



CS 115

Functional Programming

Lecture 4: April 6, 2016

Higher-order functions, part 1



Functional Programming: Spring 2016



Today

- Higher-order functions on lists
- Anonymous functions
- Simple code transformations
- Point-free style
- More on lazy evaluation





Higher-order functions

- Functions in functional languages are *data*
 - can be passed as arguments to other functions
 - can be created on-the-fly
 - can be returned from functions
- Functions which take other functions as arguments and/or return functions as results are called "higher-order" functions





Higher-order functions

- Higher-order functions are the first distinctly "functional" aspect of functional programming we've seen
- This will lead to a style of programming where new functions can often be created by "snapping together" other functions
- Benefits:
 - easier to write code
 - greater confidence that the code is correct





List functions

- Functions on lists include many very useful higher-order functions
- Two simple examples: `map` and `filter`





map

- The **map** function takes a function and a list as its arguments
 - applies the function to each element of the list
 - collects all the results in a new list
 - returns the new list
- For this to work, the function must be *unary* (taking one argument)
- Fortunately, *all* Haskell functions are unary, so not a limitation 😊





map

- What is `map`'s type signature?

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

- `map` takes
 - a function from type `a` to type `b`
 - a list of values of type `a`
- and returns
 - a list of values of type `b`
 - Note: type `b` can be the same as type `a`





map examples

```
double :: Int -> Int
```

```
double x = 2 * x
```

```
Prelude> map double [1, 2, 3, 4, 5]  
[2,4,6,8,10]
```

```
Prelude> map (*2) [1, 2, 3, 4, 5]  
[2,4,6,8,10]
```





Anonymous functions

- Often we want to create a function for a single use
- For example: `double` function on previous slide
 - maybe that's the only place `double` was ever needed
 - requiring that you write a separate function for this is overkill
 - can use an operator section like `(2*)` here, but this is not always possible





Anonymous functions

- Haskell allows you to define *anonymous functions* (functions with no name)
- This is part of what it means for functions to be data: can create them "on the fly"
- They are also referred to as *lambda expressions*
 - from lambda calculus (theoretical underpinnings of Haskell) and Lisp/Scheme





Anonymous functions

- Example:

```
Prelude> map (\x -> 2 * x) [1, 2, 3, 4, 5]  
[2, 4, 6, 8, 10]
```

- Syntax:

`\<pattern> -> <expression>`

- Often `<pattern>` is one or more variables
- The `\` is the typographic symbol most similar to "lambda" (λ)





Puzzle

- What does this return?

```
Prelude> map (\x y -> x + y) [1, 2, 3, 4, 5]
```

- *Hint:* the lambda expression `(\x y -> x + y)` has the type `(Int -> Int -> Int)` (when used with `Ints`)
- This expression returns a list of type `[Int -> Int]`
- It's a list of adder functions:

```
[ \y -> 1 + y, \y -> 2 + y, \y -> 3 + y,  
  \y -> 4 + y, \y -> 5 + y ]
```
- Equivalent to e.g. `[add1, add2, add3, add4, add5]`





Puzzle

- Useful way to think about this:

$\backslash x \ y \rightarrow x + y$

- is the same as:

$\backslash x \rightarrow \backslash y \rightarrow x + y$

- (due to currying)





Puzzle

- This is equivalent to

`map (+) [1, 2, 3, 4, 5]`

- This looks like a type error, but isn't due to curried nature of functions
- Note: if you type this into `ghci`, get error because can't print functions (no printable representation)





Definition of **map**

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





Infinite lists and `map`

- This definition works with infinite lists:

```
Prelude> take 10 (map (2*) [1..])  
[2,4,6,8,10,12,14,16,18,20]
```

- Interesting definition:

```
integers :: [Integer]  
integers = 1 : map (1+) integers
```

- Let's evaluate:

```
take 3 integers
```





Infinite lists and `map`

- `take` definition:

```
take :: Int -> [a] -> [a]
```

```
take 0 _ = []
```

```
take _ [] = error "take: no elements"
```

```
take n (x:xs) = x : take (n-1) xs
```

- *N.B.* This isn't exactly the same as the built-in `take` function

- Evaluate:

```
take 3 integers
```





Infinite lists and **map**

- Evaluate:

take 3 integers

- Reduce:

take 3 (1 : map (1+) integers)

1 : take 2 (map (1+) integers)

1 : take 2 (map (1+) (1 : map (1+) integers))

1 : take 2 (2 : map (1+) (map (1+) integers))

1 : 2 : take 1 (map (1+) (map (1+) integers))

1 : 2 : take 1 (map (1+) (map (1+) (1 : map (1+) integers)))

1 : 2 : take 1 (map (1+) (2 : map (1+) (map (1+) integers)))

1 : 2 : take 1 (3 : map (1+) (map (1+) (map (1+) integers)))





Infinite lists and **map**

- Continue:

```
1 : 2 : take 1 (3 : map (1+) (map (1+) (map (1+) integers)))
```

```
1 : 2 : 3 : take 0 (map (1+) (map (1+) (map (1+) integers)))
```

```
1 : 2 : 3 : []
```

```
[1, 2, 3]
```

- Answer: **[1, 2, 3]**

- Note that

```
map (1+) (map (1+) (map (1+) integers))
```

- was never calculated (lazy evaluation)





filter

- **filter** is a higher-order function which takes as its arguments
 - a predicate (function returning a **Bool**)
 - a list
- and returns a list of the elements in the original list that the predicate returned **True** on (in the same order)
- Type signature:

filter :: (a -> Bool) -> [a] -> [a]





filter

- Definition of **filter**:

filter :: (a -> Bool) -> [a] -> [a]

filter _ [] = []

filter p (x:xs) | p x = x : filter p xs
 | otherwise = filter p xs





filter

- Examples of **filter**:

```
Prelude> filter (\x -> x `mod` 2 == 1) [1..10]  
[1,3,5,7,9]
```

```
Prelude> filter (/= 0) [0, 1, 0, 2, 0, 3, 0]  
[1,2,3]
```

- **filter** works fine on infinite lists:

```
Prelude> take 3 (filter (\x -> x `mod` 2 == 1) [1..])  
[1,3,5]
```

- Exercise: work through this evaluation!





map and filter

- **map** and **filter**: two great tastes that taste great together!

- Consider:

```
twiceNonzeros :: [Int] -> [Int]
```

```
twiceNonzeros [] = []
```

```
twiceNonzeros (0:xs) = twiceNonzeros xs
```

```
twiceNonzeros (x:xs) = 2 * x : twiceNonzeros xs
```

- This definition is correct, but uses explicit recursion
- It's considered poor style to define like this if we can define it without explicit recursion





map and filter

- New definition:

```
twiceNonZeros xs = map (2*)  
                  (filter (\x -> x /= 0) xs)
```

- Much nicer!





The \$ operator

```
twiceNonzeros :: [Int] -> [Int]
```

```
twiceNonZeros xs = map (2*)
```

```
                    (filter (\x -> x /= 0) xs)
```

- There are a couple of simple improvements we can make to this code
- We can get rid of the last set of parentheses using the \$ (apply) operator:

```
twiceNonZeros xs =
```

```
    map (2*) $ filter (\x -> x /= 0) xs
```





The \$ operator

```
Prelude> :info $
```

```
($) :: (a -> b) -> a -> b
```

```
infixr 0 $
```

- This operator takes a function (from **a** to **b**) and a value of type **a**, and applies the function to the value to get a return value of type **b**
- Just an operator version of function application
- Why bother using this?
 - **f \$ x** is just the same as **f x**
- The answer is in the **infixr 0 \$** part





The \$ operator

- The \$ operator has the lowest possible precedence, so anything on its right-hand side gets evaluated before the function is applied
- Use case: consider a chain of function applications:

`f1 (f2 (f3 (f4 (f5 x))))`

- Using \$ makes this cleaner:

`f1 $ f2 $ f3 $ f4 $ f5 x`

- \$ associates to the right so it's equivalent to:

`f1 $ (f2 $ (f3 $ (f4 $ (f5 x))))`

- Which is the same as the expression without \$





Function composition

```
twiceNonZeros xs =
```

```
  map (2*) $ filter (\x -> x /= 0) xs
```

- Even this can be improved!
- This function is just the composition of two smaller functions:
 - `map (2*) :: [Int] -> [Int]`
 - `filter (\x -> x /= 0) :: [Int] -> [Int]`
- Haskell has a function composition operator: the dot `(.)`
 - used everywhere in Haskell code!





Function composition

```
Prelude> :info (.)
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
infixr 9 .
```

- Definition:

```
f . g = \x -> f (g x)
```

- With this, our function now becomes:

```
twiceNonZeros =
```

```
    map (2*) . filter (\x -> x /= 0)
```

- We got rid of the function's argument!





Function composition

```
twiceNonZeros =
```

```
    map (2*) . filter (\x -> x /= 0)
```

- The `.` operator has very high precedence (9) but function application is still higher, so we don't have to write this as:

```
twiceNonZeros =
```

```
    map (2*) . (filter (\x -> x /= 0))
```

- Haskell makes it very convenient to define functions by composing other functions





Function composition

```
twiceNonZeros =
```

```
  map (2*) . filter (\x -> x /= 0)
```

- But wait! We can improve this still more!
- Can rewrite `(\x -> x /= 0)` as an operator section!

```
twiceNonZeros = map (2*) . filter (/= 0)
```

- Compare to:

```
twiceNonzeros [] = []
```

```
twiceNonzeros (0:xs) = twiceNonzeros xs
```

```
twiceNonzeros (x:xs) = 2 * x : twiceNonzeros xs
```





Function composition

- Advantages of:

```
twiceNonZeros = map (2*) . filter (/= 0)
```

- Much shorter! (Less code to get wrong)
- Two actions (mapping and filtering) are separated instead of interleaved together
- Easier to write, easier to understand what's going on
- Now you see why explicit recursion is frowned upon in Haskell





(Aside) Eta equivalence

- In Haskell, these two expressions are equivalent:
 - $\lambda x \rightarrow f\ x$
 - f
- Theorists say that they are *eta-equivalent*
- Going from $\lambda x \rightarrow f\ x$ to just f is called an *eta-reduction*
- Going from f to $\lambda x \rightarrow f\ x$ is an *eta-expansion*
- Eta equivalence doesn't always hold in strict languages!
 - Just f might have to be evaluated in some context where $\lambda x \rightarrow f\ x$ would not require f to be evaluated yet





(Aside) Eta equivalence

- We can sometimes use eta equivalence to simplify function definitions
- For instance, if we had written our previous function as:

```
twiceNonZeros xs =  
    (map (2*) . filter (/= 0)) xs
```

- Eta-equivalence says we can drop the **xs** from both sides
- This is (usually) considered good style





(Aside) Eta equivalence

- However, we can't change:

```
twiceNonZeros xs =  
    map (2*) $ filter (/= 0) xs
```

- Into:

```
twiceNonZeros =  
    map (2*) $ filter (/= 0)
```

- Why not?





(Aside) Eta equivalence

- Reason:

```
twiceNonZeros xs =  
    map (2*) $ filter (/= 0) xs
```

- Can't be written as:

```
twiceNonZeros xs =  
    (map (2*) $ filter (/= 0)) xs
```

- So eta-equivalence doesn't apply
- Yet another reason to prefer the version using function composition!





Point-free style

- The style of defining functions by composing together a bunch of smaller functions, without writing out the arguments, is called *point-free style*
- Can often make code much more elegant and concise
- Occasionally can make code so "tight" it's hard to read/understand
- Use your own coding judgment!





Point-free style

- Without point-free style (AKA *point-wise* style) you might have e.g.:

-- want to build function *q* out of

-- functions *f*, *g*, *h*

```
q x = let x1 = f x in
      let x2 = g x1 in
      let x3 = h x2 in
      x3
```

- All arguments ("points") are explicitly named: **x**, **x1**, **x2**, **x3**





Point-free style

- With point-free style this is just

```
-- want to build function q out of
-- functions f, g, h
q = h . g . f
```
- Much simpler!
- NOTE: the "point" of "point-free style" does *not* refer to the function composition (`.`) operator!
- Point-free style has no (or at least fewer) "points" (explicit names for function arguments) but more "dots" (function composition operators)





Efficiency

- Our previous function has been reduced to:

```
twiceNonZeros = map (2*) . filter (/= 0)
```

- Elegance/clarity advantages are obvious
- But... what about efficiency?
- Consider applying this function to a list of 1,000,000,000 **Ints**, about 20% of which are zeros
- Can you imagine any possible problems with this?





Efficiency

```
twiceNonZeros = map (2*) . filter (/= 0)
```

- The evaluation strategy is important here
- In a *strict* language, might have to create a temporary list to hold the filtered data (80% as long as original data), then map `(2*)` over that to get final list
- A huge amount of extra memory required!
- But in a *lazy* language like Haskell, this problem doesn't come up
- Let's work through an evaluation ☺





Efficiency

```
twiceNonZeros = map (2*) . filter (/= 0)
```

- Evaluate `twiceNonZeros [1, 0, 2, 0, 3, 0]`
- First few steps:

```
twiceNonZeros [1, 0, 2, 0, 3, 0]
```

```
(map (2*) . filter (/= 0)) [1, 0, 2, 0, 3, 0]
```

```
map (2*) (filter (/= 0) [1, 0, 2, 0, 3, 0])
```

```
map (2*) (1 : filter (/= 0) [0, 2, 0, 3, 0])
```

- What is the next step?





Efficiency

```
map (2*) (1 : filter (/= 0) [0, 2, 0, 3, 0])  
2 : (map (2*) (filter (/= 0) [0, 2, 0, 3, 0]))
```

- The filtering hasn't completed, but due to lazy evaluation we're already doing the mapping!
- For instance, what if the original expression had been

```
Prelude> head $ twiceNonZeros [1, 0, 2, 0, 3, 0]  
2
```

- We could stop now!
- Items are computed *on demand*





Efficiency

- Items are computed *on demand*
- What does this mean?
- Any function that "consumes" the list returned from the map/filter is going to want to first look at the head of the list
- So the head of the list is the first thing that gets computed
- The computation runs so as to first generate the head of the list, then the next item, and so on
- Items are computed one at a time





Efficiency

- Continuing...

```
2 : (map (2*) (filter (/= 0) [0, 2, 0, 3, 0]))
2 : (map (2*) (filter (/= 0) [2, 0, 3, 0]))
2 : (map (2*) (2 : filter (/= 0) [0, 3, 0]))
2 : 4 : (map (2*) (filter (/= 0) [0, 3, 0]))
2 : 4 : (map (2*) (filter (/= 0) [3, 0]))
2 : 4 : (map (2*) (3 : filter (/= 0) [0]))
2 : 4 : 6 : (map (2*) (filter (/= 0) [0]))
2 : 4 : 6 : (map (2*) (filter (/= 0) []))
2 : 4 : 6 : (map (2*) [])
2 : 4 : 6 : []

[2, 4, 6]
```





Efficiency

- Note that operations of mapping and filtering are automatically interleaved by lazy evaluation, even though code doesn't do that explicitly
- Consequence: code doesn't have to generate large intermediate lists: *huge* space savings!
- In strict language, might have to interleave the operations explicitly (like in first version of **twiceNonZeros**) to get space efficiency
- Conclusion: *lazy evaluation improves modularity!*





Classic paper

- *Why Functional Programming Matters* by John Hughes
- Explores consequences of lazy evaluation for modularity
- Uses a language called Miranda, which is very similar to Haskell (an ancestor language)
 - you should be able to follow it
- He argues that lazy evaluation provides "better glue" to connect independent pieces of programs together to form new programs





Next time

- More higher-order list functions: `foldr` and friends
- List comprehensions

