QML: A functional quantum programming language

quantum control and orthogonality

Jonathan Grattage
with Thorsten Altenkirch & Alex Green

sneezy.cs.nott.ac.uk/qml
School of Computer Science & IT,
The University of Nottingham
What is QML?

- A high–level quantum programming language with a structure familiar to functional programmers, which supports reasoning and algorithm design.
What is QML?

- A high-level quantum programming language with a structure familiar to functional programmers, which supports reasoning and algorithm design.
- Simplifying the design of quantum programs by:
  - Allowing formal reasoning principles for quantum programs.
  - Giving a more intuitive understanding of quantum algorithms.
What is QML?

- A high-level quantum programming language with a structure familiar to functional programmers, which supports reasoning and algorithm design
- Simplifying the design of quantum programs by:
  - Allowing formal reasoning principles for quantum programs
  - Giving a more intuitive understanding of quantum algorithms
- *A functional quantum programming language*, LICS 2005 with Thorsten Altenkirch
What is QML?

- A high–level quantum programming language with a structure familiar to functional programmers, which supports reasoning and algorithm design
- Simplifying the design of quantum programs by:
  - Allowing formal reasoning principles for quantum programs
  - Giving a more intuitive understanding of quantum algorithms
- *A functional quantum programming language*, LICS 2005 with Thorsten Altenkirch
What is QML?

- A high-level quantum programming language with a structure familiar to functional programmers, which supports reasoning and algorithm design.
- Simplifying the design of quantum programs by:
  - Allowing formal reasoning principles for quantum programs
  - Giving a more intuitive understanding of quantum algorithms
- *A functional quantum programming language*, LICS 2005 with Thorsten Altenkirch
- Project Site: QML@CS.Nott – sneezy.cs.nott.ac.uk/qml
Quantum Languages

- P. Zuliani, PhD 2001, *Quantum Programming* (qGCL)
- P. Selinger, MSCS 2003, *Towards a Quantum Programming Language* (QPL)
- A. van Tonder, SIAM 2003, *A Lambda Calculus for Quantum Computation*
- A. Sabry, Haskell 2003, *Modeling quantum computing in Haskell*
- P. Selinger and B. Valiron, TLCA 2005, *A lambda calculus for quantum computation with classical control*
- ...
Quantum Languages

- P. Zuliani, PhD 2001, Quantum Programming (qGCL)
- P. Selinger, MSCS 2003, Towards a Quantum Programming Language (QPL)
- A. van Tonder, SIAM 2003, A Lambda Calculus for Quantum Computation
- A. Sabry, Haskell 2003, Modeling quantum computing in Haskell
- P. Selinger and B. Valiron, TLCA 2005, A lambda calculus for quantum computation with classical control
- ...
- Quantum data, Classical control
QML Overview

- A first-order functional language for quantum computations on finite types
QML Overview

- A first-order functional language for quantum computations on finite types
- “Quantum Data and Control”
QML Overview

- A first-order functional language for quantum computations on finite types
- “Quantum Data and Control”
- Based on strict linear logic - controlled, explicit, weakening
QML Overview

- A first-order functional language for quantum computations on finite types
- “Quantum Data and Control”
- Based on strict linear logic - controlled, explicit, weakening
- Operational Semantics: Quantum Circuit Model
QML Overview

- A first-order functional language for quantum computations on finite types
- “Quantum Data and Control”
- Based on strict linear logic - controlled, explicit, weakening
- Operational Semantics: Quantum Circuit Model
- Denotational Semantics: Superoperators
QML Overview

- A first-order functional language for quantum computations on finite types
- “Quantum Data and Control”
- Based on strict linear logic - controlled, explicit, weakening
- Operational Semantics: Quantum Circuit Model
- Denotational Semantics: Superoperators

- Notion of Finite Quantum Computations (FQC) developed by analogy with Finite Classical Computations (FCC)
<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td></td>
</tr>
</tbody>
</table>
### Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Classical Case (FCC)</td>
<td>Quantum Case (FQC)</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td></td>
</tr>
</tbody>
</table>
## Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td>Tensor product ($\otimes$)</td>
</tr>
</tbody>
</table>
### Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td>Tensor product ($\otimes$)</td>
</tr>
<tr>
<td>Bijections</td>
<td></td>
</tr>
</tbody>
</table>
### Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td>Tensor product ($\otimes$)</td>
</tr>
<tr>
<td>Bijective</td>
<td>Unitary operators</td>
</tr>
</tbody>
</table>
## Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ((\times))</td>
<td>Tensor product ((\otimes))</td>
</tr>
<tr>
<td>Bijections</td>
<td>Unitary operators</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
</tbody>
</table>
## Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (<strong>FCC</strong>)</th>
<th>Quantum Case (<strong>FQC</strong>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ((\times))</td>
<td>Tensor product ((\otimes))</td>
</tr>
<tr>
<td>Bijections</td>
<td>Unitary operators</td>
</tr>
<tr>
<td>Functions</td>
<td>Superoperators</td>
</tr>
</tbody>
</table>
## Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (<strong>FCC</strong>)</th>
<th>Quantum Case (<strong>FQC</strong>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td>Tensor product ($\otimes$)</td>
</tr>
<tr>
<td>Bijections</td>
<td>Unitary operators</td>
</tr>
<tr>
<td>Functions</td>
<td>Superoperators</td>
</tr>
<tr>
<td>Injective functions</td>
<td></td>
</tr>
</tbody>
</table>
## Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product (×)</td>
<td>Tensor product (⊗)</td>
</tr>
<tr>
<td>Bijections</td>
<td>Unitary operators</td>
</tr>
<tr>
<td>Functions</td>
<td>Superoperators</td>
</tr>
<tr>
<td>Injective functions</td>
<td>Isometries</td>
</tr>
</tbody>
</table>
# Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product (×)</td>
<td>Tensor product (∇)</td>
</tr>
<tr>
<td>Bijections</td>
<td>Unitary operators</td>
</tr>
<tr>
<td>Functions</td>
<td>Superoperators</td>
</tr>
<tr>
<td>Injective functions</td>
<td>Isometries</td>
</tr>
<tr>
<td>Projections</td>
<td></td>
</tr>
</tbody>
</table>
### Classical v. Quantum

<table>
<thead>
<tr>
<th>Classical Case (FCC)</th>
<th>Quantum Case (FQC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite sets</td>
<td>Finite dimensional Hilbert spaces</td>
</tr>
<tr>
<td>Cartesian product ((\times))</td>
<td>Tensor product ((\otimes))</td>
</tr>
<tr>
<td>Bijections</td>
<td>Unitary operators</td>
</tr>
<tr>
<td>Functions</td>
<td>Superoperators</td>
</tr>
<tr>
<td>Injective functions</td>
<td>Isometries</td>
</tr>
<tr>
<td>Projections</td>
<td>Partial trace</td>
</tr>
</tbody>
</table>

QML: A functional quantum programming language – p.5/14
QML Syntax

- Types:
  \[ \sigma = Q_1 \mid Q_2 \mid \sigma \otimes \tau \]
QML Syntax

- **Types:**
  \[ \sigma = Q_1 \mid Q_2 \mid \sigma \otimes \tau \]

- **Syntax:**
  
  (Variables) \[ x, y, \ldots \in Vars \]

  (Prob. amplitudes) \[ \kappa, \iota, \ldots \in \mathbb{C} \]

  (Patterns) \[ p, q ::= x \mid (x, y) \]

  (Terms) \[ t, u, e ::= x \mid x^g \mid () \mid (t, u) \]

  \[ \mid \text{let } p = t \text{ in } u \]

  \[ \mid \text{if } t \text{ then } u \text{ else } u' \]

  \[ \mid \text{if}^o t \text{ then } u \text{ else } u' \]

  \[ \mid \text{false} \mid \text{true} \mid \overrightarrow{0} \mid \kappa \times t \mid t + u \]
QML Syntax

- **Types:**
  \[ \sigma = Q_1 \mid Q_2 \mid \sigma \otimes \tau \]

- **Syntax:**
  - **(Variables)** \( x, y, \ldots \in Vars \)
  - **(Prob.amplitudes)** \( \kappa, \iota, \ldots \in \mathbb{C} \)
  - **(Patterns)** \( p, q ::= x \mid (x, y) \)
  - **(Terms)** \( t, u, e ::= x \mid x \vec{y} \mid () \mid (t, u) \)
    - \( \mid \text{let } p = t \text{ in } u \)
    - \( \mid \text{if } t \text{ then } u \text{ else } u' \)
    - \( \mid \text{if}^\circ t \text{ then } u \text{ else } u' \)
    - \( \mid \text{false} \mid \text{true} \mid 0 \mid \kappa \times t \mid t + u \)

- **EPR State:**
  \[
  \frac{1}{\sqrt{2}} \times (false, false) + \frac{1}{\sqrt{2}} \times (true, true)
  \]
Control of Decoherence

- Projection Function

\[ \pi_1 \in (Q_2, Q_2) \rightarrow C_2 \]

\[ \pi_1 (x, y) = x \]
Control of Decoherence

- **Projection Function**

  \[ \pi_1 \in (Q_2, Q_2) \to C_2 \]

  \[ \pi_1 (x, y) = x \]

- **Diagonal Function**

  \[ \delta \in Q_2 \to (Q_2, Q_2) \]

  \[ \delta x = (x, x) \]
Control of Decoherence

- $\pi_1.\delta \in Q_2 \rightarrow Q_2$

```
x :: Q_2
0 :: Q_2
φ_δ
φ_{π_1}
```

$x :: Q_2$

```
```

$x :: Q_2$
Control of Decoherence

- $\pi_1.\delta \in Q_2 \rightarrow Q_2$

- Classical Case:

$$x :: Q_2 \quad 0 :: Q_2 \quad x :: Q_2$$

$$Q_2 \quad \phi_\delta \quad \phi_{\pi_1} \quad Q_2$$
Control of Decoherence

- \( \pi_1 \cdot \delta \in Q_2 \rightarrow Q_2 \)

- Classical Case:

- Quantum Case:
  
  Input = \( \frac{1}{\sqrt{2}} \times false + \frac{1}{\sqrt{2}} \times true \) (equal superposition)
Control of Decoherence

- $\pi_1.\delta \in \mathbb{Q}_2 \rightarrow \mathbb{Q}_2$

- Classical Case:
  $Q_2 \xrightarrow{\delta} Q_2$

- Quantum Case:
  Input = $\frac{1}{\sqrt{2}} \times false + \frac{1}{\sqrt{2}} \times true$ (equal superposition)
  Output = $\left\{ \frac{1}{2} \right\} false + \left\{ \frac{1}{2} \right\} true$ (probability distribution)
Control of Decoherence

- $\pi_1 \cdot \delta \in Q_2 \rightarrow Q_2$

- Classical Case:

- Quantum Case:
  Input = $\frac{1}{\sqrt{2}} \times \text{false} + \frac{1}{\sqrt{2}} \times \text{true}$ (equal superposition)
  Output = $\{\frac{1}{2}\}\text{false} + \{\frac{1}{2}\}\text{true}$ (probability distribution)

Decoherence! Not the identity function
More Decoherence
• \textit{forget} mentions $x$
  
  $\text{forget} \in \mathbb{Q}_2 \rightarrow \mathbb{Q}_2$
  
  $\text{forget } x = \text{if } x \text{ then true else true}$
More Decoherence

- \textit{forget} mentions \( x \)
  
  \[ \text{forget} \in Q_2 \rightarrow Q_2 \]
  
  \[ \text{forget} \; x = \text{if} \; x \; \text{then} \; \text{true} \; \text{else} \; \text{true} \]

- but doesn't use it.
More Decoherence

- $\text{forget}$ mentions $x$
  
  $$\text{forget} \in Q_2 \rightarrow Q_2$$
  
  $$\text{forget } x = \text{if } x \text{ then } \text{true} \text{ else } \text{true}$$

- but doesn’t use it.

- Hence, it has to measure it!
More Decoherence

- **forget** mentions \( x \)
  \[
  \text{forget} \in \mathcal{Q}_2 \rightarrow \mathcal{Q}_2 \\
  \text{forget} \ x = \text{if} \ x \text{ then } \text{true} \text{ else } \text{true}
  \]

- but doesn’t use it.
- Hence, it **has** to measure it!
- **if** always measures the conditional
More Decoherence

• `forget` mentions \( x \)
  \[
  \begin{align*}
  \text{forget} & \in Q_2 \rightarrow Q_2 \\
  \text{forget } x = & \text{ if } x \text{ then true else true}
  \end{align*}
  \]

• but doesn’t use it.
• Hence, it \textbf{has} to measure it!
• if always measures the conditional
• Not, using if
  \[
  \begin{align*}
  \text{not} & \in Q_2 \rightarrow Q_2 \\
  \text{not } x = & \text{ if } x \text{ then false else true}
  \end{align*}
  \]
if\textdegree – Quantum control
if° – Quantum control

\[\text{forget}' : Q_2 \rightarrow Q_2\]

\[\text{forget}' x = \text{if}° x \text{ then true else true}\]
if° – Quantum control

- \(\text{forget}' \in Q_2 \rightarrow Q_2\)

- \(\text{forget}' \ x = \text{if}° \ x \ \text{then true else true}\)

- This program has a type error, because true \(\neq\) true.
if\(^\circ\) – Quantum control

- \(\text{forget}' \in Q_2 \rightarrow Q_2\)
  
  \(\text{forget}' \ x = \text{if}\(^\circ\) \ x \ \text{then} \ \text{true} \ \text{else} \ \text{true}\)

- This program has a type error, because \(\text{true} \not\equiv \text{true}\).

- \(qnot \in Q_2 \rightarrow Q_2\)
  
  \(qnot \ x = \text{if}\(^\circ\) \ x \ \text{then} \ \text{false} \ \text{else} \ \text{true}\)
if$^\circ$ – Quantum control

- \( \text{forget}' \in Q_2 \rightarrow Q_2 \)
  \( \text{forget}' \ x = \text{if}^\circ \ x \ \text{then} \ \text{true} \ \text{else} \ \text{true} \)

- This program has a type error, because true $\not\equiv$ true.

- \( q\text{not} \in Q_2 \rightarrow Q_2 \)
  \( q\text{not} \ x = \text{if}^\circ \ x \ \text{then} \ \text{false} \ \text{else} \ \text{true} \)

- This program typechecks, because false $\perp$ true.
• \( \text{forget}' \in Q_2 \rightarrow Q_2 \)
  \[ \text{forget'} \ x = \text{if}^{\circ} x \ \text{then} \ \text{true} \ \text{else} \ \text{true} \]

• This program has a type error, because \( \text{true} \not\perp \text{true} \).

• \( qnot \in Q_2 \rightarrow Q_2 \)
  \[ qnot \ x = \text{if}^{\circ} x \ \text{then} \ \text{false} \ \text{else} \ \text{true} \]

• This program typechecks, because \( \text{false} \perp \text{true} \).

• \( \text{cnot} \ c \ x = \text{if}^{\circ} c \ \text{then} \ (\text{true}, qnot \ x) \ \text{else} \ (\text{false}, x) \)
if° – Quantum control

- \( \text{\(\text{forget}' \in Q_2 \rightarrow Q_2\)} \)
  \(\text{\(\text{forget'} x = \text{\(\text{if}^\circ x \text{ then} \text{ true} \text{ else} \text{ true}\)} \)}\)
- This program has a type error, because \(\text{true} \not\approx \text{true}\).

- \(\text{\(\text{qnot} \in Q_2 \rightarrow Q_2\)} \)
  \(\text{\(\text{qnot} x = \text{\(\text{if}^\circ x \text{ then} \text{ false} \text{ else} \text{ true}\)} \)}\)
- This program typechecks, because \(\text{false} \not\approx \text{true}\).

- \(\text{\(\text{cnot} c x = \text{\(\text{if}^\circ c \text{ then} (\text{true}, \text{qnot} x) \text{ else} (\text{false}, x)\)} \)}\)
- Deutsch-Joza Algorithm, Quantum Teleport Algorithm, ...
Quantum Teleport Algorithm

\[
p_{\text{Zed}} \in \mathbb{Q}_2 \rightarrow \mathbb{Q}_2
\]

\[
p_{\text{Zed}} x = \text{if }^\circ x \text{ then } (-1) \times \text{true} \text{ else } \text{false}
\]

\[
h_{\text{ad}} \in \mathbb{Q}_2 \rightarrow \mathbb{Q}_2
\]

\[
h_{\text{ad}} x = \text{if }^\circ x \text{ then } (-1) \times \text{true} + \text{false} \text{ else } \text{true} + \text{false}
\]

\[
t_{\text{el}} \in \mathbb{Q}_2 \rightarrow \mathbb{Q}_2
\]

\[
t_{\text{el}} x = \text{let } (a, b) = (\text{false}, \text{false}) + (\text{true}, \text{true})
\]

\[
(a', x') = \text{cnot } a x
\]

\[
b' = \text{if } a' \text{ then } \text{qnot } b \text{ else } b
\]

\[
b'' = \text{if }^\circ \text{had } x' \text{ then } p_{\text{Zed}} b' \text{ else } b'
\]

\[
\text{in } b''
\]
We define the inner product of terms, which to any pair of terms \( \Gamma \vdash t, u : \sigma \) assigns \( \langle t \mid u \rangle \in \mathbb{C} \cup \{?\} \).
We define the inner product of terms, which to any pair of terms $\Gamma \vdash t, u : \sigma$ assigns $\langle t | u \rangle \in \mathbb{C} \cup \{?\}$.

- $t \perp u$ holds if $\langle t | u \rangle = 0$. 
- We define the inner product of terms, which to any pair of terms \( \Gamma \vdash t, u : \sigma \) assigns \( \langle t|u \rangle \in \mathbb{C} \cup \{?\} \).
- \( t \perp u \) holds if \( \langle t|u \rangle = 0 \).
- \( \langle t|t \rangle = 1 \), \( \langle \text{false}|\text{true} \rangle = 0 \), \( \langle \text{true}|\text{false} \rangle = 0 \)
Inner Product & ⊥

- We define the inner product of terms, which to any pair of terms \( \Gamma \vdash t, u : \sigma \) assigns \( \langle t|u \rangle \in \mathbb{C} \cup \{?\} \).
- \( t \perp u \) holds if \( \langle t|u \rangle = 0 \).
- \( \langle t|t \rangle = 1 \), \( \langle \text{false}|\text{true} \rangle = 0 \), \( \langle \text{true}|\text{false} \rangle = 0 \)
- \( \langle \vec{0}|\text{true} \rangle = 0 = \langle \text{true} | \vec{0} \rangle \), \( \langle \vec{0}|\text{false} \rangle = 0 = \langle \text{false} | \vec{0} \rangle \), \( \langle \vec{0}|x \rangle = 0 = \langle x|\vec{0} \rangle \)
We define the inner product of terms, which to any pair of terms \( \Gamma \vdash t, u : \sigma \) assigns \( \langle t | u \rangle \in \mathbb{C} \cup \{?\} \).

- \( t \perp u \) holds if \( \langle t | u \rangle = 0 \).

- \( \langle t | t \rangle = 1 \), \( \langle \text{false} | \text{true} \rangle = 0 \), \( \langle \text{true} | \text{false} \rangle = 0 \)

- \( \langle \overrightarrow{0} | \text{true} \rangle = 0 = \langle \text{true} | \overrightarrow{0} \rangle \), \( \langle \overrightarrow{0} | \text{false} \rangle = 0 = \langle \text{false} | \overrightarrow{0} \rangle \), \( \langle \overrightarrow{0} | x \rangle = 0 = \langle x | \overrightarrow{0} \rangle \)

- \( \langle (t, t') | (u, u') \rangle = \langle t | u \rangle \ast \langle t' | u' \rangle \)
Inner Product & \(\perp\)

- We define the inner product of terms, which to any pair of terms \(\Gamma \vdash t, u : \sigma\) assigns \(\langle t | u \rangle \in \mathbb{C} \cup \{?\}\).
- \(t \perp u\) holds if \(\langle t | u \rangle = 0\).
- \(\langle t | t \rangle = 1, \quad \langle \text{false} | \text{true} \rangle = 0, \quad \langle \text{true} | \text{false} \rangle = 0\)
- \(\langle \vec{0} | \text{true} \rangle = 0 = \langle \text{true} | \vec{0} \rangle, \quad \langle \vec{0} | \text{false} \rangle = 0 = \langle \text{false} | \vec{0} \rangle, \quad \langle \vec{0} | x \rangle = 0 = \langle x | \vec{0} \rangle\)
- \(\langle (t, t') | (u, u') \rangle = \langle t | u \rangle * \langle t' | u' \rangle\)
- \(\langle \lambda * t + \lambda' * t' | u \rangle = \lambda^* * \langle t | u \rangle + \lambda'^* * \langle t' | u \rangle, \quad \langle t | \kappa * u + \kappa' * u' \rangle = \kappa * \langle t | u \rangle + \kappa' * \langle t | u' \rangle\)
Inner Product & ⊥

- We define the inner product of terms, which to any pair of terms $\Gamma \vdash t, u : \sigma$ assigns $\langle t|u \rangle \in \mathbb{C} \cup \{?\}$.

- $t \perp u$ holds if $\langle t|u \rangle = 0$.

- $\langle t|t \rangle = 1$, $\langle \text{false}|\text{true} \rangle = 0$, $\langle \text{true}|\text{false} \rangle = 0$

- $\langle \overrightarrow{0}|\text{true} \rangle = 0 = \langle \text{true}|\overrightarrow{0} \rangle$, $\langle \overrightarrow{0}|\text{false} \rangle = 0 = \langle \text{false}|\overrightarrow{0} \rangle$,
  $\langle \overrightarrow{0}|x \rangle = 0 = \langle x|\overrightarrow{0} \rangle$

- $\langle (t, t') | (u, u') \rangle = \langle t|u \rangle \ast \langle t'|u' \rangle$

- $\langle \lambda \ast t + \lambda' \ast t' | u \rangle = \lambda \ast \langle t|u \rangle + \lambda' \ast \langle t'|u \rangle$,
  $\langle t | \kappa \ast u + \kappa' \ast u' \rangle = \kappa \ast \langle t|u \rangle + \kappa' \ast \langle t|u' \rangle$

- ...
We define the inner product of terms, which to any pair of terms \( \Gamma \vdash t, u : \sigma \) assigns \( \langle t|u \rangle \in \mathbb{C} \cup \{?\} \).

- \( t \perp u \) holds if \( \langle t|u \rangle = 0 \).

- \( \langle t|t \rangle = 1 \), \( \langle \text{false}|\text{true} \rangle = 0 \), \( \langle \text{true}|\text{false} \rangle = 0 \)

- \( \langle \overrightarrow{0}|\text{true} \rangle = 0 = \langle \text{true}|\overrightarrow{0} \rangle \), \( \langle \overrightarrow{0}|\text{false} \rangle = 0 = \langle \text{false}|\overrightarrow{0} \rangle \),
  \( \langle \overrightarrow{0}|x \rangle = 0 = \langle x|\overrightarrow{0} \rangle \)

- \( \langle (t, t') | (u, u') \rangle = \langle t|u \rangle \ast \langle t'|u' \rangle \)

- \( \langle \lambda \ast t + \lambda' \ast t' | u \rangle = \lambda^* \ast \langle t|u \rangle + \lambda'^* \ast \langle t'|u \rangle \),
  \( \langle t | \kappa \ast u + \kappa' \ast u' \rangle = \kappa \ast \langle t|u \rangle + \kappa' \ast \langle t|u' \rangle \)

- \( \ldots \)

- \( \langle t|u \rangle = ? \) otherwise
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
- Our analysis also highlights the differences between classical and quantum programming.
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
- Our analysis also highlights the differences between classical and quantum programming.
- Quantum programming introduces the problem of control of decoherence, which we address by making the ‘forgetting’ of variables explicit, and by having two if constructs and the orthogonality condition.
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
- Our analysis also highlights the differences between classical and quantum programming.
- Quantum programming introduces the problem of control of decoherence, which we address by making the ‘forgetting’ of variables explicit, and by having two if constructs and the orthogonality condition.
- Other work:
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
- Our analysis also highlights the differences between classical and quantum programming.
- Quantum programming introduces the problem of control of decoherence, which we address by making the ‘forgetting’ of variables explicit, and by having two if constructs and the orthogonality condition.
- Other work:
  - A compiler for QML, in Haskell, that produces (typed) circuits.
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
- Our analysis also highlights the differences between classical and quantum programming.
- Quantum programming introduces the problem of control of decoherence, which we address by making the ‘forgetting’ of variables explicit, and by having two if constructs and the orthogonality condition.
- Other work:
  - A compiler for QML, in Haskell, that produces (typed) circuits.
  - Extend algebra and normal form beyond pure subset of QML.
Summary

- Our semantic ideas proved useful when designing a quantum programming language, analogous concepts are modelled by the same syntactic constructs.
- Our analysis also highlights the differences between classical and quantum programming.
- Quantum programming introduces the problem of control of decoherence, which we address by making the ‘forgetting’ of variables explicit, and by having two if constructs and the orthogonality condition.
- Other work:
  - A compiler for QML, in Haskell, that produces (typed) circuits.
  - Extend algebra and normal form beyond pure subset of QML.
  - Higher-order types, composionality proofs, implement using measurement calculus...
Thanks for listening

- Papers on QML can be found at:
  - sneezy.cs.nott.ac.uk/qml
- There is also an interactive research diary:
  - sneezy.cs.nott.ac.uk/qml/weblog

- Jonathan Grattage (www.cs.nott.ac.uk/~jjg)
- Thorsten Altenkirch (www.cs.nott.ac.uk/~txa)