Functional Quantum Programming
The Quantum IO Monad

The University of Nottingham

The Quantum IO Monad is an interface from Haskell to Quantum Computation and provides a *constructive semantics* for quantum programming.

The *QIO* monad provides a functional interface to quantum programming, similar to the way Haskell's *IO* monad provides an interface to conventional stateful programming. The basic idea is that our classical computer is connected to a quantum device which contains a number of qubits.

The quantum device can be instructed to:

- Set qubits to one of the computational base states (i.e. $|0\rangle = \mathit{False}$ or $|1\rangle = \mathit{True}$).
- Perform unitary operations involving one or several qubits.
- Measure qubits and observe the outcome.

We can either *run* our quantum program using *run* or we can *simulate* the quantum program using *sim* which calculates a probability distribution.

e.g.

$$\mathit{sim}\ (\mathit{bell} \ggg= \mathit{measQ}) = [((\mathit{True}, \mathit{True}), 0.5), ((\mathit{False}, \mathit{False}), 0.5)]$$

$$\mathit{run}\ (\mathit{deutsch}\ (\lambda x \rightarrow \mathit{True})) = \mathit{False}$$

The Side effects from measurement are dealt with by the monadic structure:

$\textbf{instance}\ \mathit{Monad}\ \mathit{QIO}$

$\mathit{mkQbit} :: \mathit{Bool} \rightarrow \mathit{QIO}\ \mathit{Qbit}$

$\mathit{applyU} :: U \rightarrow \mathit{QIO}\ ()$

$\mathit{measQbit} :: \mathit{Qbit} \rightarrow \mathit{QIO}\ \mathit{Bool}$

The reversible nature of unitaries is kept seperate in a monoidal structure:

$\textbf{instance}\ \mathit{Monoid}\ U$

$\mathit{rot} :: \mathit{Qbit} \rightarrow \mathit{Rotation} \rightarrow U$

$\mathit{swap} :: \mathit{Qbit} \rightarrow \mathit{Qbit} \rightarrow U$

$\mathit{cond} :: \mathit{Qbit} \rightarrow (\mathit{Bool} \rightarrow U) \rightarrow U$

$\mathit{ulet} :: \mathit{Bool} \rightarrow (\mathit{Qbit} \rightarrow U) \rightarrow U$

- Haskell is a *pure* functional language.
- Computations are modelled as the evaluation of mathematical expressions.
- Programs are pure mathematical functions, leading to an ease of abstraction and reasoning
- Effectful computations are simulated via monads (e.g. the *IO* Monad).
- **do** notation gives monadic programs a more imperative look and feel.

QIO Examples

Creating a Bell state:

```
share :: Qbit → QIO Qbit
share qa = do qb ← |0⟩
              applyU (ifQ qa (unot qb))
              return qb
bell :: QIO (Qbit, Qbit)
bell = do qa ← |+⟩
          qb ← share qa
          return (qa, qb)
```

Deutsch's Algorithm:

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

We also have an implementation of quantum teleportation, a library of reversible arithmetic functions, the quantum Fourier transform, and an implementation of Shor's algorithm.

The code for QIO is available online from http://www.cs.nott.ac.uk/~asg/QIO and more informtation on Haskell can be found at http://www.haskell.org

Alexander S. Green and Thorsten Altenkirch