Higher Order Functions in Haskell

Evan Misshula

2017-02-14

Outline

Every function in haskell only takes one argument

- But what about 'max' or min?
- We actually apply parameters to functions one at time
  - These are called "curried" functions
    - This is after Haskell Curry

  max (Ord a) => a -> a -> a
  max (Ord a) => a -> (a -> a)
- If we call a function with to few parameters we get back a partially applied function

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
multThree 3 5 9 == multThree (multThreee (multThree 3) 5) 9
i==1
```

Here is a curried comparison

- These are the same because 'x' is on both sides of the equation

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x

compareWithHundred1 :: (Num a, Ord a) => a -> Ordering
compareWithHundred1 = compare 100
```

Let's look at an infix function

- simply surround the function with parentheses and only supply one of the parameters
- this is called 'sectioning'

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

String functions can be partially applied too

- this is written in point free style
- it is also sectioned

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

Functions can return functions

- take a function and apply it twice

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

We are going to implement ZipWith

- It joins two lists and performs a function on the corresponding elements

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

flip changes the order of the arguements

```haskell
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x

flip'' :: (a -> b -> c) -> b -> a -> c
flip'' f y x = f x y
```

Map

- map takes a function applies the function to each element of a list

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

Filter

- 'filter' take a function called a predicate and a list of any type
- the predicate takes an element of the list and returns a Bool
    - the filter returns elements for which the predicate is True

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

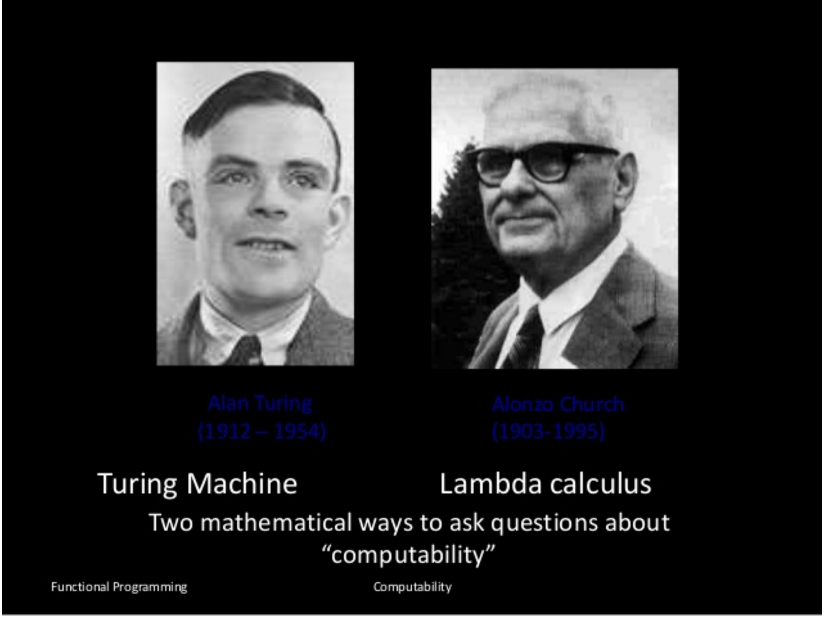
Find the largest number divisible by 3289 under 100,000

- We are going to use head and filter and a range

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

Lamdas are anonymous functions

- These are unnamed functions
- They are passed as parameters to other functions
- They work like composition in math
- They are called 'lambdas' because of the 'lambda calculus'


Alan Turing
(1912 – 1954)
Alonzo Church
(1903-1995)
Turing Machine
Lambda calculus
Two mathematical ways to ask questions about "computability"
Functional Programming
Computability

Lambda Calculus is a formal system for computation

- it is equivelent to calculation by Turing Machine
- invented by Alonzo Church in the 1930's
- Church was Turing's thesis advisor
  - a function is denoted by the greek letter λ
  - a function f(x) that maps x -> f(x) is:
    - $\lambda x.y$

We can pass a lambda to ZipWith

- a lambda function in Haskell starts with '\'
- can't define several parameters for one para,eters

```
zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
1==1
```

Folds encapsulate several functions with (x:xs) patterns

- they reduce a list to a single value
- 'foldl' is the left fold functio

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs

sum'' :: (Num a) => [a] -> a
sum'' = foldl (+) 0
1==1
```

creating elem

- we can have a boolean accumulator function

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) Fal
1==1
```

'foldr' works the same way

- it eats values from the right hand side
- folds can be used to implement any function that goes through a list once
- foldl1 and foldr1 same but start at 0

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

1==1

scanl and scanr are like foldl and foldr only

- they report the intermediate values

```
scanl (+) 0 [3,5,2,1]
scanr (+) 0 [3,5,2,1]
scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
scanl (flip (:)) [] [3,2,1]
1==1
```

Function application with $

- '$' is called the *function application*
- changes to right association
- keeps us from writing parentheses

```
map ($ 3) [(4+), (10*), (^2), sqrt]
1==1
```

```
[7.0,30.0,9.0,1.7320508075688772]
```

Function composition is just like math

- In math $f \cdot g(x) = f(g(x))$
- Let's look at Haskell function
- g takes a -> b
- f takes b -> c

Function composition is just like math

- In math $f \mathbin{.} g(x) = f(g(x))$
- Let's look at Haskell function
- g takes a -> b
- f takes b -> c
- so the composition take f . g takes a -> c

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

```
<interactive>:342:1:
    No instance for (Show ((b0 -> c0) -> (a0 -> b0) -> a0 -> c0
      arising from a use of ‘print’
    Possible fix:
      add an instance declaration for
      (Show ((b0 -> c0) -> (a0 -> b0) -> a0 -> c0))
    In a stmt of an interactive GHCi command: print it
```

Function composition examples

- with a λ
- with point free notation

```
map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
```

```
[-5,-3,-6,-7,-3,-2,-19,-24]
```

What if your function takes more than one argument

- you will need to rewrite

```
sum (replicate 5 (max 6.7 8.9))
(sum . replicate 5 . max 6.7) 8.9
sum . replicate 5 . max 6.7 $ 8.9
i==1
```

```
44.5
44.5
44.5
```

point free functions

```
-- sum' :: (Num a) => [a] -> a
let sum' xs = foldl (+) 0 xs
i==1
```

point free functions

```
let fn x = ceiling (negate (tan (cos (max 50 x))))
let fn’ = ceiling . negate . tan . cos . max 50
i==1
```