The University of Nottingham

The Quantum IO Monad

QIO

Alexander S. Green
asg@cs.nott.ac.uk

Foundations of Programming Group,
School of Computer Science & IT,
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

The University of Nottingham

- 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

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

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

'do' notation

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

$$getChar :: IO\ Char$$

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

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

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

- or in do notation

'do' notation

The University of Nottingham

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

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.
-

$$\begin{aligned} &trueBit :: QIO\ Bool \\ &trueBit = \mathbf{do}\ qb \leftarrow mkQbit\ True \\ &\qquad\qquad\qquad x \leftarrow measQbit\ qb \\ &\qquad\qquad\qquad return\ x \end{aligned}$$

API

- What can we do in the QIO Monad?

API

- What can we do in the QIO Monad?
- $$mkQbit :: Bool \longrightarrow QIO\ Qbit$$

API

- What can we do in the QIO Monad?
- $$mkQbit :: Bool \longrightarrow QIO\ Qbit$$
- When initialising a qubit the user must define which of the base states, True or False ($|0\rangle$ or $|1\rangle$), to initialise it into.

API

- What can we do in the QIO Monad?
- $$mkQbit :: Bool \longrightarrow QIO\ Qbit$$
- When initialising a qubit the user must define which of the base states, True or False ($|0\rangle$ or $|1\rangle$), to initialise it into.
- $$measQbit :: Qbit \longrightarrow QIO\ Bool$$

API

- What can we do in the QIO Monad?
- $mkQbit :: Bool \longrightarrow QIO\ Qbit$
- When initialising a qubit the user must define which of the base states, True or False ($|0\rangle$ or $|1\rangle$), to initialise it into.
- $measQbit :: Qbit \longrightarrow QIO\ Bool$
- The measurement of a qubit always results in a boolean value.

API

- What can we do in the QIO Monad?
- $mkQbit :: Bool \longrightarrow QIO\ Qbit$
- When initialising a qubit the user must define which of the base states, True or False ($|0\rangle$ or $|1\rangle$), to initialise it into.
- $measQbit :: Qbit \longrightarrow QIO\ Bool$
- The measurement of a qubit always results in a boolean value.
- Wht else can be done with these qubits?

Unitaries.

- It is possible to construct unitary operators, and apply them to the relevent qubits.

Unitaries.

- It is possible to construct unitary operators, and apply them to the relevent qubits.
- $applyU :: U \rightarrow QIO\ ()$

Unitaries.

- It is possible to construct unitary operators, and apply them to the relevent qubits.
- $$applyU :: U \rightarrow QIO\ ()$$
- There are 5 unitary constructors that are available:

Unitaries.

- It is possible to construct unitary operators, and apply them to the relevent qubits.
- $$applyU :: U \rightarrow QIO\ ()$$
- There are 5 unitary constructors that are available:
- $$unot :: Qbit \rightarrow U$$
  which will rotate the given qubit by 180^o as in the Not rotation.

Unitaries.

- It is possible to construct unitary operators, and apply them to the relevent qubits.
- $$applyU :: U \to QIO\ ()$$
- There are 5 unitary constructors that are available:
- $$unot :: Qbit \to U$$
  which will rotate the given qubit by 180^o as in the Not rotation.
- $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Unitaries..

- $$uhad :: Qbit \to U$$
  which will rotate the given qubit by 90^o as in the Hadamard rotation.

Unitaries..

- $$uhad :: Qbit \rightarrow U$$
  which will rotate the given qubit by 90^o as in the Hadamard rotation.
- $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Unitaries..

- $$uhad :: Qbit \rightarrow U$$
  which will rotate the given qubit by 90^o as in the Hadamard rotation.
- $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
- $$uphase :: Qbit \rightarrow Float \rightarrow U$$
  which will rotate the given qubit by the given phase change (ϕ).

Unitaries..

- $$uhad :: Qbit \rightarrow U$$
  which will rotate the given qubit by 90^o as in the Hadamard rotation.
- $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
- $$uphase :: Qbit \rightarrow Float \rightarrow U$$
  which will rotate the given qubit by the given phase change (ϕ).
- $\begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i\phi} \end{pmatrix}$

Unitaries...

- $$swap :: Qbit \rightarrow Qbit \rightarrow U$$
  which simply swaps the two given qubits.

Unitaries...

- $$swap :: Qbit \rightarrow Qbit \rightarrow U$$
  which simply swaps the two given qubits.
- $$cond :: Qbit \rightarrow (Bool \rightarrow U) \rightarrow U$$
  which given a control qubit, will conditionally do the corresponding unitary given by the function.
  (The control qubit must not be effected by the unitaries)

Unitaries...

- $$swap :: Qbit \rightarrow Qbit \rightarrow U$$
  which simply swaps the two given qubits.
- $$cond :: Qbit \rightarrow (Bool \rightarrow U) \rightarrow U$$
  which given a control qubit, will conditionally do the corresponding unitary given by the function.
  (The control qubit must not be effected by the unitaries)
- It is this conditional operation that can be used to entangle qubits.

Unitaries...

- $$swap :: Qbit \rightarrow Qbit \rightarrow U$$
  which simply swaps the two given qubits.
- $$cond :: Qbit \rightarrow (Bool \rightarrow U) \rightarrow U$$
  which given a control qubit, will conditionally do the corresponding unitary given by the function.
  (The control qubit must not be effected by the unitaries)
- It is this conditional operation that can be used to entangle qubits.
- The U datatype of unitaries, also forms a Monoid meaning there is an append operation for combining uniatries sequentially.

Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.

Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.
- $run :: QIO\ a \rightarrow IO\ a$

Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.
- $run :: QIO\ a \rightarrow IO\ a$
- Running a quantum computation returns a probabilistic result for each measurement.

Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.
- $run :: QIO\ a \rightarrow IO\ a$
- Running a quantum computation returns a probabilistic result for each measurement.
- $sim :: QIO\ a \rightarrow Prob\ a$

Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.
- $run :: QIO\ a \rightarrow IO\ a$
- Running a quantum computation returns a probabilistic result for each measurement.
- $sim :: QIO\ a \rightarrow Prob\ a$
- Simulating a quantum computation returns a probability distribution of all the possible measurement outcomes.

Running Quantum Computations?

- Along with creating quantum computations, the QIO Monad also provides two ways of evaluating them.
- $$run :: QIO\ a \rightarrow IO\ a$$
- Running a quantum computation returns a probabilistic result for each measurement.
- $$sim :: QIO\ a \rightarrow Prob\ a$$
- Simulating a quantum computation returns a probability distribution of all the possible measurement outcomes.
- We would also like to be able to display the internal state of the system at any time, possibly by showing the complex amplitudes for each base state.

Computations.

$$qPlus :: QIO\ Qbit$$
$$\begin{aligned} qPlus = \mathbf{do}\ & qa \leftarrow mkQbit\ False \\ & applyU\ (uhad\ qa) \\ & return\ qa \end{aligned}$$

$$randBit :: QIO\ Bool$$
$$\begin{aligned} randBit = \mathbf{do}\ & qa \leftarrow qPlus \\ & x \leftarrow measQbit\ qa \\ & return\ x \end{aligned}$$

Computations..

$$share :: Qbit \rightarrow QIO\ Qbit$$

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

$$bell :: QIO\ (Qbit, Qbit)$$

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

Computations..

$$test\_bell :: QIO\ (Bool, Bool)$$
$$test\_bell = \mathbf{do}\ qb \leftarrow bell$$
$$b \leftarrow measQ\ qb$$
$$return\ b$$

Teleportation.

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

Teleportation..

$$uZ :: Qbit \rightarrow U$$
$$uZ\ qb = (uphase\ qb\ 0.5)$$

$$bobsU :: (Bool, Bool) \rightarrow Qbit \rightarrow U$$
$$bobsU\ (False, False)\ qb = mempty$$
$$bobsU\ (False, True)\ qb = (unot\ qb)$$
$$bobsU\ (True, False)\ qb = (uZ\ qb)$$
$$bobsU\ (True, True)\ qb = ((unot\ qb)\ \text{`}mappend\text{`}\ (uZ\ qb))$$

$$bob :: Qbit \rightarrow (Bool, Bool) \rightarrow QIO\ Qbit$$
$$bob\ bsq\ cd = \textbf{do}\ applyU\ (bobsU\ cd\ bsq)$$
$$return\ bsq$$

Teleportation...

$$
\begin{aligned}
&\mathit{teleportation} :: \mathit{Qbit} \rightarrow \mathit{QIO}\ \mathit{Qbit}\\
&\mathit{teleportation}\ \mathit{iq} = \mathbf{do}\ (\mathit{bsq1}, \mathit{bsq2}) \leftarrow \mathit{bell}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{cd} \leftarrow \mathit{alice}\ \mathit{iq}\ \mathit{bsq1}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{tq} \leftarrow \mathit{bob}\ \mathit{bsq2}\ \mathit{cd}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{return}\ \mathit{tq}
\end{aligned}
$$

Qdata.

- There is a symmetry between initialising a qubit, and measuring a qubit.

Qdata.

- There is a symmetry between initialising a qubit, and measuring a qubit.
- Larger quantum data structures can be defined using qubits, in the same way classical data structures are defined using bits.

Qdata.

- There is a symmetry between initialising a qubit, and measuring a qubit.
- Larger quantum data structures can be defined using qubits, in the same way classical data structures are defined using bits.
- We have defined a class of quantum data types, *Qdata* For which an *mkQ* initialisation function and a *measQ* measurement function must be defined, between the quantum datatype and its classical counter-part.

Qdata.

- There is a symmetry between initialising a qubit, and measuring a qubit.
- Larger quantum data structures can be defined using qubits, in the same way classical data structures are defined using bits.
- We have defined a class of quantum data types, *Qdata* For which an *mkQ* initialisation function and a *measQ* measurement function must be defined, between the quantum datatype and its classical counter-part.
- $$\begin{aligned}&\textbf{instance}\ Qdata\ Bool\ Qbit\ \textbf{where}\\&\quad mkQ = mkQbit\\&\quad measQ = measQbit\end{aligned}$$

Qdata..

$$
\begin{aligned}
&\textbf{instance } (Qdata\ a\ qa, Qdata\ b\ qb) \\
&\qquad \Rightarrow Qdata\ (a, b)\ (qa, qb) \textbf{ where} \\
&\quad mkQ\ (a, b) = \quad \textbf{do } qa \leftarrow mkQ\ a \\
&\qquad\qquad qb \leftarrow mkQ\ b \\
&\qquad\qquad return\ (qa, qb) \\
&\quad measQ\ (qa, qb) = \quad \textbf{do } a \leftarrow measQ\ qa \\
&\qquad\qquad b \leftarrow measQ\ qb \\
&\qquad\qquad return\ (a, b)
\end{aligned}
$$

Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor's quantum factorisation algorithm.

Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor's quantum factorisation algorithm.
- We are going to use the QIO Monad to start reasoning about quantum computation in general.

Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor's quantum factorisation algorithm.
- We are going to use the QIO Monad to start reasoning about quantum computation in general.
- We are going to model other forms of quantum computer within the QIO Monad, such as the Measurment based model of quantum computations.

Further Work

- We are going to implement some of the famous quantum algorithms, such as Shor's quantum factorisation algorithm.
- We are going to use the QIO Monad to start reasoning about quantum computation in general.
- We are going to model other forms of quantum computer within the QIO Monad, such as the Measurment based model of quantum computations.
- Thank you all for listening!