G53NSC and G54NSC
Non-Standard Computation
Lab 1 Exercises

Dr. Alexander S. Green

28th January 2010

Introduction

The language we are using for these labs is Haskell. It is recommended that you start using GHCi (part of the Glasgow Haskell Compiler) to run and test your solutions[1]. The Glasgow Haskell Compiler is available online at:
http://www.haskell.org/ghc/

The exercises set in the labs have a firm deadline of 12:00 (midday); Thursday the 1st of April, but it is highly recommended that you submit your work on a weekly basis (E.g. 1 week after the date each exercise sheet is released) to enable you to receive ongoing feedback. I will give feedback for any exercises submitted within 2 weeks of their original release date.

The weekly submissions should be emailed to me (asg@cs.nott.ac.uk), or handed to me in the labs. The final submission of your portfolio will be through the school office by 12:00 (midday) on Thursday the 1st of April (The last day of the Spring term). The final submission through the school office should be made even if you have been submitting work to me on a weekly basis as it is this final submission that counts as your portfolio.

These exercise sheets should be attempted on your own, and at the end of the course, it is these individual submissions that will make up your portfolio project. Combined, the work submitted in your portfolio is worth 50% of the mark for this module (The other 50% consisting of the research report and presentation).

[1] Most of the early exercises will run in Hugs, but when we come to use the Quantum IO Monad later in the course, GHCi will be required

Exercise sheet 1

The following exercises should be attempted.

1. Define a Type (*Bit*) in Haskell that represents the two states a bit can be in[2].

2. A binary number can be thought of as a list of bits (with the least significant bit first). Define a function ($int2bits :: Int \rightarrow [Bit]$) that converts an integer into the corresponding list of bits that represents the binary expansion of the given integer.

   Example: *int2bits* 150 should give a list of bits representing the binary expansion 01101001 (note: the least significant bit is the first bit)

   A Haskell **case** expression will probably be useful here, but isn't the only way of doing it. Other useful library functions include:

   - $even :: Int \rightarrow Bool$, which tests whether an integer is even
   - $div :: Int \rightarrow Int \rightarrow Int$, implements integer division.

3. Implement in Haskell the inverse function to the previous exercise. E.g. write a function ($bits2int :: [Bit] \rightarrow Int$) that converts a given list of bits into the corresponding integer representation.

   Test your previous two answers by checking that the function application $\lambda x \rightarrow bits2int\ (int2bits\ x)$ corresponds to the identity on integers.

4. Implement, in Haskell, a function that computes the factors of a given integer, and write a paragraph explaining the complexity of your solution[3]. Could you have made it more efficient?

5. Functions in Haskell are *pure* so we must make use of Monads to enable us to define effectful computations. All I/O in Haskell takes place in what is known as the *IO* Monad. Monadic programs use a special notation called *do* notation which give monadic computations a more imperative style. To write a simple program that echoes characters to the screen we can make use of the following library functions:

   - $getChar :: IO\ Char$, is an effectful computation that reads in a single character from the keyboard.
   - $putChar :: Char \rightarrow IO\ ()$, is an effectful computation, that has no return value[4], but has the side effect of displaying its argument character to the screen

[2] Type and Constructor names must begin with an Upper-case letter

[3] Make sure you take into account the complexity of any library functions that you may have used

[4] Technically the return value is the unit type ()

Using *do* notation we can write the following *echo* function that binds the result of reading in a character to a variable x which is then used as the argument to the *putChar* function. (note: this program doesn't terminate, so use Ctrl-C in GHCi to break out of its execution)

$$
\begin{aligned}
&\mathit{echo} :: \mathit{IO}\ () \\
&\mathit{echo} = \mathbf{do}\ x \leftarrow \mathit{getChar} \\
&\quad \mathit{putChar}\ x \\
&\quad \mathit{echo}
\end{aligned}
$$

Using the above functions (*getChar* and *putChar*), implement functions (*getString* :: *IO String* and *putString* :: *String* → *IO* ()) that read in (and print out respectively) an entire line of text.

For example, the *getString* function would repeatedly call the *getChar* function until a newline character (`'\n'`) is detected.

6. Implement a function (*echoString* :: *IO* ()) that echoes an entire string to the screen (note: You should make use of the two functions you defined in the previous exercise).

7. Using the random number generator in the IO Monad, implement a probabilistic function that tests whether its argument is a prime number[5], returning *False* if the number isn't prime, or *True* if the number is probably prime.

   Searching online for *Primality Test* is a good starting point for finding an algorithm for this task.

   You will need to **import** the *System.Random* library to make use of the random number generator. Once imported the following function is available:

   - *randomRIO* :: (*Int*, *Int*) → *IO Int*, which is an effectful computation that returns a randomly selected integer in the given range.

8. Who is this man?
   Why might his name have come up whilst you were answering exercises 4 and 7?
   (This last exercise is unassessed)

[5] There are deterministic primality tests, but we are implementing a probabilistic one here as a practise using Monads in Haskell