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)
twice is higher-order because it takes a function as its first argument.
```

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

map applies a function to every element of a list.

map :: (a → b) → [a] → [b]

For example:

> 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:

map f xs = [f x | x <- xs]

Alternatively, it can also be defined using recursion:

map f [] = []
map f (x:xs) = f x : map f xs
The filter Function

filter selects every element from a list that satisfies a predicate.

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

For example:

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

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

filter can be defined using a list comprehension:

\[ \text{filter } p \; \text{x} \; \text{s} = [x \mid x \leftarrow \text{x}s, \; p \; x] \]

Alternatively, it can be defined using recursion:

\[ \text{filter } p \; [] = [] \]

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

\[ | \; p \; x = x : \text{filter } p \; xs \]

\[ | \; \text{otherwise} = \text{filter } p \; xs \]
The foldr Function

Many 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:

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

\[
\text{product} \ [\] = 1 \\
\text{product} \ (x:xs) = x \times \text{product} \ xs
\]

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

For example:

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

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

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

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

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:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```
For example:

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

Hence, we have:

\[
\text{length } = \text{foldr } (\lambda_\_ n \to 1+n) \emptyset
\]
Now the reverse function:

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

For example:

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

Hence, we have:

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

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

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = 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.

\[(.): (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)\]
\[f \cdot g = \lambda x \rightarrow f (g x)\]

For example:

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

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

\[\text{filterOut} \ p \ \texttt{xs} = \text{filter} \ (\text{not . } p) \ \texttt{xs}\]
\[> \text{filterOut odd [1..10]}\]
\[[2,4,6,8,10]\]
The library function \texttt{all} decides if every element of a list satisfies a given predicate.

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

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.

\[
\text{any} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \\
\text{any } p \; xs = \text{or } [p \; x \mid x \leftarrow xs]
\]

For example:

\[
> \text{any } \text{isSpace} \; "abc \; \text{def}" \\
\text{True}
\]
The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

\[
\text{takeWhile} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
\[
\text{takeWhile} \ p \ [] \ = \ []
\]
\[
\text{takeWhile} \ p \ (x:xs)
\ |
\ p \ x \ = \ x \ : \ \text{takeWhile} \ p \ xs
\ |
\ otherwise \ = \ []
\]

For example:

\[
> \ \text{takeWhile} \ \text{isAlpha} \ "abc \ def"
\]

"abc"
Dually, the function \texttt{dropWhile} removes elements while a predicate holds of all the elements.

\begin{verbatim}
dropWhile :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
dropWhile p [] = []
dropWhile p (x:xs)
    | p x = dropWhile p xs
    | otherwise = x:xs
\end{verbatim}

For example:

\begin{verbatim}
> dropWhile isSpace "   abc"
"abc"
\end{verbatim}
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
```