Quantum Programming in Haskell

with 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

The University of Nottingham

- We would like to model Quantum Computations...

Introduction

- We would like to model Quantum Computations...
- ... in a functional setting.

Introduction

- We would like to model Quantum Computations…
- … in a functional setting.
- The QIO Monad can be thought of as a register of Qubits that's controlled by a classical computer.

Introduction

- We would like to model Quantum Computations...
- ... in a functional setting.
- The QIO Monad can be thought of as a register of Qubits that's controlled by a classical computer.
- It provides a framework for constructing quantum computations...

Introduction

- We would like to model Quantum Computations...
- ... in a functional setting.
- The QIO Monad can be thought of as a register of Qubits that's controlled by a 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

The University of Nottingham

- 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

The University of Nottingham

- 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 $(>>=)$

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

- $$(\gg\!=) :: m\ \ a \longrightarrow (a \longrightarrow m\ \ b) \longrightarrow m\ \ b$$

$$return :: a \longrightarrow m\ \ a$$

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 ($>>=$)
- $$\begin{aligned}&\textbf{class}\ Monad\ m\ \textbf{where}\\ &\quad (\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\\ &\quad return :: a \rightarrow m\ a\end{aligned}$$
- The *return* function lifts values of an underlying type into the Monad.

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 ($>>=$)
- $$\textbf{class}\ Monad\ m\ \textbf{where}$$
  $$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$
  $$return :: a \rightarrow m\ a$$
- The *return* function lifts values of an underlying type into the Monad.
- The $>>=$ function lifts the application of the given function to a result already in the Monad.

The Maybe Monad

- The Maybe Monad can be used for functions that are undefined on some inputs. (E.g. division by zero)

The Maybe Monad

- The Maybe Monad can be used for functions that are undefined on some inputs. (E.g. division by zero)
- $\textbf{data}\ Maybe\ a = Just\ a \mid Nothing$

The Maybe Monad

- The Maybe Monad can be used for functions that are undefined on some inputs. (E.g. division by zero)
- $\textbf{data}\ Maybe\ a = Just\ a \mid Nothing$
- $return\ x = Just\ x$

The Maybe Monad

The University of Nottingham

- The Maybe Monad can be used for functions that are undefined on some inputs. (E.g. division by zero)
- $\textbf{data}\ Maybe\ a = Just\ a \mid Nothing$
- $return\ x = Just\ x$
- $$(Just\ x) \gg\!= f = f\ x$$
  $$Nothing \gg\!= f = Nothing$$

The Maybe Monad

- The Maybe Monad can be used for functions that are undefined on some inputs. (E.g. division by zero)
- $$\textbf{data}\ Maybe\ a = Just\ a \mid Nothing$$
- $$return\ x = Just\ x$$
- $$(Just\ x) \gg\!= f = f\ x$$
  $$Nothing \gg\!= f = Nothing$$
- The bind function allows for an undefined result to propagate through the rest of the computation.

‘do’ notation

- Haskell provides the do notation to make monadic programming easier.

‘do’ notation

- Haskell provides the do notation to make monadic programming easier.
- IO in Haskell takes place in the IO Monad.

‘do’ notation

- Haskell provides the do notation to make monadic programming easier.
- 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

- Haskell provides the do notation to make monadic programming easier.
- 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

- Haskell provides the do notation to make monadic programming easier.
- 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

‘do’ notation

- Haskell provides the do notation to make monadic programming easier.
- IO in Haskell takes place in the IO Monad.
- For example, echoing a character to the screen

$$getChar :: IO\ \ Char$$

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

- $$echo :: IO\ ()$$

$$echo = getChar \ggg= (\lambda c \rightarrow putChar\ \ c) > > echo$$

- or in do notation

$$echo = \textbf{do}\ c \leftarrow getChar$$

- $$putChar\ \ c$$

$$echo$$

The QIO Monad

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

The QIO Monad

The University of Nottingham

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The do notation provided by Haskell can be used to help this.

The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The do notation provided by Haskell can be used to help this.

$$|0\rangle :: QIO\ Qbit$$

- $$|0\rangle = \mathbf{do}\ qb \leftarrow mkQbit\ False$$
  $$return\ x$$

The QIO Monad

- The QIO Monad has been designed so that Quantum computations can be defined within Haskell.
- The do notation provided by Haskell can be used to help this.
- $$\begin{array}{l} |0\rangle :: QIO\ Qbit \\ |0\rangle = \textbf{do}\ qb \leftarrow mkQbit\ False \\ \qquad\qquad return\ x \end{array}$$
- $$\begin{array}{l} |1\rangle :: QIO\ Qbit \\ |1\rangle = \textbf{do}\ qb \leftarrow mkQbit\ True \\ \qquad\qquad return\ x \end{array}$$

QIO Examples

- Creating the state $|+\rangle$

$$
\begin{aligned}
&|+\rangle :: QIO\ Qbit \\
&|+\rangle = \mathbf{do}\ qb \leftarrow |0\rangle \\
&\qquad\qquad applyU\ (uhad\ qb) \\
&\qquad\qquad return\ qb
\end{aligned}
$$

QIO Examples

- Creating the state $|+\rangle$

$$|+\rangle :: QIO\ Qbit$$
$$|+\rangle = \mathbf{do}\ qb \leftarrow |0\rangle$$
$$\quad applyU\ (uhad\ qb)$$
$$\quad return\ qb$$

- Creating a bell state

$$share :: Qbit \rightarrow QIO\ Qbit$$
$$share\ qa = \mathbf{do}\ qb \leftarrow |0\rangle$$
$$\quad applyU\ (cond\ qa\ (\lambda a \rightarrow \mathbf{if}\ a\ \mathbf{then}\ (unot\ qb)$$
$$\quad \mathbf{else}\ (\bullet)))$$
$$\quad return\ qb$$

$$bell :: QIO\ (Qbit, Qbit)$$
$$bell = \mathbf{do}\ qa \leftarrow |+\rangle$$
$$\quad qb \leftarrow share\ qa$$
$$\quad return\ (qa, qb)$$

Deutsch's Algorithm

- $$u :: (Bool \rightarrow Bool) \rightarrow Qbit \rightarrow Qbit \rightarrow U$$
  $$u\ f\ x\ y = cond\ x\ (\lambda b \rightarrow \mathbf{if}\ f\ b\ \mathbf{then}\ unot\ y\ \mathbf{else}\ \bullet)$$

Deutsch's Algorithm

- $u :: (Bool \rightarrow Bool) \rightarrow Qbit \rightarrow Qbit \rightarrow U$
  $u\ f\ x\ y = cond\ x\ (\lambda b \rightarrow \textbf{if}\ f\ b\ \textbf{then}\ unot\ y\ \textbf{else}\ \bullet)$

  $deutsch :: (Bool \rightarrow Bool) \rightarrow QIO\ Bool$
  $deutsch\ f = \textbf{do}\ x \leftarrow |+\rangle$
  $y \leftarrow |-\rangle$
- $applyU\ (u\ f\ x\ y)$
  $applyU\ (uhad\ x)$
  $b \leftarrow measQ\ x$
  $return\ b$

QIO Design

The University of Nottingham

- The design allows unitaries to be defined outside of the monadic structure...

QIO Design

- The design allows unitaries to be defined outside of the monadic structure...
- ... the U data-type defines the available unitaries.

QIO Design

- The design allows unitaries to be defined outside of the monadic structure...
- ... the U data-type defines the available unitaries.
- The position of two qubits can be swapped.

$$swap :: Qbit \to Qbit \to U$$

QIO Design

- The design allows unitaries to be defined outside of the monadic structure...
- ... the U data-type defines the available unitaries.
- The position of two qubits can be swapped.

  $$swap :: Qbit \rightarrow Qbit \rightarrow U$$

- A conditional unitary, depending on the value of the given qubit, can be constructed.

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

QIO Design

- The design allows unitaries to be defined outside of the monadic structure...
- ... the U data-type defines the available unitaries.
- The position of two qubits can be swapped.

  $$swap :: Qbit \rightarrow Qbit \rightarrow U$$

- A conditional unitary, depending on the value of the given qubit, can be constructed.

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

- Qubits can be temporarily introduced into a unitary.

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

QIO Design.

- Single qubit rotations can be applied...

$$rot :: Qbit \rightarrow Rotation \rightarrow U$$

QIO Design.

- Single qubit rotations can be applied...

  $$rot :: Qbit \rightarrow Rotation \rightarrow U$$

- $$\textbf{type}\ Rotation = ((Bool, Bool) \rightarrow \mathbb{C})$$

QIO Design.

- Single qubit rotations can be applied...

$$rot :: Qbit \rightarrow Rotation \rightarrow U$$

- $$\textbf{type}\ Rotation = ((Bool, Bool) \rightarrow \mathbb{C})$$

- Some common rotations are defined...

$$rnot :: Rotation$$
$$rnot\ (x, y) = \textbf{if}\ x \equiv y\ \textbf{then}\ 0\ \textbf{else}\ 1$$

$$rhad :: Rotation$$
$$rhad\ (x, y) = \textbf{if}\ x \wedge y\ \textbf{then}\ -h\ \textbf{else}\ h\ \textbf{where}\ h = (1\ /\ sqrt\ 2)$$

$$rphase :: \mathbb{R} \rightarrow Rotation$$
$$rphase\ \_\ (False, False) = 1$$
$$rphase\ r\ (True, True) = exp\ (0 :+ r)$$
$$rphase\ \_\ (\_, \_) = 0$$

QIO Design...

- The U data-type also forms a Monoid

QIO Design...

- The U data-type also forms a Monoid
- There's an identity element denoted $\bullet$

QIO Design...

- The U data-type also forms a Monoid
- There's an identity element denoted $\bullet$
- ... and an append operation denoted $\rhd$.

QIO Design...

- The U data-type also forms a Monoid
- There's an identity element denoted $\bullet$
- ... and an append operation denoted $\rhd$.
- We can also define a reverse function $urev :: U \rightarrow U$ that returns the inverse of the given unitary.

QIO Design...

- The U data-type also forms a Monoid
- There's an identity element denoted $\bullet$
- ... and an append operation denoted $\rhd$.
- We can also define a reverse function $urev :: U \rightarrow U$ that returns the inverse of the given unitary.
- The choice of available unitaries has been adapted as we have implemented more quantum algorithms.

QIO Design...

- The U data-type also forms a Monoid
- There's an identity element denoted $\bullet$
- ... and an append operation denoted $\rhd$.
- We can also define a reverse function $urev :: U \rightarrow U$ that returns the inverse of the given unitary.
- The choice of available unitaries has been adapted as we have implemented more quantum algorithms.
- However, there are side-conditions that need to be imposed to ensure that all the members of U are actually unitary.

Side Conditions

The University of Nottingham

- As it stands, conditionals can be created that aren't unitary.

Side Conditions

- As it stands, conditionals can be created that aren't unitary.
- $$notUnitary :: U$$ $$notUnitary = cond\ x\ (\lambda x \rightarrow \textbf{if}\ x\ \textbf{then}\ unot\ x\ \textbf{else}\ \bullet)$$

Side Conditions

- As it stands, conditionals can be created that aren't unitary.
- $$\begin{aligned} &notUnitary :: U \\ &notUnitary = cond\ x\ (\lambda x \rightarrow \mathbf{if}\ x\ \mathbf{then}\ unot\ x\ \mathbf{else}\ \bullet) \end{aligned}$$
- The given function always leaves the qubit x in the state $|0\rangle$.

Side Conditions

- As it stands, conditionals can be created that aren't unitary.
- $$\begin{aligned} &notUnitary :: U \\ &notUnitary = cond\ x\ (\lambda x \rightarrow \textbf{if}\ x\ \textbf{then}\ unot\ x\ \textbf{else}\ \bullet) \end{aligned}$$
- The given function always leaves the qubit x in the state $|0\rangle$.
- A side condition for conditionals must be introduced, that the branches of the conditional must not reference the control qubit.

Side Conditions

- As it stands, conditionals can be created that aren't unitary.
- $$notUnitary :: U$$
  $$notUnitary = cond\ x\ (\lambda x \rightarrow \textbf{if}\ x\ \textbf{then}\ unot\ x\ \textbf{else}\ \bullet)$$
- The given function always leaves the qubit x in the state $|0\rangle$.
- A side condition for conditionals must be introduced, that the branches of the conditional must not reference the control qubit.
- Trying to run the $notUnitary$ function will result in a run-time error.

Side Conditions.

- The *ulet* constructor could easily give rise to non-unitary behaviour...

Side Conditions.

- The *ulet* constructor could easily give rise to non-unitary behaviour...
- e.g. the temporary qubit could be left entangled with the rest of the state.

Side Conditions.

- The $ulet$ constructor could easily give rise to non-unitary behaviour...
- e.g. the temporary qubit could be left entangled with the rest of the state.
- The side-condition imposed for $ulet$ is that the temporary qubit must be returned to its original state.

Side Conditions.

- The $ulet$ constructor could easily give rise to non-unitary behaviour...
- e.g. the temporary qubit could be left entangled with the rest of the state.
- The side-condition imposed for $ulet$ is that the temporary qubit must be returned to its original state.
- It would also be possible to create a non-unitary single qubit rotation.

Side Conditions.

- The *ulet* constructor could easily give rise to non-unitary behaviour...
- e.g. the temporary qubit could be left entangled with the rest of the state.
- The side-condition imposed for *ulet* is that the temporary qubit must be returned to its original state.
- It would also be possible to create a non-unitary single qubit rotation.
- The side-condition for rotations is that they must be unitary!

Side Conditions.

- The *ulet* constructor could easily give rise to non-unitary behaviour...
- e.g. the temporary qubit could be left entangled with the rest of the state.
- The side-condition imposed for *ulet* is that the temporary qubit must be returned to its original state.
- It would also be possible to create a non-unitary single qubit rotation.
- The side-condition for rotations is that they must be unitary!
- Again, in both cases, failure to comply will result in a run-time error.

The Monadic constructors

- The Monadic constructors allow the system to deal with the side-effects to the state arising from measurements

The Monadic constructors

- The Monadic constructors allow the system to deal with the side-effects to the state arising from measurements
- Qubits can be initialised, from a Boolean value.

$$mkQbit :: Bool \rightarrow QIO\ Qbit$$

The Monadic constructors

- The Monadic constructors allow the system to deal with the side-effects to the state arising from measurements
- Qubits can be initialised, from a Boolean value.

  $$mkQbit :: Bool \longrightarrow QIO\ Qbit$$

- Unitaries can be applied to the current state.

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

The Monadic constructors

- The Monadic constructors allow the system to deal with the side-effects to the state arising from measurements
- Qubits can be initialised, from a Boolean value.

  $$mkQbit :: Bool \rightarrow QIO\ Qbit$$

- Unitaries can be applied to the current state.

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

- Qubits can be measured, returning a Boolean value.

  $$measQbit :: Qbit \rightarrow QIO\ Bool$$

Teleportation

$$
\begin{aligned}
&alice :: Qbit \longrightarrow Qbit \longrightarrow QIO\ (Bool, Bool)\\
&alice\ aq\ eq = \mathbf{do}\ applyU\ (cond\ aq\ (\lambda a \longrightarrow\\
&\qquad\qquad\qquad\qquad \mathbf{if}\ a\ \mathbf{then}\ (unot\ eq)\\
&\qquad\qquad\qquad\qquad\quad \mathbf{else}\ (\bullet)))\\
&\qquad applyU\ (uhad\ aq)\\
&\qquad cd \leftarrow measQ\ (aq, eq)\\
&\qquad return\ cd
\end{aligned}
$$

Teleportation.

$$bobsU :: (Bool, Bool) \to Qbit \to U$$
$$bobsU\ (False, False)\ eq = \bullet$$
$$bobsU\ (False, True)\ eq = (unot\ eq)$$
$$bobsU\ (True, False)\ eq = (uZZ\ eq)$$
$$bobsU\ (True, True)\ eq = ((unot\ eq) \rhd (uZZ\ eq))$$

$$bob :: Qbit \to (Bool, Bool) \to QIO\ Qbit$$
$$bob\ eq\ cd = \mathbf{do}\ applyU\ (bobsU\ cd\ eq)$$
$$return\ eq$$

Teleportation..

$$
\begin{aligned}
&teleportation :: Qbit \rightarrow QIO\ Qbit\\
&teleportation\ iq = \mathbf{do}\ (eq1, eq2) \leftarrow bell\\
&\qquad cd \leftarrow alice\ iq\ eq1\\
&\qquad tq \leftarrow bob\ eq2\ cd\\
&\qquad return\ tq
\end{aligned}
$$

Running QIO Computations

The University of Nottingham

- We provide three evaluation functions for (classically) simulating the running of our QIO computations.

Running QIO Computations

The University of Nottingham

- We provide three evaluation functions for (classically) simulating the running of our QIO computations.
- $runQ$ returns a single probabilistic result.
  $>runQ\ (deutsch\ \neg)$
  $True$
  $>runQ\ (deutsch\ (\lambda x \rightarrow True))$
  $False$

Running QIO Computations

- We provide three evaluation functions for (classically) simulating the running of our QIO computations.
- $runQ$ returns a single probabilistic result.
  $> runQ\ (deutsch\ \neg)$
  $True$
  $> runQ\ (deutsch\ (\lambda x \rightarrow True))$
  $False$
- $simQ$ returns a probability distribution of the possible results.
  $> simQ\ (deutsch\ \neg)$
  $[(True, 1.0)]$
  $> simQ\ (meas\_bell)$
  $[((True, True), 0.5), ((False, False), 0.5)]$

Running QIO Computations.

- There is also the $runC$ function which efficiently simulates computations that only use the classical subset of U.

  $>runC\ (deutsch\ \neg)$

  *** Exception: not classical

Running QIO Computations.

The University of Nottingham

- There is also the $runC$ function which efficiently simulates computations that only use the classical subset of U.
  $> runC\ (deutsch\ \neg)$
  *** Exception: not classical
- The $runC$ function is useful for testing our reversible arithmetic functions

Reversible Arithmetic

- One of our goals was to implement Shor's algorithm.

Reversible Arithmetic

- One of our goals was to implement Shor's algorithm.
- The period finding sub-routine requires a function that computes modular exponentiation.

Reversible Arithmetic

- One of our goals was to implement Shor's algorithm.
- The period finding sub-routine requires a function that computes modular exponentiation.
- We have created a set of quantum arithmetic functions following the design of the circuits in [Vedral, Barenco, Ekert. 1996].

Reversible Arithmetic

- One of our goals was to implement Shor's algorithm.
- The period finding sub-routine requires a function that computes modular exponentiation.
- We have created a set of quantum arithmetic functions following the design of the circuits in [Vedral, Barenco, Ekert. 1996].
- To implement these functions we decided that it would be useful to be able to define quantum data-types, built up from qubits, and related with a classical counter-part

Reversible Arithmetic

- One of our goals was to implement Shor's algorithm.
- The period finding sub-routine requires a function that computes modular exponentiation.
- We have created a set of quantum arithmetic functions following the design of the circuits in [Vedral, Barenco, Ekert. 1996].
- To implement these functions we decided that it would be useful to be able to define quantum data-types, built up from qubits, and related with a classical counter-part
- This lead to the definition of a class of quantum data-types.

Qdata

- The *Qdata* class defines functions that a pair of corresponding classical and quantum data-types must fulfill, within the QIO setting.

Qdata

- The *Qdata* class defines functions that a pair of corresponding classical and quantum data-types must fulfill, within the QIO setting.

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

-

Qdata

- Booleans and Qubits form the simplest instance of the *Qdata* class.

Qdata

- Booleans and Qubits form the simplest instance of the *Qdata* class.

$$\textbf{instance } Qdata\ Bool\ Qbit \textbf{ where}$$
$$mkQ = mkQbit$$

- $$measQ = measQbit$$

$$letU\ b\ xu = ulet\ b\ xu$$
$$condQ\ q\ br = cond\ q\ br$$

Qdata

- Booleans and Qubits form the simplest instance of the *Qdata* class.

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

- We have also implemented a quantum data-type *QInt* related to the (positive instances of) the Haskell *Int* type.

Reversible Arithmetic ...

- The circuits in [Vedral, Barenco, Ekert. 1996] make extensive use of auxilliary qubits...

Reversible Arithmetic ...

- The circuits in [Vedral, Barenco, Ekert. 1996] make extensive use of auxilliary qubits...
- ... which we can handle nicely using the *ulet* constructor.

Reversible Arithmetic ...

- The circuits in [Vedral, Barenco, Ekert. 1996] make extensive use of auxilliary qubits...
- ... which we can handle nicely using the *ulet* constructor.

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

-

Reversible Arithmetic ...

- The circuits in [Vedral, Barenco, Ekert. 1996] make extensive use of auxilliary qubits...
- ... which we can handle nicely using the *ulet* constructor.

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

- The required modular exponentiation function ($modExp$) follows nicely.

Quantum Fourier transform

The University of Nottingham

- Shor's algorithm also required the inverse QFT.

Quantum Fourier transform

- Shor's algorithm also required the inverse QFT.
- The structure of the QFT leads to a nice functional representation using an accumulator function, recursively defined over the input register.

Quantum Fourier transform

The University of Nottingham

- Shor's algorithm also required the inverse QFT.
- The structure of the QFT leads to a nice functional representation using an accumulator function, recursively defined over the input register.

$$qft :: [\,Qbit\,] \rightarrow U$$
$$qft\ qs = condQ\ qs\ (\lambda bs \rightarrow qftAcu\ qs\ bs\ [\,])$$

$$qftAcu :: [\,Qbit\,] \rightarrow [\,Bool\,] \rightarrow [\,Bool\,] \rightarrow U$$
$$qftAcu\ [\,]\ [\,]\ \_ \qquad\qquad = \bullet$$
$$qftAcu\ (q : qs)\ (b : bs)\ cs = qftBase\ cs\ q \rhd qftAcu\ qs\ bs\ (b : cs)$$

-

$$qftBase :: [\,Bool\,] \rightarrow Qbit \rightarrow U$$
$$qftBase\ bs\ q = f'\ bs\ q\ 2$$
$$\textbf{where}\ f'\ [\,] \qquad q\ \_ = uhad\ q$$
$$f'\ (b : bs)\ q\ x = \textbf{if}\ b\ \textbf{then}\ (rotK\ x\ q) \rhd f'\ bs\ q\ (x+1)$$
$$\textbf{else}\ f'\ bs\ q\ (x+1)$$

Shor’s Algorithm

- The period finding sub-routine of Shor’s algorithm can now be given.

Shor's Algorithm

- The period finding sub-routine of Shor's algorithm can now be given.

$$hadamards :: QInt \rightarrow U$$
$$hadamards\ (QInt\ [\,]) \qquad = \bullet$$
$$hadamards\ (QInt\ (x : xs)) = uhad\ x \rhd hadamards\ (QInt\ xs)$$

$$shorU :: QInt \rightarrow QInt \rightarrow QInt \rightarrow Int \rightarrow U$$
$$shorU\ i0\ i1\ x\ n = hadamards\ i0 \rhd$$
$$condQ\ i0\ (\lambda a \rightarrow modExp\ n\ a\ x\ i1) \rhd$$
$$urev\ (qft\ i0)$$

-

$$shor :: Int \rightarrow Int \rightarrow QIO\ Int$$
$$shor\ x\ n = \mathbf{do}\ ((i0, i1), qx) \leftarrow mkQ\ ((0, 1), x)$$
$$applyU\ (shorU\ i0\ i1\ qx\ n)$$
$$p \leftarrow measQ\ i0$$
$$return\ p$$

Dependent Types

- The fact that our side-conditions can be checked at run-time follows from the fact that we're classically simulating quantum computations.

Dependent Types

- The fact that our side-conditions can be checked at run-time follows from the fact that we're classically simulating quantum computations.
- Dependent Types give us types that can depend on data...

Dependent Types

- The fact that our side-conditions can be checked at run-time follows from the fact that we're classically simulating quantum computations.
- Dependent Types give us types that can depend on data...
- ... the data that they depend on could be a proof of some property about the type.

Dependent Types

- The fact that our side-conditions can be checked at run-time follows from the fact that we're classically simulating quantum computations.
- Dependent Types give us types that can depend on data...
- ... the data that they depend on could be a proof of some property about the type.
- With dependent types, we could embed proofs that the unitaries satisfy the imposed side-conditions.

Dependent Types

- The fact that our side-conditions can be checked at run-time follows from the fact that we're classically simulating quantum computations.
- Dependent Types give us types that can depend on data...
- ... the data that they depend on could be a proof of some property about the type.
- With dependent types, we could embed proofs that the unitaries satisfy the imposed side-conditions.
- These proofs are checked at compile time by the type checker...

Dependent Types

The University of Nottingham

- The fact that our side-conditions can be checked at run-time follows from the fact that we're classically simulating quantum computations.
- Dependent Types give us types that can depend on data...
- ... the data that they depend on could be a proof of some property about the type.
- With dependent types, we could embed proofs that the unitaries satisfy the imposed side-conditions.
- These proofs are checked at compile time by the type checker...
- leading to a more "sound" implementation.

Conclusions

- A Dependently typed version of QIO could give a sound basis for reasoning about quantum computations.

Conclusions

The University of Nottingham

- A Dependently typed version of QIO could give a sound basis for reasoning about quantum computations.
- ... so we would like to implement this.

Conclusions

- A Dependently typed version of QIO could give a sound basis for reasoning about quantum computations.
- ... so we would like to implement this.
- We are also planning at looking to extend QIO as a full language.

Conclusions

The University of Nottingham

- A Dependently typed version of QIO could give a sound basis for reasoning about quantum computations.
- ... so we would like to implement this.
- We are also planning at looking to extend QIO as a full language.
- We are also looking for more examples like the Qdata class, where ideas in functional program can be used nicely in the quantum setting.

Finally...

- We are presenting a paper on the Quantum IO Monad at TFP 2008 (Trends in Functional Programming). Soon to be available on-line: http://www.cs.nott.ac.uk/~asg/research.html

Finally...

The University of Nottingham

- We are presenting a paper on the Quantum IO Monad at TFP 2008 (Trends in Functional Programming). Soon to be available on-line: http://www.cs.nott.ac.uk/˜asg/research.html
- The code from the implementation is also available on-line: http://www.cs.nott.ac.uk/˜asg/QIO/