

CSc 520

Principles of Programming Languages

16: Haskell — Higher-Order Functions

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2004 Christian Collberg

- A function is **Higher-Order** if it takes a function as an argument or returns one as its result.
- Higher-order function aren't weird; the differentiation operation from high-school calculus is higher-order:

```
deriv :: (Float->Float)->Float->Float
deriv f x = (f(x+dx) - f x)/0.0001
```

- Many recursive functions share a similar structure. We can capture such “recursive patterns” in a higher-order function.
- We can often avoid the use of explicit recursion by using higher-order functions. This leads to functions that are shorter, and easier to read and maintain.

Currying Revisited

- We have already seen a number of higher-order functions. In fact, any curried function is higher-order. Why? Well, when a curried function is applied to one of its arguments it returns a new function as the result.

Uh, what was this currying thing?

- A curried function does not have to be applied to all its arguments at once. We can supply some of the arguments, thereby creating a new specialized function. This function can, for example, be passed as argument to a higher-order function.

Currying Revisited...

How is a curried function defined?

- A curried function of n arguments (of types t_1, t_2, \dots, t_n) that returns a value of type t is defined like this:

```
fun :: t1 -> t2 -> ... -> tn -> t
```

- This is sort of like defining n different functions (one for each $->$). In fact, we could define these functions explicitly, but that would be tedious:

```
fun1 :: t2 -> ... -> tn -> t
fun1 a2 ... an = ...
```

```
fun2 :: t3 -> ... -> tn -> t
fun2 a3 ... an = ...
```

Currying Revisited...

Duh, how about an example?

- Certainly. Lets define a recursive function `get_nth n xs` which returns the `n`:th element from the list `xs`:

```
get_nth 1 (x:_) = x
get_nth n (_:xs) = get_nth (n-1) xs
```

```
get_nth 10 "Bartholomew" => 'e'
```

- Now, let's use `get_nth` to define functions `get_second`, `get_third`, `get_fourth`, and `get_fifth`, without using explicit recursion:

```
get_second = get_nth 2 | get_fourth = get_nth 4
get_third  = get_nth 3 | get_fifth  = get_nth 5
```

Currying Revisited...

```
get_fifth "Bartholomew" => 'h'
```

```
map (get_nth 3)
  ["mob", "sea", "tar", "bat"] =>
  "bart"
```

So, what's the type of `get_second`?

- Remember the **Rule of Cancellation**?
- The type of `get_nth` is `Int -> [a] -> a`.
- `get_second` applies `get_nth` to one argument. So, to get the type of `get_second` we need to cancel `get_nth`'s first type: ~~`Int`~~ `-> [a] -> a ≡ [a] -> a`.

Patterns of Computation

Mappings

- Apply a function `f` to the elements of a list `L` to make a new list `L'`. **Example**: Double the elements of an integer list.

Selections

- Extract those elements from a list `L` that satisfy a predicate `p` into a new list `L'`. **Example**: Extract the even elements from an integer list.

Folds

- Combine the elements of a list `L` into a single element using a binary function `f`. **Example**: Sum up the elements in an integer list.

The `map` Function

- `map` takes two arguments, a function and a list. `map` creates a new list by applying the function to each element of the input list.
- `map`'s first argument is a function of type `a -> b`. The second argument is a list of type `[a]`. The result is a list of type `[b]`.

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

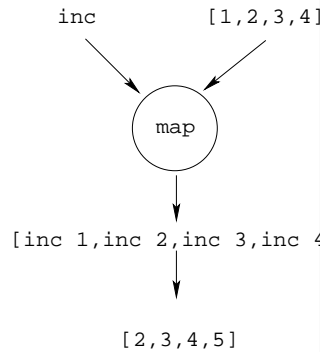
- We can check the type of an object using the `:type` command. Example: `:type map`.

The map Function...

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

```
inc x = x + 1
```

```
map inc [1,2,3,4] ⇒ [2,3,4,5]
```



The map Function...

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

map f [] = [] means: “The result of applying the function f to the elements of an empty list is the empty list.”

map f (x:xs) = f x : map f xs means: “applying f to the list $(x:xs)$ is the same as applying f to x (the first element of the list), then applying f to the list xs , and then combining the results.”

The map Function...

Simulation:

```
map square [5,6] ⇒
square 5 : map square [6] ⇒
25 : map square [6] ⇒
    25 : (square 6 : map square [ ]) ⇒
    25 : (36 : map square [ ]) ⇒
        25 : (36 : [ ]) ⇒
        25 : [36] ⇒
[25,36]
```

The filter Function

- Filter takes a predicate p and a list L as arguments. It returns a list L' consisting of those elements from L that satisfy p .
- The predicate p should have the type $a \rightarrow \text{Bool}$, where a is the type of the list elements.

Examples:

```
filter even [1..10] ⇒ [2,4,6,8,10]
filter even (map square [2..5]) ⇒
    filter even [4,9,16,25] ⇒ [4,16]
filter gt10 [2,5,9,11,23,114]
    where gt10 x = x > 10 ⇒ [11,23,114]
```

The filter Function...

- We can define `filter` using either recursion or list comprehension.

Using recursion:

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

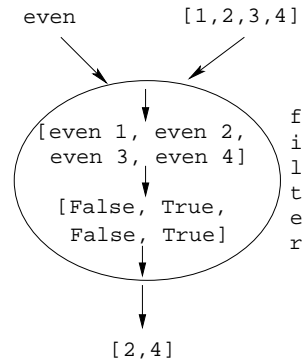
Using list comprehension:

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

The filter Function...

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

`filter even [1,2,3,4] ⇒ [2,4]`



The filter Function...

- `doublePos` doubles the positive integers in a list.

```
getEven :: [Int] -> [Int]
getEven xs = filter even xs
```

```
doublePos :: [Int] -> [Int]
doublePos xs = map dbl (filter pos xs)
  where dbl x = 2 * x
        pos x = x > 0
```

Simulations:

`getEven [1,2,3] ⇒ [2]`

`doublePos [1,2,3,4] ⇒`
`map dbl (filter pos [1,2,3,4]) ⇒`
`map dbl [2,4] ⇒ [4,8]`

fold Functions

- A common operation is to combine the elements of a list into one element. Such operations are called **reductions** or **accumulations**.

Examples:

```
sum [1,2,3,4,5] ≡
  (1 + (2 + (3 + (4 + (5 + 0)))))) ⇒ 15
concat ["H","i","!"] ≡
  ("H" ++ ("i" ++ ("!" ++ ""))) ⇒ "Hi!"
```

- Notice how similar these operations are. They both combine the elements in a list using some binary operator (+, ++), starting out with a “seed” value (0, "").

fold Functions...

- Haskell provides a function `foldr` (“fold right”) which captures this pattern of computation.
- `foldr` takes three arguments: a function, a seed value, and a list.

Examples:

```
foldr (+) 0 [1,2,3,4,5] ⇒ 15
foldr (++) "" ["H","i","!"] ⇒ "Hi!"
```

foldr:

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

fold Functions...

- Note how the fold process is started by combining the last element x_n with z . Hence the name **seed**.

$$\text{foldr}(\oplus)z[x_1 \cdots x_n] = (x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z))))$$

- Several functions in the standard prelude are defined using `foldr`:

```
and,or :: [Bool] -> Bool
and xs = foldr (&&) True xs
or xs = foldr (||) False xs
? or [True,False,False] ⇒
  foldr (||) False [True,False,False] ⇒
  True || (False || (False || False)) ⇒ True
```

fold Functions...

- Remember that `foldr` binds from the right:

```
foldr (+) 0 [1,2,3] ⇒ (1+(2+(3+0)))
```

- There is another function `foldl` that binds from the left:

```
foldl (+) 0 [1,2,3] ⇒ (((0+1)+2)+3)
```

- In general:

$$\text{foldl}(\oplus)z[x_1 \cdots x_n] = (((z \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_n)$$

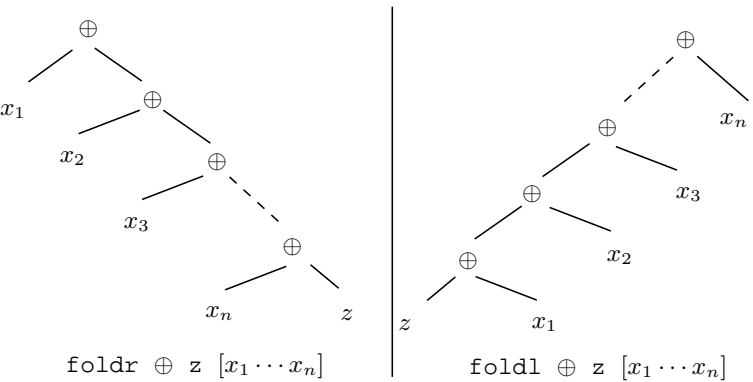
fold Functions...

- In the case of `(+)` and many other functions

$$\text{foldl}(\oplus)z[x_1 \cdots x_n] = \text{foldr}(\oplus)z[x_1 \cdots x_n]$$

- However, one version may be more efficient than the other.

fold Functions...



Operator Sections

- We've already seen that it is possible to use operators to construct new functions:

(* 2) – function that doubles its argument

(> 2) – function that returns True for numbers > 2.

- Such **partially applied operators** are known as **operator sections**. There are two kinds:

$$\frac{(\text{op } a) \ b = b \ \text{op } a}{\begin{array}{l} (*2) \ 4 = 4 * 2 = 8 \\ (>2) \ 4 = 4 > 2 = \text{True} \\ (++) \ \backslash\text{n} \) \ \text{"Bart"} = \text{"Bart"} ++ \backslash\text{n} \end{array}}$$

Operator Sections...

$$\frac{(a \ \text{op}) \ b = a \ \text{op} \ b}{\begin{array}{l} (3:) \ [1,2] = 3 : [1,2] = [3,1,2] \\ (0<) \ 5 = 0 < 5 = \text{True} \\ (1/) \ = 1/5 \end{array}}$$

Examples:

(+ 1) – The successor function.

(/ 2) – The halving function.

(: []) – The function that turns an element into a singleton list.

More Examples:

```
? filter (0<) (map (+1) [-2,-1,0,1])
[-1]
```

takeWhile & dropWhile

- We've looked at the **list-breaking** functions drop & take:

```
take 2 ['a','b','c'] => ['a','b']
drop 2 ['a','b','c'] => ['c']
```

- takeWhile and dropWhile are higher-order list-breaking functions. They take/drop elements from a list while a predicate is true.

```
takeWhile even [2,4,6,5,7,4,1] =>
[2,4,6]
dropWhile even [2,4,6,5,7,4,1] =>
[5,7,4,1]
```

takeWhile & dropWhile...

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

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

Summary

- Higher-order functions take functions as arguments, or return a function as the result.
- We can form a new function by applying a curried function to some (but not all) of its arguments. This is called **partial application**.
- **Operator sections** are partially applied infix operators.

takeWhile & dropWhile...

- Remove initial/final blanks from a string:

```
dropWhile ((==) ' ') "   Hi!" =>
  "Hi!"
```

```
takeWhile ((/=) ' ') "Hi!   " =>
  "Hi!"
```

Summary...

- The standard prelude contains many useful higher-order functions:
 - map f xs** creates a new list by applying the function *f* to every element of a list *xs*.
 - filter p xs** creates a new list by selecting only those elements from *xs* that satisfy the predicate *p* (i.e. (*p x*) should return `True`).
 - foldr f z xs** reduces a list *xs* down to one element, by applying the binary function *f* to successive elements, starting from the right.
 - scanl/scanr f z xs** perform the same functions as `foldr/foldl`, but instead of returning only the ultimate value they return a list of all intermediate results.

Homework

Homework (a):

- Define the `map` function using a list comprehension.

Template:

```
map f x = [ ... | ... ]
```

Homework (b):

- Use `map` to define a function `lengthall xss` which takes a list of strings `xss` as argument and returns a list of their lengths as result.

Examples:

```
? lengthall ["Ay", "Caramba!"]
[2,8]
```

Homework

- Give a accumulative recursive definition of `foldl`.
- Define the `minimum xs` function using `foldr`.
- Define a function `sumsq n` that returns the sum of the squares of the numbers `[1..n]`. Use `map` and `foldr`.
- What does the function `mystery` below do?

```
mystery xs =
  foldr (++) [] (map sing xs)
sing x = [x]
```

Examples:

```
minimum [3,4,1,5,6,3] ⇒ 1
```

Homework...

- Define a function `zip f xs ys` that takes a function `f` and two lists `xs=[x1,...,xn]` and `ys=[y1,...,yn]` as argument, and returns the list `[f x1 y1,...,f xn yn]` as result.
- If the lists are of unequal length, an error should be returned.

Examples:

```
zip (+) [1,2,3] [4,5,6] ⇒ [5,7,9]
```

```
zip (==) [1,2,3] [4,2,2] ⇒ [False,True,True]
```

```
zip (==) [1,2,3] [4,2] ⇒ ERROR
```

Homework

- Define a function `filterFirst p xs` that removes the first element of `xs` that does not have the property `p`.

Example:

```
filterFirst even [2,4,6,5,6,8,7] ⇒
[2,4,6,6,8,7]
```

- Use `filterFirst` to define a function `filterLast p xs` that removes the last occurrence of an element of `xs` without the property `p`.

Example:

```
filterLast even [2,4,6,5,6,8,7] ⇒
[2,4,6,5,6,8]
```