

Introduction

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.

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.
- 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.

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

```
map f []      = []  
map f (x:xs) = f x : map f xs
```

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:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p []      = []  
filter p (x:xs)  =  
    | p x        = x : filter p xs  
    | otherwise  = filter p xs
```

The Foldr Function

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

```
f []          = v
f (x:xs) = x ⊕ f xs
```

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:

```
sum [] = 0  
sum (x:xs) = x + sum xs
```

$v = 0$

$\oplus = +$

```
product [] = 1  
product (x:xs) = x * product xs
```

$v = 1$

$\oplus = *$

```
and [] = True  
and (x:xs) = x && and xs
```

$v = \text{True}$

$\oplus = \&\&$

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:

```
sum      = foldr (+) 0
product  = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True
```

Foldr itself can be defined using recursion:

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

`sum [1,2,3]`
=
`foldr (+) 0 [1,2,3]`
=
`foldr (+) 0 (1:(2:(3:[])))`
=
`1+(2+(3+0))`
=
`6`

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

For example:

`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.

Recall the length function:

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

For example:

$$\begin{aligned} & \text{length } [1,2,3] \\ = & \text{length } (1:(2:(3:[]))) \\ = & 1+(1+(1+0)) \\ = & 3 \end{aligned}$$

Replace each $(:)$
by $\lambda_n \rightarrow 1+n$
and $[]$ by 0.

Hence, we have:

$$\text{length} = \text{foldr } (\lambda_n \rightarrow 1+n) \ 0$$

Now recall the reverse function:

```
reverse []      = []  
reverse (x:xs) = reverse xs ++ [x]
```

For example:

```
reverse [1,2,3]  
= reverse (1:(2:(3:[])))  
= (([] ++ [3]) ++ [2]) ++ [1]  
= [3,2,1]
```

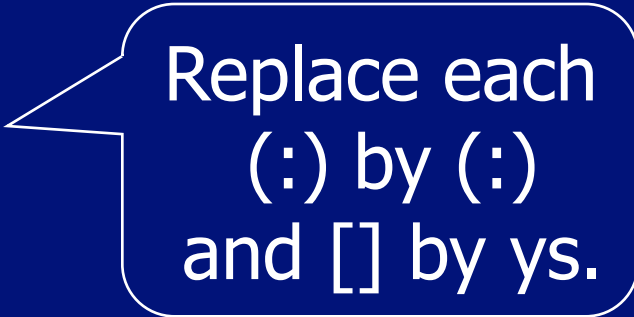
Replace each $(:)$ by
 $\lambda x \text{ xs} \rightarrow \text{xs} ++ [x]$
and $[]$ by $[]$.

Hence, we have:

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

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

```
(++ ys) = 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 optimisations 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.

```
(.)    :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

For example:

```
odd  :: Int → Bool
odd  = not . even
```

The library function all decides if every element of a list satisfies a given predicate.

```
all      :: (a → Bool) → [a] → Bool  
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]  
True
```

Dually, the library function any decides if at least one element of a list satisfies a predicate.

```
any      :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any isSpace "abc def"
True
```

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

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []          = []
takeWhile p (x:xs)
  | p x                  = x : takeWhile p xs
  | otherwise            = []
```

For example:

```
> takeWhile isAlpha "abc def"
"abc"
```

Dually, the function dropWhile removes elements while a predicate holds of all the elements.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []           = []
dropWhile p (x:xs)
  | p x                   = dropWhile p xs
  | otherwise             = x:xs
```

For example:

```
> dropWhile isSpace "   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`.