Outline of Lecture 9

- Using higher-order functions: example
- Folding revisited
- Additional Haskell libraries and other resources
- Installing external Haskell packages; Finding external documentation
- Input and output in Haskell

Functions as data

Example: recognising regular expressions – patterns on strings of characters:

- ϵ – empty string
- x – any single character
- $r_1|r_2$ – either pattern r_1 or r_2
- $r_1 r_2$ – r_1 followed by r_2
- (r)* – repeating r zero or more times

Matching arbitrary strings against such patterns

Functions as data (cont.)

A Haskell implementation of regular expressions:

```
type RegExp = String -> Bool

epsilon ::  RegExp
epsilon = (=="")

char ::  Char -> RegExp
char ch = (==[ch])

(|||) ::  RegExp -> RegExp -> RegExp
e1 ||| e2 = \x -> e1 x || e2 x
```

Functions as data (cont.)

A Haskell implementation of regular expressions (cont.):

```
(<*>) ::  RegExp -> RegExp -> RegExp
e1 <*> e2 = \x ->
    or [e1 y && e2 z | (y,z) <- splits x]

star ::  RegExp -> RegExp
star p = epsilon ||| (p <*> star p)
```

`splits :: String -> [(String,String)]` returns all the ways a string can be split into two

Folding revisited

- Folding to the right:

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
```

- Recursive definition of folding to the right:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- Evaluation unfolding according to `foldr`:

```
foldr f z [1,2,3]
1 `f` (foldr f z [2,3])
1 `f` ( 2 `f` (foldr f z [3]))
1 `f` ( 2 `f` (3 `f` (foldr f z [])))
1 `f` ( 2 `f` (3 `f` z))
```

Folding revisited (cont.)

- Folding to the left:

```
foldl ::  (b -> a -> b) -> b -> [a] -> b
```

- Recursive definition of folding to the right:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- Evaluation unfolding according to `foldl`:

```
foldl f z [1,2,3]
foldl f (z `f` 1) [2,3]
foldl f ((z `f` 1) `f` 2) [3]
foldl f (((z `f` 1) `f` 2) `f` 3) []
((z `f` 1) `f` 2) `f` 3
```

Folding revisited (cont.)

- For associative functions like `(+)`, both versions of folding produce the same results:

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

- For non-associative function, the results can be quite different:

```
foldr (^) 2 [1..3] == 1
foldl (^) 2 [1..3] == 64
```

  or

```
foldr (:) [] [1..3] == [1,2,3]
foldl (flip (:)) [] [1..3] == [3,2,1]
```

  where `flip` creates a binary function with the reversed order of parameters

Folding revisited (summary)

- `foldr` associates to the right when evaluating
- Can be thought as alternation between applications `foldr` and the folding function *f*
- The next invocation of `foldr` is thus *conditional* (if necessary), allowing to work with infinite lists:

```
foldr const 0 [1..]  == 1
```

- `foldl` associates to the left when evaluating
- `foldl` self-calls (tail-calls) through the list, only beginning to produce values after reaching the end of the list
- Because of that, `foldl` cannot be used with infinite lists

Folding revisited (summary)

- `foldl` can be also inefficient with very large lists
- The reason: evaluation and simplification is postponed until all list structure is unfolded
- `foldl'` – a more efficient version of `foldl` (located in the module Data.List)
- Forcefully evaluates and simplifies the inner expression z ‘f‘ x before a recursive call foldl f (z ‘f‘ x) xs
- More about evaluation order as well as strict and non-strict (lazy) computations in Haskell – in later lectures

Scans

- A combination of mapping and folding that produces all the intermediate results of folding as a list
- Scanning to the right:

```
scanr ::  (a -> b -> b) -> b -> [a] -> [b]
scanr (+) 0 [1..5] == [15,14,12,9,5,0]
```

- Scanning to the left:

```
scanl ::  (b -> a -> b) -> b -> [a] -> [b]
scanl (+) 0 [1..5] == [0,1,3,6,10,15]
```

- Properties of `scanr` and `scanl`:

```
head(scanr f z xs) == foldr f z xs
last(scanl f z xs) == foldl f z xs
```

Additional Haskell libraries and other resources

- Standard Haskell installation – the Haskell Platform
- In addition to the definitions in `Prelude`, many other functions/ modules/ libraries (hierarchical modules) / packages are available (either in the Haskell Platform or externally)
- Module names are often hierarchical (examples: `QuickCheck.Test`, `Data.Char`, `Data.List`, `Foreign.Marshal.Alloc.Data.Bool`)
- Moreover, additional (package) downloading and installing via using the tool Cabal (a part of the Haskell platform)

`import` command revisited

- **import Mod** – all the `Mod` definitions are imported
  (simple identifiers x, y, ... or qualified ones `Mod`.x, `Mod`.y, ...)
- **import Mod (x,y)** – only x and y are imported from `Mod`
- **import qualified Mod (x,y)** – only qualified identifiers, e.g., `Mod`.x, `Mod`.y, can be used
- **import Mod hiding (x,y)** – all except x and y are imported
- **import Mod as Foo** – the imported module is renamed

- We can use **qualified, as, hiding** keywords in one command
- `Prelude` can be hidden, qualified, and renamed as well:
  import qualified Prelude as P hiding (zip)

Some libraries from the Haskell Platform

- **Data** – contain additional datatypes (like `Data.Array`) or additional operations on the existing types (like `Data.List` or `Data.Char`)
- **Control** – provides application control (e.g., sequencing of computations), basic IO mechanism, concurrent executions, exception handling
- **Numeric** – contains functions to read and print numbers in a variety of formats
- **Foreign** – supports interworking with other programming languages
- **System** – support various forms of IO handling (e.g., interaction with command line)

Additional Haskell resources : Hackage and Cabal

- **Hackage** – an online repository for Haskell packages and libraries (currently over 5000 packages)
- `http://hackage.haskell.org`
- A package: a collection of Haskell modules. Can contain also C code, documentation, test cases, and so on
- **Cabal** – a command line tool for installing packages (and the packages they depend on)
- **Cabal** is a part of the Haskell Platform distribution (quick documentation – `https://wiki.haskell.org/Cabal-Install`)

Additional Haskell resources : Documentation

- `http://hackage.haskell.org/package` – documentation for many external packages listed by category but also searchable
- `http://www.haskell.org/hoogle` – search for many standard libraries (by name and type)
- `http://hayoo.fh-wedel.de` – search of the whole Hackage and standard libraries (only by string/name)

Input and output in Haskell

- We consider simplest programs, reading and writing to a terminal
- The described model (solution) forms a foundation for more complex interactions (e.g., with a mail or an operating system)
- The solution relies on the Haskell type `IO a`, describing programs that do some input/output before returning the value of the type `a`
- A number of such programs can be sequenced by the means of the `do` construct

Input and output: problems in functional programming

- Functional program consists of a number of definitions, associating a fixed value with the variable/identifier name
- How to implement input/output in such a programming style?
- One approach (tried in Standard ML and F#) is to include operations/special identifiers like

  ```
  inputInt :: Integer
  ```

  whose effect is to read an integer from the input. The read value becomes the value of `inputInt`
- How to interpret then the following definition?

$$inputDiff = inputInt - inputInt$$

Input and output: problems in functional programming

$$inputDiff = inputInt - inputInt$$

- Since the values of the first and second occurrences of `inputInt` may be different, evaluation of such a definition breaks the main principle of functional programming stating that an identifier/variable name always stands for a fixed value
- Moreover, the problem propagates in all other definitions relying on `inputDiff`, like

```
funny n = InputInt + n
```

- Such mutability of definitions made I/O quite an issue for functional programming

Input and output: Haskell solution

- A part of the monadic approach (more details later)
- The solution relies on the Haskell type `IO a`, describing programs that do some input/output (or any effects beyond evaluating function or expression) before returning the value of the type `a`
- The type `IO a` contains all I/O actions of the type `a` (i.e., returning, after doing some I/O, the value of the type `a`)
- Such I/O actions are usually done in sequence (read something, calculate next, return some output)
- Haskell provides a small *imperative* language (`do` notation) to sequence such actions

Reading input

- Basic I/O commands (part of Prelude)
- The built-in operation of reading a line from input:

```
getLine :: IO String
```

- Similarly, the operation of reading a single character from input:

```
getChar :: IO Char
```

- In GHCI, executing such commands is delayed until the respective input is supplied

Writing strings into output

- The built-in operation of putting a string to output:

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

- Here `()` represents the Haskell type containing one element (also denoted `()`). Used in the cases to indicate that nothing specific should be returned (similar to `void`). Here, nothing is to be returned back to Haskell after IO actions

- Using this, we can write our "Hello, World!" program in Haskell:

```
helloWorld :: IO ()
helloWorld = putStr "Hello, World!"
```

Writing values in general (printing)

- Printing can be implemented as follows (very close to the actual definition of `print`):

```
myprint ::  Show a => a -> IO ()
myprint s = putStrLn (show s)
```

where `putStrLn` is defined as

```
putStrLn ::  String -> IO ()
putStrLn st = putStr (st ++ "\n")
```

- Returning a value (by the built-in command `return`):

```
return ::  a -> IO a
```

Return nothing: `return ()`

The Main program

- If we compile a Haskell project using GHC, then it produces executable program, which runs a function :

```
main :: IO t
```

  for some type `t`
- Often, nothing is returned:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

- By default, the `main` program is expected to be in the `Main` module
- Compiling and running the main program (in the module helloworld.hs):

```
> ghc --make helloworld
> ./helloworld
```

The `do` notation

- The `do` notation is used to build IO programs from those and similar primitives we had so far
- In general, it supports sequencing simple IO programs (i.e., "glue together" several IO actions into one)
- The `do` notation also allows to capture (name) the values returned by IO actions
- This makes `do` expression appear like a simple imperative program, containing a sequence of commands and assignments

The do notation (cont.)

- Combining inputs and outputs:

```
read2lines ::  IO ()
read2lines = do
    getLine
    getLine
    putStr "Two lines read."
    putStr "\n"
```

To put several IO actions in one line, use ";"

The do notation (cont.)

- Capturing the read values:

```
reverse2lines ::  IO ()
reverse2lines = do
    line1 <- getLine
    line2 <- getLine
    putStrLn (reverse line2)
    putStrLn (reverse line1)
```

Similar to variable assignments, however, each 'var <- ' creates a new variable var. Therefore, a single assignment, not updatable assigment

The do notation (cont.)

- Local definitions in a do expression:

```
reverse2lines ::  IO ()
reverse2lines = do
    line1 <- getLine
    line2 <- getLine
    let rev1 = reverse line1
    let rev2 = reverse line2
    putStrLn rev2
    putStrLn rev1
```

Using let constructs to introduce local identifiers

Loops and recursion

- Looping is achieved via recursion within the `do` construct:

```
copy ::  IO ()
copy = do
    line <- getLine
    putStrLn line
    copy
```

Running `copy` within GHCI ⇒ looping forever; it can be interrupted by Ctrl-C

Loops and recursion (cont.)

- We can control the number of lines by passing the number as a parameter:

```
copyN ::  Integer -> IO ()
copyN n =
    if n <= 0 then
      return ()
    else do
      line <- getLine
      putStrLn line
      copyN (n-1)
```

Similar to while loop (only by recursion)

Loops and recursion (cont.)

- We can also terminate the loop by checking a condition on data:

```
copyEmpty ::  IO ()
copyEmpty = do
    line <- getLine
    if line == "" then
      return ()
    else do
      putStrLn line
      copyEmpty
```

Note: embedded `do` constructs; Anywhere we need to sequence IO actions, the `do` constructs are used