

AC21007: Haskell Lecture 4 Higher order functions, map, folds

František Farka

Recapitulation

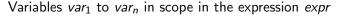


- ▶ Data type tuple (a, b)
- Non-strict semantics:
 - expressions evaluated on-demand
 - ▶ allows infinite data structures (lists)

Anonymous (lambda) functions

- Functions without a name
- Syntax:

$$\langle var_1 \rangle \dots \langle var_n \rangle \rightarrow \langle expr \rangle$$



- Anonymous functions:
 - can be applied to an argument:

$$(\x -> 2 + x) 3 ==> 5$$

- can be passed as an argument... anonymous functions are values
- ► E.g.:



Anonymous (lambda) functions (cont.)

filter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter pred (x:xs) = if (pred x)
    then x : filter pred xs
    else filter pred xs
```

► E.g:

First-class functions



► All functions can be passed as an argument, e.g standard_NDEE functions even and odd:

```
filter odd [1, 2, 3, 4, 5, 6]
==> [1, 3, 5]

filter even [1, 2, 3, 4, 5, 6]
==> [2, 4, 6]
```

- All functions are just values
- We will call functions that take a function as an argument higher order functions

Some useful higher order functions

▶ map - applies a function to each element of a list

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```



```
map (\x -> 2 * x)) [1, 2, 3, 4]
==> [2, 4, 6, 8]
```

zipWith - generalises zip, combines list elements with the
 function in its first argument, truncates the longer list
 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
 zipWith _ [] _ = []
 zipWith _ [] = []

zipWith f (a:as) (b:bs) = f a b : zipWith f as bs

```
zipWith (+) [2, 3, 4] [5, 6, 7] [7, 9, 11]
```

First-class functions (cont)

- Function type a -> b (right-associative)
- Values of this type are constructed by:
 - the usual function definitions
 - ► lambda constructions
- ▶ The following definitions of max are equivalent:

```
max :: (Int -> (Int -> Int))
-- max x y = if x > y then x else y
-- max x = \y -> if x > y then x else y
max = \x y -> if x > y then x else y
```

- Haskell compiler will figure out types from LHS patterns and type of RHS expression
- Note: In a function definition all equations must have the same number of LHS patterns



Currying

- currying translating the evaluation of a function that takes multiple arguments (a tuple of arguments) into evaluating a sequence of (higher-order) functions, each with a single under argument
- A variant of max:

```
\max' :: (d, d) -> d \max' (x, y) = if x > y then ...
```

▶ We can express this translation as higher-order function:

▶ There is also the reverse translation:

uncurry ::
$$(a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$$

uncurry f $(x, y) = f x y$

Function manipulation

UNIVERSITY O.

- Composition
 - ▶ The usual (f.g)(x) = f(g(x))
 - Operator (.), higher order function:

(.) ::
$$(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

f . g = $\ x \rightarrow f (g x)$

▶ E.g.:

```
filter even . (filter (\ x \rightarrow x \text{ 'mod' 3 == 0})
```

- Partial application
 - We can provide function only with first n arguments
 - Result is a partially applied function a new function taking the rest of arguments
 - ► E.g: max 5, (1 +), (2 *)

List folding

- Let's compare two recursive functions on lists:
 - ▶ Function sum:

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

► Function maximum:

```
maximum :: [Integer] -> Integer
maximum [] = error "empty list"
maximum (x : []) = x
maximum (x : xs) = max x (maximum xs)
```

Recursive case has the same structure:

$$recf (x : xs) = f x (recf xs)$$



List folding

- ▶ Let's compare two recursive functions on lists:
 - Function sum:

► Function maximum:

```
maximum :: [Integer] -> Integer
maximum [] = error "empty list"
maximum (x : []) = x
maximum (x : xs) = max x (maximum xs)
```

Recursive case has the same structure:

$$recf (x : xs) = f x (recf xs)$$

Base case is different . . .



List folding (cont.)

- Let's slightly modify our two functions:
- Function sum:

► Function maximum:

maximum max 3 [2, 5, 4, 2]



List folding - foldr and foldl

One generic function foldr for right-associative recursion:

► The structure of recursion is

foldr f z
$$[x_1, x_2, ..., x_n]$$

==> f x_1 (f x_2 ...(f x_n)...)

There is also function

foldl :: (b -> a -> b) -> b -> [a] -> b for left-associative recursion, i.e.:

foldl f z
$$[x_1, x_2, ..., x_n]$$

==> f x_n (...(f x_2 (f x_1)...)

List folding - examples

Our sum and maximum as folds:

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

maximum :: [Int] -> Int
maximum [] = error "empty list"
maximum (x:xs) = foldr max x xs
```

▶ A fold where a and b are different:

```
length :: [a] -> Integer
length xs = foldr f 0 xs
    where
        -- f :: a -> Integer -> Integer
        f _ b = 1 + b
```



Next time



- ▶ Monday the the 8th of February, 2-3PM, Dalhousie 3G05 LT2
- Sorting algorithms on lists
 - Selection Sort
 - Insertion Sort
 - ▶ Bubble Sort