MATHFUN Lecture FP8-9
Input/Output

Matthew Poole
moodle.port.ac.uk

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- So far, all our Haskell programs have been self-contained; i.e without interaction with the user.
- Clearly, input/output (I/O) is an essential property of all realistic programs.

Introduction to Lecture

- So far, all our Haskell programs have been self-contained; i.e without interaction with the user.
- Clearly, input/output (I/O) is an essential property of all realistic programs.
- In this lecture we will consider I/O in Haskell:
  - we begin by considering why I/O is problematic for functional programming; then
  - we discuss the practical details of writing I/O-based programs in Haskell, through a series of examples.

Introduction to Lecture

- So far, all our Haskell programs have been self-contained; i.e without interaction with the user.
- Clearly, input/output (I/O) is an essential property of all realistic programs.
- In this lecture we will consider I/O in Haskell:
  - we begin by considering why I/O is problematic for functional programming; then
  - we discuss the practical details of writing I/O-based programs in Haskell, through a series of examples.
- This lecture is supported by Chapter 8 of Thompson's book.

Why is input/output a problem?

- Why is input/output, although simple in imperative programming, problematic for functional programming?
- We have seen that the fundamental building block of functional programming is the **function definition**.
- An example function definition, from lecture FP1, is:

```
square :: Int -> Int
square n = n * n
```

- The most important property of a function is that it always gives the same result when given the same arguments.
- i.e., if `x = y` then `square x = square y`.
- This property is known as **referential transparency**.

Why is input/output a problem?

- Referential transparency allows us to more easily reason about program code (both formally and informally).
- For example, for any expression `e` that evaluates to a number:

```
e - e
```

will **always** evaluate to 0.

Why is input/output a problem?

- Referential transparency allows us to more easily reason about program code (both formally and informally).
- For example, for any expression `e` that evaluates to a number:

```
e - e
```

  will **always** evaluate to 0.
- Furthermore, it's quite easy to **prove** that a function such as:

```
fact n
  | n > 0     = n * fact (n - 1)
  | n == 0    = 1
```

Why is input/output a problem?

- Referential transparency allows us to more easily reason about program code (both formally and informally).
- For example, for any expression `e` that evaluates to a number:

```
e - e
```

will **always** evaluate to 0.

- Furthermore, it's quite easy to **prove** that a function such as:

```
fact n
  | n > 0     = n * fact (n - 1)
  | n == 0    = 1
```

is **correct** (i.e. gives the factorial of n for any n).

Why is input/output a problem?

- Referential transparency allows us to more easily reason about program code (both formally and informally).
- For example, for any expression `e` that evaluates to a number:

  ```
  e - e
  ```

  will **always** evaluate to 0.
- Furthermore, it's quite easy to **prove** that a function such as:

  ```
  fact n
    | n > 0     = n * fact (n - 1)
    | n == 0    = 1
  ```

  is **correct** (i.e. gives the factorial of n for any n).
- It is much more difficult to prove correct a factorial function written in an imperative programming language using a loop.

Why is input/output a problem?

- The approach to I/O taken by some functional languages is to provide "functions" such as:

  ```
  inputInt :: Int
  ```

  to read an integer value (e.g. from the keyboard) and to return the value read.

Why is input/output a problem?

- The approach to I/O taken by some functional languages is to provide "functions" such as:

  ```
  inputInt :: Int
  ```

  to read an integer value (e.g. from the keyboard) and to return the value read.
- This approach breaks referential transparency; for example, the expression:

  ```
  inputInt - inputInt
  ```

  is no longer guaranteed to evaluate to 0.

Why is input/output a problem?

- The approach to I/O taken by some functional languages is to provide "functions" such as:

  ```
  inputInt :: Int
  ```

  to read an integer value (e.g. from the keyboard) and to return the value read.
- This approach breaks referential transparency; for example, the expression:

  ```
  inputInt - inputInt
  ```

  is no longer guaranteed to evaluate to 0.
- The fact that `inputInt` returns different values each time is due to the **side effect** of reading a new value from the keyboard.

Haskell's approach to I/O

- Since any function in a functional program might include an `inputInt`, the whole program becomes difficult to understand.

Haskell's approach to I/O

- Since any function in a functional program might include an `inputInt`, the whole program becomes difficult to understand.
- Haskell provides a different approach to input/output.
- This approach is known as the **monadic approach** since it is based on the mathematical concept of a monad.

Haskell's approach to I/O

- Since any function in a functional program might include an `inputInt`, the whole program becomes difficult to understand.
- Haskell provides a different approach to input/output.
- This approach is known as the **monadic approach** since it is based on the mathematical concept of a monad.
- Input/output is viewed as a **sequence** of **actions** (or programs) that happen in **sequence**.

Haskell's approach to I/O

- Since any function in a functional program might include an `inputInt`, the whole program becomes difficult to understand.
- Haskell provides a different approach to input/output.
- This approach is known as the **monadic approach** since it is based on the mathematical concept of a monad.
- Input/output is viewed as a **sequence** of **actions** (or programs) that happen in **sequence**.
- Haskell provides the types:

```
IO a
```

of I/O actions or programs of type a.

Haskell's approach to I/O

- A value of type `IO a` is an action that, when executed:
  - performs some input/output; and then
  - returns a value of type a.

Haskell's approach to I/O

- A value of type `IO a` is an action that, when executed:
  - performs some input/output; and then
  - returns a value of type a.
- Haskell also provides a mechanism for sequencing actions; i.e. to allow them to execute one after another.
- In effect, it provides a simple imperative programming language for writing I/O programs.

Haskell's approach to I/O

- A value of type `IO a` is an action that, when executed:
  - performs some input/output; and then
  - returns a value of type a.
- Haskell also provides a mechanism for sequencing actions; i.e. to allow them to execute one after another.
- In effect, it provides a simple imperative programming language for writing I/O programs.
- Typically, programs in Haskell thus comprise some:
  - function definitions (free from the problems of I/O); and
  - I/O programs.

Haskell's approach to I/O

- A value of type `IO a` is an action that, when executed:
  - performs some input/output; and then
  - returns a value of type a.
- Haskell also provides a mechanism for sequencing actions; i.e. to allow them to execute one after another.
- In effect, it provides a simple imperative programming language for writing I/O programs.
- Typically, programs in Haskell thus comprise some:
  - function definitions (free from the problems of I/O); and
  - I/O programs.
- These imperative I/O programs are really an illusion: they are actually "syntactic sugar" for (purely functional) expressions.

Reading input

- There are two built-in (Prelude) actions for reading input.
- Firstly, the action:

  ```
  getLine :: IO String
  ```

  reads a line from standard input (the keyboard).
- (The type says that `getLine` does some IO and then returns a string.)

Reading input

- There are two built-in (Prelude) actions for reading input.
- Firstly, the action:

  ```
  getLine :: IO String
  ```

  reads a line from standard input (the keyboard).
- (The type says that `getLine` does some IO and then returns a string.)
- Secondly, the action:

  ```
  getChar :: IO Char
  ```

  reads a single character from standard input.

Writing output

- Performing output is different from input in that we do not expect output actions to return results.
- However, I/O programs have to be of type `IO a`, for some `a`.

Writing output

- Performing output is different from input in that we do not expect output actions to return results.
- However, I/O programs have to be of type `IO a`, for some a.
- Haskell provides a **one-element** type called `()`, which contains the single value `()`. Question: what is `()` really?

Writing output

- Performing output is different from input in that we do not expect output actions to return results.
- However, I/O programs have to be of type `IO a`, for some `a`.
- Haskell provides a **one-element** type called `()`, which contains the single value `()`. Question: what is `()` really?
- We use the `()` type to denote that an I/O program doesn't return anything of interest.
- For printing strings, Haskell includes the function:

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

- There is also a version:

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

  that adds a newline after the outputted string.

Writing output

- For example, the Haskell "Hello, World!" program is

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

- This example illustrates the **starting point** of a complete Haskell program: an action of type `IO ()` called `main`.

Writing output

- For example, the Haskell "Hello, World!" program is

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

- This example illustrates the **starting point** of a complete Haskell program: an action of type `IO ()` called `main`.
- Haskell also provides a polymorphic function:

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

  that will display data of any type that is an instance of the `Show` class (i.e. any value that can be converted to a string).

The do notation

- We now show how Haskell allows IO actions to be sequenced.
- As a simple example, suppose we wish to write a program that:
  - asks the user the enter any string;
  - reads in (but does not store) the string;
  - displays a "done" message.

The do notation

- We now show how Haskell allows IO actions to be sequenced.
- As a simple example, suppose we wish to write a program that:
  - asks the user the enter any string;
  - reads in (but does not store) the string;
  - displays a "done" message.
- Such a program includes a sequence of three actions, and we must thus use the do notation as in:

```
readALine :: IO ()
readALine = do
    putStrLn "Enter a string"
    getLine
    putStrLn "Done"
```

- Note that each line consists of an action of type `IO a` for some a.

Capturing inputted values

- Clearly, we'll usually want to "capture" the input values, and to somehow make the output depend on the input.

Capturing inputted values

- Clearly, we'll usually want to "capture" the input values, and to somehow make the output depend on the input.
- To do this, we **name** the value returned by `getLine`; our program might become:

```
readALine :: IO ()
readALine = do
    putStrLn "Enter a string"
    line <- getLine
    putStrLn ("Done reading " ++ line)
```

Capturing inputted values

- Clearly, we'll usually want to "capture" the input values, and to somehow make the output depend on the input.
- To do this, we **name** the value returned by `getLine`; our program might become:

```
readALine :: IO ()
readALine = do
    putStrLn "Enter a string"
    line <- getLine
    putStrLn ("Done reading " ++ line)
```

- The `<-` here appears to operate like an assignment (= in Java), although this analogy is not fully correct.
- For example, we cannot update the value of `line` using a subsequent action.

Example: reading integers

- We can write our own action:

  ```
  getInt :: IO Int
  ```

  for reading an integer from standard input.

Example: reading integers

- We can write our own action:

  ```
  getInt :: IO Int
  ```

  for reading an integer from standard input.

- To do this, we first need to use `getLine` to read in a string:

  ```
  do str <- getLine
  ```

Example: reading integers

- We can write our own action:

  ```
  getInt :: IO Int
  ```

  for reading an integer from standard input.
- To do this, we first need to use `getLine` to read in a string:

  ```
  do str <- getLine
  ```

- We then need to translate `str` into an integer using the `read` function declared in the `Read` type class:

  ```
  read str :: Int
  ```

  (we need to force conversion to an `Int` using `:: Int`).

Example: reading integers

- We can write our own action:

  ```
  getInt :: IO Int
  ```

  for reading an integer from standard input.

- To do this, we first need to use `getLine` to read in a string:

  ```
  do str <- getLine
  ```

- We then need to translate `str` into an integer using the `read` function declared in the `Read` type class:

  ```
   read str :: Int
  ```

  (we need to force conversion to an `Int` using `:: Int`).

- As an example of `read`:

  ```
  read " 456    " :: Int
  456
  ```

Example: reading integers

- Q: What is the type of `read str :: Int` ?
- It needs to be converted to an action (Q: of what type?) in order for it to appear within a `do` construct.

Example: reading integers

- Q: What is the type of `read str :: Int` ?
- It needs to be converted to an action (Q: of what type?) in order for it to appear within a `do` construct.
- The built-in function:

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

  is defined such that `return x` does no I/O, but simply returns `x`.

Example: reading integers

- Q: What is the type of `read str :: Int` ?
- It needs to be converted to an action (Q: of what type?) in order for it to appear within a `do` construct.
- The built-in function:

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

  is defined such that `return x` does no I/O, but simply returns `x`.
- Putting these pieces together gives the following function:

  ```
  getInt :: IO Int
  getInt = do
      str <- getLine
      return (read str :: Int)
  ```

File I/O

- The Prelude contains the following functions for reading and writing to/from files:

```
readFile :: String -> IO String
writeFile :: String -> String -> IO ()
appendFile :: String -> String -> IO ()
```

File I/O

- The Prelude contains the following functions for reading and writing to/from files:

```
readFile :: String -> IO String
writeFile :: String -> String -> IO ()
appendFile :: String -> String -> IO ()
```

- The first argument of each function gives the file's pathname.

File I/O

- The Prelude contains the following functions for reading and writing to/from files:

```
readFile :: String -> IO String
writeFile :: String -> String -> IO ()
appendFile :: String -> String -> IO ()
```

- The first argument of each function gives the file's pathname.
- The following program makes use of `readFile` to display the contents of a file on the screen:

```
displayFile :: IO ()
displayFile = do
    putStr "Enter the filename: "
    name <- getLine
    contents <- readFile name
    putStr contents
```

Using if ... then ... else ...

- Recall from lecture FP1 that Haskell includes an `if ... then ... else ...` construct (as an alternative to guards).

Using if ... then ... else ...

- Recall from lecture FP1 that Haskell includes an `if ... then ... else ...` construct (as an alternative to guards).
- These can be used within a do construct; e.g the following reads a string from the user and tests whether it is a palindrome:

```
pal :: IO ()
pal = do
    str <- getLine
    if str == reverse str
        then putStr (str ++ " is a palindrome")
        else putStr (str ++ " is not a palindrome")
```

Using if ... then ... else ...

- Recall from lecture FP1 that Haskell includes an `if ... then ... else ...` construct (as an alternative to guards).
- These can be used within a do construct; e.g the following reads a string from the user and tests whether it is a palindrome:

```
pal :: IO ()
pal = do
    str <- getLine
    if str == reverse str
        then putStr (str ++ " is a palindrome")
        else putStr (str ++ " is not a palindrome")
```

- The test is of type `Bool`, and the branches are single actions.
- Note that if one of the branches had two or more actions, then these would need to combined into another `do` construct.

Alternative solution

- An alternative solution to the above program is to move more of the computation to a normal function (i.e. without I/O):

```
isPalindrome :: String -> String
isPalindrome str
  | str == reverse str  = str ++ " is a palindrome"
  | otherwise           = str ++ " is not a palindrome"
```

Alternative solution

- An alternative solution to the above program is to move more of the computation to a normal function (i.e. without I/O):

```
isPalindrome :: String -> String
isPalindrome str
  | str == reverse str  = str ++ " is a palindrome"
  | otherwise           = str ++ " is not a palindrome"
```

  and to simplify the I/O program:

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

- (This example illustrates a typical separation of the purely functional "core" of a program from its user interface code.)

Local definitions in I/O programs

- We could break the final line of `pal` into two parts (i.e. separating the computation from the output) as follows:

```
pal = do
    line <- getLine
    response <- return (isPalindrome line)
    putStrLn response
```

- However, the middle line is ugly—we have had to introduce some IO (i.e. `return`) just so we can use `<-`.
- An better way involves a **local definition** (introduced with `let`):

```
pal = do
    line <- getLine
    let response = isPalindrome line
    putStrLn response
```

Recursion

- Like functions, IO programs can be recursive.
- The following program allows the user to enter several lines, checking whether each one is a palindrome.
- It terminates when the user enters a blank line.

Recursion

- Like functions, IO programs can be recursive.
- The following program allows the user to enter several lines, checking whether each one is a palindrome.
- It terminates when the user enters a blank line.

```
palLines :: IO ()
palLines = do
    putStr "Enter a line: "
    str <- getLine
    if str == "" then
        return ()
    else do
        putStrLn (isPalindrome str)
        palLines
```

Further Functional Programming

- We have now covered enough concepts to enable practical application of the Haskell language (e.g. for the coursework).

Further Functional Programming

- We have now covered enough concepts to enable practical application of the Haskell language (e.g. for the coursework).
- The main topic that is recommended for further study is lazy evaluation (Chapter 17).