Shor in Haskell

and the Quantum IO Monad

Alexander S. Green and Thorsten Altenkirch

asg@cs.nott.ac.uk, txa@cs.nott.ac.uk

School of Computer Science,

The University of Nottingham

Introduction

- We would like to model Quantum Computations.

Introduction

- We would like to model Quantum Computations.
- The QIO Monad, can be thought of as a register of Qubits that plugs into your classical computer.

Introduction

- We would like to model Quantum Computations.
- The QIO Monad, can be thought of as a register of Qubits that plugs into your classical computer.
- It provides a framework for constructing quantum computations...

Introduction

- We would like to model Quantum Computations.
- The QIO Monad, can be thought of as a register of Qubits that plugs into your classical computer.
- It provides a framework for constructing quantum computations...
- ... and simulates the running of these computations.

Haskell and Monads

- Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.

Haskell and Monads

- Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.
- Monads are defined by a $return$ function, and a bind function denoted ($>>=$)

Haskell and Monads

- Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.
- Monads are defined by a $return$ function, and a bind function denoted ($>>=$)

$$\textbf{class}\ Monad\ m\ \textbf{where}$$

- $$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

$$return :: a \rightarrow m\ a$$

Haskell and Monads

- Haskell is a pure functional programming language, so any computations that may involve side effects make use of Monads.
- Monads are defined by a $return$ function, and a bind function denoted ($>>=$)
- $$\begin{aligned} &\textbf{class } Monad\ m \textbf{ where} \\ &\quad (\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\ &\quad return :: a \rightarrow m\ a \end{aligned}$$
- Haskell provides the do notation to make monadic programming easier.

‘do’ notation

- IO in Haskell takes place in the IO Monad.

‘do’ notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen

$$getChar :: IO\ \ Char$$
$$putChar :: Char \rightarrow IO\ ()$$

‘do’ notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen

$$getChar :: IO\ Char$$
$$putChar :: Char \rightarrow IO\ ()$$

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

‘do’ notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen

$$getChar :: IO\ Char$$

$$putChar :: Char \to IO\ ()$$

- $$echo :: IO\ ()$$
  $$echo = getChar \gg\!= (\lambda c \to putChar\ c) >> echo$$
- or in do notation

‘do’ notation

- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen

$$getChar :: IO\ Char$$

$$putChar :: Char \rightarrow IO\ ()$$

- $$echo :: IO\ ()$$
  $$echo = getChar \gg\!= (\lambda c \rightarrow putChar\ c) >> echo$$
- or in **do** notation
- $$\begin{aligned} echo = \mathbf{do}\ & c \leftarrow getChar \\ & putChar\ c \\ & echo \end{aligned}$$

The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.

The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The do notation provided by Haskell is very useful for this purpose.

The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The do notation provided by Haskell is very useful for this purpose.
-
$$\begin{aligned} &q0 :: QIO\ Qbit \\ &q0 = \mathbf{do}\ qb \leftarrow mkQbit\ False \\ &\qquad\qquad return\ x \end{aligned}$$

QIO Examples

- Creating the state $|+\rangle$

$$
\begin{array}{l}
qPlus :: QIO\ Qbit \\
qPlus = \mathbf{do}\ qb \leftarrow q0 \\
\qquad\qquad\quad applyU\ (uhad\ qb) \\
\qquad\qquad\quad return\ qb
\end{array}
$$

QIO Examples

- Creating the state $|+\rangle$

$$
\begin{aligned}
&qPlus :: QIO\ Qbit\\
&qPlus = \mathbf{do}\ qb \leftarrow q0\\
&\qquad\qquad applyU\ (uhad\ qb)\\
&\qquad\qquad return\ qb
\end{aligned}
$$

- Creating a bell state

$$
\begin{aligned}
&share :: Qbit \rightarrow QIO\ Qbit\\
&share\ qa = \mathbf{do}\ qb \leftarrow q0\\
&\qquad\qquad applyU\ (cond\ qa\ (\lambda a \rightarrow \mathbf{if}\ a\ \mathbf{then}\ (unot\ qb)\\
&\qquad\qquad\quad \mathbf{else}\ (mempty)))\\
&\qquad\qquad return\ qb
\end{aligned}
$$

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

QIO Examples

- Deutsch's Algorithm

$$u :: (Bool \to Bool) \to Qbit \to Qbit \to U$$

$$u\ f\ x\ y = cond\ x\ (\lambda b \to \mathbf{if}\ f\ b\ \mathbf{then}\ unot\ y\ \mathbf{else}\ mempty)$$

$$deutsch :: (Bool \to Bool) \to QIO\ \ Bool$$

$$\begin{aligned} deutsch\ f = \mathbf{do}\ & x \leftarrow qPlus \\ & y \leftarrow qMinus \\ & applyU\ (u\ f\ x\ y) \\ & applyU\ (uhad\ x) \\ & b \leftarrow measQ\ x \\ & return\ b \end{aligned}$$

QIO Design

- We have single qubit rotations, swaps, and conditionals.

QIO Design

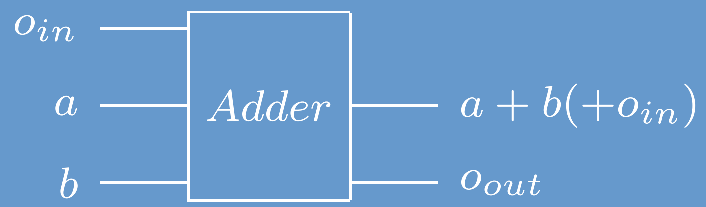
- We have single qubit rotations, swaps, and conditionals.
- We also have sequential composition in the form of the monoidal append operation, with identity $mempty$.

QIO Design

- We have single qubit rotations, swaps, and conditionals.
- We also have sequential composition in the form of the monoidal append operation, with identity $mempty$.
- QIO is for helping develop quantum algorithms, so aspects of its design follows from implementing existing algorithms.

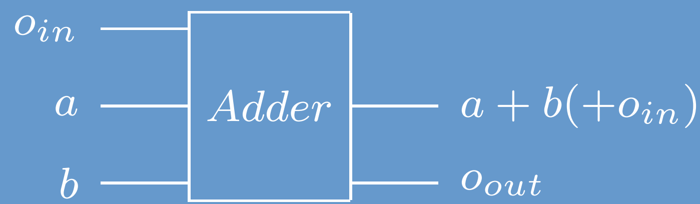
QIO Design.

- Classically, the (bit-wise) addition circuit isn't reversible.

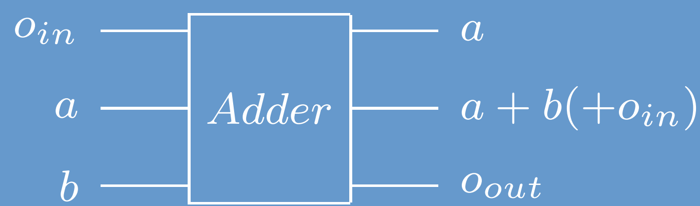


QIO Design.

- Classically, the (bit-wise) addition circuit isn't reversible.



- But this can seemingly be corrected



QIO Design..

- Classically, a full adder is created by feeding in the overflow from the previous bit-wise adder.

QIO Design..

- Classically, a full adder is created by feeding in the overflow from the previous bit-wise adder.
- However, in the quantum case, we need an auxiliary register of qubits to enable this.

QIO Design..

- Classically, a full adder is created by feeding in the overflow from the previous bit-wise adder.
- However, in the quantum case, we need an auxiliary register of qubits to enable this.
- We must also be careful to undo our carry operations so that the auxiliary qubits are not entangled with our result.

QIO Design..

The University of Nottingham

- Classically, a full adder is created by feeding in the overflow from the previous bit-wise adder.
- However, in the quantum case, we need an auxiliary register of qubits to enable this.
- We must also be careful to undo our carry operations so that the auxiliary qubits are not entangled with our result.
- The final overflow can be stored in a single qubit, giving rise to

$$adder :: [\,Qbit\,] \rightarrow [\,Qbit\,] \rightarrow [\,Qbit\,] \rightarrow Qbit \rightarrow U$$

QIO Design...

- With this definition, it would be up to the programmer to keep track of all the auxiliary qubits.

QIO Design...

- With this definition, it would be up to the programmer to keep track of all the auxiliary qubits.
- We can introduce the $ulet$ constructor

$$ulet :: Bool \rightarrow (Qbit \rightarrow U) \rightarrow U$$

QIO Design...

- With this definition, it would be up to the programmer to keep track of all the auxiliary qubits.
- We can introduce the $ulet$ constructor
  $$ulet :: Bool \rightarrow (Qbit \rightarrow U) \rightarrow U$$
- It is up to the programmer to ensure that the resulting computation is still a valid unitary.

QIO Design...

- With this definition, it would be up to the programmer to keep track of all the auxiliary qubits.
- We can introduce the *ulet* constructor

$$ulet :: Bool \to (Qbit \to U) \to U$$

- It is up to the programmer to ensure that the resulting computation is still a valid unitary.
- In Haskell, this side-condition can only be checked at run-time.

QIO Design....

- Using a $ulet$ in the definition of the adder function we now have the type
  $$adder :: [\,Qbit\,] \rightarrow [\,Qbit\,] \rightarrow Qbit \rightarrow U$$

QIO Design....

- Using a $ulet$ in the definition of the adder function we now have the type

$$adder :: [\,Qbit\,] \rightarrow [\,Qbit\,] \rightarrow Qbit \rightarrow U$$

- We can actually define the type

$$\mathbf{newtype}\ QInt = QInt\ [\,Qbit\,]$$

which ensures a fixed size of quantum register.

QIO Design....

- Using a $ulet$ in the definition of the adder function we now have the type
  $$adder :: [\,Qbit\,] \longrightarrow [\,Qbit\,] \longrightarrow Qbit \longrightarrow U$$
- We can actually define the type
  $$\mathbf{newtype}\ QInt = QInt\ [\,Qbit\,]$$
  which ensures a fixed size of quantum register.
- So the adder can have type
  $$adder :: QInt \longrightarrow QInt \longrightarrow Qbit \longrightarrow U$$

Qdata

- The $QInt$ mentioned above is an example of a quantum data type.

Qdata

- The $QInt$ mentioned above is an example of a quantum data type.
- We have implemented a class of quantum data types that formalise the correspondence between classical and quantum data.

Qdata

The University of Nottingham

- The $QInt$ mentioned above is an example of a quantum data type.
- We have implemented a class of quantum data types that formalise the correspondence between classical and quantum data.
- For example, the simplest quantum data type is the Qubit, which corresponds to the classical Boolean data type.

$$\textbf{instance}\ Qdata\ Bool\ Qbit\ \textbf{where}\ \ldots$$

Qdata

The University of Nottingham

- The $QInt$ mentioned above is an example of a quantum data type.
- We have implemented a class of quantum data types that formalise the correspondence between classical and quantum data.
- For example, the simplest quantum data type is the Qubit, which corresponds to the classical Boolean data type.

$$\textbf{instance}\ Qdata\ Bool\ Qbit\ \textbf{where}\ \ldots$$

- This can be extended to lists and pairs...

Qdata

- The $QInt$ mentioned above is an example of a quantum data type.
- We have implemented a class of quantum data types that formalise the correspondence between classical and quantum data.
- For example, the simplest quantum data type is the Qubit, which corresponds to the classical Boolean data type.

$$\textbf{instance}\ Qdata\ Bool\ Qbit\ \textbf{where}\ \ldots$$

- This can be extended to lists and pairs...
- and the QInt used above

$$\textbf{instance}\ Qdata\ Int\ QInt\ \textbf{where}\ \ldots$$

Qdata.

- The $Qdata$ class specifies that any instance of the class provides the necessary functions.

Qdata.

- The *Qdata* class specifies that any instance of the class provides the necessary functions.

$$
\begin{aligned}
&\textbf{class } Qdata\ a\ qa \mid a \rightarrow qa, qa \rightarrow a \textbf{ where}\\
&\quad mkQ :: a \rightarrow QIO\ qa\\
&\quad measQ :: qa \rightarrow QIO\ a\\
&\quad letU :: a \rightarrow (qa \rightarrow U) \rightarrow U\\
&\quad condQ :: qa \rightarrow (a \rightarrow U) \rightarrow U
\end{aligned}
$$

-

Qdata.

- The $Qdata$ class specifies that any instance of the class provides the necessary functions.

$$
\begin{aligned}
&\textbf{class } Qdata\ a\ qa \mid a \to qa, qa \to a \textbf{ where}\\
&\quad mkQ :: a \to QIO\ qa\\
&\quad measQ :: qa \to QIO\ a\\
&\quad letU :: a \to (qa \to U) \to U\\
&\quad condQ :: qa \to (a \to U) \to U
\end{aligned}
$$

- we shall see what $condQ$ is shortly.

Qdata..

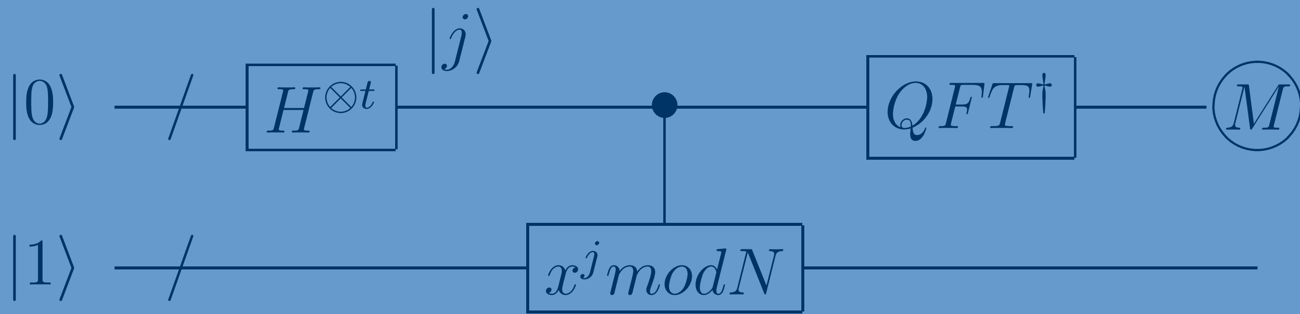
- The full Boolean-Qubit instance

$$\textbf{instance}\ Qdata\ Bool\ Qbit\ \textbf{where}$$
$$mkQ = mkQbit$$
$$measQ = measQbit$$
$$letU\ \ b\ xu = ulet\ \ b\ xu$$
$$condQ\ \ q\ \ br = cond\ \ q\ \ br$$

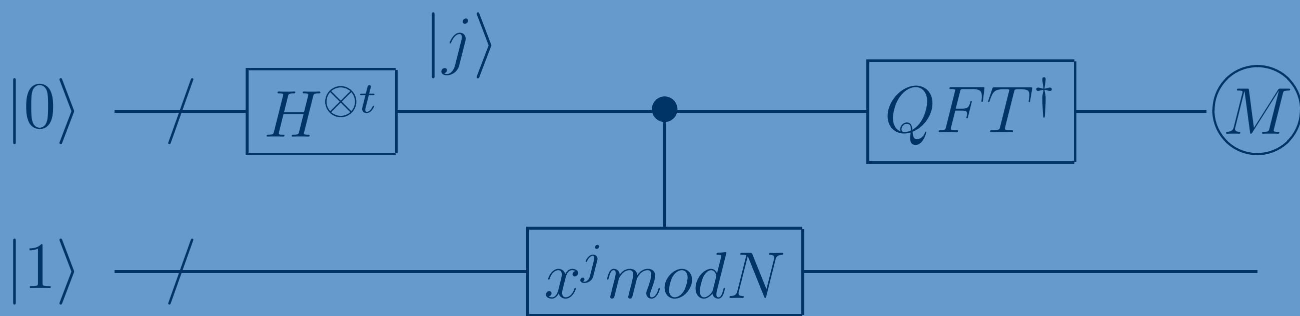
Shor's Algorithm ?

- What do we need to implement Shor's algorithm ?

Shor's Algorithm ?

- What do we need to implement Shor's algorithm ?
- 

Shor's Algorithm ?

- What do we need to implement Shor's algorithm ?
- 

- We can already create the $H^{\otimes t}$ with the available single qubit rotations, but we need to implement the (inverse) Quantum Fourier Transform, and a means of modular exponentiation.

QFT

- QFT can be given as

$$|j_0, \ldots, j_n\rangle$$

$$\downarrow$$

$$\frac{(|0\rangle + e^{2\pi i 0.j_n}\,|1\rangle)(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}\,|1\rangle)\cdots(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n}\,|1\rangle)}{2^{n/2}}$$

which is the (bit-wise) QFT for an n-qubit register.

QFT.

- Looking at the given QFT we can see that the action to be performed on each qubit depends on the value of other qubits in the register.

QFT.

- Looking at the given QFT we can see that the action to be performed on each qubit depends on the value of other qubits in the register.
- This can be done using conditional statements, but this is also where the $condQ$ constructor from the Qdata class is useful.

QFT.

- Looking at the given QFT we can see that the action to be performed on each qubit depends on the value of other qubits in the register.
- This can be done using conditional statements, but this is also where the $condQ$ constructor from the Qdata class is useful.
- The $condQ$ function allows conditional unitaries to be defined by functions on the classical counterpart of a quantum data structure.

$$condQ :: qa \rightarrow (a \rightarrow U) \rightarrow U$$

QFT..

- $$\begin{aligned}
&qft :: [\,Qbit\,] \rightarrow U \\
&qft\ \ qs = condQ\ \ qs\ (\lambda bs \rightarrow qftAcu\ \ qs\ \ bs\ [\,])\\
&qftAcu :: [\,Qbit\,] \rightarrow [\,Bool\,] \rightarrow [\,Bool\,] \rightarrow U\\
&qftAcu\ [\,]\ [\,]\ _ = mempty\\
&qftAcu\ (q : qs)\ (b : bs)\ cs = qftBase\ \ cs\ \ q \mathbin{+\!\!+} qftAcu\ \ qs\ \ bs\ (b : cs)\\
&qftBase :: [\,Bool\,] \rightarrow Qbit \rightarrow U\\
&qftBase\ \ bs\ \ q = f'\ \ bs\ \ q\ \ 2\\
&\quad \mathbf{where}\ f'\ [\,]\ q\ _ = uhad\ \ q\\
&\qquad f'\ (b : bs)\ q\ x = \mathbf{if}\ b\ \mathbf{then}\ (rotK\ \ x\ \ q) \mathbin{+\!\!+} f'\ \ bs\ \ q\ (x+1)\\
&\qquad\quad \mathbf{else}\ f'\ \ bs\ \ q\ (x+1)
\end{aligned}$$

Modular Exponentiation

- To define modular exponentiation in QIO, it is necessary to build up a set of reversible arithmetic functions.

Modular Exponentiation

- To define modular exponentiation in QIO, it is necessary to build up a set of reversible arithmetic functions.
- We have already seen that we can create an adder function

Modular Exponentiation

- To define modular exponentiation in QIO, it is necessary to build up a set of reversible arithmetic functions.
- We have already seen that we can create an adder function
- We have created a set of quantum arithmetic functions following the design of the circuits in [Vedral, Barenco, Ekert. 1996].

Modular Exponentiation

- To define modular exponentiation in QIO, it is necessary to build up a set of reversible arithmetic functions.
- We have already seen that we can create an adder function
- We have created a set of quantum arithmetic functions following the design of the circuits in [Vedral, Barenco, Ekert. 1996].
- So we have the modular exponentiation function of type

$$modExp :: Int \rightarrow Int \rightarrow QInt \rightarrow QInt \rightarrow U$$
$$modExp\ n\ a\ x\ o = \ldots$$

giving the function $x^a modn$.

Conclusion

- We have shown that we have all the necessary functions to put together Shor's algorithm in the QIO Monad.

Conclusion

- We have shown that we have all the necessary functions to put together Shor's algorithm in the QIO Monad.
- We have shown how implementing Shor's algorithm has lead to design changes to the QIO Monad.

Conclusion

- We have shown that we have all the necessary functions to put together Shor's algorithm in the QIO Monad.
- We have shown how implementing Shor's algorithm has lead to design changes to the QIO Monad.
- We have started using the features of functional programming to enrich what we can do with the QIO Monad (ie. the Qdata class).

Further Work

- We would like to use the QIO Monad to start reasoning about quantum computation in general.

Further Work

- We would like to use the QIO Monad to start reasoning about quantum computation in general.
- We would like to formalise QIO within the Coq proof assistant program.

Further Work

- We would like to use the QIO Monad to start reasoning about quantum computation in general.
- We would like to formalise QIO within the Coq proof assistant program.
- The type system in Coq will allow the formalisation of the side conditions imposed on the $ulet$ constructor.