Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 270 (1) (2011) 165–174

www.elsevier.com/locate/entcs

# An Overview of QML With a Concrete Implementation in Haskell

Jonathan Grattage

*Laboratoire d'Informatique de Grenoble, CNRS, France*
jonathan.grattage@imag.fr

**Abstract**

This paper gives an introduction to and overview of the functional quantum programming language QML. The syntax of this language is defined and explained, along with a new QML definition of the quantum teleport algorithm. The categorical operational semantics of QML is also briefly introduced, in the form of annotated quantum circuits. This definition leads to a denotational semantics, given in terms of superoperators. Finally, an implementation in Haskell of the semantics for QML is presented as a compiler. The compiler takes QML programs as input, which are parsed into a Haskell datatype. The output from the compiler is either a quantum circuit (operational), an isometry (pure denotational) or a superoperator (impure denotational). Orthogonality judgements and problems with coproducts in QML are also discussed.

*Keywords:* QML, language, functional, teleport, denotational semantics, operational semantics, Haskell.

## 1 Introduction and motivation

Language development for quantum computation is a rapidly developing research area [4], motivated by the application of established formal reasoning and verification techniques within a quantum framework, understanding the behaviour of quantum computation, aiding the development of new algorithms and gaining a deeper understanding of how they work. This paper discusses the syntax and features of, and gives a compiler for, a language allowing both classical and quantum control: QML [1,5]. The syntax and semantics for QML is a complete redevelopment of that presented previously [1], as the language has been changed to remove a problematic interpretation of coproducts (section 2.2); the interpretation of orthogonality has also been updated (section 2.1). In addition, in this work the operational semantics is made concrete by a compiler for QML programs implemented in Haskell.

[1] The author gratefully acknowledges Thorsten Altenkirch in the development of QML and the operational semantics. Ellie D'Hondt is also thanked for her constructive suggestions.

doi:10.1016/j.entcs.2011.01.015

QML has a syntax similar to other functional languages [4], and is based on strict linear logic: *i.e.* linear logic with contraction, but without implicit weakening (in contrast, Selinger's QPL uses affine linear logic [8], which allows weakening but not contraction). QML also integrates reversible and irreversible computations in a single language, where weakenings (which can give rise to the collapse of superpositions and entanglement) must be explicit. Contractions are allowed and are modelled as a form of sharing, analogous to the behaviour of classical functional languages. Differences between QML and other languages include the use of a quantum control operation, provided by the *quantum-if* construct $\mathbf{if}^{\circ}$. To use the $\mathbf{if}^{\circ}$ operation, the branches of the computation must be orthogonal (distinguishable), the proof of which is supplied automatically by the type checker at compilation, and hence the programmer need not supply the condition, nor need it appear in the syntax of terms. Classical control is provided by a second *classical-if* construct **if**, which measures the term being branched over.

QML has both an operational and denotational semantics [5], supporting formal reasoning principles with an algebra for equational reasoning and a normal form [2]. The operational semantics of QML is presented using a categorical formalisation of the quantum circuit model, which is realised by a compiler that translates QML programs into 'typed' quantum circuits. The denotational semantics is also implemented, which translates QML terms, via the operational semantics category **FQC**, into either isometries $\mathbf{Q}^{\circ}$ or superoperators $\mathbf{Q}$.

Recent developments in QML include a fully operational compiler, with a type inference algorithm for QML terms, and the automatic derivation and extension of the orthogonality judgements and circuits required for quantum control, which have all been implemented. The implementation of the orthogonality judgements is such that it can be easily expanded as new rules for proving the orthogonality of terms are added to QML. This short paper outlines the new syntax of QML and its semantics. A new and faithful interpretation of the quantum teleportation algorithm in QML is also presented as an example of using QML. Details for using the compiler, and the output it generates, are then provided. Finally, the future development of the language, the semantics and compiler, and the uses of the compiler are described.

## 2 The syntax of QML

This section introduces the syntax for QML (see also ref. [5]). The symbols $\sigma, \tau$ are used to vary over QML types, given by $\sigma = \mathcal{Q}_{\mathbf{1}} \mid \mathcal{Q}_{\mathbf{2}} \mid \sigma \otimes \tau$, where the type constructor is the tensor product $\otimes$ corresponding to a product type and $\mathcal{Q}_{\mathbf{2}}$ is a qubit type. $x, y, z$ are used to vary over names. Typing contexts $(\Gamma, \Delta)$ are given by $\Gamma = \bullet \mid \Gamma, x : \sigma$ where $\bullet$ is the empty context. Contexts correspond to functions from a finite set of variables to types.

Constants $\kappa, \iota \in \mathbb{C}$ are also used to define the syntax of expressions. Function variables are used to refer to previously defined QML programs. The terms of QML consist of those of a first-order functional language, extended with quantum data, a quantum control structure, and a measurement operator. The vector notation $\boldsymbol{y}$

is used for sequence variables to be measured (weakened). The grammar of QML terms is defined thus:

$$\begin{array}{llcl}
(\textit{Variables}) & x, y, \ldots & \in & \textit{Vars} \\
(\textit{Prob amplitudes}) & \kappa, \iota, \ldots & \in & \mathbb{C} \\
(\textit{Patterns}) & p, q & ::= & x \mid (x, y) \\
(\textit{Terms}) & t, u & ::= & x \mid x^{y} \mid () \mid (t, u) \mid \textbf{let } p = t \textbf{ in } u \\
 & & \mid & \textbf{if } t \textbf{ then } u \textbf{ else } u' \mid \textbf{if}^{\circ}\ t \textbf{ then } u \textbf{ else } u' \\
 & & \mid & \textbf{qfalse}^{y} \mid \textbf{qtrue}^{y} \mid \kappa \times t \mid t + u
\end{array}$$

Quantum data is modelled using the constructs $\kappa \times t$ and $t + u$. The term $\kappa \times t$, associates the probability amplitude $\kappa$ with the term $t$. The term $t + u$ describes a quantum superposition of $t$ and $u$. Quantum superpositions are first class values, and can be used in conditional expressions to provide quantum control. For example: $\textbf{if}^{\circ}$ (**qtrue** + **qfalse**) **then** $t$ **else** $u$ evaluates both $t$ and $u$ and combines the results in a quantum superposition. Note that the term $\lambda_0 \times t + \lambda_1 \times u$, where $t, u$ are not qubits, is syntactic-sugar for $\textbf{if}^{\circ}$ ($\lambda_0 \times$ **qtrue** + $\lambda_1 \times$ **qfalse**) **then** $t$ **else** $u$. The type-checker and orthogonality judgements ensure that this is a valid operation, by providing a proof that $t$ and $u$ are orthogonal (distinguishable in some way), that their types match, and that they are strict terms (they produce no garbage).

In a quantum control operation, the two branches must be orthogonal, otherwise the type system would accept terms that implicitly perform measurements. Without this restriction "valid" programs could be written in QML that are not physically realisable by a quantum computer. Orthogonality judgements are inferred automatically by static analysis of terms (see section 2.1).

As an example of superposition formation, the term $(\frac{1}{\sqrt{2}}) \times$ **qfalse** + $(\frac{1}{\sqrt{2}}) \times$ **qtrue** is an equal superposition of **qfalse** and **qtrue**. Normalisation factors that are equal may be omitted.

Finally, a QML program is a sequence of function definitions, where a function definition is given by $f\ \Gamma = t : \tau$. A Haskell-style syntax is used to present program examples. For example, the QML function below (left) is equivalent to the following Haskell-like code (right):

$$\begin{array}{ll}
f\ (x_1 : \sigma_1, x_2 : \sigma_2, .., x_n : \sigma_n) & \qquad f : \sigma_1 \multimap \sigma_2 .. \multimap \sigma_n \multimap \tau \\
\qquad = t : \tau & \qquad f\ x_1\ x_2 .. x_n = t
\end{array}$$

## 2.1 QML orthogonality judgements

QML has a basic type system that tracks the use of variables, preventing them from being weakened inappropriately. However, the type system still accepts terms which implicitly perform measurements. As a consequence QML would accept programs which are not realisable as quantum computations.

Consider the expression $\textbf{if}^{\circ}$ $x$ **then qtrue else qtrue**. This expression returns **qtrue** without using any information about $x$. In order to maintain the invariant that all measurements are explicit, the type system should reject the above expression. More precisely, the expression $\textbf{if}^{\circ}$ $x$ **then** $t$ **else** $u$ should only be accepted if $t \perp u$. This notion intuitively ensures that the conditional operator does not implicitly discard any information about $x$ during the evaluation. The branches of a superposition should also be orthogonal for similar reasons.

Mathematically, two terms, $t, u$, are orthogonal if their inner-product is equal to zero, $\langle t|u\rangle = 0$. If this is the case then the judgement $t \perp u$ is true, but if the inner-product yields any other value then $t$ is not orthogonal to $u$. In the presentation of an equational theory for QML [2] the orthogonality judgements are replaced by an inner-product judgement on terms, to much the same effect. However, the inner-product approach is more informative and flexible, and gives a method of reasoning about orthogonality, hence in future this method may be adopted for all terms.

The following rules give the current QML orthogonality judgements:

$$\frac{}{\mathbf{qtrue} \perp \mathbf{qfalse}} \qquad \frac{}{\mathbf{qfalse} \perp \mathbf{qtrue}}$$

$$\frac{t \perp u}{(t, v) \perp (u, w)} \perp \mathbf{pair}_0 \qquad \frac{t \perp u}{(v, t) \perp (w, u)} \perp \mathbf{pair}_1$$

$$\frac{t \perp u \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\lambda_0 \times t + \lambda_1 \times u \perp \kappa_0 \times t + \kappa_1 \times u} \perp \mathbf{sup}$$

$$\frac{t \perp u \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\mathbf{if}^\circ\ (\lambda_0 \times \mathbf{qtrue} + \lambda_1 \times \mathbf{qfalse})\ \mathbf{then}\ t\ \mathbf{else}\ u \perp \mathbf{if}^\circ\ (\kappa_0 \times \mathbf{qtrue} + \kappa_1 \times \mathbf{qfalse})\ \mathbf{then}\ t\ \mathbf{else}\ u} \perp \mathbf{supif}^\circ$$

$$\frac{t \perp u \qquad t \perp u'}{t \perp \mathbf{if}^\circ\ c\ \mathbf{then}\ u\ \mathbf{else}\ u'} \perp \mathbf{if}^\circ_0 \qquad \frac{t \perp u \qquad t \perp u'}{\mathbf{if}^\circ\ c\ \mathbf{then}\ u\ \mathbf{else}\ u' \perp t} \perp \mathbf{if}^\circ_1$$

The first two axioms state that the basic states of **qtrue** and **qfalse** are orthogonal. The third and fourth rules state that pairs of terms can be orthogonal, provided that one component of a pair is orthogonal to the other pair's corresponding component. The two **sup** rules state when superpositions of terms can be orthogonal; the second is a restatement of the first, translating superpositions using $\mathbf{if}^\circ$. The final two rules state that $\mathbf{if}^\circ$ statements can be orthogonal if all the component terms are.

The use of $\mathbf{if}^\circ$ in QML programs is valid only if the two branches are orthogonal. Hence, for the Hadamard operation (section 4), it is required that $-\mathbf{qtrue}+\mathbf{qfalse} \perp \mathbf{qtrue} + \mathbf{qfalse}$, with the appropriate renormalisation. In this case the $\perp$ **sup** rule verifies orthogonality.

The rules for orthogonality given so far are incomplete, and may be extended. Orthogonality judgements must also be interpreted by the operational semantics, discussed in ref [5].

## *2.2 Removing coproducts from QML*

In a previous version of QML [1], here referred to as QML$^\oplus$, the language included the notion of a tensorial coproduct, denoted by $\oplus$. This coproduct has now been removed. The types of QML$^\oplus$ were generated by $\mathcal{Q}_\mathbf{1}$, $\sigma \otimes \tau$, and $\sigma \oplus \tau$. Qubits were not primitive, but defined as $\mathcal{Q}_\mathbf{2} = \mathcal{Q}_\mathbf{1} \oplus \mathcal{Q}_\mathbf{1}$. The coproduct allowed any finite type to be directly represented in this way; not just limited to $\mathcal{Q}_{\mathbf{2}}$. The introduction rules used for $\oplus$ were the usual coproduct rules of a left and a right injection:

$$\frac{\Gamma \vdash^a s : \sigma}{\Gamma \vdash^a \mathrm{inl}\ s : \sigma \oplus \tau}\ {+}\mathrm{intro_l} \qquad \frac{\Gamma \vdash^a t : \tau}{\Gamma \vdash^a \mathrm{inr}\ t : \sigma \oplus \tau}\ {+}\mathrm{intro_r}$$

The coproduct type was interpreted as $\sigma \oplus \tau = \mathcal{Q}_{\mathbf{2}} \otimes |\sigma \sqcup \tau|$, where $|\sigma \sqcup \tau|$ could store a value of either $|\sigma|$ or $|\tau|$, by padding the smaller type. Using the coproduct and injection rules, **qfalse** and **qtrue** were defined in QML$^{\oplus}$ as inl() : $\mathcal{Q}_2$ and **qfalse** = inr() : $\mathcal{Q}_2$, respectively, omitting the weakening property of QML$^{\oplus}$.

Instead of **if** and **if**$^{\circ}$ rules, QML$^{\oplus}$ implemented two $\oplus$-elimination rules: **case**, providing classical-control (a generalisation of **if**), and a quantum-control operation **case**$^{\circ}$ (generalising **if**$^{\circ}$). The quantum (non-measuring) $\oplus$-elimination rule is similar to the standard coproduct elimination rule, and is given as:

$$\frac{\begin{array}{l}\Gamma \vdash^a c : \sigma \oplus \tau \\ \Delta,\ x : \sigma \vdash^{\circ} t : \rho \\ \Delta,\ y : \tau \vdash^{\circ} u : \rho \quad\ t \perp u\end{array}}{\Gamma \otimes \Delta \vdash^a \mathbf{case}^{\circ}\ c\ \mathbf{of}\{\mathbf{inl}\ x \Rightarrow t \mid \mathbf{inr}\ y \Rightarrow u\} : \rho}\ \oplus\mathrm{elim}^{\circ}$$

The non-strict (measuring) case removes the orthogonality requirement, and does not require sub-terms to be strict. The **if**$^a$ rules ($a \in \{\circ, -\}$; if $a = \circ$ then the rules are strict, *i.e.* measurement-free) would then be derived as:

$$\mathbf{if}^a\ b\ \mathbf{then}\ t\ \mathbf{else}\ u = \mathbf{case}^a\ b\ \mathbf{of}\ \{\mathbf{inl}\ \_ \Rightarrow t \mid \mathbf{inr}\ \_ \Rightarrow u\}$$

The branches of a **case**$^{\circ}$ operation can be of different sizes, and this was dealt with in the semantics of QML$^{\oplus}$ by padding the type of the smaller branch. The padding of one type in this way can lead to the garbage becoming entangled with the useful output in some way. This happens, for example, if branching over $\mathcal{Q}_{\mathbf{1}} \otimes \mathcal{Q}_{\mathbf{2}}$. The garbage created by padding may indirectly measure the qubit which is being branched over. Consequently, this approach is not compositional, and has therefore been rejected.

This version of QML resolves this issue by removing the coproduct. Qubits are now primitive, as are **if** and **if**$^{\circ}$. Additionally, the strict **if**$^{\circ}$ does not allow any garbage to be produced. Coproducts may be reintroduced, possibly limited to classical types.

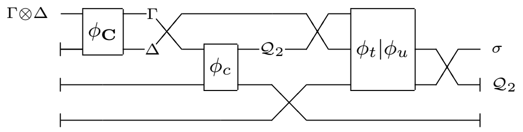
## 3 An operational semantics for QML

The new operational semantics for QML is briefly discussed here, which is an updated version of that presented in [1]. This semantics is defined by giving a translation from QML terms into morphisms in the category **FQC**. FQC morphisms consist of a reversible quantum circuit $\phi$, the input context $\Gamma$, the output type $\sigma$, and the size of the auxiliary heap $h$ and any garbage $g$. Any heap qubits are initially set to 0 (*false*) in the computational basis, and garbage qubits can be removed by the partial trace operation at the end of the computation. A full development of **FQC** is given in references [5,6].

As an example, the classical **if** construct is defined by the following typing rule and operational semantic:

$$\frac{\begin{array}{c}\Gamma \vdash c : \mathcal{Q}_2 \\ \Delta \vdash t, u : \sigma\end{array}}{\Gamma \otimes \Delta \vdash \mathbf{if}\ c\ \mathbf{then}\ t\ \mathbf{else}\ u : \sigma}\ \mathbf{if} \qquad \frac{\begin{array}{c}\mathbf{c} \in \mathbf{FQC}\ \Gamma\ \mathcal{Q}_\mathbf{2} \\ \mathbf{t} \in \mathbf{FQC}\ \Delta\ \sigma \\ \mathbf{u} \in \mathbf{FQC}\ \Delta\ \sigma\end{array}}{\begin{array}{l}\textsc{if}_{\mathrm{Op}}\ \mathbf{c\ t\ u} \in \mathbf{FQC}\ (\Gamma \otimes \Delta)\ \sigma \\ \textsc{if}_{\mathrm{Op}}\ \mathbf{c\ t\ u} = (h_\mathbf{C} + h_\mathbf{c} + h_{\mathbf{t}|\mathbf{u}}, g_\mathbf{c} + 1, \phi)\end{array}}$$

where $\phi$ in the operational semantics is the following circuit:



with $\phi_\mathbf{C}$ as a "context-splitting" operation that copies any variables used by both subterms. $\phi_t|\phi_u$ denotes a conditional circuit which performs $\phi_t$ if the result of the conditional circuit $c$ is true ($|1\rangle$) and $\phi_u$ if it is false ($|0\rangle$).

In this way, the semantics of QML is defined recursively over the syntax of terms, and results in a valid FQC morphism for a valid QML term. Given a QML term, the QML compiler follows the operational semantics and outputs an FQC morphism represented as a *typed circuit*, which can then either be displayed, exported for use with other programs, or used to directly implement the program on a quantum-circuit based quantum computer. The FQC morphism can be further evaluated via the denotational semantics, producing a unitary matrix $\mathbf{Q}^{\simeq}$ (for strict-QML terms without heap), an isometry $\mathbf{Q}^{\circ}$ (for strict-QML terms), or a superoperator $\mathbf{Q}$ (for QML terms that produce garbage). In addition, orthogonality judgements and circuits for quantum control and superpositions of terms are automatically inferred by the compiler at compile-time, so there can be no orthogonality errors during run-time. The relationships between QML without measurement ($\mathbf{QML}^{\circ}$), full QML ($\mathbf{QML}$), typed quantum circuits ($\mathbf{FQC}$), and isometries ($\mathbf{Q}^{\circ}$) and superoperators ($\mathbf{Q}$), is shown in the following diagram:

$$\begin{tikzcd}
& \mathbf{QML}^{\circ}\ \Gamma\ \sigma \arrow[r, hook] \arrow[dl, "{[\![\cdot]\!]^{\circ}_{\mathrm{Op}}}"'] & \mathbf{QML}\ \Gamma\ \sigma \arrow[dr, "{[\![\cdot]\!]_{\mathrm{Op}}}"] & \\
\mathbf{FQC}^{\circ}\ \Gamma\ \sigma \arrow[rrr, hook] \arrow[dr, "{[\![\cdot]\!]_d}"'] & & & \mathbf{FQC}\ \Gamma\ \sigma \arrow[dl, "{[\![\cdot]\!]_d}"] \\
& \mathbf{Q}^{\circ}\ \Gamma\ \sigma \arrow[r, tail, "\widehat{\cdot}"] & \mathbf{Q}\ \Gamma\ \sigma &
\end{tikzcd}$$

where $[\![\cdot]\!]_{\mathrm{Op}}$ denotes the operational semantics, and $[\![\cdot]\!]_{\mathrm{D}}$ denotes the additional translation from quantum circuits into the denotational semantics. The full semantics can be found in reference [5].

## 4 Example: Teleportation in QML

The quantum teleportation describes how to transport a quantum state using a small amount of classical communication. A QML interpretation of the teleportation circuit with deferred measurement has previously been presented, along with a full description of the algorithm [5]. However, this circuit relies on the existence of a quantum channel. In order to demonstrate and explain the syntax of QML and the compiler, a new, faithful, implementation of the quantum teleport algorithm, which explicitly makes use of measurement, is presented.[2] This algorithm is similar to that developed by Selinger and Valiron [9], and also in reference [3], which includes a relevant discussion of teleportation, both with and without deferred measurement. These examples show the elegance of allowing quantum control via the quantum **if**° construct (*CNot*) and term superpositions (*Epr*), in writing functions. For simple one qubit functions (*Had*), the branches of the quantum control are the columns from the unitary matrices that describe the operation. A simple measurement operation, using classical control, is also defined in the following example which implements the teleportation algorithm (*Tele*):

```
Had, Qnot, Meas ∈ Q2 ⊸ Q2
Had b  = if° b  then qfalse + − qtrue    -- The Hadamard operation
                else  qfalse + qtrue
Qnot b = if° b then qfalse else qtrue    -- The Not operation (Pauli X)
Meas b = if b then qtrue else qfalse     -- A measurement operator, using if
CNot     ∈ Q2 ⊸ Q2 ⊸ Q2 ⊗ Q2             -- A quantum CNOT operation
CNot s t = if° s then (qtrue, Qnot t)
                 else  (qfalse, t)
Epr ∈ Q2 ⊗ Q2                            -- The EPR pair, |00⟩ + |11⟩
Epr = (qtrue, qtrue) + (qfalse, qfalse)
Bmeas     ∈ Q2 ⊸ Q2 ⊸ Q2 ⊗ Q2            -- The Bell-measurement operation
Bmeas x y = let (x′, y′) = CNot x y
            in (Meas (Had x′), Meas y′)
U      ∈ Q2 ⊸ Q2 ⊗ Q2 ⊸ Q2               -- The unitary correction operations
U q xy =  let (x, y) = xy in if x then   (if y then $U_{11}$ q else $U_{10}$ q)
                                  else   (if y then $U_{01}$ q else q)
$U_{01}$, $U_{10}$, $U_{11}$ ∈ Q2 ⊸ Q2
$U_{01}$ x = if° x then   qfalse else qtrue
$U_{10}$ x = if° x then − qtrue  else qfalse
$U_{11}$ x = if° x then − qfalse else qtrue
  -- The quantum teleportation algorithm
Tele    ∈ Q2 ⊸ Q2
Tele q  =  let (a, b)    = Epr        -- a is given to Alice, b to Bob
                f        = Bmeas q a  -- Result of Alice's Bell-measurement is classical data
            in U b f                  -- Bob applies U to his qubit, using classical data f
```

## 5 The QML compiler

This section briefly describes the design and operation of the Haskell QML compiler, which can be found on the project website.[3] The objective is to take a file containing a QML program, consisting of QML function definitions, and output an annotated (typed) quantum circuit which realises the QML program as an **FQC** object. This is achieved by implementing the operational semantics in Haskell. Additionally, the circuit produced by the compiler can be further processed to produce either a unitary matrix ($\mathbf{Q}^{\simeq}$), isometry ($\mathbf{Q}^{\circ}$), or superoperator ($\mathbf{Q}$), as appropriate, giving an implementation of the denotational semantics of QML, as factored through the operational semantics.

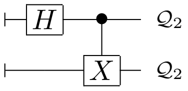
The compiler has a modular design, giving a clear logical structure. For example,

[2] The code is included with the compiler as `teleport.qml`, see section 5.

[3] Instructions for downloading the QML compiler can be found on the QML compiler website `http://fop.cs.nott.ac.uk/qml/compiler`. The Haskell compiler GHC is also required. To use the compiler, the QML system must be loaded into GHC via the command "*ghci qml*". This will initialise the system with all the modules required to interpret and evaluate QML programs.

the *QOrth* module contains all the code for generating the orthogonality judgements and circuits, while *QCirc* contains the definition of the circuit datatypes and associated functions. The compiler exploits advanced Haskell features, such as monads and pattern matching, and making use of the ideas put forward by Vizzotto *et al* [10]. The operational semantics is realised in the *QComp* (*Q*ML *Comp*ilation) module.

To compile QML functions into the operational semantics (**FQC** morphisms, represented as typed circuits), the "run typed circuit" command is used: *runTC* `"#filename" "#function"`. For example, to evaluate the typed circuit of the function *Epr* (from section 4) the command is *runTC* `"teleport.qml" "Epr"`. This outputs the following typed circuit (as a Haskell datatype):



where there is no input context, the heap is two qubits (marked $\vdash$), and a pair of qubits are the output. $H$ and $X$ are the Hadamard and Pauli-$X$ operations. As heap is initialised to **qfalse**, this circuit describes the function $|00\rangle \rightarrow |00\rangle + |11\rangle$, producing an EPR pair. The compilation of a QML term into a circuit is efficient; each QML term is recursively translated directly into an FQC morphism (an annotated circuit), as described in section 3. No quantum computation is simulated, so this does not effect the efficiency of the compiler - it is a simple recursive translation into circuits. The output is further optimised after compilation, by collapsing identities and permutations and other simple circuit manipulations which do not effect the action of the circuit (see *QCirc*).

There are three main options for further evaluation of a QML term:

- The QML term could be evaluated to a unitary matrix ($\mathbf{Q}^{\simeq}$), interpreting only the reversible circuit $\phi$ from the **FQC** morphism. As this option classically computes the full mathematical interpretation of the program it is inherently inefficient. The output is actually a triple $(h, g, \phi) \in (Int, Int, \text{Unit})$, where $h, g$ gives the size of any heap required or garbage produced. The command for producing a unitary matrix is *runM*;
- The function could be evaluated to an isometry ($\mathbf{Q}^{\circ}$), which initialises any required heap, and is the full description for terms that produce no garbage. This option is no less efficient than the *runM* option, and is the preferred evaluation option. The output is actually a pair $(g, \phi) \in (Int, \mathbf{Q}^{\circ})$, where $g$ gives the size of any garbage (if the QML function is impure). The command for evaluating to an isometry is *runI*;
- Thirdly, the QML term can be interpreted as a superoperator ($\mathbf{Q}$), which initialises heap and traces out any garbage, using the command *runS*. This option is substantially less efficient than the previous two options, as the state space is doubled and the partial trace is a computationally expensive operation.

Together, the options *runI* and *runS* give an interpretation of the denotational semantics of QML factored through the category **FQC**, as shown in the diagram in section 3. A direct implementation of the denotational semantics, without using the operational semantics, is an extension currently being developed. Please refer to the project website for full details.

## 6 Conclusions and further work

This paper introduces the language QML and presents its semantics with a compiler and example programs. The semantics and compiler give a realisable interpretation of QML programs as quantum circuits in a formal, categorical, setting. The semantics can be extended in many ways, such as expanding the current orthogonality judgements, or by the addition of non-linear, classical, data. The algebra for the pure fragment of QML [2] is currently being extended to include measurement, following work on van Tonder's quantum lambda calculus [3]. It would be instructive to implement this algebra, especially the normal form, as part of the QML compiler. Future possibilities for the development of the language also including developing a notion of higher-order functions for QML, and adding iteration to the language.

The development of QML and the compiler is an ongoing project which has already reached a functional state. As the language and semantics evolve, extensions and new features can be incorporated into the compiler; which also provides a useful testbed for the development of new language features and capabilities. For example, an extension of the orthogonality circuits given in [5] was developed using the compiler in this way. The compiler also facilitates the testing and development of new QML algorithms, such as the described teleportation algorithm. It has also been useful in allowing others to experiment with quantum programming and get immediate feedback on the behaviour of their functions, in a style that is familiar to computer scientists, logicians, and physicists with functional programming experience.

Further extensions to the compiler include adding the ability to export typed circuits as images, or in notation compatible with tools such as MatLab and Mathematica. Possible relationships with the measurement calculus, the Haskell QIO monad [7], and other formalisms are being studied, and may provide new insights. This will lead to new features being developed, such as basis independence, and further useful abstractions.

## References

[1] Altenkirch, T. and J. Grattage, *A functional quantum programming language*, in: *LICS 2005 proceedings* (2005), pp. 249–258, also arXiv:quant-ph/0409065.
URL `http://fop.cs.nott.ac.uk/qml`

[2] Altenkirch, T., J. Grattage, J. K. Vizzotto and A. Sabry, *An algebra of pure quantum programming*, ENTCS **170** (2007), pp. 23–47, also arXiv:quant-ph/0506012v1.
URL `http://fop.cs.nott.ac.uk/qml`

[3] Díaz-Caro, A., P. Arrighi, M. Gadella and J. Grattage, *Measurements and confluence in quantum lambda calculi with explicit qubits* (2008), accepted by DCM/QPL (ICALP 2008).
URL `http://equipes-lig.imag.fr/capp/qcg/people/jgrattage/papers/adc-lambdameas-qpl.pdf`

[4] Gay, S. J., *Quantum programming languages: Survey and bibliography*, Mathematical Structures in Computer Science **16** (2006).
URL http://www.dcs.gla.ac.uk/~simon/publications/QPLsurvey.pdf

[5] Grattage, J., "QML: A functional quantum programming language," Ph.D. thesis, School of Computer Science and School of Mathematical Physics, The University of Nottingham (2006).
URL http://etheses.nottingham.ac.uk/archive/00000250/

[6] Green, A. and T. Altenkirch, *From reversible to irreversible computations*, in: P. Selinger, editor, *QPL 2006 proceedings*, ENTCS (2006).
URL http://fop.cs.nott.ac.uk/qml

[7] Green, A. and T. Altenkirch, *Shor in Haskell: The quantum IO monad* (2008), accepted by Trends in Functional Programming 2008.
URL http://www.cs.nott.ac.uk/~asg/pdfs/tfp08.pdf

[8] Selinger, P., *Towards a Quantum Programming Language*, Mathematical Structures in Computer Science **14** (2004), pp. 527–586.
URL http://www.mathstat.dal.ca/~selinger/papers.html#qpl

[9] Selinger, P. and B. Valiron, *A lambda calculus for quantum computation with classical control*, in: *TLCA 2005 proceedings*, LNCS **3461** (2005).
URL http://www.mathstat.dal.ca/~selinger/papers.html#qlambda

[10] Vizzotto, J. K., T. Altenkirch and A. Sabry, *Structuring Quantum Effects: Superoperators as Arrows* (2005), also arXiv:quant-ph/0501151.