ITI0212 Functional programming

Lecture 3

Pawel Sobocinski

Homework assignment - instructions

Higher order functions

- One of the killer features of functional programming is **first-class functions**

- Since functions are values, they can be passed as arguments

- Functions that take functions as arguments are called **higher-order functions**

map

```
Idris> :t map
map : Functor f => (a -> b) -> f a -> f b
```

```
Idris> :doc map
Prelude.Functor.map : Functor f => (func : a -> b) -> f a -> f b
    Apply a function across everything of type 'a' in a parameterised type

    The function is: Total & public export
```

Example - add 1 to every element of a List

```
addone : List Int -> List Int
addone xs = map (\x => x + 1) xs
```

```
*Addone> addone [1,2,3]
[2, 3, 4] : List Int
```

Example - prefix every string in a list with “not ”

```
addnot : List String -> List String
addnot xs = map (\x => "not " ++ x) xs
```

```
*Addnot> addnot ["good","decent","honourable"]
["not good", "not decent", "not honourable"] : List String
```

filter

```
Idris> :t filter
filter : (a -> Bool) -> List a -> List a
```

```
Idris> :doc filter
Prelude.List.filter : (a -> Bool) -> List a -> List a
    filter, applied to a predicate and a list, returns the list of those elements that satisfy the
    predicate; e.g.,

        > filter (\ARG => ARG < 3) [0, 1, 2, 3, 4]
        [0, 1, 2]

    The function is: Total & public export
```

Example - take all non-negative elements from a list

```
nonnegative : List Int -> List Int
nonnegative xs = filter (>=0) xs
```

```
*Nonnegative> nonnegative [-2,0,2,-3,4]
[0, 2, 4] : List Int
```

Example - take all strings of length equal to 3 from a list

```
*Threeletterstrings> threeletterstrings ["I", "am", "the", "walrus"]
["the"] : List String
```

```haskell
threeletterstrings : List String -> List String
threeletterstrings xs = filter (\x => (length x) == 3) xs
```

Example - quicksort

```
quicksort : (Ord ty) => List ty -> List ty
quicksort [] = []
quicksort (x :: xs) = quicksort (filter (<=x) xs) ++ [x] ++ quicksort (filter (>x) xs)
```

```
*Quicksort> quicksort [21,5,63,436,5,-1]
[-1, 5, 5, 21, 63, 436] : List Integer
```

Other

- Certain higher-order functions called `fold`s are particularly useful
  - `foldl`
  - `foldr`
- More on this in Chad's lectures

Structuring code

- `let` blocks
- `where` blocks
- `modules`

`let in`

- Let blocks allow to bind local variables, whose scope is only visible inside the function body

```
let x1 = something
    x2 = something_else
in f
```

- Use to break up complicated function definitions into more manageable/readable code

Example - more readable

`quicksort`

```
quicksort : (Ord ty) => List ty -> List ty
quicksort [] = []
quicksort (x :: xs) = let lessthanorequaltox = quicksort (filter (<=x) xs)
                          greaterthanx = quicksort (filter (>x) xs)
                      in lessthanorequaltox ++ [x] ++ greaterthanx
```

where

- Where blocks allow one to define local function definitions

Example - take the even nats from a list

```
takeevens : List Nat -> List Nat
takeevens xs = filter even xs
   where
      even : Nat -> Bool
      even Z = True
      even (S k) = not (even k)
```

```
*Takeevens> takeevens [0..10]
[0, 2, 4, 6, 8, 10] : List Nat
```

modules

- Modules allow the logical division of a larger program into several source files, each with its own purpose

- A module **exports** the definition of one or several functions

- A module can be **imported** and its functions used

- A module declaration means that that a namespace for the definitions is created

    - sometimes this means that **fully qualified** function names must be used

Example - average (listing 2.7 in Brady)

```
module Average

export
average : String -> Double
average str = let numWords = wordCount str
                  totalLength = sum (allLengths (words str))
              in
                  cast totalLength / cast numWords
              where
                  wordCount : String -> Nat
                  wordCount str = length (words str)

                  allLengths : List String -> List Nat
                  allLengths strs = map length strs
```

```
*Average> average "The quick fox jumped over the lazy dog"
3.875 : Double
```

Importing

```
module Main

import Average

main : IO ()
main = repl "Enter a string: " stringStats
       where stringStats : String -> String
             stringStats x = "Average word length:" ++ show (average x) ++ "\n"
```

Functions used

```
repl : String -> (String -> String) -> IO ()
```

```
show : Show ty => ty -> String
```