Input/Output in Haskell

COMP2209 - Programming III

Dr Julian Rathke
Black Box Functions

• One of the core design principles of Haskell is that of using pure functions

• A function can be thought of in black box terms as something that takes an input value and produces an output value.

• A **pure** function is such that the output value depends solely on the input value and not some other external condition.

• If that is true then the following should hold:
  • No matter how many times you do so, or at what point in program execution, if you call function $f$ on input $v$ then you should always get the same output.

• This is clearly not true for methods in Java -
  • Methods may access heap stored values that change during execution
  • Methods may read input from files etc
  • Method may generate random output

We call these things “side-effects”
Functions with side-effects

• The name “side-effect” is terribly misleading as it indicates that the side-effect is something incidental to the main event of calculating the function.
  • Input/Output often is the main event!
• So the question is then, how do we allow for side-effects in a language of pure functions?
• It’s all a matter of reconsidering what the actual inputs to the function are. e.g. take a function that affects the heap

We can think of the state of the system an input to the function to be modified and returned as output

input → heap → output

input → output

heap

input

output

heap

heap

this can be rewritten as
You can change the World

• So a general view of an IO based function is a function that transforms a representation of the system state.
  • e.g. write characters to a file, read characters from stdin
• Suppose we have a type World that represents system state
• Then we could define a type as such a transformation
  
  \[
  \text{type IO} = \text{World} \rightarrow \text{World}
  \]

• However, functions need to return values so we should allow for that in an IO based function
  
  \[
  \text{type IO a} = \text{World} \rightarrow (a,\text{World})
  \]

• And functions need to take arguments so we need to allow for that. By Currying though we can do that immediately
  
  \[
  \text{Char} \rightarrow \text{IO a} \quad \text{expands to} \quad \text{Char} \rightarrow \text{World} \rightarrow (a,\text{World})
  \]

• is the type of a function that accepts a Char, the current World and returns an a with a modified World
IO as a built-in

• Of course we don’t have such a type as World and it would be infeasible to package up such a thing as a Haskell value.

• But the type \texttt{data IO a = …}\n
does exist in Haskell as a built-in type.

• It is important to understand the difference between a computation of type \texttt{Int}, say, and \texttt{IO Int}.

• The former is a pure computation - an integer - that will be the same integer every time you evaluate the computation.

• The latter is a \texttt{function} that, given the system state, will produce an integer and a modified system state.

  • The integer produced may depend on the system state and hence may be different each time the function is evaluated.

  • There is an element of purity retained though - if the same system state were given as input then the same answer would be produced.
Actions

• Haskell expressions of type \texttt{IO \ a} are called \texttt{actions}

• The three most basic actions are Char based actions that use the standard input/output

• Read a single character from stdin

\texttt{getChar :: IO Char}

• Take a Char and return an action that writes that Char to standard output.

\texttt{putChar :: Char \rightarrow IO ()}

• Recall that () is the unit type containing just a single value.

• Take a pure expression with no IO and casts it in to an expression with IO - it simply leaves the World unaffected.

\texttt{return :: a \rightarrow IO a}

• This is surprisingly useful as it allows us to map non side-effecting computations in to the world of side-effecting ones.
Sequencing Actions

• Single actions provide the very basics of IO but are limited in their use.

• By combining actions into sequences of actions we obtain better expressive power.

• For example, we could create a `putStr` function that writes a string to stdin by calling `putChar` several times.

• But in Haskell, we “sequence” evaluation of expressions essentially by function composition
  - e.g. `map (+1) . sort . removeDuplicates`

    is read as performing this function, then that one, etc.

• However, for actions, all of type `IO a` (for some `a`) the types are not correct for using function composition.

• We require something similar but for `IO a` types
Do notation in Haskell

• There is a syntactic sugar (more on this later) in Haskell that allows us to do exactly this.

• The general form is as follows

```
  do p₁ <- act₁
    p₂ <- act₂
    .
    .
    .
    pₙ <- actₙ
  act_Final
```

• We list actions act₁ to actₙ and allow any result of the action to be pattern matched and bound to p₁ to pₙ

• The final action act_Final must not be bound and is typically a return e action.

• The layout rule applies so each action must be column aligned

• If no value is required for action actᵢ the “pᵢ <-“ binder may be omitted.

• The type of the overall expression is the type of the final action.
Example of doing a do

- Let’s write a simple example where we create an IO action that reads three characters from standard input and return the first and third of these:

```haskell
firstThird :: IO (Char, Char)
firstThird = do x <- getChar
                getChar
                y <- getChar
                return (x, y)
```

Although this looks very much imperative in style we’ll see in the next lecture how this is really some sort of function composition and it is important to remember that lazy evaluation still applies.
Deriving actions using sequence

Let’s make use of “do” notation now to derive some new actions

Write a string to standard out:

```
putStr :: String → IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
              putStr xs
```

Write a string to standard out with new line:

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                putChar '\n'
```
Deriving actions using sequence

Read a line from standard in:

```haskell
getLine :: IO String
getLine = do x <- getChar
  if x = '\n' then return []
  else do xs <- getLine
    return (x:xs)
```

This expression has type `IO String` and thus is an action.

These actions are sequenced in to a single action due to nesting and layout.
Main function in Haskell

• You may have noticed that we haven’t yet actually compiled any Haskell code.

• We’ve used GHCi - the Haskell interactive interpreter a lot.

• To write a Haskell program that compiles using GHC we need to provide a function called main

• The type of main must be IO ()

• That is, a function that returns no significant value but is executed for its side-effects.

• So to see any effect of evaluating functions in a compiled program you must actually output something!
Files in Haskell

- In the standard Prelude, Haskell provides basic actions for reading and writing files

\[
\text{type FilePath} = \text{String}
\]

- Read contents of a file

\[
\text{readFile} :: \text{FilePath} \rightarrow \text{IO String}
\]

- Write to a file

\[
\text{writeFile} :: \text{FilePath} \rightarrow \text{String} \rightarrow \text{IO ()}
\]

- Add to the contents of a file

\[
\text{appendFile} :: \text{FilePath} \rightarrow \text{String} \rightarrow \text{IO ()}
\]

Reading files is done lazily; files are only partially read in to memory as needed.

Remember to convert values to Strings using show to write in to files.
System.IO

• The module System.IO also has a collection of functions for reading and writing files etc
• It introduces the IO type (actually a Monad cf. next lecture)
• It introduces the Handle type as an addressing mechanism for IO also
  • stdin, stdout, stderr are all Handles
  • openFile returns a Handle
• There are analogous functions to getChar, getLine etc where the IO handle to be used is given as an argument
  • These are mostly called hGetChar, hGetLine etc
  • e.g. getChar = hGetChar stdin
  • e.g. putStrLn = hPutStrLn stdout
Random actions

• Another interesting action in Haskell is that of random number generation.

• Remember - pure functions return the same result every time you apply them to the same value.

• This is not true of a (good) random number generator.

• The module System.Random provides a fairly complicated framework for random numbers that allows for different generator algorithms.

• To use it in its simplest form, import System.Random and use:
  • For a random number within the range of the type determined by context: \( \text{randomIO} :: \text{IO} \ a \)
  • For a random number in the given range: \( \text{randomRIO} :: (a,a) \rightarrow \text{IO} \ a \)
Writing a simple interactive game

• We’ll put together what we have learned so far to write a simple interactive game: Hangman
• One player types in a secret word
• The other player guesses a character that they think may appear in that word
  • If it does then the character is displayed in the word at all the positions it appears
• The game ends when all characters have been guessed.

Top Down Design: first thing we need is a main method!

```
main = hangman
hangman :: IO ()
hangman = ..
```
The hangman method

- We need to output the initial message
- We need to read the secret word - so we must disable echoing of characters to screen on standard out
  - Use hSetEcho from System.IO for this
- We need to prompt the user to input
- We need to enter a “play loop” where repeated guesses are made and checked.
  - The play loop needs to know:
    - The secret word to be guessed - a string
    - The characters correctly guessed so far - we can record this as a string with dummy characters in place of unknown characters
The hangman method

```haskell
hangman :: IO ()
hangman = do putStrLn "Think of a secret word:"
            word <- secretGetLine
            putStrLn "Enter characters to guess it:"
            play word (replicate (length word) '-')
```

Need to implement these functions next

Just create a string of ‘-’s the same length as the inputted word for the characters guessed so far.
secretGetLine and getCharHidden

**secretGetLine :: IO String**

```haskell
secretGetLine = do x <- getCharHidden
  if x=='\n' then
    do putChar '\n'
    return []
  else
    do putChar '-'
    xs <- secretGetLine
    return (x:xs)
```

**getCharHidden :: IO Char**

```haskell
getCharHidden = do hSetEcho stdin False
  x <- getChar
  hSetEcho stdin True
  return x
```

- Use the input method with no echoing
- Newline indicates word completed.
- Finish echoing ‘-’s with a newline.
- Echo ‘-’s to indicate the length of word entered.
- Switch echo off/on
Implementing play

- We now look at the main play loop

```haskell
play :: String -> String -> IO ()
play word answerSoFar =
    if answerSoFar == word then
        putStrLn "Correct!!"
    else
        do putStrLn "Enter a character: "
            guess <- getChar
            updatedAnswer <- putUpdate (updateMatch word answerSoFar guess)
        play word updatedAnswer
```

Loop using recursion - pass the answer so far as “state”

Just remains to define `updateMatch`

```
putUpdate :: String -> IO String
putUpdate s = do putStrLn "Your answer so far is: "
                putStrLn s
                return s
```
Implementing updateMatch

- Notice that this function does not involve any I/O
- This is reflected in the type - it is a pure function!

```haskell
updateMatch :: String -> String -> Char -> String
updateMatch [] [] c = []
updateMatch (x:xs) (y:ys) c | x==y = x : updateMatch xs ys c
updateMatch (x:xs) (y:ys) c | x==c = x : updateMatch xs ys c
updateMatch (x:xs) (y:ys) c | otherwise = '-' : updateMatch xs ys c
```

If this char has previously been guessed then copy it to the updated answer.

If this char is currently being guessed then copy it to the updated answer.

Otherwise the current guess does not match this character so this character is still unknown in the updated answer.
YOUR QUESTIONS

Next Lecture:
Applicatives