Higher-Order Functions

COMP2209 - Programming III

Dr Julian Rathke

These slides use modified content from Graham Hutton’s slides for the module textbook Programming in Haskell
Higher-Order Functions

- A function is called **higher-order** if it takes a function as an argument or returns a function as a result.
- Occasionally “higher-order” is reserved to mean just the former.
- We’ve seen lots of higher-order functions that return functions as results - cf. curried functions.
- Here is an example of a function that takes a function as an argument:

  ```haskell
  twice :: (a -> a) -> a -> a
  twice f a = f (f a)
  ``
Why use higher-order functions?

• Consider the following three functions:

```haskell
doubleList :: [Int] → [Int]
doubleList [] = []
doubleList (x:xs) = 2*x : doubleList xs

squareList :: [Int] → [Int]
squareList [] = []
squareList (x:xs) = x^2 : squareList xs

negateList :: [Int] → [Int]
negateList [] = []
negateList (x:xs) = -x : negateList xs
```

They are quite similar aren’t they?
Why use higher-order functions?

- Consider the following three functions:

  ```haskell
doubleList :: [Int] → [Int]
doubleList [] = []
doubleList (x:xs) = 2*x : doubleList xs

squareList :: [Int] → [Int]
squareList [] = []
squareList (x:xs) = x^2 : squareList xs

negateList :: [Int] → [Int]
negateList [] = []
negateList (x:xs) = -x : negateList xs
  ```

The only thing changing is the name and the operation. Surely we should just define this once somehow?
Why use higher-order functions?

- Let’s rewrite this function in a more abstract way

\[ \text{doThisList} :: [\text{Int}] \to [\text{Int}] \]
\[ \text{doThisList} \; [] = [] \]
\[ \text{doThisList} \; (x:xs) = \text{doThisOperation} \; x : \text{doThisList} \; xs \]

- The beauty of functional languages is that we can abstract the “doThis” operation as an argument to a higher-order function

\[ \text{abstractList} :: (\text{Int} \to \text{Int}) \to [\text{Int}] \to [\text{Int}] \]
\[ \text{abstractList} \; f \; [] = [] \]
\[ \text{abstractList} \; f \; (x:xs) = f \; x : \text{abstractList} \; xs \]
Introducing: map

- In fact, the above function is so common across all list types that it generalises into a polymorphic function known as **map**
- Map applies a given operation to every element in a given list

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map } f \; [] = []
\]

\[
\text{map } f \; (x:xs) = f \; x : \text{map } f \; xs
\]

See how it can be used to create our three functions above:

\[
\text{doubleList } xs = \text{map } (2^*) \; xs
\]

\[
\text{squareList } xs = \text{map } (^2) \; xs
\]

\[
\text{negateList } xs = \text{map } (0-) \; xs
\]
More common patterns

- A common pattern in defining functions is to use a list comprehension to pick out certain elements from a list.

  - evensOnly xs = [ x | x <- xs, even x ]
  - oddsOnly xs = [ x | x <- xs , odd x ]
  - primesOnly xs = [ x | x <- xs , prime x ]

- Notice that we simply use a predicate as a single guard on the declared variable x in each case.
- We can write this in more abstract form as

  testOnly xs = [ x | x <- xs, test x ]

- And again abstract the “test” predicate as an argument to this function:

  abstractOnly p xs = [ x | x <- xs, p x ]
Introducing: filter

• Again, this is a heavily used higher-order function that is generalised to be polymorphic

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

filter p xs = [x | x <- xs, p x]
```

```haskell
> filter even [1,2,3,4,5]
[2,4]
> filter (>3) [1,2,3,4,5]
[4,5]
```
It’s folding time

• After map and filter, perhaps the most used higher-order function for list processing is fold (actually foldl and foldr).

• Folds describe functions that follow the recursive pattern below:

\[
\begin{align*}
  f \, [] &= v \\
  f \, (x:xs) &= x \# f \, xs
\end{align*}
\]

• So an empty list is mapped to some initial value \( v \) and every instance of the constructor \((:)\) is replaced with some operation \( \# \) instead.

• For example:

\[
\begin{align*}
  \text{sum} \, [] &= 0 \\
  \text{sum} \, (x:xs) &= x + \text{sum} \, xs
\end{align*}
\]

\[
\begin{align*}
  \text{and} \, [] &= \text{True} \\
  \text{and} \, (x:xs) &= x \&\& \text{and} \, xs
\end{align*}
\]
The abstract \texttt{foldr} function

- We use the function \texttt{foldr} to capture this pattern and of course, this needs to take as arguments both
  - the initial value that replaces the empty list and
  - the operation that replaces (:) as an argument

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

\[
\text{foldr } f \ v \ [] = v
\]

\[
\text{foldr } f \ v \ (x:xs) = f \ x \ (\text{foldr } f \ v \ xs)
\]
Some example foldr functions

```

foldr :: (a → b → b) → b → [a] → b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

sum :: Num a => [a] → a
sum = foldr (+) 0

product :: Num a => [a] → a
product = foldr (*) 1

and :: [Bool] → Bool
and = foldr (&&) True

sum [1,2,3] = foldr (+) 0 [1,2,3]
= 1 + (foldr (+) 0 [2,3]
= 1 + (2 + foldr (+) 0 [3])
= 1 + (2 + (3 + foldr (+) 0 []))
= 1 + (2 + (3 + 0))
= ... 6

product [2,3] = foldr (*) 1 [2,3]
= 2 * (foldr (*) 1 [3]
= 2 * (3 * foldr (*) 1 [])
= 2 * (3 * 1)
```
Deriving a foldr from a recursive definition

Consider the function length and an application of it:

\[
\text{length } [ ] = 0 \\
\text{length } (x:xs) = 1 + \text{length } xs
\]

\[
\text{length } [1,2,3] = \text{length } 1 : (2 : (3 : [])) \\
= 1 + (\text{length } 2 : (3 : [])) \\
= 1 + (1 + (1 + \text{length } []) ) \\
= 1 + (1 + (1 + 0) ) \\
= 1 + (1 + (1 + \text{foldr } f 0 []) ) \\
= 1 + (1 + (\text{foldr } f 0 [3])) \\
= 1 + \text{foldr } f 0 [2,3] \\
= \text{foldr } f 0 [1,2,3] \\
\]

where \( f \) is \( \lambda x y \rightarrow 1 + y \)

The function \( f \) is applied at every step.

Here is the initial value.

So \( \text{length } = \text{foldr } f 0 \)
Some other foldr functions: append

\[
(++) \; [] \; ys = ys \\
(++) \; (x:xs) \; ys = x : (++) \; xs \; ys
\]

Rewrite this as a section:

\[
(++) \; ys \; [] = ys \\
(++) \; ys \; (x:xs) = x : (++) \; xs \; ys
\]

Rewrite this as a foldr: The initial value is ys and the operation to replace each (:) with is (:) again!

\[
(++) \; ys = \text{foldr} \; (:) \; ys
\]
Some other foldr functions: reverse

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse [1,2,3] = reverse 1 : (2 : (3 : []))
= (reverse 2 : (3 : [])) ++ [1]
= (((reverse []) ++ [3]) ++ [2]) ++ [1]
= ([] ++ [3]) ++ [2] ++ [1]
= f 1 ( [] ++ [3]) ++ [2]
= f 1 ( f 2 ( [] ++ [3]) )
= f 1 ( f 2 ( f 3 [ ] ) )
= foldr (f) []. where f = λx xs → xs ++ [x]
```

So reverse = foldr (λx xs → xs ++ [x]) []
Associativity and fold

- The name foldr is short for “fold right” and this reflects the use of an operator that is assumed to associate to the right.
- For example, \( \text{foldr} (+) 0 [1,2,3] = 1+(2+(3+0)) \)
- Generally we think of foldr as

\[
\text{foldr} (#) v [x0, x1, \ldots, xn] = x0 \# (x1 \# (\ldots(xn \# v)\ldots))
\]

But what about operations that associate to the left?
Left Associativity

- Consider the (tail recursive) version of \texttt{sum}

\[
\text{sum} = \text{sum'} 0 \\
\text{where sum'} \text{ acc } [] = \text{acc} \\
\text{sum'} \text{ acc } (x:xs) = \text{sum'} (\text{acc}+x) \text{ xs}
\]

\[
\text{sum} [1,2,3] = \text{sum'} 0 [1,2,3] \\
= \text{sum'} (0+1) [2,3] \\
= \text{sum'} (((0+1)+2) [3] \\
= \text{sum'} (((0+1)+2)+3) [] \\
= (((0+1)+2)+3) \\
= 6
\]

This forms a left associated sum.

Some functions are defined more naturally this way.
Reverse as left associative

Recall the tail recursive version of reverse?

\[
\text{reverse} = \text{reverse'} []
\]

\[
\text{where} \quad \text{reverse'} \ acc \ [] = acc
\]

\[
\text{reverse'} \ acc \ (x:xs) = \text{reverse'} \ (x:acc) \ xs
\]

\[
\text{reverse'} [] [1,2,3] = \text{reverse'} 1 : (2 : (3 : []))
\]

\[
= \text{reverse'} (1:[]) (2 : (3 : []))
\]

\[
= \text{reverse'} (3:(2:(1:[]))) []
\]

\[
= (3:(2:(1:[])))
\]

\[
= (f (f (f [] 1) 2) 3)
\]

where \( f = \lambda xs x \to x : xs \)

Just “flip” the constructor!
foldl as left associative fold

- This suggests a left associated version of fold
- We can build this using the idea of tail recursive functions

\[
\text{foldl} :: (a \to b \to a) \to a \to [b] \to a \\
\text{foldl} f \text{ acc} \text{ []} = \text{ acc} \\
\text{foldl} f \text{ acc} \text{ (x:xs)} = \text{ foldl} f \text{ (f acc x)} \text{ xs}
\]

- In many cases there is not much to choose between foldr and foldl but in some cases there are simpler definitions and efficiency gains. Be wary of foldl on infinite lists!

\[
\text{sum} = \text{ foldl} \ (+) \ 0 \\
\text{product} = \text{ foldl} \ (*) \ 1 \\
\text{reverse} = \text{ foldl} \ (\lambda x:xs \ x \to x:xs) \text{ []} \\
\text{and} = \text{ foldl} \ (&&) \ True
\]
A useless operator?

• There is an interesting operator ($) in Haskell that takes a function and an argument and simply applies the function to the argument.

\[
($) :: (a \rightarrow b) \rightarrow a \rightarrow b \\
f \ $ \ x = f \ x
\]

• So why not just write \( f \ x \) ?
  • Function application has high precedence and ($) has low precedence
  • Function application associates to the left and ($) associates to the right.
  • Can write \( \text{sqrt} \ $ \ a^2 + b^2 \) instead of \( \text{sqrt} \ (a^2 + b^2) \) for fewer parentheses.
  • ($) can be used as an operator section, or passed to a higher-order function
Function Composition

• Another convenient operator in Haskell is function composition, written (.)

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

\[
f \cdot g = \lambda x \rightarrow f ( g x )
\]

• Again, it is a means of using fewer parentheses in definitions as demonstrated in the following examples.

odd \ x = not (even \ x)

\[
\text{odd} = \text{not} \cdot \text{even}
\]

twice \ f \ x = f (f \ x)

\[
\text{twice} \ f = f \cdot f
\]

sumsqeven \ ns =
sum ( map (^2) (filter even \ ns) )

\[
\text{sumsqeven} = \text{sum} \cdot \text{map} (^2) \cdot \text{filter even}
\]
Quick Quiz - Higher Types

What are the types of the following functions?

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>symm f = \(x \ y \rightarrow f \ y \ x\)</code></td>
<td>???</td>
</tr>
<tr>
<td><code>zipwith f [ ] _ = []</code></td>
<td>???</td>
</tr>
<tr>
<td><code>zipwith f _ [ ] = []</code></td>
<td>???</td>
</tr>
<tr>
<td><code>zipwith (x:xs) (y:ys) = f x y : zipwith xs ys</code></td>
<td>???</td>
</tr>
<tr>
<td><code>uncurry f = \((x,y) \rightarrow f \times y\)</code></td>
<td>???</td>
</tr>
<tr>
<td><code>iterate f x = x : map f (iterate f x)</code></td>
<td>???</td>
</tr>
</tbody>
</table>
YOUR QUESTIONS

Next Lecture:
Declaring Types and Classes