Functional Quantum Programming

Thorsten Altenkirch

School of Computer Science and IT
University of Nottingham

April 26, 2007
People at Nottingham

Jonathan Grattage
recently successfully defended his PhD on QML
*A Quantum Programming Language*

Alex Green
works on quantum programming in a functional setting.

Slava Belavkin
Prof in Mathematical Physics,
Quantum Information Theory
Where do we come from . . .

- Functional Programming
e.g. functional treatment of concurrency
- Type Theory
e.g. Epigram: program+specify+prove
- Category Theory
e.g. Containers for generic programming
- Quantum Programming
e.g. QML, QIO monad
What is functional programming? And why?

- Based on function abstraction and application ($\lambda$ calculus).
- Ease of building abstractions, reasoning about programs
- Programming $\sim$ constructive mathematics.
- Popular functional language: Haskell
Functional Quantum Programming?

- QML - a quantum programming language
  - Design ideas
  - Operational and denotational semantics
  - An algebra of quantum programs
- The QIO monad in Haskell
- Questions for QICS
Starting point: Selinger’s QPL or Simon Perdrix’s quantum programming language.

Simple language with a nice mathematical semantics (Superoperators)

Unitary operators are represented as combinatorical expressions built up from some primitives, e.g. Hadmard, CNOT, etc

Slogan: Quantum data — classical control.

Quantum variables can only be used in a linear fashion (no contraction).
QML

- Contraction by sharing
- Explicit weakening by measurement
- Reversible $\text{if}^\circ$ and irreversible $\text{if}$
- Quantum data and control.
- No while loops.
Contraction by sharing

\[ \delta \in Q_2 \to Q_2 \otimes Q_2 \]
\[ \delta \ x = (x, x) \]

\[ \delta (false +_Q true) \neq (false +_Q true, false +_Q true) \]
\[ \delta (false +_Q true) \equiv (false, false) +_Q (true, true) \]
Explicit weakening by measurement

\[ \pi_1 \in Q_2 \otimes Q_2 \Rightarrow Q_2 \]

\[ \pi_1 (x, y) = x \uparrow \{ y \} \]

\[ \pi_1 (\delta x) \equiv x? \]

\[ \pi_1 (\delta (false +_Q true)) \equiv false +_P true \]
Reversible \texttt{if$^\circ$} and irreversible \texttt{if}

\[
\neg \in Q_2 \to Q_2 \\
\neg x = \texttt{if$^\circ$} x \texttt{ then false else true} \\
\neg (\neg x) \equiv x \\
\neg^c \in Q_2 \to Q_2 \\
\neg^c x = \texttt{if} x \texttt{ then false else true} \\
\neg^c (\neg^c (\texttt{false} +_Q \texttt{true})) \equiv \texttt{false} +_P \texttt{true}
\]
Why do we need \textbf{if}?

\[
cswap \in Q_2 \rightarrow Q_2 \otimes Q_2 \rightarrow Q_2 \otimes Q_2
\]
\[
cswap \ x \ (y, z) = \text{if}^\circ \ x \ \text{then} \ (z, y) \ \text{else} \ (y, z)
\]

is \textbf{not well-typed}, because we cannot show \((z, y) \perp (y, z)\).

\[
cswap' \in Q_2 \rightarrow Q_2 \otimes Q_2 \rightarrow Q_2 \otimes Q_2
\]
\[
cswap' \ x \ (y, z) = \text{if} \ x \ \text{then} \ (z, y) \ \text{else} \ (y, z)
\]

is well-typed, since \textbf{if} does not require orthogonality.
QML’s type system

We introduce the following judgements:

Programs

\[ \Gamma \vdash t : \sigma \]

Pure programs

\[ \Gamma \vdash ^{°} t : \sigma \]

programs without weakening and \textbf{if}.

Orthogonality

\[ t \perp u \]

given that \[ \Gamma \vdash ^{°} t, u : \sigma. \]
QML’s operational semantics

Given (non-empty) finite sets $A$ (input) and $B$ (output), we define $\text{FQC}(A, B)$ as:

\[
\begin{array}{c}
A \hspace{1cm} B \\
\hline
\phi \\
\hline
h \hspace{1cm} H \\
\hline
G
\end{array}
\]

- a finite set $H$, the base of the space of initial heaps,
- a heap initialisation vector $\vec{h} \in \mathbb{C}^H$,
- a finite set $G$, the base of the space of garbage states,
- a unitary operator $\phi \in A \otimes H \rightarrow_{\text{unitary}} B \otimes G$. 
QML’s operational semantics

We write:

\[ \text{FQC}(A, B) \] quantum circuits with heap and garbage.
\[ \text{FQC}^\circ(A, B) \] quantum circuits with heap but no garbage.

\[
\Gamma \vdash t : \sigma \\
\hline
\llbracket t \rrbracket^{\text{op}} \in \text{FQC}(\llbracket \Gamma \rrbracket, \llbracket \sigma \rrbracket)
\]

\[
\Gamma \vdash^\circ t : \sigma \\
\hline
\llbracket t \rrbracket^{\text{op}} \in \text{FQC}^\circ(\llbracket \Gamma \rrbracket, \llbracket \sigma \rrbracket)
\]
QML’s denotational semantics

We write:

\( \text{Super}(A, B) \) \hspace{1em} \text{Superoperators}

\( \text{Isom}(A, B) \) \hspace{1em} \text{Isometries}

\[
\frac{\Gamma \vdash t : \sigma}{\llbracket t \rrbracket^{\text{den}} \in \text{Super}(\llbracket \Gamma \rrbracket, \llbracket \sigma \rrbracket)}
\]

\[
\frac{\Gamma \vdash^\circ t : \sigma}{\llbracket t \rrbracket^{\text{den}} \in \text{Isom}(\llbracket \Gamma \rrbracket, \llbracket \sigma \rrbracket)}
\]
Relating operational and denotational semantics

We can assign denotations to circuits:

\[
\begin{align*}
\text{\(c \in FQC^\circ(A, B)\)} & \quad \text{\(\Rightarrow\)} & \quad \text{\(D(c) \in \text{Isom}(A, B)\)} \\
\text{\(c \in FQC(A, B)\)} & \quad \text{\(\Rightarrow\)} & \quad \text{\(D(c) \in \text{Super}(A, B)\)}
\end{align*}
\]

and state soundness of the operational semantics:

\[
D([t]^\text{op}) = [t]^\text{den}
\]

for \(\Gamma \vdash t : \sigma\) (\(\Gamma \vdash^\circ t : \sigma\)).
QUICS questions

- Can we extend QML by classical coproducts (corresponding to biproducts)?
- Can we extend QML by quantum views (corresponding to change of base)?
- Can we extend QML by higher order types?
It may seem that higher order is no problem because in every compact closed category:

$$C(A \otimes B, C) \simeq C(A, B^* \otimes C)$$

But is this the right structure?

Superoperators are not compact closed (but completely positive maps are).

The category of relations

$$\text{Rel}(A, B) = A \rightarrow \mathcal{P}(B)$$

is compact closed, but

$$\text{Rel}_{<\omega}(A, B) = A \rightarrow \mathcal{P}_{<\omega}(B)$$

is not.
Day’s construction

- For any category $C$ the category of presheaves $\text{PSh}(C)$ is given by:
  - **Objects** Contravariant functors from $C$ to $\text{Set}$.
  - **Morphisms** Natural transformations.
- There is an embedding $Y$ (the Yoneda embedding) from $C$ to $\text{PSh}(C)$
  \[ Y(A) = C(-, A) \]
- A monoidal structure on $C$ induces a monoidal structure in $\text{PSh}(C)$:
  \[ (F \otimes G)(X) = \int^{A,B} F(A) \times G(B) \times C(X, A \otimes B) \]
- $Y$ preserves the monoidal structure.
This structure is always closed:

\[(F \to G)(X) = \text{Nat}(Y(X) \otimes F, G)\]

This is semantically the interpretations of higher order computations as chunks (delayed computations).
An algebra of quantum programs?

*joint work with Amr Sabry and Juliana Vizotto.*

- QPL 2005: restricted to the pure fragment (no weakening, no if)
  - Denotational semantics: isometries
- Extends the rules for the classical sublanguage (no superpositions, if and if° behave the same)
  - Denotational semantics: sets and (injective) functions
- Sound and complete.
- Completeness also gives rise to a normalisation algorithm (*Normalisation by evaluation*).
Equations for $\texttt{if}^\circ$

$$\beta$$

\[ \texttt{if}^\circ \texttt{false} \texttt{then} \ t \ \texttt{else} \ u \equiv u \]
\[ \texttt{if}^\circ \texttt{true} \texttt{then} \ t \ \texttt{else} \ u \equiv t \]

$$\eta$$

\[ \texttt{if}^\circ \texttt{true} \texttt{then} \ \texttt{true} \ \texttt{else} \ \texttt{false} \equiv t \]

Commuting conversion

\[
\text{let } p = \texttt{if}^\circ t \texttt{then} \ u_0 \ \texttt{else} \ u_1 \\
\text{in } e \\
\equiv \texttt{if}^\circ t \texttt{then} (\text{let } p = u_0 \texttt{ in } e) \\
\texttt{else} (\text{let } p = u_1 \texttt{ in } e)
\]

Thorsten Altenkirch qics07
Equations for `let`

\[
\text{Val}^C ::= x \mid () \mid false \mid true \mid (val_1, val_2)
\]

\[\beta\]
\[
\text{let } p = \text{val} \text{ in } u \equiv u [\text{val} / p]
\]

\[\eta\]
\[
\text{let } x = t \text{ in } x \equiv t
\]

**Commuting conversion**
\[
\text{let } p = t \text{ in let } q = u \text{ in } e
\]
\[
\equiv \text{let } q = u \text{ in let } p = t \text{ in } e
\]
Quantum equations

\[(\text{if}^\circ)\]

\[\text{if}^\circ (t_0 + t_1) \text{ then } u_0 \text{ else } u_1 \]
\[\equiv (\text{if}^\circ t_0 \text{ then } u_0 \text{ else } u_1) + (\text{if}^\circ t_1 \text{ then } u_0 \text{ else } u_1)\]

\[\text{if}^\circ (\lambda \ast t) \text{ then } u_0 \text{ else } u_1 \]
\[\equiv \lambda \ast (\text{if}^\circ t \text{ then } u_0 \text{ else } u_1)\]

\[(\text{superpositions})\]

\[t + u \equiv u + t\]
\[t + 0 \equiv t\]
\[t + (u + v) \equiv (t + u) + v\]
\[\lambda \ast (t + u) \equiv \lambda \ast t + \lambda \ast u\]
\[\lambda \ast t + \kappa \ast t \equiv (\lambda + \kappa) \ast t\]
\[0 \ast t \equiv 0\]

Thorsten Altenkirch  qics07
QUICS questions

- Relation to the linear $\lambda$ calculus by Paolo and Gilles?
- Extend the theory to the full language (including measurements). 
  
  related to Ross’s question?
- Higher order?!
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.
Monads in Haskell

\[\text{class } Monad \ m \ \text{where}\]
\[>\triangleleft \in m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b\]
\[\text{return} \in a \rightarrow m \ a\]

Equations:

\[\text{return } a >\triangleleft f = f \ a\]
\[c >\triangleleft \text{return} = c\]
\[(c >\triangleleft f) >\triangleleft g = c >\triangleleft \lambda a \rightarrow f \ a >\triangleleft g\]

Computations are represented by morphisms in the Kleisli category

\[a \rightarrow_{\text{Kleisli}} b = a \rightarrow m \ b\]
Haskell’s IO monad

\textbf{instance} \textit{Monad IO}

\textit{getChar} \in \textit{IO Char}
\textit{putChar} \in \textit{Char} \rightarrow \textit{IO} ()

\textit{echo} \in \textit{IO} ()
\textit{echo} = \textit{getChar} \gg= \lambda c \rightarrow \textit{putChar} c \gg= \lambda x \rightarrow \textit{echo}

\textit{echo} = \textbf{do} c \leftarrow \textit{getChar}
\hspace{1em} \textit{putChar} c
\hspace{1em} \textit{echo}
QIO

\textbf{type} \textit{Qbit}
\textbf{type} \textit{QIO} \ a
\textbf{type} \textit{U}

\textbf{instance} \textit{Monad QIO}
\textit{mkQbit} \in \textit{Bool} \rightarrow \textit{QIO} \ \textit{Qbit}
\textit{applyU} \in \textit{U} \rightarrow \textit{QIO} \ ()
\textit{meas} \in \textit{Qbit} \rightarrow \textit{QIO} \ \textit{Bool}

Thorsten Altenkirch
Reversible Ops

instance Monoid U

unot ∈ Qbit → U
uhad ∈ Qbit → U
uphase ∈ Qbit → ℝ → U
swap ∈ Qbit → Qbit → U
cond ∈ Qbit → (Bool → U) → U

cond x (λb → if b then unot x else mempty)
leads to a runtime error!
run or sim

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

  \[
  \text{run} \in \text{QIO } a \rightarrow \text{IO } a
  \]

- or a real quantum computer…

- *sim* calculates the probability distribution of possible answers:

  \[
  \text{sim} \in \text{QIO } a \rightarrow \text{Prob } a
  \]

- where

  \[
  \text{data } \text{Prob } a = \text{Prob } (\text{Vec } \mathbb{R} \ a)
  \]
Example: a random bit

$qran \in QIO \ Qbit$

$qran = do \ \{ \ qb \leftarrow mkQbit \ True \$

\hspace{1cm} applyU (uhad qb) \$

\hspace{1cm} return qubit \$

$\}

test\_qran \in QIO \ Bool$

$test\_qran = do \ \{ \ qb \leftarrow qran \$

\hspace{1cm} meas qubit \$

$\}

* $Qio > run \ test\_qran$

False

* $Qio > run \ test\_qran$

True

* $Qio > sim \ test\_qran$

[[(True, 0.5), (False, 0.5)]]
The Bell state

\[ \text{share} \in \text{Qbit} \rightarrow \text{QIO Qbit} \]

\[
\text{share qa} = \text{do} \begin{align*}
qb &\leftarrow \text{mkQbit} \ \text{False} \\
\text{applyU} (\text{cond qa} \lambda a \rightarrow \begin{align*}
\text{if} \ a \\
\text{then} \ \text{unot qb} \\
\text{else} \ \text{mempty}
\end{align*})
\end{align*}
\]

\[ \return qb \]

\[ \text{bell} \in \text{QIO (Qbit, Qbit)} \]

\[
\text{bell} = \text{do} \begin{align*}
qa &\leftarrow \text{qran} \\
qb &\leftarrow \text{share qa} \\
\return (qa, qb)
\end{align*}
\]
QUICS questions

- Measurement Calculus $\rightarrow$ QIO, or vice versa.
- Formal reasoning about QIO (factor through superoperators).
Hypotheses

- Interesting interactions between functional and quantum programming
- Design programming language as a vehicle to express high level patterns of quantum programming.
- Denotational semantics reflect our understanding of quantum programming.