 1:2:3:[]
y = 1 : 2 : y
z = 1 : 1 : zipWith (+) z (tail z)
```

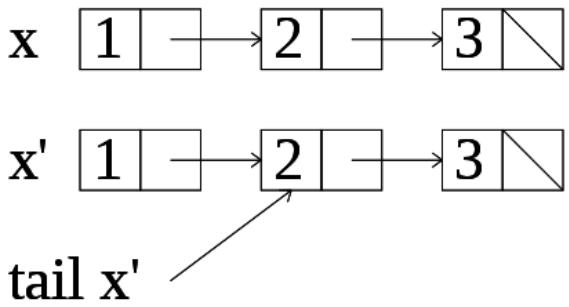


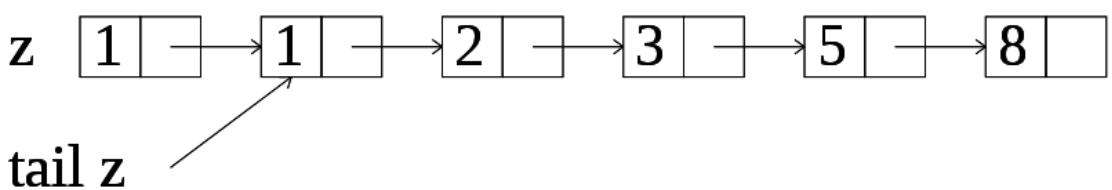




Linked lists - Boxes and Pointers!

```
x = [1,2,3]
x' = 1:2:3:[]
y = 1 : 2 : y
z = 1 : 1 : zipWith (+) z (tail z)
```





Map and Reduce

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

map' f = foldr ((:) . f) [] -- equivalent definition

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

`map` applies its function to every element of a list.
`foldr` uses the provided function to reduce/aggregate the list into some value, starting from a seed.
Folds are very general, and can be used to implement many types of loops.
Definition of `foldl` in Python:

```
def foldl(f, init, xs):
    acc = init
    for x in xs:
        acc = f(acc, init)
    return acc
```

Reduce (continued)

More definitions of things in terms of `foldr`:

```
map f = foldr ((:) . f) []

sum = foldr (+) 0

product = foldr (*) 1

concat = foldr (++) []

all p = foldr ((&&) . p) True

any p = foldr ((||) . p) False

average = uncurry (/) . foldr (\x (s, t) -> (s+x, t+1)) (0, 0)
```

Typeclasses

With this definition:

```
instance Num a => Num [a] where
    (+) = zipWith (+)
    (*) = zipWith (*)
    (-) = zipWith (-)
    negate = map negate
    abs = map abs
    signum = map signum
    fromInteger = cycle . return . fromInteger
```

Expressions like these become valid:

```
1 + [1,2,3]
2 * [2..10]
-[6,2,8]
abs [-5..5]
```

Currying, Sections, and Infix

The following are all equivalent (via currying):

```
f g x y = zipWith g x y
f g x = zipWith g x
f g = zipWith g
f = zipWith
f g x = \y -> zipWith g x y
f g = \x y -> zipWith g x y
f = \g x y -> zipWith g x y
```

These are equivalent triples (via sections, and infix):

```
g x = x + 1
g = (+1)
g x = (+) x 1
```

```
h x = x `mod` 2
h = (`mod` 2)
h x = mod x 2
```

```
i x = 2 `mod` x
i = (2 `mod`)
i x = mod 2 x
```

Note that `h` $\neq$ `i`, since `mod` is not commutative.

TIMTOWTDI - There's More Than One Way To Do It

Functions introduced:

```
ord :: Char -> Int
chr :: Int -> Char

interact :: (String -> String) -> IO ()

forever :: Monad m => m a -> m b
```

```
import Control.Monad
import Data.Char

caesarCipher shift = map (chr . (`mod` 256) . (+ shift) . ord)

main1 = forever $ do
    plaintext <- getLine
    let ciphertext = caesarCipher 3 plaintext
    putStrLn (caesarCipher 3 plaintext)

main2 = forever (getLine >>= (putStrLn . caesarCipher 3))

main3 = interact (caesarCipher 3)
```

Datatype declarations

```
data TrafficLight = RedLight | YellowLight | GreenLight
```

```
import qualified Data.Foldable as F

data List a = Node a (List a) | Empty

instance Foldable List where
    foldr f acc Empty = acc
    foldr f acc (Node x xs) = f x (F.foldr f acc xs)
```

```
import qualified Data.Map as M

data JSVal =
    JSUndefined
    | JSNull
    | JSBoolean Bool
    | JSString String
    | JSNumber Double
    | JSObject (M.Map String JSVal)
```

Binary Search Trees

```haskell
{-# LANGUAGE NoMonomorphismRestriction #-}
import Data.Foldable as F

data BinaryTree a = Node a (BinaryTree a) (BinaryTree a) | Empty
    deriving (Eq, Show, Ord)

instance Foldable BinaryTree where
    foldr f acc Empty = acc
    foldr f acc (Node x l r) = F.foldr f (f x (F.foldr f acc r)) l
```

```cpp
#include <memory>
using std::shared_ptr;
using std::make_shared;

template <class A> struct BinaryTree {
    A elem;
    shared_ptr<BinaryTree<A>> left, right;
};
template <class A> shared_ptr<BinaryTree<A>> makeTree(A x,
        shared_ptr<BinaryTree<A>> l, shared_ptr<BinaryTree<A>> r) {
    return make_shared({elem: x, left: l, right: r});
}
template <class A> shared_ptr<BinaryTree<A>> emptyTree() {
    return shared_ptr<BinaryTree<A>>(nullptr);
}
```

Binary Search Trees (continued)

```
treeInsert x Empty = Node x Empty Empty
treeInsert x (Node y l r) = case compare x y of
    LT -> Node y (treeInsert x l) r
    GT -> Node y l (treeInsert x r)
    EQ -> Node y l r
```

```
template <class A> shared_ptr<BinaryTree<A>>
treeInsert(A x, shared_ptr<BinaryTree<A>> tree) {
    if(tree.get() == nullptr) {
        return makeTree(x, emptyTree<A>(), emptyTree<A>());
    } else if(x < tree->elem) {
        return makeTree(tree->elem, treeInsert(x, tree->left), tree->right);
    } else if(tree->elem < x) {
        return makeTree(tree->elem, tree->left, treeInsert(x, tree->right));
    } else { // equal
        return tree;
    }
}
```

Binary Search Trees (continued)

```
treeFind x Empty = Empty
treeFind x (Node y l r) = case compare x y of
    LT -> treeFind x l
    GT -> treeFind x r
    EQ -> Node y l r
```

```
template <class A> shared_ptr<BinaryTree<A>>
treeFind(A x, shared_ptr<BinaryTree<A>> tree) {
    if(tree.get() == nullptr) {
        return tree;
    } else if(x < tree->elem) {
        return treeFind(x, tree->left);
    } else if(tree->elem < x) {
        return treeFind(x, tree->right);
    } else { // equal
        return tree;
    }
}
```

Binary Search Trees (continued)

```
treeRemove x Empty = Empty
treeRemove x (Node y l r) = case compare x y of
    LT -> Node y (treeRemove x l) r
    GT -> Node y l (treeRemove x r)
    EQ -> case (l, r) of
        (Empty, Empty) -> Empty
        (_, Empty) -> l
        (Empty, _) -> r
        (_, _) -> let lMax = F.maximum l in
            Node lMax (treeRemove lMax l) r
```

Binary Search Trees (continued)

```
template <class A> shared_ptr<BinaryTree<A>>
treeRemove(A x, shared_ptr<BinaryTree<A>> tree) {
    if(tree.get() == nullptr) {
        return tree;
    } else if(x < tree->elem) {
        return treeRemove(x, tree->left);
    } else if(tree->elem < x) {
        return treeRemove(x, tree->right);
    } else { // equal
        if((tree->left.get() == nullptr) &&
            (tree->right.get() == nullptr)) {
            return emptyTree<A>();
        } else if(tree->right.get() == nullptr) {
            return tree->left;
        } else if(tree->left.get() == nullptr) {
            return tree->right;
        } else {
            // requires implementation of tree iterators, which was not done
            //  for this presentation due to space constraints
            A lMax = std::max(begin(tree->left), end(tree->left));
            return makeTree(lMax, treeRemove(lMax, tree->left), tree->right);
        }
    }
}
```

Quicksort

```
{-# LANGUAGE NoMonomorphismRestriction #-}
import Control.Monad
import Control.Monad.ST
import Data.STRef
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM

import Test.QuickCheck

naiveQuicksort [] = []
naiveQuicksort lst = naiveQuicksort less ++ eq ++ naiveQuicksort more where
    pivot = lst !! (length lst `div` 2)
    [less, eq, more] = map (\op -> filter (`op` pivot) lst) [(<), (==), (>)]

partitionByST cmp vec pivot lo hi = aux lo lo hi where
    aux i j n | j <= n = do
        a_j <- VM.read vec j
        case cmp a_j pivot of
            LT -> VM.swap vec i j >> aux (i+1) (j+1) n
            GT -> VM.swap vec j n >> aux i j (n-1)
            EQ -> aux i (j+1) n
    aux i j n = return (i, j)
```

Quicksort (continued)

```
quicksortByST cmp vec = aux 0 (VM.length vec - 1) where
    aux lo hi = when (lo < hi) $ do
        let pivotIdx = lo + ((hi-lo) `div` 2)
        pivot <- VM.read vec pivotIdx
        (leftMid, rightMid) <- partitionByST cmp vec pivot lo hi
        aux lo leftMid
        aux rightMid hi

quicksortBy cmp vec = V.modify (quicksortByST cmp) vec

quicksort = quicksortBy compare

quicksortList = V.toList . quicksort . V.fromList

runTests = quickCheck naiveSort_matches_STSort where
    naiveSort_matches_STSort :: [Int] -> Bool
    naiveSort_matches_STSort x = naiveQuicksort x == quicksortList x
```

Questions?

Thanks

- RCOS
- Professor Goldschmidt
- Professor Moorthy
- Sean O'Sullivan