Session 4

Higher Order Functions

Universiteit Antwerpen

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.

Universiteit Antwerpen

Why Are They Useful?

- Common programming idioms, such as applying a function twice, can naturally be encapsulated as general purpose higher-order functions;
- Special purpose languages can be defined within Haskell using higher-order functions, such as for list processing, interaction, or parsing;
- 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]
```

Universiteit Antwerpen

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
```

Universiteit Antwerpen

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 a value v, and any non-empty list to a function ⊕ applied to its head and f of its tail.

Universiteit Antwerpen

For example:

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

v = 0

⊕ = +

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

v = 1

⊕ = *

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

v = True

⊕ = &&

Universiteit Antwerpen

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

and     = foldr (&&) True
```

Foldr itself can be defined using recursion:

```
foldr (⊕) v []     = v

foldr (⊕) v (x:xs) =

   x ⊕ foldr (⊕) v xs
```

In practice, however, it is better to think of foldr non-recursively, as simultaneously replacing each cons in a list by a function, and [] by a value.

Universiteit Antwerpen

For example:

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

Replace each cons by + and [] by 0.

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.

Exercises

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

(2) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

(3) Show how the functions length, reverse, map f and filter p could be re-defined using foldr.