1

The Quantum IO Monad

Thorsten Altenkirch and Alexander S. Green
School of Computer Science, The University of Nottingham

Abstract

The Quantum IO monad is a purely functional interface to quantum programming implemented as a Haskell library. At the same time it provides a constructive semantics of quantum programming. The QIO monad separates reversible (i.e. unitary) and irreversible (i.e. probabilistic) computations and provides a reversible let operation (*ulet*), allowing us to use *ancillas* (auxiliary qubits) in a modular fashion. QIO programs can be simulated either by calculating a probability distribution or by embedding it into the IO monad using the random number generator. As an example we present a complete implementation of Shor's algorithm.

1.1 Introduction

We present an interface from a pure functional programming language, Haskell, to quantum programming: the Quantum IO monad, and use it to implement Shor's factorisation algorithm. The implementation of the QIO monad provides a **constructive semantics** for quantum programming, i.e. a functional program which can also be understood as a mathematical model of quantum computing. Actually, the Haskell QIO library is only a first approximation of such a semantics, we would like to move to a more expressive language which is also logically sound. Here we are thinking of a language like Agda (Norell (2007)), which is based on Martin Löf's Type Theory. We have already investigated this approach of *functional specifications of effects* in a classical context (Swierstra and Altenkirch (2007, 2008); Swierstra (2008)). At the same

time the QIO monad provides a high level interface to a hypothetical quantum computer.

To make the presentation accessible to the non-Haskell programmer, we shall give a brief introduction to the language in section 1.2. The QIO monad separates reversible (i.e. unitary) and irreversible (i.e. probabilistic) computations and provides a reversible let operation (*ulet*), allowing us to use *ancillas* (auxiliary qubits) in a modular fashion. QIO programs can be simulated either by calculating a probability distribution or by embedding it into the IO monad using the random number generator. An overview of the Quantum IO Monad (QIO) is given in section 1.3.

Exploiting Haskell's class system we can present our algorithms in a high level way, implementing abstractions using functional programming technology. We describe the implementation of Shor's algorithm in some detail (section 1.5) also covering the necessary reversible arithmetic (section 1.4). The actual implementation of QIO is covered in section 1.6, where we focus in detail on the design of the QIO quantum simulator functions *sim* and *run*.

Quantum programming is able to exploit the strange nature of quantum physics to achieve classically impossible, or rather infeasible, tasks. Most famously Shor's algorithm shows that on a quantum computer we can factor a number in polynomial time, hence we could break many encryption schemes. While physicists are working on building working quantum circuits with more than a handful of qubits, we computer scientists grapple with the challenges quantum computing creates for software: in designing algorithms, like Shor's which exploit quantumness; but also in designing languages which support abstractions relevant for quantum computing. See Gay (2006) and Rüdiger (2007) for recent language surveys.

Here we investigate a different approach: instead of implementing a new language from scratch we provide a monadic interface to do quantum programming in Haskell - the quantum IO monad (*QIO*). This has a number of advantages:

- we can exploit existing means of abstraction present in Haskell to structure our quantum programs, indeed we will give an example of this by implementing the class *Qdata* which relates classical data-types with their quantum counterparts;
- We can explore which semantic primitives are most useful to structure quantum programs, without committing to a particular view too early;

- we can exploit the existing support for Haskell, in the form of tools and libraries to develop our quantum programs.

We believe that a purely functional approach is ideally suited for this venture, since it already makes effects explicit (in Haskell via the IO monad) and is close to a mathematical semantics of quantum programming (see our introductory comments on constructive semantics).

While *QIO* realises the infrastructure we need to control a quantum computer from Haskell, we don't have to wait until the physicists develop a practical quantum computer, we can use the same interface to run a quantum simulator. Our approach is inspired by the first author's work with Wouter Swierstra on functional specifications of IO (Swierstra and Altenkirch (2007)). Indeed, we provide some choice here: we can embed *QIO* into the IO monad using pseudo-random numbers to simulate quantum randomness, we can statically calculate the probability distribution of possible results given a quantum program and we can simulate the classical subset of our quantum operations directly. The latter is useful for testing components efficiently since the quantum simulation generates a considerable overhead.

All the code described in this chapter, i.e. the implementation of QIO and the quantum algorithms implemented in it are available from the second author's web-page (Green (2008)).

There are a number of papers on modelling quantum programming in Haskell, (Mu and Bird (2001); Sabry (2003); Karczmarczuk (2003); Vizzotto et al. (2006)). They describe different abstractions one can use to simulate quantum computation in a functional setting - however, our approach is different in that we provide a high level interface in the spirit of the IO monad which provides an interface to a hypothetical quantum computer. Our previous work on QML (Altenkirch and Grattage (2005)) proposed a first order functional quantum programming language, the present work is more modest but gives us a stepping stone to experiment with various alternative structures useful for structuring quantum programming and also to implement future versions of languages like QML.

Acknowledgements

We have profited from Ralf Hinze's and Andres Löh's implementation of *lhs2TeX* to typeset the Haskell code in this paper. The research reported in this paper has benefited from EPSRC's QNET framework and from

the QICS framework 7 STREP. We would also like to thank the editors (Simon Gay and Ian Mackie) for their efforts to publish this book and the anonymous referees for their helpful feedback. We have had useful discussions with members of the Functional Programming Laboratory in Nottingham in particular we would like to thank Wouter Swierstra for his advice on Haskell programming techniques.

1.2 Functional Programming and the Real World

Haskell is a pure functional programming language, treating computations as the evaluation of pure mathematical functions. A function is said to be pure if it always returns the same result when given the same arguments, and in producing that result has not caused any side-effects to occur within the system. Haskell employs lazy evaluation which allows expressions to only be evaluated when needed. This allows infinite data structures to be defined in Haskell, whereby a finite part of the data structure can be used as and when required. For example, you could define an infinite list of all the prime numbers, and then define a function that requires the first 10 elements of this list. The definition of a pure function as given above may seem to inhibit the creation of any sort of *useful* real world programs, as if a program cannot depend on any state, and cannot change any state, ie. have any input or output, then it can be argued it is the same as a program that does nothing. However, we can make use of the categorical notion of a monad which is a kind of abstract data-type that can be used to model side-effects within a purely functional setting. In section 1.2.3 we'll introduce the notion of a monad within Haskell, and show how they can be used to define stateful computations. We'll also look at how Haskell deals with I/O using the IO monad. In the following section (section 1.2.1) we'll see how functions are defined in Haskell, starting with some simple bitwise operations and then a look at how higher order functions and recursion are a mainstay of Haskell programs.

Another useful aspect of programming in Haskell is its use of *type-classes* which allow polymorphic functions to be defined for any type which satisfies the class requirements. Other functions can then be defined that have the pre-requisite that the input type is a member of one of the type-classes (as apposed to just a specific type). A common example of a type-class in Haskell is the equality class (Eq). A type

can become a member of the *Eq* class if it can define a function $\equiv$† that defines equality for that type. The programmer could then define a polymorphic function whose only restriction on the input type is that it is a member of *Eq*. We will see in section 1.2.2 an example of how type-classes can be used to define monoids within Haskell. You'll also notice that monads are defined by a type-class within Haskell.

1.2.1 Haskell by example

The Haskell language definition (Jones (2003)), Haskell compilers and interpreters, the description of the standard libraries and much more can be found on the Haskell homepage: `www.haskell.org`. There are a number of introductory textbooks available, e.g. (Hutton (2007)). We restrict ourselves here to a small number of introductory examples, relevant to our development.†

We have chosen the example of defining a full-adder function in Haskell as in section 1.4 we go on to design reversible arithmetic functions, starting off with a reversible adder function. You'll see that the basic idea is the same, defining a function that adds single bits, and keeps a track of overflow. Addition is then achieved by calling this function recursively.

In Haskell it is useful to first define the type of the function. For example, if we wanted to define a bitwise full-adder function we could first define the function that adds together the two-bits to be added and the input carry bit, and returns the pair consisting of the output bit and the output carry bit. Using the *Bool* type (of Booleans) to represent bits, this would give us a function which takes three input bits and returns a pair of output bits.

$$addBit :: Bool \to Bool \to Bool \to (Bool, Bool)$$

Currying allows us to chain the input bits in this way. Note that it is not always necessary for the programmer to define the type of a function as Haskell employs type-inference at compile time to infer the type of a function, however it is deemed good programming practise to do so. Now we have the type for our function, we can define the function itself.

$$addBit\ c\ a\ b = ((a \wedge b) \vee (a \wedge c) \vee (b \wedge c), (a \not\equiv b) \not\equiv c)$$

† We use lhs2tex to typeset Haskell code, hence `==` becomes $\equiv$.

† It is useful to note that the current code (Green (2008)) makes use of some of the extensions only available with the Glasgow Haskell Compiler.

Given the three input bits (a,b the bits to be summed, and c the input carry bit) we can define the output as the pair of the two functions that define the two output bits respectively. The left hand side of the pair computes the output carry bit (which occurs when at least 2 of the 3 input bits are true), and the right hand side computes the sum of the three input bits (where addition modulo 2 is the same as the logical $\not\equiv$ (or xor)).

Addition can now be extended to lists of Booleans, recursively. This is achieved by feeding the carry-bit through the function.

$$
\begin{array}{l}
add' :: Bool \to [Bool] \to [Bool] \to [Bool] \\
add'\ c\ [\,]\ [\,] = [\,] \\
add'\ c\ (a : as)\ (b : bs) = a' : (add'\ c'\ as\ bs) \\
\quad \textbf{where}\ (c', a') = addBit\ c\ a\ b
\end{array}
$$

When evaluated, Haskell will use pattern matching to apply the correct function definition to the current arguments, ie. starting from the first line and going down until the current arguments match the patterns given in the definition. In the function add' above the first line will only pattern match when both the list arguments are empty ([ ] in Haskell), and the second line will only pattern match when both the list arguments have at least one element (The cons operator in Haskell : is used to construct lists, so $a : as$ is the list with head element a and (possibly empty) tail as).

The first line of the definition gives us the base case, where the sum of two empty lists of Booleans is also the empty list. This base case doesn't take into account the final carry bit, showing us that our adder as defined here doesn't deal with overflow. The second line of the definition is the recursive call and uses our *addBit* function from above to add the current bits, and then recursively calls add' with the new carry bit and the tails of the input lists. The **where** keyword is used to give variable names to the result of calling *addBit*.

The full adder is finished off by always starting the call to the addition function above with a False input carry-bit. The use of currying to define our functions means that this is a valid function as required.

$$
\begin{array}{l}
add :: [Bool] \to [Bool] \to [Bool] \\
add = add'\ False
\end{array}
$$

As they stand, these functions aren't defined for all inputs, as there is no definition of the function when only one of the input lists is empty. Our intention is that they are only used for lists of the same length but

it is not straightforward to express this constraint using Haskell's type system (we'd be better off with Agda). Indeed, we will be using the functions only over a fixed wordsize which is useful when defining *zero*:

```
wordSize :: Int
wordSize = 8

zero :: [Bool]
zero = take wordSize (repeat False)

data Word = Word [Bool]
```

Here *Word* is a datatype with one constructor which is also called *Word*, a common Haskell convention.

The zero function above shows how we can make use of infinite data structures, as the function *repeat* creates an infinite list of its argument (in this case *False*), but calling zero is possible due to lazy evaluation and creates the data structure:
[*False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*]

1.2.2 Monoids

A monoid, or a semi-group, is defined as an algebraic structure with an associative binary operation, along with an identity element. In Haskell, we can realise this as a type-class that provides an element of the underlying type as the identity element, and a binary function acting on and returning the underlying type as the monoidal operation. In Haskell the identity element is usually referred to as *mempty* and the binary operator as *mappend*.

```
class Monoid a where
  mempty :: a
  mappend :: a → a → a
```

A nice example of a type which can fulfil the requirements of the monoid type-class is the *Word* type which we have defined above.

```
instance Monoid Word where
  mempty = Word zero
  (Word a) `mappend` (Word b) = Word (a `add` b)
```

The identity element is the *zero* (lifted to a *Word*), and the binary operation is the *add* function (again lifted to words).

With monoids it is also possible to introduce an idea of stateful programs within the pure world of Haskell. However, it is not possible to

access the state at intermediary stages within a computation. A stateful computation over the state type s is defined by a transition function:

$$\textbf{data}\ \mathit{State}\ s = \mathit{State}\{\,\mathit{runState} :: s \to s\,\}$$

The destructor function *runState*, i.e. the inverse to the constructor, is generated automatically, satisfying the equation $\mathit{runState}\ (\mathit{State}\ f) = f$.

The monoidal structure is used to define how these transition functions can be sequenced:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monoid}\ (\mathit{State}\ s)\ \textbf{where}\\ &\quad \mathit{mempty} = \mathit{State}\ \mathit{id}\\ &\quad (\mathit{State}\ f)\ \text{`}\mathit{mappend}\text{`}\ (\mathit{State}\ g) = \mathit{State}\ (g \circ f)\end{aligned}$$

The sequencing of these state transition functions is simply just functional composition.

1.2.3 Monads

Monads are a natural generalisation of the concept of a monoid in Category Theory, by looking at monoids in a given category. In functional programming they are used as a notion for defining computations. These computations in themselves don't necessarily need to be pure computations, which can be achieved by defining datatypes that describe the side-effects that a computation may have. In this way, it is possible to define effectful computations within Haskell as computations within a monad. The idea of a monad in Haskell is quite simple, but is used extensively to create any sort of computations that would otherwise not be pure. In section 1.2.4 we'll see that all I/O in Haskell takes place in the IO monad, and that the use of monads within Haskell is so common that Haskell provides the **do** notation which is a form of syntactic sugar that enables monadic programs to be written in a more imperative style. In this section we shall introduce monads, starting with their definition as a type-class within Haskell:

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\ &\quad \mathit{return} :: a \to m\ a\\ &\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

Here $m :: * \to *$ is an operator on types, this fact is inferred by Haskell. A monad is defined by a *return* function whose job it is to lift a member of the underlying type into the monadic version of the same type, and

a bind function (denoted $\gg=$) which defines how computations can be sequenced within the monad.

A simple example of a monad is the *Maybe* monad, which is defined by the data type

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

a member of this data type is either *Just* a member of the underlying data-type, or it's *Nothing*. A simple way of thinking of this, is that *Nothing* defines an error value in the monad, and it is the job of the monadic structure to propagate *Nothing* values through any computation where they may occur. This is achieved by defining *Maybe* as an instance of the monad type class as follows:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where}\\ &\quad \mathit{return} = \mathit{Just}\\ &\quad \mathit{Nothing} \gg= f = \mathit{Nothing}\\ &\quad (\mathit{Just}\ x) \gg= f = f\ x\end{aligned}$$

the *return* function lifts values into the monad by making them a *Just* value, and the bind function pattern matches on the monadic type to either propagate a *Nothing* or apply a computation to the underlying *Just* value.

A more interesting example of monads in Haskell is the state monad, which like the state monoid above allows stateful computations to be defined. However, unlike the monoid example, the state monad also allows values to be accessed, giving us effects (e.g. changes in state), and values which may depend on previous effectful computations. A stateful computation over the state type s that can return values of type a is defined by a transition function that returns a value along with the new state.

$$\textbf{data}\ \mathit{StateM}\ s\ a = \mathit{StateM}\{\,\mathit{runStateM} :: s \to (a, s)\,\}$$

We can now define the monadic behaviour required to thread state through a computation. The *return* function can simply create a transition function that returns the underlying value, but has no effect on the state. The bind function returns a transition function that extracts the value and new state from applying its left hand argument to the given state, and uses these new values in applying its right hand argument.

```
instance Monad (StateM s) where
  return a          = StateM (λs → (a, s))
  (StateM x) >>= f = StateM (λs → let (v, s') = x s
                                  in runStateM (f v) s')
```

In order to actually use the state monad it is necessary to create some sort of an interface to it. The *MonadState* class is often used for this: †

```
class MonadState m s | m → s where
  get :: m s
  put :: s → m ()
instance MonadState (State s) s where
  get = State (λs → (s, s))
  put s = State (λ_ → ((), s))
```

The *get* function is used to return the current state as the returned value, and leaves the overall state unchanged, and the *put* function updates the state to the given value. Since *put* doesn't return any information we are using Haskell's unit type () corresponding to `void` in C.

We shall now give some examples of monads which we have created specifically for the implementation of the QIO monad. Firstly, a type of vectors which are used in QIO for the creation of probability distributions, which we will use when defining the *sim* function later.

A vector over types x and a is defined as a list of pairs of those types.

```
data Vec x a = Vec{unVec :: [(a, x)]}
```

‡ The following functions are defined over vectors:

```
empty :: Vec x a
empty = Vec []
(@) :: (Num x, Eq a) ⇒ Vec x a → a → x
(Vec ms)@a = foldr (λ(b, k) m → if a ≡ b then m + k else m) 0 ms
(⊗) :: Num x ⇒ x → (Vec x a) → Vec x a
l ⊗ (Vec as) = (Vec (map (λ(a, k) → (a, l ∗ k)) as))
(⊕) :: (Vec x a) → (Vec x a) → Vec x a
(Vec as) ⊕ (Vec bs) = (Vec (as ++ bs))
```

empty is simply the empty vector, @ is a lookup function that requires the type x to be in the *Num* class (of numeric types), and the type a

† The functional dependency $m \to s$ is a hint to the Haskell type checker and may be ignored by the reader.

‡ *unVec* is the inverse of the constructor *Vec*.

to have equality (Eq), and returns the sum of the numeric argument of every member of the vector that matches the given argument. $\otimes$ is a scalar multiplication function that again requires the x type to be a numeric type, such that it can multiply the numeric part of each member of the vector by the given scalar. The $\oplus$ is simply the concatenation function for vectors, and joins two vectors by using list concatenation.

We can now define vectors over numeric types as a monad, specifically such that they can be used to hold the probability distributions required. The *return* function lifts an underlying value into the monad type by just giving it a probability of 1 as the only member of the probability distribution. The bind operation, applies the bound function to the first argument of every member of the current probability distribution, and multiplies the numeric argument by this new result, with the effect of updating the probability distribution depending on the effect of the given function.

$$\begin{aligned}&\textbf{instance}\ \mathit{Num}\ n \Rightarrow \mathit{Monad}\ (\mathit{Vec}\ n)\ \textbf{where}\\&\quad \mathit{return}\ a = \mathit{Vec}\ [(a, 1)]\\&\quad (\mathit{Vec}\ ms) \gg\!= f = \mathit{Vec}\ [(b, i * j) \mid (a, i) \leftarrow ms, (b, j) \leftarrow \mathit{unVec}\ (f\ a)]\end{aligned}$$

To make use of these vectors in the implementation of QIO we now define a *PMonad* or a probability monad as a monad along with the extra function *merge*. The *merge* function can be thought of as a function that defines how two computations (c,d) of the same type can be merged depending on the given $\mathbb{R}$ argument, where $\mathit{merge}\ p\ c\ d$ means that the probability of c occurring is p, and the probability of d occurring is $1-p$.

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{PMonad}\ m\ \textbf{where}\\&\quad \mathit{merge} :: \mathbb{R} \rightarrow m\ a \rightarrow m\ a \rightarrow m\ a\end{aligned}$$

To create our probability distributions for the *sim* function we can now define the actual type of our probability distributions ($\mathit{Prob}\ a$).

$$\textbf{data}\ \mathit{Prob}\ a = \mathit{Prob}\{\,\mathit{unProb} :: \mathit{Vec}\ \mathbb{R}\ a\,\}$$

In this case, the distribution can be over any type a with a $\mathbb{R}$ argument as its current probability. We continue by defining the type *Prob* as a monad by lifting the monadic operations for *Vec*.

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ \mathit{Prob}\ \textbf{where}\\&\quad \mathit{return} = \mathit{Prob} \circ \mathit{return}\\&\quad (\mathit{Prob}\ ps) \gg\!= f = \mathit{Prob}\ (ps \gg\!= \mathit{unProb} \circ f)\end{aligned}$$

$$\begin{aligned}&\textbf{instance } \mathit{PMonad}\ \mathit{Prob}\ \textbf{where}\\&\quad \mathit{merge}\ pr\ (\mathit{Prob}\ \mathit{ift})\ (\mathit{Prob}\ \mathit{iff}) = \mathit{Prob}\ ((pr \otimes \mathit{ift})\\&\qquad\qquad\qquad\qquad\qquad\qquad \oplus ((1 - pr) \otimes \mathit{iff}))\end{aligned}$$

The *PMonad* is then defined with the *merge* function multiplying each probability distribution by its respective probability, and joining them. In comparison, the *run* function requires that the *IO* monad is defined as a *PMonad* which is achieved by using the built in random number generator to probabilistically return one of the given arguments depending on the given probability:

$$\begin{aligned}&\textbf{instance } \mathit{PMonad}\ \mathit{IO}\ \textbf{where}\\&\quad \mathit{merge}\ pr\ \mathit{ift}\ \mathit{iff} = \mathit{Random.randomRIO}\ (0, 1.0)\\&\qquad\qquad\qquad\qquad \gg\!= \lambda pp \rightarrow \textbf{if } pr > pp \textbf{ then } \mathit{ift} \textbf{ else } \mathit{iff}\end{aligned}$$

We'll see more on how the *PMonad* type is used in the implementation of QIO in section 1.6.

1.2.4 Haskell's IO Monad, and do *notation*

All I/O in Haskell takes place within the IO Monad, in fact in a compiled Haskell program, the *main* function which is called by the system at run-time has type $\mathit{IO}\ ()$. This should not be that surprising as any interaction with the system makes use of I/O. A simple *Hello World* program written in Haskell would be:

$$\begin{aligned}&\mathit{main} :: \mathit{IO}\ ()\\&\mathit{main} = \mathit{putStrLn}\ \texttt{"Hello, world!"}\end{aligned}$$

where the function $\mathit{putStrLn} :: \mathit{String} \rightarrow \mathit{IO}\ ()$ defines an effectful computation that outputs the given string to **stdout**, and returns the unit type as its value.

The IO Monad defines many standard I/O functions that can be used within Haskell programs, however, we shall not discuss them all in detail here.

As the IO monad is indeed a monad within Haskell, it is necessary to bind all *IO* computations together using the monadic bind function. For example, if we wanted to write a function that echoes a character to the screen we could use the two following *IO* functions:

$$\begin{aligned}&\mathit{getChar} :: \mathit{IO}\ \mathit{Char}\\&\mathit{putChar} :: \mathit{Char} \rightarrow \mathit{IO}\ ()\end{aligned}$$

getChar is a computation that reads a character from **stdin**, *putChar* is a computation that outputs the given character to **stdout**, returning the unit type. The *echo* function below binds these two functions together such that the character obtained from the *getChar* function is fed as the argument to the *putChar* function and hence *echoed* to the screen. The echo function then goes on to call itself again.

$$echo :: IO\ ()$$
$$echo = getChar \gg\!= (\lambda c \rightarrow putChar\ c) \gg echo$$

This is quite a simple example, and already the monadic notation is looking quite complicated. Imagine if we wanted to bind lots of monadic operations together, such as prompting for a file name, reading in the file name, opening the file, reading in the file contents, doing a computation with the file contents, then saving the new data back to the file. This isn't an unlikely request for a program, but it would seem that a complicated function would need to be defined in order to bind all these actions together. However, Haskell once again comes to our rescue, providing what is known as the **do** notation to simplify the creation of monadic programs. **do** notation is in fact just syntactic sugar, and the compiler converts it back to the necessary binds at compile time. The **do** notation is designed to give monadic programs a more imperative look and style. For example the echo example from above written using **do** notation:

$$\begin{aligned} echo = \mathbf{do}\ & c \leftarrow getChar \\ & putChar\ c \\ & echo \end{aligned}$$

For the rest of this chapter we shall be making extensive use of **do** notation which can be used for any monadic programs. In the next section (1.3) we shall introduce the QIO monad, in the style of the IO monad, whereby we shall be introducing its constructors and how they are used as apposed to the details of their implementation. For more details on the implementation please see section 1.6 towards the end of the chapter.

1.3 The QIO interface

The *QIO* monad (figure 1.1) provides a functional interface to quantum programming, similar to the way the *IO* monad provides an interface to conventional stateful programming. We will provide a constructive semantics of *QIO* later in section 1.6, but for the moment we will explain

```
Qbit :: ∗
QIO :: ∗ → ∗
U :: ∗
instance Monad QIO
mkQbit :: Bool → QIO Qbit
applyU :: U → QIO ()
measQbit :: Qbit → QIO Bool

instance Monoid U
swap :: Qbit → Qbit → U
cond :: Qbit → (Bool → U) → U
rot :: Qbit → ((Bool, Bool) → ℂ) → U
ulet :: Bool → (Qbit → U) → U
urev :: U → U

Prob :: ∗ → ∗
instance Monad Prob
run  :: QIO a → IO a
sim  :: QIO a → Prob a
runC :: QIO a → a
```

Fig. 1.1. The QIO API

its constructs informally. The basic idea is that our classical computer is connected to a quantum device which contains a number of qubits. The quantum device can be instructed to set qubits to one of the computational base states (i.e. $|0\rangle = \mathit{False}$ or $|1\rangle = \mathit{True}$), to perform unitary operations involving one or several qubits, or to measure qubits and observe the outcome, this operation is probabilistic.

Figure 1.1 gives a quick overview of the API: it consists of two types† *Qbit* and *U* and an operator on types *QIO*. We read *QIO a* as the type of quantum operations which return values of type *a*. The API specifies that *QIO* is a monad, i.e. provides operations for embedding functional computation (*return*) and allows sequential composition ($\gg=$). Similarly, *U* is a monoid, i.e. it has a neutral element *mempty* and an operation *mappend*. The difference reflects the fact that we cannot extract any information from a reversible computation.

As for conventional IO we use Haskell's **do** notation, which enables us to write imperative looking programs which are translated into pure functional programs using monadic combinators. As a first trivial example lets write our quantum *hello world* program, which initialises a qubit and then measures it:

```
hqw :: QIO Bool
hqw = do q ← mkQbit False
         measQbit q
```

† The kind of types is written ∗ in Haskell.

We can now either *run* our quantum program, using *run hqw*, or since it doesn't involve any non-classical steps, using *runC hqw*. *runC* is only able to run QIO computations consisting of the classical subset of available operations, and will throw an error if any quantum operations occur. We provide this functionality since the classical simulation is much more efficient and can be used to test classical (but reversible) components of a quantum program. Alternatively we can *simulate* the quantum program using *sim hqw* which calculates a probability distribution, in this case the difference is unremarkable.

A more interesting example is to apply the Hadamard transformation

$$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

to our qubit before measuring it. Given a qubit q the expression *uhad q*:: U represents this unitary, i.e. reversible computation - we'll provide the implementation later. The function $applyU :: U \rightarrow QIO\ ()$ allows us to run a unitary computation, thereby embedding reversible into non-reversible quantum computations. We arrive at the following piece of code:

```
rnd :: QIO Bool
rnd = do q ← mkQbit False
         applyU (uhad q)
         measQbit q
```

The expression *run rnd* will now produce a random result. How can this be, given that Haskell is a *pure* functional language? Indeed *run* embeds *QIO* into Haskell's IO-monad which allows us to access effects such as random computations. In contrast *sim rnd* doesn't use *IO* but returns the probability distribution: $[(True, 0.5), (False, 0.5)]$

To demonstrate that *QIO* goes beyond classical random computation we can produce a pair of entangled qubits. To do this we use the unitary conditional *cond* which allows us to construct *branching* reversible quantum programs. I.e. given $q :: Qbit$ and $t, u :: U$ the expression *cond q* $(\lambda b \rightarrow$ **if** b **then** t **else** $u)$ intuitively runs the programs t, u depending on q. Actually, if the current value of q is not a base state, both computations t and u will contribute to the result. This is the source of *quantum parallelism*.

We exploit the conditional to produce a bell pair, and measure it. Given a qubit *qa* in any state and another qubit *qb* prepared in the base state *False* we can entangle *qb* with *qa* using the one-sided *ifQ* which is

implemented as a conditional that applies the empty computation when the control qubit is False. The expression $unot\ qb :: U$ negates qb.

Putting everything together we obtain:

```
testBell :: QIO (Bool, Bool)
testBell = do qa ← mkQbit False
              applyU (uhad qa)
              qb ← mkQbit False
              applyU (ifQ qa (unot qb))
              a ← measQbit qa
              b ← measQbit qb
              return (a, b)
```

Evaluating *sim testBell* reveals that the two apparently independent measurements always agree: $[((True, True), 0.5), ((False, False), 0.5)]$. The reason is that we created an entangled quantum state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ and once we measure one of the bits it is projected into one of the base states $|00\rangle$ and $|11\rangle$ with probability $\frac{1}{2} = |\frac{1}{\sqrt{2}}|^2$.

The reader may wonder, whether *cond* or *ifQ* will always produce a unitary, i.e. reversible operation. Couldn't we irreversibly reset a qubit q by running $ifQ\ q\ (unot\ q)$? Indeed, the conditional has a semantic side condition that the computations in the branches will not change the qubit we are branching over. Haskell's type system is too weak to express this constraint but this violation will be caught by *sim* or *run* at *runtime*. This could be fixed by using a more expressive type system, e.g. Coq (The Coq development team (2004)) or Agda (Norell (2007)).

Using functional abstraction we can organise our quantum programs more succinctly. E.g. the *bell* example exploits the fact that we can *share* the state of a qubit. This can be realised by the following function:

```
share :: Qbit → QIO Qbit
share qa = do qb ← mkQbit False
              applyU (ifQ qa (unot qb))
              return qb
```

It is important to realise that the quantum state is not copied (this would contradict the no-cloning theorem), but merely shared †. That is *share qa* produces a new qubit which shares the state of the given qubit.

† Quantum sharing is used in QML (Altenkirch and Grattage (2005)) and for the linear-algebraic λ-calculus (Arrighi and Dowek (2008)) to model non-linear use of quantum variables.

We can also separate the production of a qubit in the $|+\rangle$ or $|-\rangle$ states:

```
|+⟩ :: QIO Qbit
|+⟩ = do q ← mkQbit False
         applyU (uhad q)
         return q

|−⟩ :: QIO Qbit
|−⟩ = do q ← mkQbit True
         applyU (uhad q)
         return q
```

More over we can separate the production of a bell pair and its measurement:

```
bell :: QIO (Qbit, Qbit)
bell = do qa ← |+⟩
          qb ← share qa
          return (qa, qb)
```

Now the function *bell* simply generates a bell pair. The main function *testBell* now becomes

```
testBell = do (qa, qb) ← bell
              a ← measQbit qa
              b ← measQbit qb
              return (a, b)
```

This shows that functional abstraction and quantum programming interact well, extending Haskell's functional approach for classical effects to quantum effects.

We can go further and use Haskell's class system to organise quantum data. The reader may have noticed that the functions $mkQbit :: Bool \to QIO\ Qbit$ and $measQbit :: Qbit \to QIO\ Bool$ look like inverses, relating the classical type *Bool* with the *quantum representation Qbit*. This idea extends to other types, i.e. in the example *testBell* we may want to exploit the correspondence of $(Bool, Bool)$ and $(Qbit, Qbit)$ by providing functions $mk2Qbits : (Bool, Bool) \to QIO\ (Qbit, Qbit)$ and $meas2Qbits : (Qbit, Qbit) \to QIO\ (Bool, Bool)$. Indeed, also the conditional can be lifted to pairs, while *cond* had the type $Qbit \to (Bool \to U) \to U$ the corresponding operation for pairs allows us to *branch over* a pair: $cond2Qbits : (Qbit, Qbit) \to ((Bool, Bool) \to U) \to U$.

More generally, we can introduce a **class** which allows us to relate classical data and quantum data in a systematic fashion: †

† The definition also exploits *functional dependencies* $a \to qa, qa \to a$ which are again hints to the Haskell type checker.

```
class Qdata a qa | a → qa, qa → a where
  mkQ :: a → QIO qa
  measQ :: qa → QIO a
  condQ :: qa → (a → U) → U
```

An instance of this class is qubits with booleans:

```
instance Qdata Bool Qbit where
  mkQ = mkQbit
  measQ = measQbit
  condQ q br = cond q br
```

and we can show that *Qdata* is *closed* under pairing:

```
instance (Qdata a qa, Qdata b qb) ⇒ Qdata (a, b) (qa, qb) where
  mkQ (a, b) = do qa ← mkQ a
                  qb ← mkQ b
                  return (qa, qb)
  measQ (qa, qb) = do a ← measQ qa
                      b ← measQ qb
                      return (a, b)
  condQ (qa, qb) br = condQ qa (λx → condQ qb (λy → br (x, y)))
```

Other instances of this class include the closure of *Qdata* over lists:

```
instance Qdata a qa ⇒ Qdata [a] [qa] where
  mkQ n = sequence (map mkQ n)
  measQ qs = sequence (map measQ qs)
  letU as xsu = letU' as []
    where letU' [] xs = xsu xs
          letU' (a : as) xs =
                letU a (λx → letU' as (xs ++ [x]))
  condQ qs qsu = condQ' qs []
    where condQ' [] xs = qsu xs
          condQ' (a : as) xs =
                condQ a (λx → condQ' as (xs ++ [x]))
```

and in section 1.4 we'll even present a quantum integer data type (*QInt*), that is a member of *Qdata* along with the classical *Int* data type.

We can now use the generic operation *measQ* to define *testBell*:

$$\begin{aligned}&\mathit{testBell} = \mathbf{do}\ \mathit{qab} \leftarrow \mathit{bell}\\ &\qquad\qquad\ \ \mathit{measQ}\ \mathit{qab}\end{aligned}$$

The complete overview of the *QIO* API is given in figure 1.1. Let's explain the operations we haven't yet covered in our examples. First note that U is actually a monoid, i.e. we can execute unitary operations sequentially using *mappend*. The neutral element is the empty computation *mempty* which is actually needed to derive *ifQ* from *cond*:

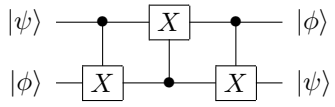
$$\begin{aligned}&\mathit{ifQ} :: \mathit{Qbit} \to U \to U\\ &\mathit{ifQ}\ q\ u = \mathit{cond}\ q\ (\lambda x \to \mathbf{if}\ x\ \mathbf{then}\ u\ \mathbf{else}\ \mathit{mempty})\end{aligned}$$

For brevity, we shall now write *mappend* as $\blacktriangleright$ and *mempty* as $\bullet$. Our examples also already exploited the fact that *QIO* is a monad, using the **do** notation which translates programs with an *imperative look* into pure monadic functional programs.

Our primitive unitary operations are *rot* and *swap*. The function *rot* allows us to apply any unitary 2×2 complex valued matrix represented as a function $(\mathit{Bool}, \mathit{Bool}) \to \mathbb{C}$ to a given qubit. We can derive the operations *uhad*, *unot* and *uphase*:

$$\begin{aligned}&\mathit{unot} :: \mathit{Qbit} \to U\\ &\mathit{unot}\ x = \mathit{rot}\ x\ (\lambda(x, y) \to \mathbf{if}\ x \equiv y\ \mathbf{then}\ 0\ \mathbf{else}\ 1)\\ &\mathit{uhad} :: \mathit{Qbit} \to U\\ &\mathit{uhad}\ x = \mathit{rot}\ x\ (\lambda(x, y) \to \mathbf{if}\ x \wedge y\ \mathbf{then} - h\ \mathbf{else}\ h)\\ &\qquad\quad \mathbf{where}\ h = (1\ /\ \mathit{sqrt}\ 2)\\ &\mathit{uphase} :: \mathit{Qbit} \to \mathbb{R} \to U\\ &\mathit{uphase}\ x\ r = \mathit{rot}\ x\ (\mathit{rphase}\ r)\\ &\mathit{rphase} :: \mathbb{R} \to \mathit{Rotation}\\ &\mathit{rphase}\ \_\ (\mathit{False}, \mathit{False}) = 1\\ &\mathit{rphase}\ r\ (\mathit{True}, \mathit{True}) \ = \mathit{exp}\ (0 :+ r)\\ &\mathit{rphase}\ \_\ (\_, \_) \qquad\quad = 0\end{aligned}$$

We also include *swap* x y as a primitive for reasons of efficiency even though this function could be derived: The swap operation is equivalent to three controlled not operations in a row as in the following circuit diagram:



which could easily be implemented in QIO as:

$$
\begin{aligned}
&\mathit{swap} :: \mathit{Qbit} \to \mathit{Qbit} \to U \\
&\mathit{swap}\ \mathit{qa}\ \mathit{qb} = \mathit{ifQ}\ \mathit{qa}\ (\mathit{unot}\ \mathit{qb}) \\
&\qquad\qquad \blacktriangleright\ \mathit{ifQ}\ \mathit{qb}\ (\mathit{unot}\ \mathit{qa}) \\
&\qquad\qquad \blacktriangleright\ \mathit{ifQ}\ \mathit{qa}\ (\mathit{unot}\ \mathit{qb})
\end{aligned}
$$

Many useful quantum computations use ancillas, i.e. auxiliary qubits, an example is quantum addition as we will see later (section 1.4). A quantum computation using ancillas is unitary, if it doesn't change the state of the ancilla. This may sound useless, however, we are allowed to use the ancilla as long as we make sure that we leave it back in the state in which we found it. The QIO API supports ancillas with the function $\mathit{ulet} :: \mathit{Bool} \to (\mathit{Qbit} \to U) \to U$: the expression $\mathit{ulet}\ b\ f$ temporarily creates a new qubit q initialised in the base state b, runs $f\ q$ and then returns q to the pool of unused qubits. Again *ulet* imposes a semantic condition: After running $f\ q$ the qubit q has to again be in the base state b. As before, this semantic condition is caught at runtime. As the *ulet* constructor is again a function relating Booleans and qubits, we are able to extend the *Qdata* class with the function:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Qdata}\ a\ \mathit{qa} \mid a \to \mathit{qa}, \mathit{qa} \to a\ \textbf{where} \\
&\quad \vdots \\
&\quad \mathit{letU} :: a \to (\mathit{qa} \to U) \to U
\end{aligned}
$$

meaning any members of the *Qdata* class must also define a letU function to allow any *Qdata* to be used as an auxiliary quantum data structure. For example the letU functions for qubits and for pairs of *Qdata* are as follows:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Qdata}\ \mathit{Bool}\ \mathit{Qbit}\ \textbf{where} \\
&\quad \vdots \\
&\quad \mathit{letU}\ b\ \mathit{xu} = \mathit{ulet}\ b\ \mathit{xu} \\
&\textbf{instance}\ (\mathit{Qdata}\ a\ \mathit{qa}, \mathit{Qdata}\ b\ \mathit{qb}) \Rightarrow \mathit{Qdata}\ (a, b)\ (\mathit{qa}, \mathit{qb})\ \textbf{where} \\
&\quad \vdots \\
&\quad \mathit{letU}\ (a, b)\ \mathit{xyu} = \mathit{letU}\ a\ (\lambda x \to \mathit{letU}\ b\ (\lambda y \to \mathit{xyu}\ (x, y)))
\end{aligned}
$$

Finally, we have an operation *urev* which calculates the inverse of a given invertible operation. The following subsections go on to give some slightly more in depth examples of quantum computations written in QIO. Section 1.3.1 gives details of our implementation of Deutsch's algorithm, and section 1.3.2 gives details of our implementation of quantum teleportation.

1.3.1 Deutsch's algorithm

Deutsch's Algorithm (Deutsch (1985)) was presented as one of the first and simplest quantum algorithms that could be proven to provide a solution to its problem quicker than any classical solution. The problem involves being given a function $f :: \mathit{Bool} \to \mathit{Bool}$ and being asked to calculate whether the function is balanced or constant. There are only four possible functions that f can be, which relate to the identity function, the not function, the constant False function or the constant True function. Classically it can be shown that two applications of f are required to tell whether it is one of the balanced or one of the constant functions, but in a quantum computer it is possible to get the answer having only to run the function f once (albeit over a quantum state).

In the QIO monad the algorithm can easily be modelled: we initialise two qubits in the $|+\rangle$ and $|-\rangle$ states, and then conditionally negate the second qubit depending on the outcome of applying f to the first qubit. We then apply the Hadamard transformation to the first qubit and measure it. This is confusing at the first glance because classically it seems that the first qubit should be unaffected by the operation we have performed. But indeed, doing the operation in the $|+\rangle$, $|-\rangle$-basis does the trick and we have to consult f only once.

$$
\begin{aligned}
&\mathit{deutsch} :: (\mathit{Bool} \to \mathit{Bool}) \to \mathit{QIO}\ \mathit{Bool}\\
&\mathit{deutsch}\ f = \mathbf{do}\ x \leftarrow |+\rangle\\
&\qquad\qquad\quad\ \ y \leftarrow |-\rangle\\
&\qquad\qquad\quad\ \ \mathit{applyU}\ (\mathit{cond}\ x\ (\lambda b \to \mathbf{if}\ f\ b\ \mathbf{then}\ \mathit{unot}\ y\ \mathbf{else}\ \bullet))\\
&\qquad\qquad\quad\ \ \mathit{applyU}\ (\mathit{uhad}\ x)\\
&\qquad\qquad\quad\ \ \mathit{measQ}\ x
\end{aligned}
$$

In either of the cases where f was a constant function then the measurement will yield *False* (with probability 1), and in the cases where f is a balanced function the measurement will yield *True* (again with probability 1).

Evaluating, $\mathit{sim}\ (\mathit{deutsch}\ \neg)$ gives $[(\mathit{True}, 1.0)]$, $\mathit{sim}\ (\mathit{deutsch}\ \mathit{id})$ also gives $[(\mathit{True}, 1.0)]$. $\mathit{sim}\ (\mathit{deutsch}\ (\lambda x \to \mathit{False}))$ gives $[(\mathit{False}, 1.0)]$, and $\mathit{sim}\ (\mathit{deutsch}\ (\lambda x \to \mathit{True}))$ also gives $[(\mathit{False}, 1.0)]$.

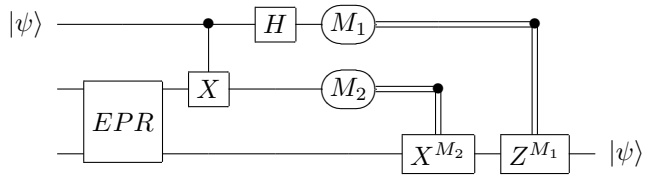
1.3.2 Quantum Teleportation

Quantum teleportation can be thought of as a process that transfers the state of a single qubit to another single qubit. It makes use of an

entangled pair of qubits that are shared between the transmitting and receiving parties (usually referred to as Alice and Bob). It doesn't break the rule of no-cloning as the state of the original qubit is lost in the process. An example text-book description of quantum teleportation as taken from (Nielsen and Chuang (2000)) is as follows:

> Alice and Bob met long ago but now live far apart. While together they generated an EPR pair, each taking one qubit of the EPR pair when they separated. Many years later, Bob is in hiding, and Alice's mission, should she choose to accept it, is to deliver a qubit $|\psi\rangle$ to Bob. She does not know the state of the qubit, and moreover can only send *classical* information to Bob. Should Alice accept the mission?
>
> Intuitively, things look pretty bad for Alice. She doesn't know the state $|\psi\rangle$ of the qubit she has to send to Bob, and the laws of quantum mechanics prevent from determining the state when she only has a single copy of $|\psi\rangle$ in her possession. What's worse, even if she did know the state $|\psi\rangle$, describing precisely takes an infinite amount of classical information since $|\psi\rangle$ takes values in a *continuous* space. So even if she did know $|\psi\rangle$, it would take forever for Alice to describe the state to Bob. It's not looking good for Alice. Fortunately for Alice, quantum teleportation is a way of utilising the entangled EPR pair in order to send $|\psi\rangle$ to Bob, with only a small overhead of classical communication.
>
> In outline, the steps of the solution are as follows: Alice interacts the qubit $|\psi\rangle$ with her half of the EPR pair, and then measures the two qubits in her possession, obtaining one of four classical results, 00, 01, 10, and 11. She sends this information to Bob. Depending on Alice's classical message, Bob performs one of four operations on his half of the EPR pair. Amazingly by doing this he can recover the original state $|\psi\rangle$!

The following quantum circuit gives a more precise description of quantum teleportation.



The top two lines represent Alice's qubits, that is the input qubit (which is in the state $|\psi\rangle$), and her qubit from the EPR pair. The bottom line represents Bob's qubit. Alice entangles the input qubit with her EPR pair qubit using a controlled Not operation, as we have shown previously in the sharing example. Then she performs a Hadamard rotation on the input qubit before measurement (which is equivalent to a measurement in the Hadamard basis). The double lines coming

from the measurements represent the classical data that she sends to Bob, who correspondingly has to perform (conditionally depending on the classical bits) an X and a Z rotation on his qubit, which will then be in the state $|\psi\rangle$. It is also clear to see that the state of the original input qubit has been lost as it will be in one of the base states, $|0\rangle$ or $|1\rangle$, depending upon the measurement outcome.

Within the QIO monad, quantum teleportation can be considered as given by 3 functions. Firstly there is what Alice has to do, secondly there is what Bob has to do, but thirdly there is the requirement that they each have one qubit from an entangled pair of qubits. This third requirement means that each part of the teleportation algorithm must take part within the same quantum system.

Alice has her initial qubit (aq) and one of the entangled pair of qubits (eq). All she has to do is apply a controlled not between these two qubits, and then perform the hadamard rotation on the first one. Finally she has to measure these two qubits and send the results of this measurement to Bob. In the QIO monad this can be coded as:

```
alice :: Qbit → Qbit → QIO (Bool, Bool)
alice aq eq = do applyU (ifQ aq (unot eq))
                 applyU (uhad aq)
                 measQ (aq, eq)
```

Bob has his qubit from the entangled pair (eq), and receives the classical data from Alice (cd). Depending on the classical data, Bob must apply the necessary unitary. Again, in the QIO Monad this can be coded as:

```
uZ :: Qbit → U
uZ qb = (uphase qb 0.5)

bobsU :: (Bool, Bool) → Qbit → U
bobsU (False, False) eq = •
bobsU (False, True)  eq = (unot eq)
bobsU (True, False)  eq = (uZ eq)
bobsU (True, True)   eq = ((unot eq) ► (uZ eq))

bob :: Qbit → (Bool, Bool) → QIO Qbit
bob eq cd = do applyU (bobsU cd eq)
               return eq
```

The teleportation algorithm can be defined by creating an entangled pair, Alice uses the input qubit and one of the pair to return the clas-

sical data. Bob can then use this classical data and his element of the entangled pair to return the teleported qubit. In the QIO Monad this is coded as:

```
teleportation :: Qbit → QIO Qbit
teleportation iq = do (eq1, eq2) ← bell
                      cd ← alice iq eq1
                      tq ← bob eq2 cd
                      return tq
```

1.4 Reversible arithmetic

Quantum computation is a superset of classical reversible computation, enabling classical reversible computations to be defined within QIO. Reversible computation can by definition have no side-effects, and indeed our *runC* function for the evaluation of the classical subset of QIO doesn't need to embed the result of the evaluation in a monad: $runC :: QIO\ a \rightarrow a$ just returns a pure result. However, as we are working in the quantum realm, it is possible to run these purely classical computations over a quantum state. In essence, this has the effect of running the computation over each base state within a given superposition. We'll see later, in section 1.5, that being able to run an albeit classical modular exponentiation function over a super-position of states, plays a major role in Shor's algorithm, and in the rest of this section we shall be looking at our library of reversible arithmetic functions in QIO, that will build up to give us this modular exponentiation function that is required. Much of our work on reversible arithmetic follows from the work in (Vedral et al. (1995)).

For this section we have created the quantum data-type *QInt* which is essentially a wrapper for a list of qubits.

```
data QInt = QInt [Qbit] deriving Show
```

We provide functions which convert between classical integers and lists of Booleans of fixed length. The length is defined by:

```
qIntSize :: Int
```

These functions (*int2bits* and *bits2int*) are then used to create an instance of the *Qdata* class between *Int* and *QInt*. The definitions mostly use the definitions for lists of *Qdata* but lifted to the *QInt* datatype.

```
instance Qdata Int QInt where
  mkQ n = do qn ← mkQ (int2bits n)
             return (QInt qn)
  measQ (QInt qbs) = do bs ← measQ qbs
                        return (bits2int bs)
  letU n xu = letU (int2bits n) (λbs → xu (QInt bs))
  condQ (QInt qi) qiu = condQ qi (λx → qiu (bits2int x))
```

In simple boolean arithmetic circuits, (such as we have defined previously in section 1.2.1) the addition of integers is performed by going through the bits, adding the corresponding bits, and keeping track of any overflow. We can express both the calculation of the current sum and the calculation of the carry as reversible algorithms:

```
sumq :: Qbit → Qbit → Qbit → U
sumq qc qa qb =
  cond qc (λc →
    cond qa (λa → if a ≢ c then unot qb else •))
carry :: Qbit → Qbit → Qbit → Qbit → U
carry qci qa qb qcsi =
  cond qci (λci →
    cond qa (λa →
      cond qb (λb →
        if ci ∧ a ∨ ci ∧ b ∨ a ∧ b then unot qcsi else •)))
```

We note that *carry* needs access to the current and the next carry-bit, while *sumq* only depends on the current qubits. Using these functions we could now implement reversible addition as a function of type

```
qadd :: QInt → QInt → QInt → Qbit → U
qadd (QInt qas) (QInt qbs) (QInt qcs) qc = qadd′ qas qbs qcs qc
    where qadd′ [] [] [] qc
                = •
          qadd′ [qa] [qb] [qci] qc
                = carry qci qa qb qc ▶
                  sumq qci qa qb
          qadd′ (qa : qas) (qb : qbs) (qci : qcsi : qcs) qc
                = carry qci qa qb qcsi ▶
                  qadd′ qas qbs (qcsi : qcs) qc ▶
                  urev (carry qci qa qb qcsi) ▶
                  sumq qci qa qb
```

The algorithm requires an additional 3rd register which needs to be initialised to $|\vec{0}\rangle$, i.e. a quantum register where each qubit is $|0\rangle$, to store the auxiliary carry bits. We have designed the algorithm so that it leaves this register in the same state $|\vec{0}\rangle$, as it has found it. This is achieved by undoing the computation of the carry bit using *urev* after it has been used. Hence, we could measure this additional register without affecting the rest of the computation. However, measuring the register means that we have to define a potentially irreversible operation living in *QIO*, which means that we cannot use addition to derive other unitary operations, which is exactly what we want to do for Shor's algorithm. The other alternative is to thread the auxiliary qubits through all the arithmetic operations we define, reusing it at other places where we need temporary qubits. This leads to a very low level design, where memory management is explicit — this leads to a drastic loss of modularity.

This is exactly the reason why we need *ulet*, which temporarily creates qubits which can be used in a unitary operation under the condition that they are restored to the state they were found in.

```
qadd :: QInt → QInt → Qbit → U
qadd (QInt qas) (QInt qbs) qc' =
  ulet False (qadd' qas qbs)
  where qadd' [] [] qc = ifQ qc (unot qc')
        qadd' (qa : qas) (qb : qbs) qc =
          ulet False (λqc' → carry qc qa qb qc' ►
                             qadd' qas qbs qc' ►
                             urev (carry qc qa qb qc')) ►
          sumq qc qa qb
```

Extending on this function for reversible addition, we can carry on following (Vedral et al. (1995)) and create the necessary functions to build up to the goal of a reversible modular exponentiation algorithm.

The next step is to create a modular addition function using the addition function we have already created.

```
adderMod :: Int → QInt → QInt → U
```

Then we can use the modular addition function to create a modular multiplication function.

```
multMod :: Int → Int → QInt → QInt → U
```

This can easily be adapted into a controlled operation with our *ifQ* constructor:

$condMultMod :: Qbit \to Int \to Int \to QInt \to QInt \to U$
$condMultMod\ q\ n\ a\ x\ y = ifQ\ q\ (multMod\ n\ a\ x\ y)$

Modular exponentiation is achieved by using iterated squaring, and takes advantage of the fact that modular inverses can be computed efficiently classically.

$modExp :: Int \to Int \to QInt \to QInt \to U$

1.5 Shor's algorithm

Shor's algorithm (Shor (1994)) is perhaps the most famous of all algorithms that have been designed specifically for use on quantum computers. It can be used to find the factors of large numbers, which in classical computation is currently believed to be computationally infeasible. The best known classical solution to the problem is an algorithm of exponential time complexity in the size of its input (the number of bits used). However, Shor's algorithm, which is a quantum algorithm, has only a polynomial time complexity. This is an exponential speed up over the best known classical solution.

Shor's algorithm is sometimes referred to as the "Killer Application" for quantum computers. This nomenclature came about because factorising large numbers was thought to be so computationally infeasible that it forms the basis of the RSA encryption protocol (Rivest et al. (1977)). The RSA encryption protocol is a very widely used public-key protocol for sending secure information over public channels such as the Internet. It works on the principle that multiplying 2 large prime numbers (p and q) is computationally easy, and a public and private key can be computed from the result ($n = p * q$). However, if it is possible for an eaves-dropper to compute p and q from n then it is also possible for them to work out the private key. If we can now factor large numbers in polynomial time this encryption scheme has effectively been broken allowing anyone with a sufficiently large quantum computer to intercept, and decode encrypted data. It is not much of a coincidence that since 1994 when Peter Shor first published his algorithm, that governments and other large organisations have been giving more and more money for the research into quantum computers and quantum computation!

Peter Shor exploited the fact that the factors of a number can be computed from the period of a given modular exponentiation function. In fact, the exponentiation function required was shown to be of the form

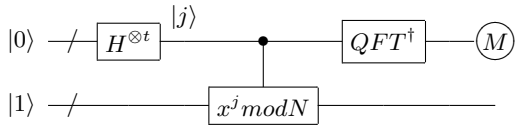


Fig. 1.2. Shor's algorithm

$f(x) = a^x mod N$ where N is the number we wish to factorise, and a is known to be co-prime to N. That is, that the greatest common divisor of a and N is 1. Calculating the greatest common divisor of two numbers can be done efficiently classically, using the Euclidean algorithm, and hence finding values for a can also be done classically. Depending on the value of a, it is possible that the period of this function can be used to find factors of the input. Shor discovered that finding the period of this function (and other periodic functions) can be done efficiently on a quantum computer. The efficiency of this period-finding algorithm comes directly from the use of quantum parallelism.

Two quantum registers (of sufficient number of qubits to represent N) are initialised into the state $|\vec{0}\rangle$. The first of these two quantum registers is placed into an equal super-position of all its possible base states (using Hadamard rotations). This register ($|k\rangle$) is then used as x in our exponential function $f(x) = a^x mod N$ such that the result of the function is stored in the second quantum register. At this point in the computation our quantum registers will be in the state ($|k, a^k mod N\rangle$) in such a way that each value of k in the first register is entangled with its corresponding $a^k mod N$ in the second register. Shor goes on to show that the application of a discrete Fourier transform to the first register will (with high probability) yield the period of the given input function.

Figure 1.2 shows us what we essentially need to implement Shor's algorithm within QIO. We can easily set up the required Hadamard rotations, and section 1.4 shows us how we've implemented the necessary modular exponentiation function, all we need now is the function implementing the inverse quantum Fourier transform.

1.5.1 Quantum Fourier transform

The Quantum Fourier Transformation (QFT) is basically the fast, discrete Fourier Transformation applied to a quantum register, where the discrete Fourier transform maps functions in the time domain into functions in the frequency domain, or in other words, decomposes a function

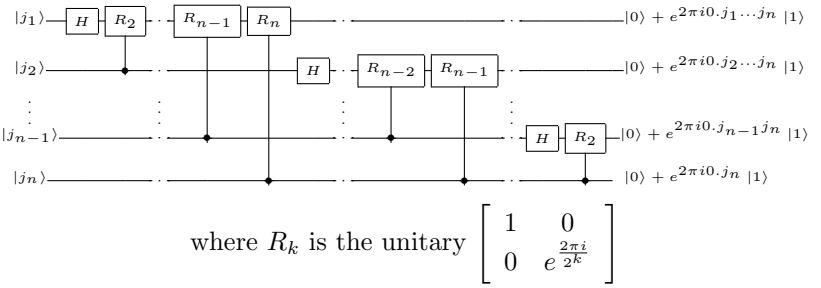


Fig. 1.3. A circuit for the Quantum Fourier transform

in terms of sinusoidal functions of different frequencies. In Shor's algorithm the inverse Fourier transformation is used to recover the frequency representation of the modular exponential, thus giving direct access to the period. The QFT as developed in (Nielsen and Chuang (2000)), pp. 216-221 (given here in figure 1.3) can be easily encoded in the QIO monad, giving rise to a nice functional implementation over a list of qubits:

```
qft :: [Qbit] → U
qft qs = condQ qs (λbs → qftAcu qs bs [])
```

The *qft* functions makes use of a conditional statement over a list of qubits, allowing the definition to be decomposed into terms of an accumulator function: †

```
qftAcu :: [Qbit] → [Bool] → [Bool] → U
qftAcu [] [] _                = •
qftAcu (q : qs) (b : bs) cs = qftBase cs q ► qftAcu qs bs (b : cs)
qftBase :: [Bool] → Qbit → U
qftBase bs q = f' bs q 2
  where f' []       q _ = uhad q
        f' (b : bs) q x = if b then (rotK x q) ► f' bs q (x + 1)
                               else f' bs q (x + 1)
rotK :: Int → Qbit → U
rotK k q = uphase q (1.0 / (2.0 ↑ k))
```

† Accumulator functions are often used in Haskell to create more efficient implementations of list processing functions, e.g. fast reverse.

Although we have created the QFT here, Shor's algorithm requires the inverse QFT. Fortunately, because of the reversible nature of unitaries, the inverse QFT can be given by *urev qft*.

1.5.2 Shor in QIO

We've now seen how we can create all the necessary pieces for Shor's algorithm, we can define the *hadamards* function recursively over the list representation of the *QInt*.

```
hadamards :: QInt → U
hadamards (QInt [ ]) = •
hadamards (QInt (q : qs)) = uhad q ► hadamards (QInt qs)
```

The inverse quantum Fourier transform can also be lifted to the *QInt* data-type.

```
qftI :: QInt → U
qftI (QInt i) = urev (qft i)
```

The unitary used by Shor's algorithm can now be given as:

```
shorU :: QInt → QInt → Int → Int → U
shorU k i1 x n = hadamards k ►
                 modExp n x k i1 ►
                 qftI k
```

To ensure that the given *QInt* arguments to the *shorU* function are correct ($|0\rangle$ and $|1\rangle$) we can now define *shor* as a function which takes two classical inputs, n the number to be factorised, and x a number which is co-prime to n. The *shor* function initialises the necessary quantum integers, the *shorU* unitary applies the Hadamard rotations to the zeroed input, then the *modExp* unitary is applied over both of the quantum integers (along with the necessary classical inputs), and then the inverse quantum Fourier transform is applied. Finally, the period is measured and returned.

```
shor :: Int → Int → QIO Int
shor x n = do i0 ← mkQ 0
              i1 ← mkQ 1
              applyU (shorU i0 i1 x n)
              p ← measQ i0
              return p
```

Finding the x argument can be done classically, and extracting the result from the returned period is also a classical problem. These can easily be programmed in Haskell and used with the QIO monad to create the overall factorisation function ($factor :: Int \to QIO\ (Int, Int)$). A typical evaluation of $run\ (factor\ 15)$ is $(5, 3)$.

1.6 Implementing QIO

In this section we shall go through the implementation of the QIO API, specifically how we realise *run*, *runC*, and *sim*. If we had a real quantum computer we could implement *run* much more efficiently, but here we use a classical random number generator to *simulate* the quantum behaviour.

To represent quantum states within a classical system, it is necessary to define a data structure that represents a super-position of classical base states. For this purpose we have defined a class of *Heaps* which are used to represent the classical base states, and then we have defined a class of *vectors* we have dubbed *VecEq*, that can hold multiple occurrences of the heaps, whereby each one is associated with its corresponding complex amplitude in the current super-position. The name *VecEq* for these vectors is derived from the way that the primitive operations available for the vectors should automatically keep the overall state normalised, and hence never allow a super-position to take up more space in memory than the number of classical base states that make up the super-position. Building the normalisation directly into the vector class in this way, follows from the fact that only types with equality can be normalised (such as our Heaps).

1.6.1 Heaps

We define the *Heap* class by

```
class Eq h ⇒ Heap h where
  initial :: h
  update :: h → Qbit → Bool → h
  (?) :: h → Qbit → Maybe Bool
  forget :: h → Qbit → h
  hswap :: h → Qbit → Qbit → h
  hswap h x y = update (update h y (fromJust (h ? x)))
                       x (fromJust (h ? y))
```

The *Maybe* monad is used here so that un-initialised qubits can have the state *Nothing* and don't lead to an undefined Heap. A Heap must provide an *initial* element, which represents an empty *Heap*, before any qubits have been initialised.

To accommodate the application of *mkQbit* the *Heap* must also come with an *update* function. The *update* function, when given a heap, a qubit, and a boolean value, should just return a new heap in which the given qubit has been updated to the given boolean value. This update function is also used whenever a rotation is applied to a state, as the only effect a rotation may have on the individual base states of a superposition is to apply the classical not function to one of the qubits in the base state.

To deal with measurements, a query function (?) must also be provided, which just applies the given heap to the given qubit, and returns the boolean value of that qubit. If the qubit isn't initialised, meaning the heap computes to *Nothing* then a suitable error message is returned. The main case where an error of this sort occurs in practise is in a conditional where the control qubit appears as a variable in one of the branches. To accommodate the throwing of a run-time error whenever such a conditional occurs, the control qubit is temporarily *forgotten* from each base state using the provided *forget* function and hence, if it is used in one of the branches, the query function will fail, returning the error.

A generic swap function on heaps, *hswap* is also provided. This swaps the positions of two qubits within a *Heap*, and is how the swap unitary is dealt with at the level of heaps.

We use Haskell's Map data type for our implementation of a *Heap*. The Map data type is an efficient implementation of key-value maps, which is exactly what we require for our heaps (between qubits and booleans), and the operations required for heaps can be directly translated into the primitive operations provided by the Maps.

```
type HeapMap = Map.Map Qbit Bool

instance Heap HeapMap where
  initial = Map.empty
  update h q b = Map.insert q b h
  h ? q = Map.lookup q h
  forget h q = Map.delete q h
```

The classical simulator *runC* uses a single *Heap* to represent the entire

state of the system, along with an integer representing the next free qubit.

$$\textbf{data}\ \mathit{StateC} = \mathit{StateC}\{\mathit{freeC} :: \mathit{Int}, \mathit{heap} :: \mathit{HeapMap}\}$$

Any QIO program that only uses classical rotations is just translated into the primitive *Heap* operations given above.

1.6.2 Vectors

To hold our quantum states we define a **class** of vectors similar to the vectors we introduced in section 1.2.3. The main difference being that these vectors (*VecEq*) require that the type they hold is a member of the *Eq* class. This is a requirement here as we know that we shall be using these vectors over the *Heap* data type, along with their complex amplitude in the current super-position. *Heap* is a member of *Eq*, and it allows us to keep the vectors normalised at the level of vector operations (specifically $\oplus$). This is achieved by combining the members of the vector that are equal, when concatenating two vectors. We have defined *VecEq* as a **class** such that we can change the specific implementation if we find a more efficient data type that fulfils the **class**.

$$\begin{aligned}
&\textbf{class}\ \mathit{VecEq}\ v\ \textbf{where}\\
&\quad \mathit{vzero} :: v\ x\ a\\
&\quad (\oplus) :: (\mathit{Eq}\ a, \mathit{Num}\ x) \Rightarrow v\ x\ a \to v\ x\ a \to v\ x\ a\\
&\quad (\otimes) :: (\mathit{Num}\ x) \Rightarrow x \to v\ x\ a \to v\ x\ a\\
&\quad (@) :: (\mathit{Eq}\ a, \mathit{Num}\ x) \Rightarrow a \to v\ x\ a \to x\\
&\quad \mathit{fromList} :: [(a, x)] \to v\ x\ a\\
&\quad \mathit{toList}\ \ \ :: v\ x\ a \to [(a, x)]
\end{aligned}$$

For the moment the implementation we're using for *VecEq* is again lists of pairs, as in the *Vec* example. The main difference now being how the $\oplus$ operation works:

$$\begin{aligned}
&\textbf{data}\ \mathit{VecEqL}\ x\ a = \mathit{VecEqL}\{\mathit{unVecEqL} :: [(a, x)]\}\ \textbf{deriving}\ \mathit{Show}\\
&\mathit{vEqZero} :: \mathit{VecEqL}\ x\ a\\
&\mathit{vEqZero} = \mathit{VecEqL}\ [\,]
\end{aligned}$$

The $\oplus$ function (*vEqPlus*) uses a fold operation to *add* each element from the first vector to the second vector. The *add* function takes care of this by combining elements that occur in both vectors by adding their numeric values.

```
vEqPlus :: (Eq a, Num x) ⇒
           VecEqL x a → VecEqL x a → VecEqL x a
(VecEqL as) `vEqPlus` vbs = foldr add vbs as

add :: (Eq a, Num x) ⇒ (a, x) → VecEqL x a → VecEqL x a
add (a, x) (VecEqL axs) = VecEqL (addV' axs)
  where addV' [] = [(a, x)]
        addV' ((by@(b, y)) : bys) | a ≡ b = (b, x + y) : bys
                                  | otherwise = by : (addV' bys)
```

The $\otimes$ and @ functions (*vEqTimes* and *vEqAt*), are equivalent to the functions we saw for the vectors in section 1.2.3.

```
vEqTimes :: (Num x) ⇒ x → VecEqL x a → VecEqL x a
l `vEqTimes` (VecEqL bs) | l ≡ 0 = VecEqL []
                         | otherwise
                           = VecEqL (map (λ(b, k) → (b, l ∗ k)) bs)
vEqAt :: (Eq a, Num x) ⇒ a → VecEqL x a → x
a `vEqAt` (VecEqL []) = 0
a `vEqAt` (VecEqL ((a', b) : abs)) | a ≡ a' = b
                                   | otherwise
                                     = a `vEqAt` (VecEqL abs)
instance VecEq VecEqL where
  vzero = vEqZero
  (⊕) = vEqPlus
  (⊗) = vEqTimes
  (@) = vEqAt
  fromList as = VecEqL as
  toList (VecEqL as) = as
```

The requirement that the underlying type is a member of *Eq* leads to a problem that we can no longer define *VeqEq* as a monad, which we require so we can sequence the QIO operations over it. To overcome this problem we use a technique suggested in (Sittampalam (2008)). We define an *EqMonad* class

```
class EqMonad m where
  eqReturn :: Eq a ⇒ a → m a
  eqBind :: (Eq a, Eq b) ⇒ m a → (a → m b) → m b
```

of which *VecEq* is a member:

```
instance (VecEq v, Num x) ⇒ EqMonad (v x) where
  eqReturn a = fromList [(a, 1)]
  eqBind va f = case toList va of
    ([]) → vzero
    ((a, x) : []) → x ⊗ f a
    ((a, x) : vas) → (x ⊗ f a) ⊕ ((fromList vas) `eqBind` f)
```

Members of the *EqMonad* class can now be *embedded* into a monad using the *AsMonad* data-type.

```
data AsMonad m a where
  Embed :: (EqMonad m, Eq a) ⇒ m a → AsMonad m a
  Return :: EqMonad m ⇒ a → AsMonad m a
  Bind   :: EqMonad m ⇒
            AsMonad m a → (a → AsMonad m b) → AsMonad m b
instance EqMonad m ⇒ Monad (AsMonad m) where
  return = Return
  (>>=) = Bind
```

In order to make use of our *embedded EqMonad* we provide the following *unEmbed* function.

```
unEmbed :: Eq a ⇒ AsMonad m a → m a
unEmbed (Embed m) = m
unEmbed (Return a) = eqReturn a
unEmbed (Bind (Embed m) f) = m `eqBind` (unEmbed ∘ f)
unEmbed (Bind (Return a) f) = unEmbed (f a)
unEmbed (Bind (Bind m f) g) = unEmbed
                              (Bind m (λx → Bind (f x) g))
```

Now we have our *VecEq* as monads we can see how our unitary operations (U) in QIO are translated into operations on them.

1.6.3 Evaluating QIO computations

We start by defining the data-type *Pure* as a vector of heaps along with their complex amplitudes:

```
type Pure = VecEqL ℂ HeapMap
```

We now define a type *Unitary* which are functions that take a *Heap* and return a new *Pure* value. We will see that all members of our U data-type are converted into a *Unitary* which is then used to evaluate

the specific U. The Unitary data-type is also defined as a monoid so that unitaries can be combined using the (embedded) monadic structure of the underlying *Pure* data-type.

```
data Unitary = U{unU :: Int → HeapMap → Pure}
instance Monoid Unitary where
  • = U (λfv h → unEmbed (return h))
  (U f) ▶ (U g) = U (λfv h → unEmbed (do h' ← Embed (f fv h)
                                         h'' ← Embed (g fv h')
                                         return h''
  ))
```

Defining the *Unitary* data-type as a monoid in this way allows the unitary operations to be mapped over a whole *Pure* data-type as is required to run a unitary over the whole quantum state:

```
runU :: U → Unitary
runU UReturn = •
runU (Rot x a u) = uRot x a ▶ runU u
runU (Swap x y u) = uSwap x y ▶ runU u
runU (Cond x us u) = uCond x (runU ∘ us) ▶ runU u
runU (Ulet b xu u) = uLet b (runU ∘ xu) ▶ runU u
```

The functions *uRot*, *uSwap*, *uCond* and *uLet* actually convert each of the underlying U type into a *Unitary*. For example, the *uRot* function is as follows:

```
uRot :: Qbit → Rotation → Unitary
uRot q r = (uMatrix q (r (False, False),
                       r (False, True),
                       r (True, False),
                       r (True, True)))
uMatrix :: Qbit → (ℂ, ℂ, ℂ, ℂ) → Unitary
uMatrix q (m00, m01, m10, m11) = U (λfv h → (
          if (fromJust (h ? q))
          then (m01 ⊗ (unEmbed (return (update h q False))))
               ⊕ (m11 ⊗ (unEmbed (return h)))
          else (m00 ⊗ (unEmbed (return h)))
               ⊕ (m10 ⊗ (unEmbed (return (update h q True))))))
```

We can define our state as a free variable (the next available qubit), along with a *Pure*.

```
data StateQ = StateQ{free :: Int, pure :: Pure}
```

We make use of the *State* monad to thread our state throughout our computations

```
evalWith :: PMonad m ⇒ QIO a → State StateQ (m a)
```

The *QReturn* value is just the *return* of the monad.

```
evalWith (QReturn a) = return (return a)
```

initialising a qubit updates the current *Pure* state with the value of the new qubit, and increases the free variable by one.

```
evalWith (MkQbit b g) = do (StateQ f p) ← get
                           put (StateQ (f + 1)
                             (updateP p (Qbit f) b))
                           evalWith (g (Qbit f))
```

applying a unitary makes use of the *runU* function as described above.

```
evalWith (ApplyU u q) = do (StateQ f p) ← get
                           put (StateQ f (unEmbed (
                                         do x ← Embed (p)
                                            x′ ← Embed (uu f x)
                                            return x′
                               )))
                           evalWith q
                             where U uu = runU u
```

measurement splits the *Pure* state about the given qubit, then continues the computation in both *Pure* states, before leaving the merging of these two results up to the *merge* function of whichever *PMonad* we are working within.

```
evalWith (Meas x g) = do (StateQ f p) ← get
                         (let Split pr ift iff = split p x
                           in if pr < 0 ∨ pr > 1
                              then error "pr < 0 or >1"
                              else do put (StateQ f ift)
                                      pift ← evalWith (g True)
                                      put (StateQ f iff)
                                      piff ← evalWith (g False)
                                      return (merge pr pift piff))
```

Lazy evaluation is specifically useful here as if the *merge* function only requires one of the *Pure* states (as is the case for the *run* function), then the other *Pure* state is not evaluated.

The final definitions for *run* and *sim* simply call the *eval* function, but their given types inform the type-system which *PMonad* it should use.

$$
\begin{aligned}
&\mathit{eval} :: \mathit{PMonad}\ m \Rightarrow \mathit{QIO}\ a \to m\ a \\
&\mathit{eval}\ p = \mathit{evalState}\ (\mathit{evalWith}\ p)\ \mathit{initialStateQ} \\
&\mathit{run} :: \mathit{QIO}\ a \to \mathit{IO}\ a \\
&\mathit{run} = \mathit{eval} \\
&\mathit{sim} :: \mathit{QIO}\ a \to \mathit{Prob}\ a \\
&\mathit{sim} = \mathit{eval}
\end{aligned}
$$

1.7 Conclusions and further work

This work has proved to be a nice example of monadic programming, and is hopefully also a footstep into the world of quantum computation for many functional programmers. The work also leads us onto many more ideas of what we would like to achieve in the realm of quantum computation. Our first thought on extending *QIO* is to implement a dependently typed version of *QIO* in either Coq or Agda, thus moving the checking of the semantic side conditions from runtime to the type-checker at compile time. This change will allow the type U of unitary operations to be easier to reason about, leading to a nicer algorithmic approach to the design of quantum computations within *QIO*. Moving from functional programming to Type Theory also turns our programs into a formal semantics for quantum programs, with the possibility to develop formally verified quantum programs.

An underlying design factor of the *QIO* monad is its relation to the circuit model of quantum computation. However, we are interested in providing a different semantics more in keeping with the ideas presented in the measurement calculus (Danos et al. (2007)). The idea would be the same, that *QIO* will provide a interface from functional programming to quantum computation, but this different semantics will be available if the ideas of one-way quantum computation become the standard way in which physicists are able to create *real world* quantum computers.

Bibliography

Altenkirch, T. and Grattage, J. (2005) A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society. Also arXiv:quant-ph/0409065.

Arrighi, P. and Dowek, G. (2008) Linear-Algebraic A-Calculus: Higher-Order, Encodings, and Confluence. In *Rewriting Techniques and Applications: 19th International Conference, RTA 2008 Hagenberg, Austria, July 15-17, 2008, Proceedings*, page 17. Springer.

Danos, V., Kashefi, E. and Panangaden, P. (2007) The measurement calculus. *Journal of the ACM* **54**(2). Preliminary version in arXiv:quant-ph/0412135.

Deutsch, D. (1985) Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* **400**(1818):97–117.

Gay, S. J. (2006) Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science* **16**(4).

Green, A. (2008) The Quantum IO Monad, source code and examples. http://www.cs.nott.ac.uk/˜asg/QIO/.

Hutton, G. (2007) *Programming in Haskell*. Cambridge University Press.

Jones, S. P. (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Karczmarczuk, J. (2003) Structure and interpretation of quantum mechanics — a functional framework. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press.

The Coq development team (2004) *The Coq proof assistant reference manual*. LogiCal Project. Version 8.0.

Mu, S.-C. and Bird, R. (2001) Functional quantum programming. In *Proceedings of the 2nd Asian Workshop on Programming Languages and Systems*.

Nielsen, M. A. and Chuang, I. L. (2000) *Quantum Computation and Quantum Information*. Cambridge University Press.

Norell, U. (2007) *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Rivest, R. L., Shamir, A. and Adelman, L. M. (1977) A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82.

Rüdiger, R. (2007) Quantum programming languages: An introductory overview. *The Computer Journal* **50**(2):134–150.

Sabry, A. (2003) Modelling quantum computing in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press.

Shor, P. (1994) Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*. CA: IEEE Press.

Sittampalam, G. (2008) Restricted monads in Haskell, live journal entry. http://hsenag.livejournal.com/11803.html.

Swierstra, W. (2008) *A Functional Specification of Effects*. Ph.D. thesis, University of Nottingham.

Swierstra, W. and Altenkirch, T. (2007) Beauty in the beast: A functional

semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*.

Swierstra, W. and Altenkirch, T. (2008) Dependent types for distributed arrays. Presented at Trends in Functional Programming (TFP 2008). Submitted for final poceedings.

Vedral, V., Barenco, A. and Ekert, A. (1995) Quantum networks for elementary arithmetic operations.

Vizzotto, J. K., Altenkirch, T. and Sabry, A. (2006) Structuring quantum effects: Superoperators as arrows. *Mathematical Structures in Computer Science* **16**(3). Also arXiv:quant-ph/0501151.