Higher-order Functions

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

\[
\text{twice} :: (a \rightarrow a) \rightarrow a \rightarrow a \\
\text{twice } f \ x = f (f \ x)
\]
Why Are They Useful?

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

z Domain specific languages can be defined as collections of higher-order functions.

z 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.

\[
\text{map } f \hspace{1em} \left[ a, b, c, d \right] \\
\rightarrow \left[ fa, fb, fc, fd \right]
\]

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

For example:

\[
> \text{map (+1)} \left[ 1, 3, 5, 7 \right] \\
\left[ 2, 4, 6, 8 \right]
\]
The map function can be defined in a particularly simple manner using a list comprehension:

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

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

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

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]
[2,4,6,8,10]
```
Filter can be defined using a list comprehension:

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

Alternatively, it can be defined using recursion:

\[
\begin{align*}
\text{filter } p \ [] & = [] \\
\text{filter } p \ (x:xs) & = \begin{cases} 
  x : \text{filter } p \ xs & \text{if } p \ x \\
  \text{filter } p \ xs & \text{otherwise}
\end{cases}
\end{align*}
\]
Examples:

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

\[ v = 0 \quad \oplus = + \]

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

\[ v = 1 \quad \oplus = \ast \]

\[ \text{and} [\,] = \text{True} \]
\[ \text{and} (x:x\,s) = x \&\& \text{and} \, s \]

\[ v = \text{True} \quad \oplus = \&\& \]
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.
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:

```haskell
sum     = foldr (+) 0
product = foldr (*) 1
or      = foldr (||) False
and     = foldr (&&) True
```
Foldr itself can be defined using recursion:

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
\text{foldr} f v [] = v \\
\text{foldr} f v (x:x) = f x (\text{foldr} f v x) 
\]
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.

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

Replace each (:) by (+) and [] by 0.
product [1,2,3]

= foldr (*) 1 [1,2,3]

= 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.

```
length :: [a] → Int
length [] = 0
length (_:xs) = 1 + length xs
```
Hence, we have:

\[
\text{length} = \text{foldr} (\_ \ n \to 1+n) \ 0
\]
Now recall the reverse function:

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

For example:

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

Replace each 
(:) by \( \lambda x \text{ xs} \rightarrow \text{xs} ++ [x] \) and [] by [].
Hence, we have:

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

Finally, we note that the append function (++) has a particularly compact definition using foldr:

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

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, such as fusion and the banana split rule.

- Advanced program optimizations can be simpler if foldr is used in place of explicit recursion.
Other Library Functions

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

\[
(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)
\]
\[
f \cdot g = \lambda x \to f(g \, x)
\]

For example:

\[
\text{odd} :: \text{Int} \to \text{Bool}
\]
\[
\text{odd} = \text{not} \cdot \text{even}
\]
The library function `all` decides if every element of a list satisfies a given predicate.

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

\[
\text{all } p \text{ } xs = \text{and } [p \text{ } x \mid x \leftarrow xs]
\]

For example:

\[
> \text{all even } [2,4,6,8,10]
\]

True
Dually, the library function \texttt{any} decides if at least one element of a list satisfies a predicate.

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

For example:

\[
> \ \texttt{any} \ (== \ ' ') \ "abc \ def"
\]

\[
\text{True}
\]
The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

\[
\begin{align*}
takeWhile :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
takeWhile p [] &= [] \\
takeWhile p (x:xs) &=\begin{cases} \\
   p x &= x : takeWhile p xs \\
otherwise &= [] \\
\end{cases}
\end{align*}
\]

For example:

```plaintext
> takeWhile (/= ' ') "abc def"
"abc"
```
Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

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

For example:

\[
> \text{dropWhile (== ' ')} " abc"
"abc"
\]
Exercises

(1) What are higher-order functions that return functions as results better known as?

(2) Express the comprehension \([f \ x \mid x \leftarrow xs, \ p \ x]\) using the functions map and filter.

(3) Redefine map \(f\) and filter \(p\) using foldr.