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:

\[
\text{deriv} :: (\text{Float} \to \text{Float}) \to \text{Float} \to \text{Float}
\]

\[
\text{deriv}\ f\ x = (f(x+dx) - f\ x)/0.0001
\]

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:

\[
\text{fun} :: t_1 \to t_2 \to \cdots \to t_n \to t
\]

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

\[
\text{fun}_1 :: t_2 \to \cdots \to t_n \to t
\]

\[
\text{fun}_1\ a_2 \cdots a_n = \cdots
\]

\[
\text{fun}_2 :: t_3 \to \cdots \to t_n \to t
\]

\[
\text{fun}_2\ a_3 \cdots a_n = \cdots
\]
Currying Revisited...

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 \( \text{xs} \):

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

\( \text{get\_nth} \ 10 \ "\text{Bartholomew}" \Rightarrow 'e' \)

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:

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

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 \( \text{map} \) Function

\( \text{map} \) takes two arguments, a function and a list. \( \text{map} \) creates a new list by applying the function to each element of the input list.

\( \text{map} \)’s first argument is a function of type \( \text{a} \to \text{b} \). The second argument is a list of type \( \text{[a]} \). The result is a list of type \( \text{[b]} \).

\[
\begin{align*}
\text{map} \ : & \quad (a \to b) \to [a] \to [b] \\
\text{map} \ f \ [\ ] &= [\ ] \\
\text{map} \ f \ (x::xs) &= f \ x : \ \text{map} \ f \ xs
\end{align*}
\]

We can check the type of an object using the \texttt{:type} command. Example: \texttt{:type map}. 

Currying Revisited...

get\_fifth "\text{Bartholomew}" \Rightarrow 'h'

map (get\_nth 3) "\text{mob},"\text{sea},"\text{tar},"\text{bat}" \Rightarrow "\text{bart}"

So, what's the type of \( \text{get\_second} \)?

Remember the Rule of Cancellation?

The type of \( \text{get\_nth} \) is \( \text{Int} \to [\text{a}] \to \text{a} \).

\( \text{get\_second} \) applies \( \text{get\_nth} \) to one argument. So, to get the type of \( \text{get\_second} \) we need to cancel \( \text{get\_nth} \)'s first type: \( \text{Int} \to [\text{a}] \to \text{a} \equiv [\text{a}] \to \text{a} \).
**The map Function**

\[
\text{map} :: (a \to b) \to [a] \to [b] \\
\text{map } f \ [\ ] = [\ ] \\
\text{map } f \ (x:xs) = f \ x : \text{map } f \ xs \\
\text{inc } x = x + 1 \\
\text{map inc } [1,2,3,4] \Rightarrow [2,3,4,5]
\]

**Simulation:**

\[
\text{map square } [5,6] \Rightarrow \\
\text{square 5 : map square } [6] \Rightarrow \\
25 : \text{map square } [6] \Rightarrow \\
25 : (\text{square 6 : map square } [\ ]) \Rightarrow \\
25 : (36 : \text{map square } [\ ]) \Rightarrow \\
25 : [36] \Rightarrow \\
[25,36]
\]

**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 \( a \to \text{Bool} \), where \( a \) is the type of the list elements.

**Examples:**

\[
\begin{align*}
\text{filter even } [1..10] & \Rightarrow [2,4,6,8,10] \\
\text{filter even } (\text{map square } [2..5]) & \Rightarrow [4,9,16,25] \\
\text{filter gt10 } [2,5,9,11,23,114] & \Rightarrow [11,23,114]
\end{align*}
\]
We can define `filter` using either recursion or list comprehension.

**Using recursion:**

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

**Using list comprehension:**

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

**Examples:**

- `filter even [1,2,3,4] ⇒ [2,4]`

`doublePos` doubles the positive integers in a list.

```haskell
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] ⇒ [2,4]`
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:**

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

`foldr`:

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

Remember that `foldr` binds from the right:

```haskell
foldr (+) 0 [1,2,3] \Rightarrow (1+(2+(3+0)))
```

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

```haskell
foldl (+) 0 [1,2,3] \Rightarrow (((0+1)+2)+3)
```

In general:

```haskell
foldl(\oplus)z[x_1 \ldots x_n] = foldr(\oplus)z[x_1 \ldots x_n]
```

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

\[
\begin{align*}
    \text{foldr } \oplus z [x_1 \cdots x_n] & \quad \text{foldl } \oplus z [x_1 \cdots x_n]
\end{align*}
\]

Operator Sections

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

\[
\begin{align*}
    (*2) & \quad \text{function that doubles its argument} \\
    (>2) & \quad \text{function that returns True for numbers > 2.}
\end{align*}
\]

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

\[
\begin{align*}
    (\text{op } a) b &= b \text{ op } a \\
    (*2) 4 &= 4 \times 2 = 8 \\
    (>2) 4 &= 4 > 2 = \text{True} \\
    (++ "\n") "Bart" &= "Bart" ++ "\n"
\end{align*}
\]

Operator Sections...

\[
\begin{align*}
    (a \text{ op } b) &= a \text{ op } b \\
    (3:) & \quad [1,2] = 3 : [1,2] = [3,1,2] \\
    (0<) & \quad 5 = 0 < 5 = \text{True} \\
    (1/) & \quad = 1/5
\end{align*}
\]

Examples:

- \( (+1) \) – The successor function.
- \( (/2) \) – The halving function.
- \( (\cdot[]) \) – The function that turns an element into a singleton list.

More Examples:

\[
\begin{align*}
    ?\ & \text{filter} \ (0<) \ (\text{map} \ (+1) \ [-2,-1,0,1]) \\
    & \text{map} \ (+1) \ [-2,-1,0,1]) = [-1]
\end{align*}
\]

takeWhile & dropWhile

- We’ve looked at the list-breaking functions \( \text{drop} \) & \( \text{take} \):

\[
\begin{align*}
    \text{take} 2 \ ['a','b','c'] & \Rightarrow ['a','b'] \\
    \text{drop} 2 \ ['a','b','c'] & \Rightarrow ['c']
\end{align*}
\]

- \( \text{takeWhile} \) and \( \text{dropWhile} \) are higher-order list-breaking functions. They take/drop elements from a list while a predicate is true.

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

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=[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:
```
zipp (+) [1, 2, 3] [4, 5, 6] ⇒ [5, 7, 9]
zipp (==) [1, 2, 3] [4, 2, 2] ⇒ [False, True, True]
zipp (==) [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]
```