Lecture 7 – Input and Output

Violet Ka I Pun

violet@ifi.uio.no

Learn You a Haskell for Great Good!
Chapter 9

http://learnyouahaskell.com/input-and-output

Haskell is purea pure functional language

Functional: (Haskell)

- Define what “something” is
- Cannot change the state of a program (or world):
  e.g.: cannot change the value of a variable
  ⇒ no side-effect
- Functions always return the same value with the same parameters
- good for reasoning

Imperative:

- Gives a sequences of steps (operations) to a computer to execute
- Can change the state of a program (or world)
  ⇒ has side-effects
- Methods may return different values with the same parameters depending on the state

Communicating to the outside world

How to see what's going on inside a program?

Usually, we see it through the screen for IP...

For FP, change the screen $\Rightarrow$ change of state

Haskell has a system to handle functions which have side-effect.

IO is special case of side-effects

IO is special case of side-effects "in the outside world"

- `f :: State TypeA -> StateIO TypeB`
- `IO` refers that `f(a)` has a side-effect, while TypeB is what `f` "actually" returns. Usually, it means nothing when TypeB = empty tuple = (), while the function returns a null tuple `() :: ()`.
- `import System.IO` at the top of the program.
- "Functions" with `IO` "actions", e.g.:

```
putChar  ::               Char  ->  IO ()
putStr   ::             String  ->  IO ()
putStrLn ::             String  ->  IO ()
print    :: Show a  =>  a       ->  IO ()
getChar  ::                         IO Char
getLine  ::                         IO String
```

!! In general, getChar == getChar is not valid

- putStr ("Welcome" ++ getLine) .. is not allowed – because ill-typed

  Actions cannot be mixed with/called by (usual) functions.

IO: result and sequencing of actions

- <- creates as action which
  – remove IO and
  – extracts the action's "actual result"
- action :: IO TypeB
  res <- action
  res :: TypeB
- res <- getLine

IO: result and sequencing of actions

- main = do

```
putStrLn "What is your name"
name <- getLine
putStrLn ("Welcome " ++ name)
```

  This is the normal sequence as in imperative programs

do collects a sequence of actions (not functions!)
and turns them into one

$\rightarrow$ returns the type (IO-type) of the last action in the list.

- ▶ To run the actions (do-sequences), we have to put them in a unique main-block, which is executed in the loaded program.
- ▶ main-block usually is of type `IO someType`
  `someType` is a concrete type

IO: sequencing of actions

```
import Data.Char

main = do
    putStrLn "Your first name?"
    firstname <- getLine
    putStrLn "Your last name?"
    lastname <- getLine
    putStrLn ($"Hei, " ++(map toUpper firstname)++(map toUpper
  lastname)++ ". Everything's fine?")
```

- a <- something
  – only (and always) when 'something' is an action (with IO side-effect)

IO: sequencing of actions

```
import Data.Char
  main = do
        putStrLn "Your first name?"
        firstname <- getLine
        putStrLn "Your last name?"
        lastname <- getLine
        let upFirst = map toUpper firstname
            upLast = map toUpper lastname without 'in'
        putStrLn ($"Hei, " ++upFirst++upLast++ ". Everything's fine?")
```

- a <- something
  – only (and always) when 'something' is an action (with IO side-effect)
- let a = something-else (without 'in')
  – only (and always) when 'something-else' is a usual function

More action syntax

- do sequences actions – like nothing else FP
- actions can be combined in usual control structures: if, case, recursion, etc.

```
main = do
    putStrLn "What to do?"
    c <- getLine
    if (not (c == " ")) then
      if (c == "a") then do
        print ("The character is "++c)
        main
      else do
        print ("I don't understand the character "++c)
        main
    else return ()
```

More action syntax – return x

- the use of `return x` in Haskell is different from other languages
- `return x` creates an IO action out of the value x; can be thought as `return :: t -> IO t`

```
main = do
        return ()
        return "first"
        line <- getLine
        return line
        putStrLn ("return " ++ line)
```

What does the program do?

`return x` does not terminate the do-sequences.

Some examples

```
main = do putStrLn "Hello, "
  putStr "how "
  putStr "are you?" ................................ Hello,
                                                     how are you?
main = do putChar 'h'
   putChar 'e'
   putChar 'y' ............................................. Hey

main = do
   c <- getChar
   if c /= ' '
   then do
     putChar c
     main
   else return ()

> Hello, how are you ................................. Hello,
```

Run the program

- Using the interpreter (GHCi)
  `ghci` *file*`.hs`
  `*Main> main`

- First compile then run
  `ghc --make` *file*
  `./`*file*

- Execute on the fly
  `runhaskell` *file*`.hs`

Some functions

```
words ::                                      String -> [String]
words "a"...........................................["a"]
words "Hello, world" ....................["Hello,","world"]

unwords ::                                    [String] -> String
unwords ["a"] ....................................."a"
unwords ["ha", "ha"] .............................."ha ha"
```

E.g.:

```
main = do
  putStrLn Enter a line:"
  ln <- getLine
  let newln = map (++"!") $ words ln
  putStrLnprint $ unwords newln
```

```
> hey hi hello.......................... hey! hi! hello!
> hey hi hello......................... "hey! hi! hello!"
```

More functions

```
unlines ::                                    [String] -> String
unlines ["a"] .................................... "a\n"
unlines ["a",b","c"] .......................... "a\nb\nc\n"

intercalate ::                           [a] -> [[a]] -> [a]
intercalate " "["a",b","c"] ......................... "a b c"
intercalate "\n" ["a",b","c"] ..................... "a\nb\nc"

lines ::                                   String -> [String]
lines "a" ........................................ ["a"]
lines "a\n" ...................................... ["a"]
lines "How\nare\nyou?" ............... ["How", "are", "you?"]
lines "How are you?" ..................... ["How are you?"]
```

Example: lines and unlines

```
main = do
  let ln = "hey\nhi\n"
      newln = map (++"!") $ lines ln
  putStrLn $ unlines newln ............................ hey!
                                                        hi!
  print $ unlines newln ...................... "hey!\nhi!\n"
```

Files

main = do
putStrLn "Filename?"
filename <- getLine
file <- openFile filename ReadMode
content <- hGetContents file
...
hClose file

- openFile :: FilePath -> Mode -> IO Handle, where
  FilePath :: String

data Mode = ReadMode | WriteMode | AppendMode | ReadWriteMode

- ...
  content <- readFile filename
  ...
- readFile :: FilePath -> IO String

Write to files

- file <- openFile filename WriteMode / ReadWriteMode / AppendMode

  gives a "handle" to the file
- To write to *file* (of type Handle):

  hPutStr, hPutStrLn :: Handle -> String -> IO ()

  and must be closed at the end: hClose file
- writeFile, appendFile :: FilePath -> String -> IO ()
  open/close the file in the background

  (writeFile delete everything it finds in the file from before)

Write to files: read a pair of numbers and write to an file

```
import System.IO
import Data.Char
main = inter []
inter pair = do
      putStr "f name / x y / q / s: "
      cmd <- getLine
      let (h:xs) = words cmd
      if (h == "f") then do
         writeFile (last xs) (pairString pos)
         inter pair
      else if (isInt h) then
         inter ((h, last xs):pair)
      else if (h == "s") then do
         putStrLn (pairString pair)
         inter pair
      else return ()
isInt s:_ = isDigit s
pairString [] = ""
pairString ((x,y):xs) = x++" "++y ++" "++pairString xs
```

Example: file handling

```
main = do
  putStr ``create / read / write / delete / rename / quit''
  s <- getLine
  putStr ``Enter a file name''
  fn <- getLine
  case s of
    ``create'' -> do doCreate fn; main
    ``read'' -> do doRead fn; main
    ``write'' -> do doWrite fn; main
    ``delete'' -> do doDelete fn; main
    ``rename'' -> do doRename fn; main
    ``quit'' -> return ()
    _ -> do putStrLn ``unknown command''; main
```

Example: file handling

```
doCreate fn = writeFile fn some strings

doRead fn = do
  contents <- readFile fn
  putStrLn contents

doWrite fn = appendFile fn some strings

doDelete fn = removeFile fn                          System.Directory

doRename fn = renameFile fn new name                 System.Directory

doReadFirstLine fn = do
  contents <- readFile fn
  putStrLn $ head $ lines contents

doReadFirstWord fn = do              read the first charater in each line
  contents <- readFile fn
  let hl = map head $ map words $ lines content
  putStrLn $ unlines hl
```

Example: modify a file

```
doMoveLnToEnd fn = do                         move the first line to the end
  contents <- readFile fn
  let (fl:rest) = lines content
      newContent = rest++[fl]
  writeFile fn newContent ................................. error

doMoveLnToEnd2 fn = do
  file <- openFile fn ReadMode
  content <- hGetContents file
  let (fl:rest) = lines content
      newContent = rest++[fl]
  hClose file
  file2 <- openFile fn WriteMode
  hPutStr file2 newContent
  hClose file2 ............................................ error
```

Example: modify a file (cont)

```
doMoveLnToEnd3 fn = do
  file <- openFile fn ReadMode
  content <- hGetContents file
  let (fl:rest) = lines content
      newContent = rest++[fl]
  (fn2, file2) <- openTempFile "." " temp"                System.IO
  hPutStr file2 newContent
  hClose file
  hClose file2
  removeFile fn
  renameFile fn2 fn
```

```
openTempFile :: FilePath -> String -> IO (FilePath, Handle)
```

What about if fn does not exist?

```
doesFileExist :: FilePath -> IO Bool                System.Directory
```

exist <- doesFileExist fn

if (exist) then ... else return ()

Arrays in FP

Lists in Haskell:

- Access the i-th element
- Traverse the list until index i . . . . i steps
- linear time

Haskell also has arrays

- Like functions: from indices to values
- Can access any element at constant time
- `import Data.Array`

Array creation – array

```
a = array (1,5) [(1,1),(2,4),(3,9),(4,16),(5,25)]
a :: Array Int Int(Ix i, Num e, Num i) => Array i e
```

```
array :: Ix i => (i,i) -> [(i,e)] -> Array i e
(i,i):    bounds of an array
          a pair of indices
[(i,e)]: a list of pairs of indices and values
          an association list
```

Usually, use list comprehension to define an array:

```
a = array (1,5) [ (i, i*i) | i <- [1..5]]
class (Ord a) => Ix i where ...
```

```
m = array ((0,0),(1,2)) [((i,j), i*j) | i<-[0..1], j<-[0..2]]
  = array ((0,0),(1,2)) [((0,0),0), ((0,1),0), ((0,2),0),
                         ((1,0),0), ((1,1),1), ((1,2),2)]
```

```
a = array (1,3) [(1,1),(2,4),(3,9)] a = listArray (1,3)
[1,4,9]
  = array (1,3) [(1,1),(2,4),(3,9)]
```

```
listArray :: Ix i => (i, i) -> [e] -> Array i e
```

Some examples:

```
listArray (1,3) [4,2,3]
        == array (1,3) [(3,4),(1,2),(2,3)] ................ False
array (1,3) [(i,i+1)| i <- [1..3]]
        == array (1,3) [(3,4),(1,2),(2,3)] ................. True
a == array (1,3) [(1,1),(2,4),(3,10),(3,9)] .................. True
a == array (1,3) [(1,1),(2,4),(3,9),(2,8)]array (1,3)
[(1,1),(2,8),(3,9)]................................... False
```

More about indices Ix

```
class (Ord a) => Ix i where
  range :: (a,a) -> [a]
  index :: (a,a) -> a -> Int
  inRange :: (a,a) -> a -> Bool
```

Examples:

```
range (0,4) ...................................... [0,1,2,3,4]
range ((0,0),(1,1)) .................. [(0,0), (0,1),(1,0),(1,1)]
range ('m','p') ............................................ "mnop"
inRange (-50,60) 35 ...................................... True
inRange ('m','p') 'a' .................................... False
index (1,9) 2 ................................................ 1
index ((0,0),(1,1)) (1,1) .................................... 3
```

Some functions on arrays

```
a = array (1,5) [(1,1),(2,4),(3,9),(4,16),(5,25)]
m = array ((0,0),(1,2)) [((0,0),0), ((0,1),0), ((0,2),0),
                         ((1,0),0), ((1,1),1), ((1,2),2)]

(!) ::                        Ix i => Array i e -> i -> e
a ! 5 ....................................................... 25
m ! (1,1) ................................................... 1

bounds ::                     Ix i => Array i e -> (i,i)
bounds a .................................................. (1,5)
bounds m ............................................ ((0,0),(1,2))

indices ::                    Ix i => Array i e -> [i]
indices a ........................................... [1,2,3,4,5]
indices m ................. [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]

elems ::                      Ix i => Array i e -> [e]
elems a ............................................ [1,4,9,16,25]
elems m ............................................ [0,0,0,0,1,2]

(//) ::              Ix i => Array i e -> [(i,e)] -> Array i e
a // [(3,18)] ..... array (1,5) [(1,1),(2,4),(3,18),(4,16),(5,25)]
m // [((0,0),10)] ...... array (...) [((0,0),10), ..., ((1,2),2)]
```

Examples using arrays

Fibonacci numbers
– given a number, calculate a sequence of fibonacci numbers

```
fibs :: Int -> Array Int Int
fibs n = fa where
  fa = array (0,n) ([(0, 1), (1, 1)] ++
                    [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

1. Given a number, produce a matrix in which the elements of the 1st row and 1st column all have the value 1.
2. The value of all other elements is the sum of their neighbours on W, NW, and N.

```
f :: Int -> Array (Int,Int) Int
f n = a where
  a = array ((1,1),(n,n))
        ([((1,j), 1) | j <- [1..n]] ++
         [((i,1), 1) | i <- [2..n]] ++
         [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j))
                   | i <- [2..n], j <- [2..n]])
```