G53NSC and G54NSC
Non-Standard Computation
Lab 2 Exercises

Dr. Alexander S. Green

4th February 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] GHC is required by the Quantum IO Monad, and is now installed as part of the Haskell platform in the main school lab

Exercise sheet 2

These exercises carry on from "Exercise sheet 1" and will use some of the types and functions you have previously defined (E.g the type *Bit* is used in the first few exercises).

Universality

The following exercises on universality should be attempted:

1. In Haskell, implement a function $nor :: (Bit, Bit) \rightarrow Bit$ that computes the NOR of the two input arguments (The truth table for NOR can easily be found online, or in the lecture notes).

2. To show that a Boolean function is universal, we can show that we can use it to define another universal set of functions. In Haskell, using only the *nor* you have defined previously, implement the universal set $\{\wedge, \vee, \neg\}$ of functions.
   E.g. define the following functions

   - $and' :: (Bit, Bit) \rightarrow Bit$
   - $or' :: (Bit, Bit) \rightarrow Bit$
   - $neg :: Bit \rightarrow Bit$

3. Implement, in Haskell, a function $fredkin :: (Bit, Bit, Bit) \rightarrow (Bit, Bit, Bit)$ that mimics the action of a Fredkin gate.

4. In Haskell, using only the *fredkin* function you have defined previously, (re)implement the universal set $\{\wedge, \vee, \neg\}$ of functions.

5. In Haskell, use the three functions defined for exercise 4, to implement a function that has the following truth table:

$input_1$	$input_2$	$input_3$	$input_4$	$output$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Reversible Computation

The following exercises use the classical subset of *QIO*. An introduction to the classical subset of *QIO* is given here, followed by the exercises on reversible computation.

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.

The following section is an introduction to the classical subset of *QIO*, and assumes that you have *QIO* installed using the cabal method described above. We will be using the full quantum power of *QIO* in future exercise sheets, so please make sure you have *QIO* installed and working as soon as possible.

An introduction to the classical subset of *QIO*

QIO gives us access to a Haskell library of quantum computation. The classical subset of *QIO* restricts the unitary operations that we can define, but has the benefit that running the computations can be done much more efficiently (There is upto an exponential overhead in simulating quantum computation on a classical computer, but not reversible computation). This section aims to introduce the classical subset of *QIO*, including how we are able to run the computations that we define.

Although we are able to use *QIO* to define classical reversible computations, and run them, a lot of the syntax is quantum computation oriented. For now, please treat certain entities as their classical counterparts. E.g. The type *Qbit* merely represents bits in the classical subset of *QIO*.

Computations exist within the monadic *QIO* data-type. Therefore, when we are defining a *QIO* computation we must use the monadic syntax provided by Haskell. Namely, we make use of **do** notation.

The *QIO* type of quantum computations, defines only three functions that can be used in defining a computation. These functions are the same for when we are defining a reversible computation as for when we are defining a quantum computation. Namely the functions provided are the following:

$mkQbit :: Bool \rightarrow QIO\ Qbit$

Initialises a bit to the given boolean value. The return type tells us that this function defines a *QIO* computation that returns a reference to a *Qbit*, which for today is just a type synonym for a reference to a bit.

$applyU :: U \rightarrow QIO\ ()$

The *applyU* function is used to apply a unitary operator (which inhabit the type U) to the current state the *system* is in. Here, the *system* refers to any bits that are currently in scope (or in other words, have previously been initialised by a call to the *mkQbit* function). The reversible nature of the unitary operators is shown by this function's return type. The return value is a *QIO* computation, but no information is returned as it is just the state of the system that has been updated in an information preserving manner. We shall look at the U data-type shortly.

$measQbit :: Qbit \rightarrow QIO\ Bool$

This function is used to measure what state a bit (*Qbit*) is currently in. We shall be covering the importance of the measurement operation in quantum computation in the next few lectures, but for reversible computation, you can just think of this operation as returning the current value a bit is in. It is a *QIO* computation that returns the Boolean value corresponding to the value of the bit it is measuring.

Computations in *QIO* roughly follow the same structure. First, an input state is initialised using as many calls to *mkQbit* as is necessary, then some form of processing takes place, in the form of applying a unitary operator to the input state. Finally, whichever bits form the output are measured and the results are returned.

These functions and the U data-type that we shall look at shortly, are all

defined as part of the syntax of *QIO*. In order to use them, you will need to import the *QioSyn* library. Eg. **import** *QIO.QioSyn*

Before looking at what unitary operators we can construct, we can already test out the initialisation and measurement operations. Look at the following code:

```
testQIO :: QIO (Bool, Bool)
testQIO = do r1 ← mkQbit False
  r2 ← mkQbit True
  b1 ← measQbit r1
  b2 ← measQbit r2
  return (b1, b2)
```

We can infer from the code what it is doing. First it initialises a bit to the state *False*, and gives that bit the reference *r1*, then it initialises another bit to the state *True* and gives it the reference *r2*. The result of measuring the bit referenced by *r1* is stored in *b1*, and the result of measuring the bit referenced by *r2* is stored in *b2*. The pair of these two values (*b1*, *b2*) is then returned as the result of the computation.

In order to actually run this computation, we must now use one of the run functions provided by *QIO*. We'll look at the choices in more detail next week, but as we are only using the classical subset of *QIO* this week, we only need to introduce the *runC* function that can be used to run classical reversible *QIO* computations. The *runC* function is part of the *QioClass* library, and as such it is necessary to import it too (E.g. **import** *QIO.QioClass*).

The *runC* function has type $QIO\ a \rightarrow a$ and is able to just run a computation that only uses the classical subset of *QIO*, returning the pure value which the computation evaluates to. So, running the *testQIO* function from above, using *runC testQIO* gives a pure value of type (*Bool*, *Bool*). In fact, it gives the value (*False*, *True*) just as we should have been expecting.

We can now start to look at the classical subset of the *U* data-type, which will allow us to define arbitrary reversible computations. It is the members of this *U* data-type that correspond to the reversible circuits we have been looking at in the lecture slides.

The *U* data-type gives us the following constructors:

```
mempty :: U
mappend :: U → U → U
unot :: Qbit → U
swap :: Qbit → Qbit → U
cond :: Qbit → (Bool → U) → U
ulet :: Bool → (Qbit → U) → U
urev :: U → U
```

The following list gives a run through of the behaviour of each construct:

- The first two constructs (*mempty* and *mappend*) are just derived from the fact that the *U* data-type is defined as a monoid in Haskell. What this means for us, is that we can use *mempty* to define the identity operation, and *mappend* to combine two members of the *U* type sequentially. So

that we can use these operations, we must also import the *Data.Monoid* library (E.g. **import** *Data.Monoid*).

- The *unot* construct is used to negate the bit referenced by its argument, and corresponds exactly to the X gate we have in the circuit notation.

- The *swap* construct is used to swap the states of the two bits referenced by its arguments, and corresponds to the swapping of wires in the circuit notation.

- The *cond* construct corresponds closely to the control structures we saw in the circuit notation. The bit referenced by its first argument is the control bit, and its value is used to decide which sub-unitary is run. The Boolean function that makes up the second argument must return a member of the *U* type for both members of the *Bool* data-type and is evaluated over the value of the control wire to decide which of these two sub-unitaries is run. As an example we can define the controlled-X operation as follows:

  ```
  controlledX :: Qbit → Qbit → U
  controlledX r1 r2 = cond r1 (λx → if x then unot r2 else mempty)
  ```

  if the control bit is *True* then the *unot* operation is performed, but if the control wire is false then the *mempty* operation is performed.

  You must be careful when defining control structures, and ensure that the sub-unitaries do not make use of the control bit (this can lose the reversibility requirement).

- The *ulet* construct allows us to introduce ancilliary bits into our computations. If part of a circuit requires an extra bit in order to compute a result, but that bit doesn't form part of the output, and is returned to its original value, then we can use an ancilliary bit for this job. We shall look more at ancilliary bits and the *ulet* construct in the exercises below.

- The last construct, *urev*, is possible because all the members of the *U* type are unitary operators (or reversible). This function just returns the inverse of its argument unitary.

Before going on to exercises using the classical subset of *QIO* we shall look at a few more example reversible computations written in *QIO*.

Possibly the simplest example we can give is the negation function, defined using *QIO*. It takes a Boolean value as input, which is used to initialise a bit, this bit is then negated by applying the *unot* unitary, and the result of measuring it is the return value of the computation.

```
notQIO :: Bool → QIO Bool
notQIO b = do r1 ← mkQbit b
              applyU (unot r1)
              measQbit r1
```

The results of running this computation are as follows:

```
runC (notQIO False) = True
runC (notQIO True) = False
```

A slightly more complicated example would be to use a controlled-X operation to implement the XOR operation.

```
xor :: Bool → Bool → QIO Bool
xor a b = do r1 ← mkQbit a
  r2 ← mkQbit b
  applyU (controlledX r1 r2)
  measQbit r2
```

The results of running this computation are as follows:

```
runC (False `xor` False) = False
runC (False `xor` True) = True
runC (True `xor` False) = True
runC (True `xor` True) = False
```

One last example before we move onto the exercises is to implement the NAND function using the circuit that follows the generalised model of reversible computation as presented in this weeks lecture. We can make use of a single ancilliary bit in order to achieve this. The firs thing we need to define is a unitary that corresponds to the tofolli gate

```
toffoli :: Qbit → Qbit → Qbit → U
toffoli r1 r2 r3 = cond r1 (λx → (if x then (controlledX r2 r3) else mempty))
```

now we can define the full unitary which includes the ancilliary bit

```
reversibleNAND :: Qbit → Qbit → Qbit → U
reversibleNAND r1 r2 r4 = ulet True (λr3 → (toffoli r1 r2 r3)
                                           `mappend`
                                           (controlledX r3 r4)
                                           `mappend`
                                           (toffoli r1 r2 r3))
```

Because the *ulet* takes a value into which the ancilliary bit is initialised, we don't need to worry about the extra X gates that appear in the circuit diagram. We can now define the computation which returns the NAND of its two inputs, making sure we set the bit that will contain the result to 0.

```
nand :: Bool → Bool → QIO Bool
nand a b = do r1 ← mkQbit a
  r2 ← mkQbit b
  r3 ← mkQbit False
  applyU (reversibleNAND r1 r2 r3)
  measQbit r3
```

The results of running this computation are as follows:

```
runC (False `nand` False) = True
runC (False `nand` True) = True
runC (True `nand` False) = True
runC (True `nand` True) = False
```

In the following section, you will be expected to use the classical subset of *QIO* as introduced here in order to implement your solutions. However, the exercises will contain hints and tips on using *QIO*

Exercises on reversible computation

The following exercises on reversible computation should be attempted using the classical subset of *QIO*:

1. The first few exercises on reversible computation relate to defining an addition function. The first task is to implement a function, similar to the *int2bits* function from last week, that initialises a list of bits in *QIO* such that it represents the binary expansion of a given integer. To make the definition of the addition function easier, this function must return a list of fixed length, that is, it represents fixed precision integers in *QIO*. Define in Haskell a *QIO* computation that takes an *Int* as an argument and returns a list of 32 bits initialised to the binary expansion of that number. E.g. define a function $int2qbits :: Int \to QIO\ [Qbit]$

   Hint: you may like to use the function *int2bits* that you defined last week, modified slighty so that it always returns 32 Booleans ($int2bits' :: Int \to [Bool]$), and define a *QIO* computation that turns a list of Booleans into a list of bits ($bits2qbits :: [Bool] \to QIO\ [Qbit]$).

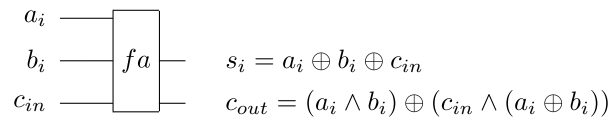
   Other useful functions on lists include:

   - $take :: Int \to [a] \to [a]$ which takes a list and returns a list that is the first n elements of the given list, where n is the given *Int*.
   - $repeat :: a \to [a]$ returns an infinite list of the given element a.

2. Define a function $qbits2int :: [Qbit] \to QIO\ Int$ that returns the integer represented by the list of bits in the argument. This should be the inverse of the previous exercise. Test your solution to make sure that the following function is indeed the identity on *Int*:

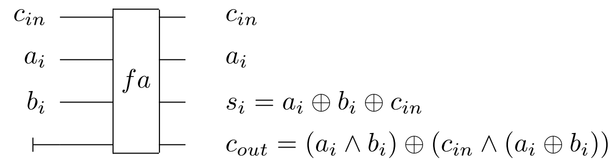
   $$\begin{aligned} &int2int :: Int \to QIO\ Int \\ &int2int\ n = \textbf{do}\ qn \leftarrow int2qbits\ n \\ &\qquad\qquad\qquad\quad qbits2int\ qn \end{aligned}$$

3. Now we have a representation of numbers in *QIO* we can move on to defining the unitary operations (E.g. members of the U data-type) that represent a circuit for reversible addition. In irreversible computation, we would use full-adders for this purpose:

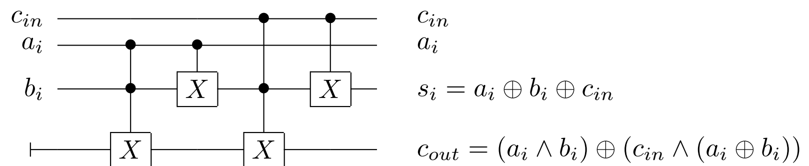
   

   However, a full-adder defined like this is not reversible. We can define a new reversible full-adder that keeps track of two of its inputs, and uses an

extra heap bit for one of the outputs:



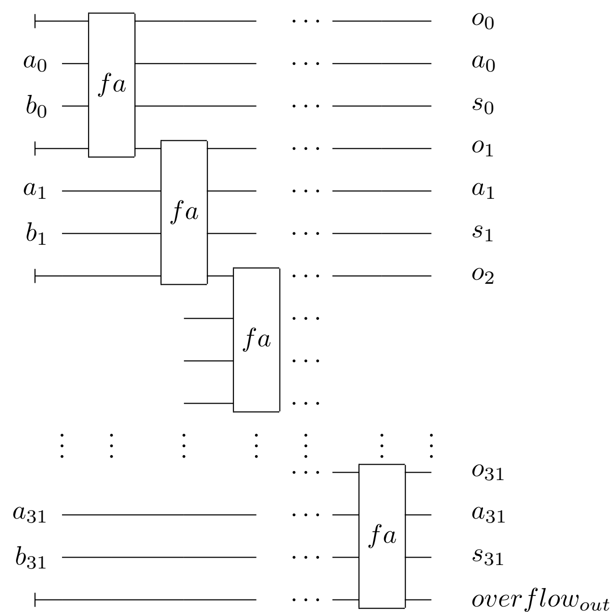
This circuit can be implemented using two Toffoli gates, and two controlled-X gates. Remembering that controlled-X gates introduce an XOR operation ($\oplus$) and the toffoli gate introduces an AND operation (within an XOR):



Define using *QIO* in Haskell, a unitary operator that represents the above circuit. E.g define a function *fullAdder* :: $Qbit \rightarrow Qbit \rightarrow Qbit \rightarrow Qbit \rightarrow U$ that defines the circuit above over its four argument bits.

note. You may find it useful to use the definitions for a Toffoli gate, and a controlled-X gate given in the introduction to *QIO*.

4. Now that we have a full-adder unitary, we need to string 32 of them together to define an addition unitary over our entire lists of bits. The following circuit diagram is gives a mock up of how this can be achieved:

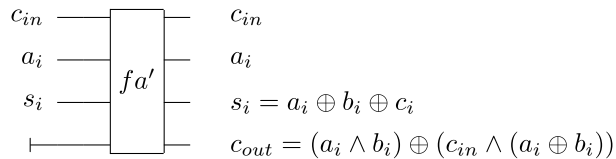


where $[a_0, a_1 \ldots a_{31}]$ and $[b_0, b_1 \ldots b_{31}]$ represent the input bit strings, $[s_0, s_1 \ldots s_{31}]$ represents the output sum bit string, $[o_0, o_1 \ldots o_{31}]$ represent the intermediary carry bits, and $overflow_{out}$ is an extra bit that carries any overall overflow information.

Implement this circuit as a member of the U data-type. E.g. write a function $adder :: [Qbit] \rightarrow [Qbit] \rightarrow [Qbit] \rightarrow Qbit \rightarrow U$. Note that we cannot currently use ancilliary bits for the carries, so we must have an extra list of bits to hold the carry bits. The final *Qbit* argument is the overflow bit.

5. So that we can use ancilliary bits for the carries, we need to use the generalised model of reversible computation that was introduced in the lectures. Implement a member of the U data-type that represents a generalised version of the adder defined in the previous exercise.

   Hint. The easiest way to do this, isn't to copy out the result and undo the whole computation, but to actually just undo the parts of the computation that calculated the overflow bit. Hence the only thing we need to *copy out* is the overflow bit itself, and define a member of the U data-type that is simlilar to the inverse of the adder, but doesn't undo the summing step. E.g. define a function $undoCarry :: Qbit \rightarrow Qbit \rightarrow Qbit \rightarrow Qbit \rightarrow U$ that defines the inverse of the following circuit:

   Hint: You should make use of the following function (in conjunction with your new *int2bits* function) that is an extension of *ulet* over a whole list of bits:

$$letBits :: [Bool] \rightarrow ([Qbit] \rightarrow U) \rightarrow U$$
$$letBits\ bs\ fbs = letBits'\ bs\ [\,]$$
$$\textbf{where}\ letBits'\ [\,]\ \ bs = fbs\ bs$$
$$letBits'\ (b : bs)\ bs' = ulet\ b\ (\lambda x \rightarrow letBits'\ bs\ (bs' +\!\!+ [x]))$$

6. Using the functions defined in the above exercises, implent a *QIO* computation that sums two input integers. E.g. define a function $add :: Int \rightarrow Int \rightarrow QIO\ Int$ that intialises two lists of bits to represent the input integers, then sums them using the unitary operator defined in the previous exercise, and finally returns the resulting list of bits as an integer.

   Hint. You can chose how you want to deal with overflow.

7. Can you now define a subtraction operation using only the unitaries you have defined already? Implement it as a function $sub :: Int \rightarrow Int \rightarrow QIO\ Int$

   Hint. This is possible, so you should attempt it.