The Quantum IO Monad

Thorsten Altenkirch

School of Computer Science and IT
University of Nottingham

December 18, 2006

Motivation

- Explain quantum programming to (functional) programmers.
- Sell functional programming to people in quantum computing..
- Provide an intermediate language for the implementation of high level quantum languages (like QML).
- Framework to discover and implement patterns for quantum programming.

Haskell

- Pure functional programming language.
- Close to constructive Mathematics (terminating fragment).
- go further: Type Theory (Epigram).
- **Effects** (e.g. Input/Output, State, Concurrency, . . .) are encapsulated in the IO monad.
- Proposal: Use *Functional specifications of IO* to reason about programs with IO.

Example: quick sort

$$
\begin{aligned}
&qsort :: (Ord\ a) \Rightarrow [a] \rightarrow [a] \\
&qsort\ [\,] = [\,] \\
&qsort\ (a : as) = qsort\ (filter\ (\lambda b \rightarrow b \leqslant a)\ as) \\
&\qquad\qquad\qquad + [a] \\
&\qquad\qquad\qquad + qsort\ (filter\ (\lambda b \rightarrow a < b)\ as)
\end{aligned}
$$

Example: Sieve of Erasthostenes

$$\mathit{primes} :: [\mathbb{Z}]$$

$$\mathit{primes} = \mathit{sieve}\ [2\,..]$$
$$\quad \textbf{where}\ \mathit{sieve}\ (p : \mathit{ns}) = p : (\mathit{sieve}\ [n \mid n \leftarrow \mathit{ns}, n\ \text{`}\mathit{mod}\text{`}\ p \not\equiv 0])$$

Monads in Haskell

class *Monad m* **where**

$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$

$return :: a \to m\ a$

Equations:

$$
\begin{array}{rcl}
return\ a \gg= f & = & f\ a \\
c \gg= return & = & c \\
(c \gg= f) \gg= g & = & c \gg= \lambda a \to f\ a \gg= g
\end{array}
$$

Computations are represented by morphisms in the Kleisli category

$$a \to_{\mathrm{Kleisli}} b = a \to m\ b$$

The state monad

$$\textbf{newtype}\ State\ s\ a = State\ (s \rightarrow (a, s))$$
$$\textbf{instance}\ Monad\ (State\ s)\ \textbf{where}$$
$$return\ a = State\ (\lambda s \rightarrow (a, s))$$
$$\begin{aligned}(State\ f) \gg\!= g = State \lambda s \rightarrow\ & \textbf{let}\ (a, s') = f\ s \\ & \qquad (State\ h) = g\ a \\ & \textbf{in}\ h\ s')\end{aligned}$$

Haskell's IO monad

instance *Monad IO*

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

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

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

Referential transparency

$$
\begin{array}{l}
\mathit{dotwice} :: \mathit{IO}\ () \to \mathit{IO}\ () \\
\mathit{dotwice}\ p = p \gg p
\end{array}
$$

The two following lines have the same behaviour:

$$\mathit{dotwice}\ (\mathit{putStrLn}\ \texttt{"Hello"})$$

$$(\mathit{putStrLn}\ \texttt{"Hello"}) \gg (\mathit{putStrLn}\ \texttt{"Hello"})$$

IORefs

$$
\begin{aligned}
&\textbf{type}\ \mathit{IORef}\ a \\
&\mathit{newIORef}\ ::\ a \rightarrow \mathit{IO}\ (\mathit{IORef}\ a) \\
&\mathit{writeIORef}\ ::\ \mathit{IORef}\ a \rightarrow a \rightarrow \mathit{IO}\ () \\
&\mathit{readIORef}\ ::\ \mathit{IORef}\ a \rightarrow \mathit{IO}\ a
\end{aligned}
$$

Functional specification of IO

type *Loc*

type *Data*

data *MyIO a* =
    NewIORef Data (*Loc* → *MyIO a*)
  | *ReadIORef Loc* (*Data* → *MyIO a*)
  | *WriteIORef Loc Data* (*MyIO a*)
  | *ReturnState a*

Functional specification of IO

$$
\begin{array}{l}
\textbf{type } \mathit{Heap} = \mathit{Loc} \to \mathit{Data} \\
\textbf{data } \mathit{Store} = \mathit{Store}\{\mathit{free} :: \mathit{Loc}, \mathit{heap} :: \mathit{Heap}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{run} :: \quad \mathit{MyIO}\ a \to a \\
\mathit{runState} :: \mathit{MyIO}\ a \to \mathit{State}\ \mathit{Store}\ a
\end{array}
$$

QIO

type *Qbit*

type *QIO a*
type *U*

instance *Monad QIO*

$$\begin{array}{lll} mkQbit & :: & Bool \rightarrow QIO\ Qbit \\ applyU & :: & U \rightarrow QIO\ () \\ meas & :: & Qbit \rightarrow QIO\ Bool \end{array}$$

Reversible Ops

instance *Monoid U*

$unot :: Qbit \rightarrow U$
$uhad :: Qbit \rightarrow U$
$uphase :: Qbit \rightarrow \mathbb{R} \rightarrow U$
$swap :: Qbit \rightarrow Qbit \rightarrow U$
$cond :: Qbit \rightarrow (Bool \rightarrow U) \rightarrow U$

$cond\ x\ (\lambda b \rightarrow$ **if** b **then** $unot\ x$ **else** $mempty)$
leads to a runtime error!

$urev :: U \rightarrow U$

run or sim

- *run* embeds QIO into IO using a random number generator:

  $$run :: QIO\ a \rightarrow IO\ a$$

- or a real quantum computer...
- *sim* calculates the probability distribution of possible answers:

  $$sim :: QIO\ a \rightarrow Prob\ a$$

- where

  $$\textbf{data}\ Prob\ a = Prob\ (Vec\ \mathbb{R}\ a)$$

Example: a random bit

$qran :: QIO\ Qbit$

$$\begin{aligned} qran = \mathbf{do}\ & qb \leftarrow mkQbit\ True \\ & applyU\ (uhad\ qb) \\ & return\ qb \end{aligned}$$

$test\_qran :: QIO\ Bool$

$$\begin{aligned} test\_qran = \mathbf{do}\ & qb \leftarrow qran \\ & meas\ qb \end{aligned}$$

$* Qio > run\ test\_qran$
$False$
$* Qio > run\ test\_qran$
$True$
$* Qio > sim\ test\_qran$
$[(True, 0.5), (False, 0.5)]$

The Bell state

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

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

Qdata

$$
\begin{array}{l}
\textbf{class } \mathit{Qdata}\ a\ \mathit{qa}\ \textbf{where} \\
\quad \mathit{mkQ} :: a \rightarrow \mathit{QIO}\ \mathit{qa} \\
\quad \mathit{measQ} :: \mathit{qa} \rightarrow \mathit{QIO}\ a
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Qdata}\ \mathit{Bool}\ \mathit{Qbit}\ \textbf{where} \\
\quad \mathit{mkQ} = \mathit{mkQbit} \\
\quad \mathit{measQ} = \mathit{measQbit}
\end{array}
$$

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

Quantum registers

$$\textbf{type}\ QR = [Qbit]$$

$$\textbf{instance}\ Qdata\ a\ qa \Rightarrow Qdata\ [a]\ [qa]\ \textbf{where}$$

$$
\begin{aligned}
&mkQ\ [\,] = return\ [\,]\\
&mkQ\ (b : bs) = \textbf{do}\ qb \leftarrow mkQ\ b\\
&\qquad\qquad\qquad\qquad qbs \leftarrow mkQ\ bs\\
&\qquad\qquad\qquad\qquad return\ (qb : qbs)\\
&measQ\ [\,] = return\ [\,]\\
&measQ\ (qx : qxs) = \textbf{do}\ x \leftarrow measQ\ qx\\
&\qquad\qquad\qquad\qquad\quad xs \leftarrow measQ\ qxs\\
&\qquad\qquad\qquad\qquad\quad return\ (x : xs)
\end{aligned}
$$

A quantum adder

$$
\begin{array}{l}
add1 :: Qbit \to Qbit \to Qbit \to U \\
add1\ qc\ qa\ qb = \\
\quad cond\ qc \lambda c \to \\
\qquad cond\ qa \lambda a \to \mathbf{if}\ a \not\equiv c \\
\qquad\qquad \mathbf{then}\ unot\ qb \\
\qquad\qquad \mathbf{else}\ mempty
\end{array}
$$

$$
\begin{array}{l}
carry :: Qbit \to Qbit \to Qbit \to Qbit \to U \\
carry\ qci\ qa\ qb\ qcsi = \\
\quad cond\ qci \lambda ci \to \\
\qquad cond\ qa \lambda a \to \\
\qquad\quad cond\ qb \lambda b \to \\
\qquad\qquad \mathbf{if}\ ci \wedge a \vee ci \wedge b \vee a \wedge b \\
\qquad\qquad \mathbf{then}\ unot\ qcsi \\
\qquad\qquad \mathbf{else}\ mempty
\end{array}
$$

A quantum adder

$$addc :: QR \rightarrow QR \rightarrow QR \rightarrow Qbit \rightarrow U$$

$$addc\ [\,]\ [\,]\ [\,]\ qc = mempty$$

$$\begin{aligned}&addc\ [qa]\ [qb]\ [qci]\ qc = \\ &\quad carry\ qci\ qa\ qb\ qc \\ &\quad `mappend` \\ &\quad add1\ qci\ qa\ qb\end{aligned}$$

$$\begin{aligned}&addc\ (qa : qas)\ (qb : qbs)\ (qci : qcsi : qcs)\ qc = \\ &\quad carry\ qci\ qa\ qb\ qcsi \\ &\quad `mappend` \\ &\quad addc\ qas\ qbs\ (qcsi : qcs)\ qc \\ &\quad `mappend` \\ &\quad urev\ (carry\ qci\ qa\ qb\ qcsi) \\ &\quad `mappend` \\ &\quad add1\ qci\ qa\ qb\end{aligned}$$

A quantum adder

$$
\begin{aligned}
&add :: QR \rightarrow QR \rightarrow Qbit \rightarrow QIO\ () \\
&add\ qaa\ qbb\ qc = \textbf{do}\ qcc \leftarrow mkQ\ (take\ ((length\ qaa)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (repeat\ False)) \\
&\qquad\qquad\qquad\qquad\quad applyU\ (addc\ qaa\ qbb\ qcc\ qc) \\
&\qquad\qquad\qquad\qquad\quad measQ\ qcc \\
&\qquad\qquad\qquad\qquad\quad return\ ()
\end{aligned}
$$

quantum garbage collection

- Qbits cannot be reclaimed by the garbage collector, because they may be entangled with other qbits.
- However, a measured qbit can be disposed.
- Hence, we have to remember in the classical state whether the qbit has been measured.

U and QIO as traces

$$\textbf{data } U = UReturn \mid Unot\ Qbit\ U \mid Uhad\ Qbit\ U$$
$$\mid Uphase\ Qbit\ Float\ U$$
$$\mid Swap\ Qbit\ Qbit\ U \mid Cond\ Qbit\ (Bool \to U)\ U$$

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

$$urev :: U \to U$$

$$urev\ UReturn = UReturn$$
$$urev\ (Unot\ x\ u) = urev\ u\ \text{`}mappend\text{`}\ unot\ x$$
$$urev\ (Uhad\ x\ u) = urev\ u\ \text{`}mappend\text{`}\ uhad\ x$$
$$urev\ (Uphase\ x\ phi\ u) = urev\ u\ \text{`}mappend\text{`}\ uphase\ x\ (-phi)$$
$$urev\ (Swap\ x\ y\ u) = urev\ u\ \text{`}mappend\text{`}\ swap\ x\ y$$
$$urev\ (Cond\ x\ br\ u) = urev\ u\ \text{`}mappend\text{`}\ cond\ x\ (urev \circ br)$$

Implementing Unitary

type *Heap*

instance $Num\ n \Rightarrow Monad\ (Vec\ n)$

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

newtype $Unitary = U\ (Heap \rightarrow Pure)$

instance *Monoid Unitary*

$runU :: U \rightarrow Unitary$

Implementing run and sim

$$\textbf{data}\ State = State\{free :: \mathbb{Z}, pure :: Pure\}$$

$$\textbf{class}\ Monad\ m \Rightarrow PMonad\ m\ \textbf{where}$$
$$\quad merge :: \mathbb{R} \to m\ a \to m\ a \to m\ a$$

$$\textbf{instance}\ PMonad\ IO$$

$$\textbf{data}\ Prob\ a = Prob\ (Vec\ \mathbb{R}\ a)$$

$$\textbf{instance}\ PMonad\ Prob$$

$$eval :: PMonad\ m \Rightarrow QIO\ a \to m\ a$$

$$run :: QIO\ a \to IO\ a$$
$$run = eval$$

$$sim :: QIO\ a \to Prob\ a$$
$$sim = eval$$

What next?

- Implement standard quantum algorithms...
- Measurement Calculus $\rightarrow$ QIO, or vice versa.
- Try to identify and implement *quantum programming patterns*.
- Formal reasoning about QIO (factor through superoperators).