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 © 2005 Christian Collberg
Higher-Order Functions

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:

```haskell
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.
How is a curried function defined?

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

\[
\text{fun} :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t
\]

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

\[
\begin{align*}
\text{fun}_1 :: & \quad t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \\
\text{fun}_1 \ a_2 \ldots a_n &= \ldots
\end{align*}
\]

\[
\begin{align*}
\text{fun}_2 :: & \quad t_3 \rightarrow \ldots \rightarrow t_n \rightarrow t \\
\text{fun}_2 \ a_3 \ldots a_n &= \ldots
\end{align*}
\]
Duh, how about an example?

Certainly. Let's define a recursive function \( \text{get\_nth} \ n \ \text{xs} \) which returns the \( n \):th element from the list \( \text{xs} \):

\[
\begin{align*}
\text{get\_nth} \ 1 \ (x::_) &= x \\
\text{get\_nth} \ n \ (_::xs) &= \text{get\_nth} \ (n-1) \ xs
\end{align*}
\]

\[
\begin{align*}
\text{get\_nth} \ 10 \ "Bartholomew" &= \ 'e' \\
\end{align*}
\]

Now, let's use \( \text{get\_nth} \) to define functions \( \text{get\_second} \), \( \text{get\_third} \), \( \text{get\_fourth} \), and \( \text{get\_fifth} \), without using explicit recursion:

\[
\begin{align*}
\text{get\_second} &= \text{get\_nth} \ 2 \\
\text{get\_third} &= \text{get\_nth} \ 3 \\
\text{get\_fourth} &= \text{get\_nth} \ 4 \\
\text{get\_fifth} &= \text{get\_nth} \ 5
\end{align*}
\]
Currying Revisited...

\[
\text{get\textunderscore fifth} \ "\text{Bartholomew}\text{"} \Rightarrow \ 'h'
\]

\[
\text{map} \ (\text{get\textunderscore nth} \ 3) \\
[\text{"mob", "sea", "tar", "bat"}] \Rightarrow \ "bart"
\]

So, what's the type of \text{get\textunderscore second}?

Remember the \textbf{Rule of Cancellation}?

- The type of \text{get\textunderscore nth} is \texttt{Int \to [a] \to a}.
- \text{get\textunderscore second} \textbf{applies} \text{get\textunderscore nth} to one argument. So, to get the type of \text{get\textunderscore second} we need to cancel \text{get\textunderscore nth's first type}: \texttt{Int \to [a] \to a} \equiv \texttt{[a] \to a}. 

---

520—Spring 2005—16
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 \( a \rightarrow b \). The second argument is a list of type \([a]\). The result is a list of type \([b]\).

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

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

\[
\text{map } f \ (x:x:s) \quad = \ f \ x : \ \text{map } f \ x:s
\]

- 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 \texttt{map} Function...

\begin{verbatim}
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
\end{verbatim}

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

\texttt{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 \texttt{map} Function... 

Simulation:

\begin{align*}
\text{map square } [5,6] & \Rightarrow \\
\text{square 5 : map square } [6] & \Rightarrow \\
25 & : \text{map square } [6] \Rightarrow \\
25 & : (\text{square 6 : map square } [\ ] ) \Rightarrow \\
25 & : (36 : \text{map square } [\ ]) \Rightarrow \\
25 & : (36 : [\ ]) \Rightarrow \\
25 & : [36] \Rightarrow \\
[25,36]
\end{align*}
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 \texttt{filter} Function...

We can define \texttt{filter} using either recursion or list comprehension.

\underline{Using recursion:}

\begin{verbatim}
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
\end{verbatim}

\underline{Using list comprehension:}

\begin{verbatim}
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
\end{verbatim}
The `filter` Function...

\[
\begin{align*}
\text{filter} & : \ (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{filter} \_ \ [] & = [] \\
\text{filter} \ p \ (x:xs) & \quad \left| \begin{array}{l}
p \ x = x : \ \text{filter} \ p \ xs \\
\text{otherwise} = \ \text{filter} \ p \ xs
\end{array}\right.
\end{align*}
\]

\[\text{filter even \ [1,2,3,4] } \Rightarrow \ [2,4]\]
The filter Function...

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

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

**Examples:**

\[
\text{sum } [1,2,3,4,5] \equiv \left(1 + (2 + (3 + (4 + (5 + 0)))))\right) \Rightarrow 15
\]

\[
\text{concat } ["H","i","!"] \equiv \left("H" ++ ("i" ++ ("!" ++ ""))\right) \Rightarrow "Hi!"
\]

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, "").
Haskell provides a function \texttt{foldr} ("fold right") which captures this pattern of computation. \texttt{foldr} takes three arguments: a function, a seed value, and a list.

**Examples:**

\begin{align*}
\text{foldr} (+) 0 [1,2,3,4,5] &\Rightarrow 15 \\
\text{foldr} (++) "\" ["H","i","!"] &\Rightarrow "Hi!"
\end{align*}

\textbf{foldr}:

\begin{verbatim}
foldr :: (a->b->b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
\end{verbatim}
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 \textit{foldr}:

\begin{verbatim}
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
\end{verbatim}
**fold Functions...**

- Remember that `foldr` binds from the right:
  \[
  \text{foldr} \ (\ + \ ) \ 0 \ [1,2,3] \ \Rightarrow \ (1+(2+(3+0)))
  \]

- There is another function `foldl` that binds from the left:
  \[
  \text{foldl} \ (\ + \ ) \ 0 \ [1,2,3] \ \Rightarrow \ (((0+1)+2)+3)
  \]

- In general:
  \[
  \text{foldl}(\oplus)z[x_1 \cdots x_n] = (((z \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_n)
  \]
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...

\[
\text{foldr} \oplus z \ [x_1 \cdots x_n]
\]
\[
\text{foldl} \oplus z \ [x_1 \cdots x_n]
\]
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:

\[
\begin{align*}
(*^2) & \quad 4 \\
(&>2) & \quad 4 \\
(\text{++ } "\text{n}") & \quad "\text{Bart"} = "\text{Bart}" \text{ ++ } "\text{n}\"
\end{align*}
\]
Operator Sections...

\[(a \ op) \ b = a \ op \ b\]

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

\[(0<) \ 5 = 0 < 5 = True\]

\[(1/) = 1/5\]

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:

  \[
  \text{take 2 } [\text{'a','b','c'] } \Rightarrow \text{ [\text{'a','b']}
  \]
  \[
  \text{drop 2 } [\text{'a','b','c'] } \Rightarrow \text{ [\text{'c']}
  \]

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

  \[
  \text{takeWhile even } [2,4,6,5,7,4,1] \Rightarrow [2,4,6]
  \]
  \[
  \text{dropWhile even } [2,4,6,5,7,4,1] \Rightarrow [5,7,4,1]
  \]
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
**takeWhile & dropWhile...**

- Remove initial/final blanks from a string:

  ```haskell
  dropWhile (==( )) "___Hi!" ⇒ "Hi!"
  takeWhile (=/= ) "Hi!___" ⇒ "Hi!"
  ```
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.
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 \texttt{map} function using a list comprehension.

\textbf{Template:}

\begin{verbatim}
map f x = [...] \\
\end{verbatim}

Homework (b):

Use \texttt{map} to define a function \texttt{lengthall xss} which takes a list of strings \texttt{xss} as argument and returns a list of their lengths as result.

\textbf{Examples:}

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

1. Give a accumulative recursive definition of foldl.
2. Define the minimum xs function using foldr.
3. Define a function sumsq n that returns the sum of the squares of the numbers [1⋯n]. Use map and foldr.
4. 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 \( zipp \ f \ xs \ ys \) that takes a function \( f \) and two lists \( xs=[x_1, \ldots, x_n] \) and \( ys=[y_1, \ldots, y_n] \) as argument, and returns the list \( [f \ x_1 \ y_1, \ldots, f \ x_n \ y_n] \) as result.

If the lists are of unequal length, an error should be returned.

Examples:

\[ zipp (+) [1,2,3] [4,5,6] \Rightarrow [5,7,9] \]
\[ zipp (==) [1,2,3] [4,2,2] \Rightarrow [False,True,True] \]
\[ zipp (==) [1,2,3] [4,2] \Rightarrow ERROR \]
Define a function \( \text{filterFirst} \ p \ \text{xs} \) that removes the first element of \( \text{xs} \) that does not have the property \( p \).

Example:

\[
\text{filterFirst} \ \text{even} [2, 4, 6, 5, 6, 8, 7] \Rightarrow [2, 4, 6, 6, 8, 7]
\]

Use \( \text{filterFirst} \) to define a function \( \text{filterLast} \ p \ \text{xs} \) that removes the last occurrence of an element of \( \text{xs} \) without the property \( p \).

Example:

\[
\text{filterLast} \ \text{even} [2, 4, 6, 5, 6, 8, 7] \Rightarrow [2, 4, 6, 5, 6, 8]
\]