Introduction to Functional Programming with Haskell - RCOS Presentation

Avi Weinstock

February 27, 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
  ```

- ```
  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 ()
```

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
```

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)
```

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 Data.List
import Test.QuickCheck

partitionByST cmp vec i j = do
    let pivotIdx = (i + j) 'div' 2
    pivotVal <- VM.read vec pivotIdx
    VM.swap vec pivotIdx j
    storeIdx <- newSTRef i
    forM_ [i..j-1] $ \i -> do
        val <- VM.read vec i
        when (cmp val pivotVal == LT) $ do
            readSTRef storeIdx >>= VM.swap vec i
            modifySTRef storeIdx (+1)
    readSTRef storeIdx >>= VM.swap vec j
    readSTRef storeIdx >>= return
```

Quicksort (continued)

```
quicksortByST cmp vec = aux 0 (VM.length vec - 1) >> return () where
    aux i j = when (i < j) $ do
        mid <- partitionByST cmp vec i j
        aux i (mid-1)
        aux (mid+1) j

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

quicksort = quicksortBy compare

runTests = quickCheck matchesListSort where
    matchesListSort x = (sort x) == (V.toList . quicksort . V.fromList)
```