CSCE 314
Programming Languages

Haskell: Higher-order Functions

Dr. Hyunyoung Lee
Higher-order Functions

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Note:
- Higher-order functions are very common in Haskell (and in functional programming).
- Writing higher-order functions is crucial practice for effective programming in Haskell, and for understanding others’ code.
Why Are They Useful?

- **Common programming idioms** can be encoded as functions within the language itself.

- **Domain specific languages** can be defined as collections of higher-order functions. For example, higher-order functions for processing lists.

- **Algebraic properties** of higher-order functions can be used to reason about programs.
The Map Function

The higher-order library function called `map` applies a function to every element of a list.

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

For example:

\[
> \text{map } (+1) [1,3,5,7] \\
[2,4,6,8]
\]

The map function can be defined in a particularly simple manner using a list comprehension:

\[
\text{map } f \; \text{x} = [f \; \text{x} \mid \text{x} \leftarrow \text{x}]
\]

Alternatively, it can also be defined using recursion:

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (\text{x}:\text{xs}) &= f \; \text{x} : \text{map } f \; \text{xs}
\end{align*}
\]
The Filter Function

The higher-order library function \texttt{filter} selects every element from a list that satisfies a predicate.

\[
\texttt{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

For example:

\[
> \texttt{filter even [1..10]}
\]

\[
[2,4,6,8,10]
\]

Filter can be defined using a list comprehension:

\[
\texttt{filter p xs} = [x \mid x \leftarrow xs, p x]
\]

Alternatively, it can be defined using recursion:

\[
\texttt{filter p []} = []
\]

\[
\texttt{filter p (x:xs)}
\]

\[
\mid p x = x : \texttt{filter p xs}
\]

\[
\mid \text{otherwise} = \texttt{filter p xs}
\]
The foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

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

- \( f \) maps the empty list to some value \( v \), and any non-empty list to some function \( \oplus \) applied to its head and \( f \) of its tail.
For example:

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

\[
\begin{align*}
\text{product } [] &= 1 \\
\text{product } (x:xs) &= x \times \text{product } xs \\
\end{align*}
\]

\[
\begin{align*}
\text{and } [] &= \text{True} \\
\text{and } (x:xs) &= x \&\& \text{and } xs \\
\end{align*}
\]
The higher-order library function `foldr` (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value $v$ as arguments.

For example:

\[
\begin{align*}
\text{sum} & = \text{foldr} \ (+) \ 0 \\
\text{product} & = \text{foldr} \ (* \ 1 \\
\text{or} & = \text{foldr} \ (||) \ True \\
\text{and} & = \text{foldr} \ (&&) \ True
\end{align*}
\]
foldr itself can be defined using recursion:

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

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.
For example:

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

Replace each (:) by (+) and [] by 0.
For example:

\[
\text{product } [1,2,3] \\
= \text{foldr } (*) \ 1 \ [1,2,3] \\
= \text{foldr } (*) \ 1 \ (1:(2:(3:[]))) \\
= 1*(2*(3*1)) \\
= 6
\]

Replace each (:) by (*) and [] by 1.
Other foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

\[
\begin{align*}
\text{length} & \quad :: \ [a] \rightarrow \text{Int} \\
\text{length} \ [] & \quad = \ 0 \\
\text{length} \ (_{_{}} : xs) & \quad = \ 1 + \text{length} \ xs
\end{align*}
\]
For example:

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

Hence, we have:

\[
\text{length} = \text{foldr } (\_ \ n \rightarrow 1+n) 0
\]

Replace each (:) by \( \lambda \_ n \rightarrow 1+n \) and [] by 0
Now the reverse function:

\[
\begin{align*}
\text{reverse} & \quad [\ ] \quad = \quad [\ ] \\
\text{reverse} & \quad (x:xs) \quad = \quad \text{reverse} \quad xs \quad ++ \quad [x]
\end{align*}
\]

For example:

\[
\begin{align*}
\text{reverse} \quad [1,2,3] \\
& \quad = \quad \text{reverse} \quad (1:(2:(3:[]))) \\
& \quad = \quad ((([] \quad ++ \quad [3]) \quad ++ \quad [2]) \quad ++ \quad [1] \\
& \quad = \quad [3,2,1]
\end{align*}
\]

Hence, we have:

\[
\text{reverse} \quad = \quad \text{foldr} \quad (\lambda x \quad xs \quad \rightarrow \quad xs \quad ++ \quad [x]) \quad []
\]

Here, the append function (++) has a particularly compact definition using foldr:

\[
(\quad ++ \quad ys) \quad = \quad \text{foldr} \quad (\quad ::\quad) \quad ys
\]

Replace each (:) by (:) and [] by ys.
Why Is foldr Useful?

- Some recursive functions on lists, such as sum, are simpler to define using foldr.

- Properties of functions defined using foldr can be proved using algebraic properties of foldr.

- Advanced program optimizations can be simpler if foldr is used in place of explicit recursion.
foldr and foldl

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

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

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

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

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

\[
\text{foldl} \ f \ v \ (x:xs) = \text{foldl} \ f \ (f \ v \ x) \ xs
\]

- foldr \ 1 : 2 : 3 : [] => (1 + (2 + (3 + 0)))
- foldl \ 1 : 2 : 3 : [] => (((0 + 1) + 2) + 3)
Other Library Functions

The library function (.) returns the composition of two functions as a single function.

\[
\begin{align*}
(\cdot) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\
\text{f . g} &\ = \ \backslash x \rightarrow f (g \ x)
\end{align*}
\]

For example:

\[
\text{odd} :: \text{Int} \rightarrow \text{Bool} \\
\text{odd} \ = \ \text{not} \ . \ \text{even}
\]

Exercise: Define \text{filterOut} \ p \ \text{xs} that retains elements that do not satisfy \ p.

\[
\text{filterOut} \ p \ \text{xs} = \text{filter} \ (\text{not} \ . \ p) \ \text{xs}
\]

> \text{filterOut} \ odd \ \ [1..10]

[2,4,6,8,10]
The library function \texttt{all} decides if every element of a list satisfies a given predicate.

\begin{verbatim}
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
\end{verbatim}

For example:

\begin{verbatim}
> all even [2,4,6,8,10]
True
\end{verbatim}
Dually, the library function `any` decides if at least one element of a list satisfies a predicate.

\[
\text{any} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{any} \ p \ \text{xs} = \text{or} \ [p \ x \mid x \leftarrow \text{xs}]
\]

For example:

\[
> \text{any} \ \text{isSpace} \ "abc \ def"
\]

True
The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

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

For example:

```haskell
> takeWhile isAlpha "abc def"
"abc"
```
Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

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

For example:

```haskell
> dropWhile isSpace "   abc"
"abc"
```
filter, map and foldr

Typical use is to select certain elements, and then perform a mapping, for example,

```
sumSquaresOfPos ls
  = foldr (+) 0 (map (^2) (filter (>= 0) ls))
```

> sumSquaresOfPos [-4,1,3,-8,10]
110

In pieces:

```
keepPos = filter (>= 0)
mapSquare = map (^2)
sum = foldr (+) 0
sumSquaresOfPos ls = sum (mapSquare (keepPos ls))
```

Alternative definition:

```
sumSquaresOfPos = sum . mapSquare . keepPos
```