Chapter 1

Shor in Haskell
The Quantum IO Monad

Alexander S. Green[1], Thorsten Altenkirch[1]
Category: Research

Abstract: We present an interface from Haskell to quantum programming: the Quantum IO monad, and use it to implement Shor's factorisation algorithm. 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. Exploiting Haskell's class system we can present our algorithms in a high level way, implementing abstractions in the functional paradigm. We describe the implementation of Shor's algorithm in some detail also covering the necessary reversible arithmetic. QIO programs can be simulated either by calculating a probability distribution or by embedding it into the IO monad using the random number generator.

1.1 INTRODUCTION

Quantum programming exploits the strange nature of quantum physics to achieve classically impossible 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 [6], 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 [4] for a recent survey.

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*). One big attraction of this approach is

[1]The University of Nottingham, UK; E-mails: {asg,txa}@cs.nott.ac.uk

that we can exploit the 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.

While *QIO* realises the infrastructure we need to control a quantum computer from Haskell, we don't have to wait until the physicists get their act together, we can use the same interface to run a quantum simulator. Our approach is inspired by the 2nd authors work with Wouter Swierstra on functional specifications of IO [11]. 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 paper, i.e. the implementation of QIO and the quantum algorithms implemented in it are available from the first authors web-page [5].

Related work

There are a number of papers on modelling quantum programming in Haskell, [8, 10, 7, 13] describe different abstractions one can use to simulate quantum computation in a functional setting - however none uses a monadic approach in the spirit of the IO monad. Also none of the previous work is intended to provide an interface to a hypothetical quantum computer. Our previous work on QML [1] 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.

Overview of the paper

In section 1.2 we introduce the quantum IO monad as an interface to quantum programming, then in section 1.3 we discuss the *Qdata* class which relates quantum and classical data types. In section 1.4 we give some simple examples: we implement sharing of quantum data and Deutsch's algorithm. Our main goal is to describe our implementation of Shor's algorithm (section 1.5), to achieve this goal we have to realise reversible arithmetic (section 1.6) and the Quantum Fourier transformation (section 1.7). Finally, we discuss how QIO can actually be implemented in Haskell (section 1.8).

1.2 THE QIO API

In figure 1.1 we give an overview over the QIO monad: irreversible operations live in *QIO* which is the quantum analogue of *IO*. Locations for quantum bits are given by the type *Qbit*. Reversible operations, which correspond to unitary operations

$Qbit :: *$
$QIO :: * \to *$
$U :: *$
instance *Monad QIO*

$mkQbit :: Bool \to QIO\ Qbit$
$applyU :: U \to QIO\ ()$
$measQbit :: Qbit \to QIO\ Bool$

instance *Monoid U*

$swap :: Qbit \to Qbit \to U$
$cond :: Qbit \to (Bool \to U) \to U$
$rot :: Qbit \to ((Bool, Bool) \to \mathbb{C}) \to U$
$ulet :: Bool \to (Qbit \to U) \to U$

$urev :: U \to U$

$Prob :: * \to *$
instance *Monad Prob*

$run :: QIO\ a \to IO\ a$
$sim :: QIO\ a \to Prob\ a$
$runC :: QIO\ a \to a$

FIGURE 1.1. The QIO API

on a finite dimensional Hilbert space are elements of *U*, which is a Monoid. Irreversible operations are constructed using *mkQbit*, *applyU* and *measQBit*, while reversible ones are constructed using *swap*, *cond*, *rot* and *ulet*. Since computations living in *U* are reversible, there is an operation *urev* constructing the reverse computation. As mentioned in the introduction, we can *run* our quantum operations using the random number generator, we can simulate (*sim*) them, giving rise to a probability distribution *Prob*. Computations living in the classical subset can be more efficiently simulated using *runC*, however this operation will return an error if applied to a non-classical computation.

Qubit initialisation and Measurement

The basic type on which quantum computations can be performed is the qubit (*Qbit*) representing the location of a qubit in the quantum memory. *Qbit* resembles *IORefs* in the conventional *IO* monad, but we are restricted to only one data-type here. As is the case for *IORefs* we can create new *Qbit*s and initialise them using $mkQbit :: Bool \to QIO\ Qbit$. This operation affects the allocation of quantum cells in the classical part of the computer and the state of the quantum memory. We can access qubits by measuring them using $measQbit :: Qbit \to QIO\ Bool$ which measures and hence collapses it to a classical base state.

Unitary transformations

Along with the ability to initialise qubits and then measure them, we need to define the unitary transformations corresponding to reversible operations on quantum data, which can be applied to these qubits. The unitary transformations are in essence the building blocks of quantum computation, and are used (possibly in conjunction with measurements) to actually define the programs that can be run. The Quantum IO Monad uses the function $applyU :: U \to QIO\ ()$ which when

given a unitary (U), is able to return a QIO computation with the unit type, and having the effect of running the unitary with the current state of the system.

The unitaries that are available form a complete model of quantum computation, the collection we have chosen is actually not minimal as some operations can be defined in term of others, but has proven useful when designing quantum algorithms:

- We can swap two qubits using $swap :: Qbit \to Qbit \to U$.
- We can branch conditionally on the state of a qubit using $cond :: Qbit \to (Bool \to U) \to U$, unlike measurement this operation doesn't change the quantum state irreversibly. A special case is the one-sided *ifQ*:

  $ifQ :: Qbit \to U \to U$
  $ifQ\ q\ u = cond\ q\ (\lambda x \to \textbf{if}\ x\ \textbf{then}\ u\ \textbf{else}\ mempty)$
- We can *rotate* a single qubit using $rot :: Qbit \to ((Bool, Bool) \to \mathbb{C}) \to U$ This corresponds to a rotation of the *Bloch sphere* which represents the state of a single qubit. A rotation is given as a unitary matrix represented as a function $(Bool, Bool) \to \mathbb{C}$. We use some predefined rotations such as
  $unot = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $uhad = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ and $uphase\ \phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i\phi} \end{bmatrix}$.
- We can temporarily allocate a qubit and use it in a local computation using $ulet :: Bool \to (Qbit \to U) \to U$, here the sub-computation is applied to the temporarily allocated qubit.

These operations are subject to some side conditions to ensure that the resulting operation is indeed a unitary transformation. E.g. *cond* requires that the branches do not change the state of the qubit we are branching over. Otherwise we could define the following operation

$notUnitary :: U$
$notUnitary = cond\ x\ (\lambda x \to \textbf{if}\ x\ \textbf{then}\ unot\ x\ \textbf{else}\ mempty)$

which will always leave the qubit x in the state *False* and is hence not reversible (and not unitary). Other conditions which have to be checked are that the specified rotation is indeed a unitary and that the local computation restores the auxiliary qubit to the state it was initialised in. Failure to meet these conditions results in a run-time error.

Combining Unitaries

As mentioned previously, quantum computations are built up using these simple unitary transformations. It is important to note that they form a Monoid and thus can be sequentially combined onto one another using Haskell's monoidal operations, we shall write ► for *mappend* and • for *mempty* to improve readability. Another useful note is that, by definition, all the computations that can be constructed with the available unitaries, are themselves unitaries, and hence reversible. We provide a function $urev :: U \to U$ which returns the inverse of the given unitary.

Evaluation of QIO Programs

We have now seen the main API for creating programs in the QIO Monad, but what can we do with these programs? QIO also provides three functions for evaluating the programs. First, there is the quantum simulator function, $sim :: QIO\ a \rightarrow Prob\ a$, which given a QIO program will return a probability distribution of the measured states. Here *Prob* is a monad derived from our monadic representation of (generalised) vector spaces (*Vec*), which we will explain later in section 1.8. For example, simulating the *randomBool* function (given in section 1.4) gives the distribution $[(True, 0.5), (False, 0.5)]$. The second means of evaluation is the quantum run function, $run :: QIO\ a \rightarrow IO\ a$, which uses the random number generator from the IO Monad to (probabilistically) return a single value for each measurement. So running the *randomBool* function will give *True* half the time, and *False* half the time. The last function that we introduce for evaluating QIO programs is the classical run function, $runC :: QIO\ a \rightarrow a$, which can only be used to run QIO programs that consist of the classical subset of the available unitaries. As these programs don't have any side-effects the returned value is just a pure value. If the *runC* function is called with a QIO program containing non-classical unitaries then it will return an error.

1.3 REPRESENTING QUANTUM DATA

In classical computation, we hardly ever think of computations acting on single bits, and it is much more useful to think of computations as acting on larger data-types. The *Qdata* class can be used for constructing quantum data-types from qubits (and other previously defined quantum data-types) in much the same way.

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

The constructors of *Qdata* define a relation between the new quantum data-type and its classical counter-part by defining three functions which must be provided. It must be possible for a member of the quantum data-type to be initialised from its classical counter-part and it must also be possible to measure the quantum data and to get a member of the classical data-type. The third constructor is used to create conditional operations that can depend on the state of the whole quantum data-type. This *condQ* operation is useful for many of the quantum algorithms that we wish to model. The simplest example of an instance of *Qdata* would be with booleans and qubits as follows:

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

It is easy to see that *Qdata* is closed under pairing:

```
instance (Qdata a qa, Qdata b qb) ⇒ Qdata (a, b) (qa, qb)
```

but even more interesting we can also create quantum lists:

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

Using the previous instance we can implement quantum integers, once we fix the size of quantum registers, e.g. $qIntSize = 8$. Quantum integers are simply a wrapper around lists of qubits:

```
newtype QInt = QInt [Qbit]

instance Qdata Int QInt
```

1.4 SIMPLE EXAMPLES

To illustrate some simple uses of the Quantum IO monad we describe how to implement sharing and then present one of the simplest but still interesting quantum algorithms: Deutsch's algorithm. We also have implemented the quantum teleport protocol but we have to omit this from the this paper due to lack of space.

Quantum data sharing

The simplest example is to just initialise a qubit into one of its base states, and simply return that qubit. The two following examples are for each base state ($|0\rangle, |1\rangle$) [1] respectively.

```
|0⟩, |1⟩ :: QIO Qbit
|0⟩ = mkQbit False
|1⟩ = mkQbit True
```

A slightly more interesting example would be to create a qubit that is in a super-position. We know that we have the Hadamard operation available to us which takes a qubit from either of its base states into an equal super-position of both, so we can just use that to produce either of the states $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + |1\rangle$ or $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - |1\rangle$.

```
|+⟩ :: QIO Qbit                 |−⟩ :: QIO Qbit
|+⟩ = do q ← |0⟩                |−⟩ = do q ← |1⟩
        applyU (uhad q)                 applyU (uhad q)
        return q                        return q
```

Interestingly, these two qubits are in different quantum states, but their behaviour upon measurement is the same. Both of these qubits, when measured, will collapse into one of the base states ($|0\rangle, |1\rangle$) with equal probability. So we could create a quantum computation that returns a random Boolean value by creating either of the above states and measuring it. For example, using $|+\rangle$, we'd get:

[1] We use the ket notation introduced by Dirac to denote states.

$$
\begin{aligned}
&\mathit{randomBool} :: \mathit{QIO\ Bool}\\
&\mathit{randomBool} = \mathbf{do}\ q \leftarrow |+\rangle\\
&\qquad\qquad\quad c \leftarrow \mathit{measQbit}\\
&\qquad\qquad\quad \mathit{return}\ c
\end{aligned}
$$

Another interesting aspect of quantum computing is the no-cloning theorem, which states that for an arbitrary quantum state $|\psi\rangle$ there is no operation to create a clone of that state. For example you cannot use the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to create the state $(\alpha|0\rangle + \beta|1\rangle) \otimes (\alpha|0\rangle + \beta|1\rangle)$. However, it is possible to create the state $\alpha|00\rangle + \beta|11\rangle$ whereby the complex amplitudes of the original state are now "shared" in an entangled state of the two qubits. This sharing operation can be achieved using a controlled not operation with the input state $|\psi\rangle$ acting as the control qubit, over a new qubit initialised in the state $|0\rangle$. This operation is not equivalent to a cloning of the original state because the two output qubits are entangled, meaning that operations on one of them may have side-effects on the other. This sharing operation can easily be modelled in the QIO Monad.

$$
\begin{aligned}
&\mathit{share} :: \mathit{Qbit} \rightarrow \mathit{QIO\ Qbit}\\
&\mathit{share}\ \mathit{qa} = \mathbf{do}\ \mathit{qb} \leftarrow |0\rangle\\
&\qquad\qquad\quad \mathit{applyU}\ (\mathit{ifQ}\ \mathit{qa}\ (\mathit{unot}\ \mathit{qb}))\\
&\qquad\qquad\quad \mathit{return}\ \mathit{qb}
\end{aligned}
$$

It is modelled as a function that takes a qubit, and returns the new qubit with which the input state is now entangled.

A bell state [2] is a maximally entangled quantum state of two-qubits, and follows from John S. Bell's famous Bell inequality. The correlations between the two entangled qubits cannot be explained without quantum mechanics, and are the foundations behind the concept of quantum teleportation. The *share* function can easily be used to create a bell state as follows:

$$
\begin{aligned}
&\mathit{bell} :: \mathit{QIO}\ (\mathit{Qbit}, \mathit{Qbit})\\
&\mathit{bell} = \mathbf{do}\ \mathit{qa} \leftarrow |+\rangle\\
&\qquad\qquad\quad \mathit{qb} \leftarrow \mathit{share}\ \mathit{qa}\\
&\qquad\qquad\quad \mathit{return}\ (\mathit{qa}, \mathit{qb})
\end{aligned}
$$

The $|+\rangle$ function creates a qubit in the state $|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and this is then shared with a new qubit. The function returns a pair of qubits in the state $|\phi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, which is a bell state in the $|0\rangle$, $|1\rangle$ basis.

Deutsch's Algorithm

Deutsch's Algorithm [3] 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 $f :: \mathit{Bool} \rightarrow \mathit{Bool}$ 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 had 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 *qplus* and *qminus* states, and then conditionally negate the second qubit depending on f applied to the first qubit. Then we 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 *qminus*, *qplus*-base does the trick and we have to consult f only once.

```
deutsch :: (Bool → Bool) → QIO Bool
deutsch f = do x ← |+⟩
               y ← |−⟩
               applyU (cond x (λb → if f b
                                    then unot y
                                    else •)
               applyU (uhad x)
               measQ x
```

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).

1.5 SHOR'S ALGORITHM

In this section we present Shor's algorithm to factor integers, which consists of a classical probabilistic algorithm reducing factorisation to period finding and a quantum part which solves the latter problem. The quantum algorithm relies on implementations of reversible arithmetic and on the Quantum Fourier Transform, which we describe in subsequent sections.

Reduction of factorisation to period finding

The reduction of factorisation to period finding shows us that for finding the factors of N, we need to find a value $x < N$ which is co-prime to N, e.g. $x < N$ such that $gcd(x,N) = 1$. Which can be done classically with the help of the random number generator in the IO monad.

```
rand_coprime :: Int → IO Int
rand_coprime n = do x ← Random.randomRIO (2, n)
                    if gcd x n ≡ 1 then return x else rand_coprime n
```

Using the values of x and N it is necessary to create a function $f(j) = x^j mod N$ which can be implemented such that j can be in a quantum state. Shor's algorithm is used to find the period, a, of this function, which can then (hopefully) be used to find factors of N. In fact, the value a returned by this method cannot always be used to find the factors of N (e.g. if a is odd, or $x^{a/2} = -1 mod N$), if this is the case then it is necessary to start again with a different value for x. It can be shown that a suitable value for a will be returned using this method with a probability of at least $\frac{1}{2}$. The following code assumes that we have already implemented

shor :: *Int* → *Int* → *QIO Int* to return a value for *a*.

```
factor :: Int → IO (Int, Int)
factor n | even n      = return (2,2)
         | otherwise = do x ← rand_coprime n
                          a ← run (shor x n)
                          let xa = x ↑ (a / 2)
                            in if odd a ∨ xa ≡ (n − 1) `mod` n
                               then factor n
                               else return (gcd (xa + 1) n, gcd (xa − 1) n)
```

Once a suitable value for *a* is found then we know that one of $gcd(x^{a/2} \pm 1, N)$ is a non-trivial factor of N. So at least one of the values returned by the *factor* function will be a non-trivial factor of the input. The only part of the algorithm that requires a quantum computer to be calculated efficiently is the use of Shor's algorithm to find the period of the function $x^j mod N$.

Period finding

We will now explain how to implement *shor* using reversible arithmetic and the quantum Fourier transform whose implementations are explained in sections 1.6 and 1.7.

The circuit in Figure 1.2 shows a simplified solution to Shor's algorithm, over the necessary $x^j mod N$ function. The inputs to the circuit are two quantum regis-

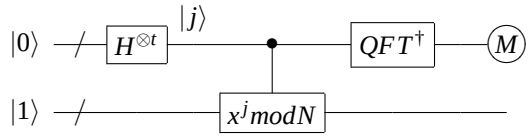

FIGURE 1.2. Shor's algorithm

ters. The algorithm first uses Hadamard rotations to put one of the quantum registers into a super-position, then uses this super-position to conditionally apply the given $x^j mod N$ function to the second qubit register. An application of the inverse Quantum Fourier Transform finishes off the algorithm before a measurement of the top register gets the result

```
hadamards :: QInt → U
hadamards (QInt [])       = •
hadamards (QInt (x : xs)) = uhad x ► hadamards (QInt xs)

shorU :: QInt → QInt → QInt → Int → U
shorU i0 i1 x n = hadamards i0 ►
                  condQ i0 (λa → modExp n a x i1) ►
                  urev (qft i0)

shor :: Int → Int → QIO Int
shor x n = do ((i0, i1), qx) ← mkQ ((0,1), x)
```

```
applyU (shorU i0 i1 qx n)
p ← measQ i0
return p
```

1.6 REVERSIBLE ARITHMETIC

Our presentation here is based on the circuits for reversible arithmetic as described in [12]. The paper describes the circuits building up from a simple reversible adder, up-to a circuit that performs modular exponentiation. These circuits have to be able to act on registers of qubits such that they perform the correct operation even when for example adding two quantum registers that are in super-positions of any of their possible base states. This basically means that every circuit for performing quantum arithmetic must be a reversible circuit. We are using the Quantum IO Monad, and not quantum circuits, so we shall go through in detail how our arithmetic functions work and how they relate to the circuits in [12].

Reversible Addition

In simple boolean arithmetic circuits, 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 abs 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 0 to store the auxiliary carry bits. We have designed the algorithm so that it leaves this register in the same state 0, as it has found it. 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 in the Shor 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.

$$
\begin{array}{l}
qadd :: QInt \rightarrow QInt \rightarrow Qbit \rightarrow U \\
qadd\ (QInt\ qas)\ (QInt\ qbs)\ qc = \\
\quad ulet\ False\ (qadd'\ qas\ qbs) \\
\quad \textbf{where}\ qadd'\ [\,]\ \ [\,]\ \ qc = ifQ\ qc\ (unot\ qc') \\
\qquad qadd'\ \ (qa : qas)\ (qb : qbs)\ qc = \\
\qquad\quad ulet\ False\ (\lambda qc' \rightarrow carry\ qc\ qa\ qb\ qc' \blacktriangleright \\
\qquad\qquad aadd'\ qas\ qbs\ qc' \blacktriangleright \\
\qquad\qquad urev\ (carry\ qc\ qa\ qb\ qc')) \blacktriangleright \\
\qquad\quad sumq\ qc\ qa\ qb
\end{array}
$$

Extending on this function for reversible addition, we can carry on following [12] and create the necessary functions to build up to the goal of a reversible modular exponentiation algorithm.

1.7 THE 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 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 [9], pp. 216-221 can be easily encoded in the QIO monad, giving rise to a foldr on a list of qubits:

$$
\begin{array}{l}
qft :: [Qbit] \rightarrow U \\
qft\ qs = condQ\ qs\ (\lambda bs \rightarrow qftAcu\ qs\ bs\ [\,]) \\
qftAcu :: [Qbit] \rightarrow [Bool] \rightarrow [Bool] \rightarrow U \\
qftAcu\ [\,]\ [\,]\ \_ \quad\quad = \bullet \\
qftAcu\ (q : qs)\ (b : bs)\ cs = qftBase\ cs\ q \blacktriangleright qftAcu\ qs\ bs\ (b : cs) \\
qftBase :: [Bool] \rightarrow Qbit \rightarrow U \\
qftBase\ bs\ q = f'\ bs\ q\ 2
\end{array}
$$

$$\begin{aligned}&\textbf{where } f'\ [\,] \qquad q\ \_ = \mathit{uhad}\ q\\ &\qquad f'\ (b:bs)\ q\ x = \textbf{if } b \textbf{ then } (\mathit{rotK}\ x\ q) \blacktriangleright f'\ bs\ q\ (x+1)\\ &\qquad\qquad\qquad\qquad \textbf{else } f'\ bs\ q\ (x+1)\end{aligned}$$

$\mathit{rotK} :: \mathit{Int} \to \mathit{Qbit} \to U$

$\mathit{rotK}\ k\ q = \mathit{uphase}\ q\ (1.0 \,/\, (2.0 \uparrow k))$

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*.

1.8 IMPLEMENTING QIO

We give here only a very high level sketch of our implementation, for details please consult our code, which is available on-line [5]. As suggested in [11] we follow a two-level approach, representing quantum computation first syntactically:

$\textbf{data } U = \mathit{UReturn} \mid \mathit{Rot}\ \mathit{Qbit}\ ((\mathit{Bool},\mathit{Bool}) \to \mathbb{C})\ U$
$\quad \mid \mathit{Swap}\ \mathit{Qbit}\ \mathit{Qbit}\ U \mid \mathit{Cond}\ \mathit{Qbit}\ (\mathit{Bool} \to U)\ U \mid \mathit{Ulet}\ \mathit{Bool}\ (\mathit{Qbit} \to U)\ U$

$\textbf{data } \mathit{QIO}\ a = \mathit{QReturn}\ a \mid \mathit{MkQbit}\ \mathit{Bool}\ (\mathit{Qbit} \to \mathit{QIO}\ a) \mid \mathit{ApplyU}\ U\ (\mathit{QIO}\ a)$
$\quad \mid \mathit{Meas}\ \mathit{Qbit}\ (\mathit{Bool} \to \mathit{QIO}\ a)$

and then interpreting both data-types in the appropriate semantical domain. One advantage of this approach is that we can interpret *urev* as a function on the syntax, another one relevant for future work is that we can also express compilation.

The classical case

It is useful to first look at the implementation of the classical fragment, before describing the quantum case.We represent a classical heap as $\textbf{type } \mathit{Heap} = \mathit{Qbit} \to \mathit{Maybe\ Bool}$, where *Nothing* corresponds to an uninitialised bit. Classically, a unitary is represented as

$\textbf{newtype } \mathit{Unitary} = U\{\mathit{unU} :: \mathit{Int} \to \mathit{Heap} \to \mathit{Heap}\}$

where the integer argument corresponds to the number of currently allocated bits. It is straightforward to derive an instance of *Monoid*:

$\textbf{instance } \mathit{Monoid\ Unitary} \textbf{ where}$
$\quad \bullet = U\ (\lambda \mathit{fv}\ bs \to bs)$
$\quad U\ f \blacktriangleright U\ g = U\ (\lambda \mathit{fv}\ h \to g\ \mathit{fv}\ (f\ \mathit{fv}\ h))$

It is then relatively straightforward to implement the remaining operations on unitaries, e.g.

$\mathit{uLet} :: \mathit{Bool} \to (\mathit{Qbit} \to \mathit{Unitary}) \to \mathit{Unitary}$
$\mathit{uLet}\ b\ ux = U\ (\lambda \mathit{fv}\ h \to \mathit{unU}\ (ux\ \mathit{fv})\ (\mathit{fv}+1)\ (\mathit{update}\ h\ \mathit{fv}\ b))$

which uses $\mathit{update} :: \mathit{Heap} \to \mathit{Qbit} \to \mathit{Bool} \to \mathit{Heap}$. We define a quantum state as

$\textbf{data } \mathit{State} = \mathit{State}\{\mathit{fv} :: \mathit{Int}, \mathit{heap} :: \mathit{Heap}\}$

and implement the classical *QIO* fragment by interpreting both *U* and *QIO* using:

$\mathit{runU} :: U \to \mathit{Unitary}$

$\mathit{runQState} :: \mathit{QIO}\ a \to \mathit{State} \to a$

The 2nd function could have been defined using the state monad. We derive *runC* by applying *runQState* to an initial empty state.

Representing vectors

Before we can embark on implementing the quantum case, we need a representation of vectors. Our current approach is based on earlier work, in particular the representation used in [13]. We basically represent a vector as an association list associating amplitudes with values, e.g. heaps:

$\textbf{newtype}\ \mathit{Vec}\ x\ a = \mathit{Vec}\{\mathit{unVec} :: [(a,x)]\}\ \textbf{deriving}\ \mathit{Show}$

This structure gives rise to a monad:

$\textbf{instance}\ \mathit{Num}\ n \Rightarrow \mathit{Monad}\ (\mathit{Vec}\ n)\ \textbf{where}$

$\quad \mathit{return}\ a = \mathit{Vec}\ [(a,1)]$

$\quad (\mathit{Vec}\ ms) \gg\!= f = \mathit{Vec}\ [(b, i*j) \mid (a,i) \leftarrow ms, (b,j) \leftarrow \mathit{unVec}\ (f\ a)]$

In this approach we are not immediately adding up tuples whose first component is equal. While it would be more efficent to do so, we would loose that *Vec* is a monad, thus complicating our implementation. Instead we define an operation:

$\mathit{norm} :: \mathit{Num}\ x \Rightarrow (a \to a \to \mathit{Bool}) \to (\mathit{Vec}\ x\ a) \to (\mathit{Vec}\ x\ a)$

which normalises a vector with respect to a given equality predicate. We can't use the class *Eq* here since we are going to use it on *Heap* which only has equality if we know the number of qubits currently used. We also implement the classical operation on vector spaces, i.e. addition and multiplication with a scalar:

$(\odot) :: \mathit{Num}\ x \Rightarrow x \to (\mathit{Vec}\ x\ a) \to \mathit{Vec}\ x\ a$

$(\oplus) :: (\mathit{Vec}\ x\ a) \to (\mathit{Vec}\ x\ a) \to \mathit{Vec}\ x\ a$

The quantum case

We represent a pure quantum state as a vector of complex valued heaps:

$\textbf{type}\ \mathit{Pure} = \mathit{Vec}\ \mathbb{C}\ \mathit{Heap}$

Unitaries are represented as functions from *Heap* to *Pure* indexed by the number of qubits currently in use:

$\textbf{newtype}\ \mathit{Unitary} = U\{\mathit{unU} :: \mathit{Int} \to \mathit{Heap} \to \mathit{Pure}\}$

It is useful to note that for any number of qubits n the relevant functions $\mathit{Heap} \to \mathit{Pure}$ correspond to $n \times n$ complex matrices. Indeed showing that *Unitary* is a monoid is very similar to the classical case, replacing application by a monadic bind:

$\textbf{instance}\ \mathit{Monoid}\ \mathit{Unitary}\ \textbf{where}$

$\quad \bullet = U\ (\lambda fv\ h \to \mathit{return}\ h)$

$\quad U\ f \blacktriangleright U\ g = U\ (\lambda fv\ h \to f\ fv\ h \gg\!= g\ fv)$

A quantum state is given by

$\textbf{data}\ \mathit{State} = \mathit{State}\{\mathit{free} :: \mathit{Int}, \mathit{pure} :: \mathit{Pure}\}$

To model measurement we introduce the concept of a probability monad, which allows us to merge computations with different probabilities:

$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{PMonad}\ m\ \textbf{where}$

$\quad \mathit{merge} :: \mathbb{R} \to m\ a \to m\ a \to m\ a$

The real number argument p corresponds to the probability that the first computation is chosen, the 2nd one is chosen with probability $1-p$. It is straightforward to show that *IO* is a *PMonad*:

```
instance PMonad IO where
  merge pr ift iff = do pp ← Random.randomRIO (0,1.0)
                        if pr > pp then ift else iff
```

Another useful *PMonad* is the type of probability distributions:

```
data Prob a = Prob{unProb :: Vec ℝ a}
```

which uses multiplication and addition of vectors:

```
instance PMonad Prob where
  merge pr (Prob ift) (Prob iff) = Prob ((pr ⊙ ift) ⊕ ((1 − pr) ⊙ iff))
```

To implement measurements we have to be able to calculate the probability amplitude of a given pure state:

```
pa :: Pure → ℝ
pa (Vec as) = foldr (λ(_,k) p → p + amp k) 0 as
```

We can split a pure state depending on the value of a certain qubit whilst calculating the probability amplitude:

```
data Split = Split{p :: ℝ, ifTrue, ifFalse :: Pure}

split :: Pure → Qbit → Split
```

Finally we implement evaluation wrt to any probability monad:

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

In the case for measurement we use the function *split*:

```
evalWith (Meas x g) (State f p) =
  let Split pr ift iff = split p x
  in merge pr
           (evalWith (g True) (State f ift))
           (evalWith (g False) (State f iff))
```

We can now obtain both *sim* and *run* by using the fact that both *IO* and *Prob* are probability monads and using an initial state as before in the classical case.

1.9 CONCLUSIONS AND FURTHER WORK

With the Quantum IO monad we have proposed a simple, low level interface to quantum programming, naturally extending Haskell's IO monad. Using the power of functional abstraction we can structure our quantum programming, exploiting the existing Haskell mechanisms such as type classes as we have already demonstrated with the *Qdata* class. With the *ulet*-construct we have also identified an important control structure which is essential for modular quantum algorithms.

Our current implementation is actually too inefficent to even calculate simple instances of Shor's algorithm. We will look into using more efficent data structures to be able to actually simulate the algorithm.

At various places we have come across side conditions which are inexpressible in Haskell, e.g. the side conditions for *cond* or *ulet*. We would like to explore how dependent types can be used to overcome this limitation, reimplementing

QIO in Agda, Epigram or Coq. This would also provide a base for reasoning about quantum algorithms, using a more abstract model based on density matrices. Another line of work is to implement a compiler from the Quantum IO monad into a more low level combinatorial model which could be directly executed on a quantum computer, either based on the traditional quantum gate model or on a more recent proposal such as measurement based quantum computing. At the other end, as we have already mentioned we would like to use the QIO monad as a testbed for new quantum control and data structures, continuing our work on QML.

REFERENCES

[1] T. Altenkirch and J. J. Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005*, pages 249–258. IEEE Computer Society Press, 2005.

[2] J. S. Bell. On the Einstein–Podolsky–Rosen paradox. *Physics*, 1(??):195–200, 1964.

[3] D. Deutsch. 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, 1985.

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

[5] A. Green. The Quantum IO Monad, source code. http://www.cs.nott.ac.uk/˜asg/QIO/, 2008.

[6] W. Hensinger. Quantum computing with trapped ions. invited talk, at the second QNET Workshop, December 2007.

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

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

[9] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, October 2000.

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

[11] W. Swierstra and T. Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[12] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations, 1995.

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