G53NSC and G54NSC
Non-Standard Computation
Lab 3 Exercises

Dr. Alexander S. Green

11th February 2010

Exercise sheet 3

These exercises carry on from Exercise sheets 1 and 2, and may use some of the types and functions you have previously defined.

The exercises are listed here, but more information can be found in the hints and tips section below.

1. Implement, in Haskell, a function that converts a single qubit state, given as two complex amplitudes, into the corresponding point on the Bloch sphere given by its polar co-ordinates.

2. Implement, in Haskell, the inverse of the previous exercise. That is, implement a function that converts a point on the Bloch sphere, given in polar co-ordinates, into the corresponding single qubit state given as two complex amplitudes.

3. In Haskell, define the single qubit states corresponding to $|0\rangle, |1\rangle, |+\rangle$ and $|-\rangle$. What are the polar co-ordinates for each of these states on the Bloch sphere?

4. Using the Quantum IO Monad, define a unitary operator that performs the Hadamard rotation on its argument qubit (see hints and tips below for more information).

5. Define a *QIO* computation that returns a qubit in the $|+\rangle$ state

6. Define a *QIO* computation that returns a qubit in the $|-\rangle$ state

7. Define a *QIO* computation that measures the $|+\rangle$ and $|-\rangle$ states from above. What is the difference in the measured values for each of the states? Why is this?

8. Define, using *QIO*, unitary operators that represent each of the three Pauli operators.

9. Write a *QIO* computation of type $Bool \rightarrow QIO\ Bool$, that initialises a qubit into the given Boolean base state, applies a hadamard operation to that qubit, then a Pauli-Z operation, and then another hadamard operation. Finally returning the measured value of the qubit.

10. What operation has the above *QIO* computation defined? Could you have calculated this without running the computation?

Hints and Tips

Exercise information

1. I would suggest defining a data-type to represent a single qubit state (*QubitState*), such as a pair of complex numbers, and a data-type to represent the polar co-ordinates of a point on a Bloch sphere (*PolarCoordinates*), such as a pair of real numbers (The types *RR* and *CC* are defined in *QIO.QioSyn* and use floating point numbers to represent the real numbers and complex numbers respectively).

   The function you need to define will then have type $toPolar :: QubitState \rightarrow PolarCoordinates$.

2. You should define a function of the type $toQubit :: PolarCoordinates \rightarrow QubitState$.

3. You should define the four states $|0\rangle, |1\rangle, |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ as members of your *QubitState* type. What is the output of the *toPolar* function applied to each of these states?

4. Using the Quantum IO Monad for defining quantum computations, is very similar to using it to define reversible computations. The actual members of the *QIO* type are the same, but there is one new constructor in the *U* data-type that replaces the *unot* operator[1]. This new constructor has type $rot :: Qbit \rightarrow Rotation \rightarrow U$ and corresponds to single qubit "rotations". With the *Rotation* type, we are able to define any 2x2 complex valued matrix, and use the *rot* constructor to make this into a unitary operator over a specific qubit. It is upto the programmer to ensure that the matrices defined are indeed unitary.

   The *Rotation* data type is just a type synonym for $(Bool, Bool) \rightarrow CC$, which means a rotation is defined by a function that takes a pair of Booleans and returns a complex number. The matrix defined by $f :: Rotation$ can be thought of as:

$$\begin{bmatrix} f\ (False, False) & f\ (False, True) \\ f\ (True, False) & f\ (True, True) \end{bmatrix}$$

[1] In fact, *unot* is an instance of the new constructor

For example, we could define the not rotation as follows

$$
\begin{aligned}
&\mathit{notRot} :: \mathit{Rotation} \\
&\mathit{notRot}\ (\mathit{False}, \mathit{False}) = 0 \\
&\mathit{notRot}\ (\mathit{False}, \mathit{True}) = 1 \\
&\mathit{notRot}\ (\mathit{True}, \mathit{False}) = 1 \\
&\mathit{notRot}\ (\mathit{True}, \mathit{True}) = 0
\end{aligned}
$$

although there are simpler ways to do this, such as

$$
\begin{aligned}
&\mathit{notRot} :: \mathit{Rotation} \\
&\mathit{notRot}\ (a, b) = \textbf{if}\ (a \equiv b)\ \textbf{then}\ 0\ \textbf{else}\ 1
\end{aligned}
$$

we can even now reimplement *unot*

$$
\begin{aligned}
&\mathit{unot} :: \mathit{Qbit} \to U \\
&\mathit{unot}\ q = \mathit{rot}\ q\ \mathit{notRot}
\end{aligned}
$$

this is in fact already done in the *QioSyn* library file, meaning that all our previously defined reversible computations are also quantum computations.

For this exercise, you need to define a function $\mathit{hadamard} :: \mathit{Qbit} \to U$ that uses a *Rotation* which implements the 2x2 matrix corresponding to the Hadamard rotation.

5. Think about the effect that the Hadamard rotation has on the base states, and define a *QIO* computation that initialises a qubit, and then applies the *hadamard* unitary to that qubit. The type of this computation should be $\mathit{plus} :: \mathit{QIO}\ \mathit{Qbit}$

6. This exercise is very simlar to the previous one. The type of this computation should be $\mathit{minus} :: \mathit{QIO}\ \mathit{Qbit}$

7. This computations should have type $\mathit{plusMinus} :: \mathit{QIO}\ (\mathit{Bool}, \mathit{Bool})$, and should initialise two qubits using the functions defined in the previous two exercises. The result of measuring these two qubits should be returned.

   Now that we actually want to run a quantum computation we need to look at the options available to us (the *run* and *sim* functions are both in the *QIO.Qio* library file). Try both, what does the ouput of *sim* tell us about measuring a $|+\rangle$ or a $|-\rangle$ state?

8. The matrices for the three Pauli operators are given in the lecture notes. You can simply use *unot* for the pauli-X operator, but should define $uY :: \mathit{Qbit} \to U$ and $uZ :: \mathit{Qbit} \to U$.

9. You can either combine the three rotations in one unitary using the *mappend* function, or use three *applyU* operations in your *QIO* computation.

10. Try multiplying the three matrices together and seeing what matrix you are left with. Alternatively, try running the computation and see what operation is achieved.

The Quantum IO Monad

The Quantum IO Monad, or *QIO* is a monadic interface from Haskell to quantum computation. More precisely, it is a library that allows you to define unitary operators and effectful quantum computations, along with simulator functions that allow you to *run* the quantum computations that you define. A lot of information on *QIO* including its implementation are available online (see the links on the course webpage). Installation of *QIO* is relatively straightforward if you can make use of cabal (cabal is part of the Haskell platform, and as such should already be installed on the machines in A32).

The following list of instructions will install *QIO* on the windows machines in A32 (but you may need to re-install it for every session). The following commands should be entered in a command prompt:

- Set the http proxy in the current command prompt

```
set HTTP_PROXY=wwwcache.cs.nott.ac.uk:3128
```

- Make sure the cabal list of packages is up to date:

```
cabal update
```

- Install *QIO* (in your own user space, as you don't have global permissions)

```
cabal install QIO --user
```

(note: if you don't have a proxy, and you are using your own machine, then you should just have to update the list of packages as above, and install the *QIO* package without the –user flag)

If you are having difficulties installing *QIO* you can always download the source from: http://www.cs.nott.ac.uk/ asg/QIO/ and import the files as necessary. However, i would recommend this as a last resort, and suggest that you contact me for support.

Information

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[2]. 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.

[2]GHC is required by the Quantum IO Monad, and is now installed as part of the Haskell platform in the main school lab

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).