QML: Quantum data and control

Thorsten Altenkirch and Jonathan Grattage

February 17, 2005

Abstract

We introduce the language QML, a functional language for quantum computations on finite types. QML introduces quantum data and control structures, and integrates reversible and irreversible quantum computation. QML is based on strict linear logic, hence weakenings, which may lead to decoherence, have to be explicit. We present an operational semantics of QML programs using quantum circuits, and a denotational semantics using superoperators.

1 Introduction

Quantum programming is now a firmly established field, as we can see from the availability of introductory text books such as [Gru99, NC00, Pit00, Hir01] and, not to forget, Preskill's excellent online notes [Pre99]. However, quantum programs are usually presented in a semi-formal style and on a very low level, usually as families of quantum circuits. We believe that high level quantum programming languages can improve the presentation, further our understanding of the power of quantum computing, and lead to new applications — as they have done in conventional programming.

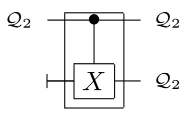
One of the first proposals towards a quantum programming language were Knill's conventions for quantum pseudo code [Kni96]. More recently, Ömer implemented an imperative language QCL with quantum primitives [Öme03]. Sanders and Zuliani [SZ00] proposed qGCL, which extends the probabilistic guarded command language by quantum primitives. A very promising venue is the integration of quantum programming with functional programming, [MB01], [Kar03] and [Sab03]. Recently, in joint work with the first author, Vizotto and Sabry [VAS04] have shown that quantum programming can be modelled using Haskell's arrow library [Hug00], presenting a high level, but constructive, view of quantum effects.

Andre van Tonder has proposed a quantum λ-calculus incorporating higher order programs [vT03a, vT03b], however he does not consider measurements as part of his language. In [vT03b] he suggests a semantics for a finitary, but higher order, calculus, based on Hilbert bundles. However, at least for the moment it is not clear how his calculus could be realized operationally, e.g. using quantum circuits.

Peter Selinger's influential paper [Sel04] introduces a single-assignment (essentially functional) quantum programming language, which is based on the separation of *classical control* and *quantum data*. This language combines high-level classical structures with operations on quantum data, and has a clear mathematical semantics in the form of superoperators. Quantum data can be manipulated by using unitary operators or by measurement, which can affect the classical control flow. Recently, Selinger and Valiron [SV05] have presented a functional language based on the *classical control* and *quantum data* paradigm.

None of the approaches discussed so far introduces quantum control structures, i.e. quantum data can only be processed using combinators corresponding to quantum circuits or by measurement. In contrast, QML, which was first introduced in [AG04], features both quantum data structures, using the connectives $\otimes$ and $\oplus$, and quantum control structures, in particular a quantum case construct $\mathbf{case}^{\circ}$ – which analyses quantum data without measuring, and hence without changing the data.

QML's type system is based on *strict linear logic*, that is linear logic with contraction, but without implicit weakening. This is in contrast to Selinger and Valiron [SV05], whose language has an affine type system, i.e. without implicit contraction. The absence of contraction in their language is motivated by the no-cloning property of quantum states. QML's type system allows implicit contraction, this is possible since contraction is modelled, as in classical functional programming, by sharing and not by cloning. Indeed, on the level of reversible circuits, classical or quantum, sharing can be realized using a conditional not (CNOT) circuit:



ϕ_δ

with the 2^{nd} input to the gate initialised with $|0\rangle$. The circuit is, as expected, the diagonal for classical states; i.e. it maps $|0\rangle$ to $|00\rangle$ and $|1\rangle$ to $|11\rangle$. It doesn't clone quantum states like $|0\rangle + |1\rangle$ [1] , but shares them: the circuit would output $|00\rangle + |11\rangle$ and not $(|0\rangle + |1\rangle)(|0\rangle + |1\rangle)$, which would correspond to cloning. The contraction as sharing interpretation been independently suggested in [**?**].

In our view it is not contraction which has to be policed, but weakening. The reason is that we cannot forget a quantum bit without measuring it first. This may affect other parts of the computation, for example it will change qbits which are entangled with the qbit we want to dispose of.

As an example, consider the following simple program which should swap two qbits:

$$swap \in \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2}$$
$$swap\ p = \mathbf{let}\ (x, y) = p$$

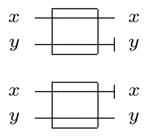
[1] To be precise, we mean $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, however, we avoid the clutter since the necessary factors can be easily inferred by the reader.

in (y, x)

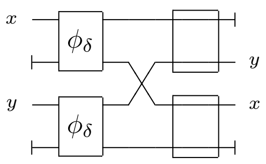
If we use a conventional type system, with rules such as

$$\frac{\Gamma \vdash t : \sigma \quad \Gamma \vdash u : \tau}{\Gamma \vdash (t, u) : \sigma \otimes \tau} \qquad \frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

where the variables x and y are interpreted by projections:



we end up interpreting *swap* by the following circuit:

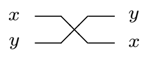


Indeed, this is essentially how a conventional functional language is implemented; using the stack as a temporary and easily disposable data storage. However, in a quantum setting this implementation doesn't give the desired effect. While base states are properly swapped, i.e. $|01\rangle$ is mapped to $|10\rangle$, a superposition like $(|0\rangle - |1\rangle)(|0\rangle + |1\rangle)$, is mapped to one of $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ with equal probability.

Hence, in QML, we use a strict linear type system, with the rules:

$$\frac{\Gamma \vdash^\circ t : \sigma \quad \Delta \vdash^\circ u : \tau}{\Gamma \otimes \Delta \vdash^\circ (t, u) : \sigma \otimes \tau} \qquad \frac{}{x : \sigma \vdash x : \sigma}$$

where $\Gamma \otimes \Delta$ is an operation which allows us to split the context. As a consequence, we interpret *swap* by



which behaves as we would expect: $(|0\rangle - |1\rangle)(|0\rangle + |1\rangle)$, is mapped to $(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$.

However, weakening is not the only source of decoherence. How do we interpret the following definition of negation?

$mnot \in \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$

$mnot\ x =$ **if** x **then** qfalse **else** qtrue

If we follow the classcial control paradigm then branching over a qbit entails measuring it. As a consequence the interpretation of *mnot* doesn't work as expected on superpositions. For example, the input $|0\rangle - |1\rangle$ gets mapped to

$|0\rangle$ or $|1\rangle$ with equal probability. A proper quantum negation operator should return $|1\rangle - |0\rangle$. Indeed, in QML we can implement this behaviour by

$$qnot \in \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}}$$

$$qnot\ x = \mathbf{if}^{\circ}\ x\ \mathbf{then}\ \text{qfalse}\ \mathbf{else}\ \text{qtrue}$$

if$^{\circ}$ is a quantum control structure; it allows us to analyse a quantum bit without measuring it. However, we cannot always replace **if** by **if**$^{\circ}$. Consider the conditional swap program:

$$cswap \in \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}}$$

$$\begin{aligned} cswap\ c\ p = {} & \mathbf{if}\ c \\ & \mathbf{then}\ swap\ p \\ & \mathbf{else}\ \ p \end{aligned}$$

If both components of p are the same then *cswap* in effect forgets the control qbit. However, forgetting is not possible without measuring and hence we cannot replace **if** by **if**$^{\circ}$. The only way to avoid this is to include the control qbit in the output:

$$cswap \in \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \multimap \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}} \otimes \mathcal{Q}_{\mathbf{2}}$$

$$\begin{aligned} cswap\ c\ p = {} & \mathbf{if}^{\circ}\ c \\ & \mathbf{then}\ (\text{qtrue}, swap\ p) \\ & \mathbf{else}\ \ (\text{qfalse}, p) \end{aligned}$$

The design of QML is based on these considerations: we use a strict linear logic with an explicit weakening operator, but implicit contractions. This is justified by the fact that the meaning of a program can be affected by changing the weakenings, but not by moving contractions. We also introduce two case operators: **case** which measures a qbit; and **case**$^{\circ}$ which doesn't, but which requires that its branches are orthogonal, reflected by introducing an orthogonality judgement on QML terms.

1.1 Structure of the paper

We begin by introducing QML's syntax and typing rules, based on strict linear logic, in section 2 and present some small example programs: a variant of Deutsch's algorithm, and a formalisation of quantum teleportation. The operational semantics of QML is presented in section 3 by translating QML into quantum circuits, which for the purposes of the operational semantics we consider as black boxes. In section 4 we present a denotational semantics of quantum circuits and QML programs by interpreting the operational semantics by superoperators. We close with conclusions and indicate areas for further work (section 5).

2 Syntax and typing rules

We introduce the syntax and typing rules of QML based on strict linear logic: contractions are implicit, while weakenings are an explicit operation which correspond to measurements. QML's types are first order and finite. There are no recursive types, so, for example, we cannot represent a type of quantum natural numbers. How to overcome those limitations is discussed in section 5.

QML's type constructors are the tensor product, $\otimes$, which plays the rule of a product type, and a tensorial coproduct, $\oplus$, which corresponds to a disjoint union in conventional programming. Qbits are not primitive, but are definable as $\mathcal{Q}_\mathbf{2} = \mathcal{Q}_\mathbf{1} \oplus \mathcal{Q}_\mathbf{1}$, using the coproduct. QML has two case constructs: **case** which measures a qbit in the data it analyses; and **case**$^\circ$, which doesn't measure, but requires that the results will always live in orthogonal subspaces. The proofs of orthogonality can be inserted automatically by the compiler, and hence don't feature in the syntax of terms.

We use σ, τ, ρ to vary over QML types which are given by the following grammar:

$$\sigma = \mathcal{Q}_\mathbf{1} \mid \sigma \otimes \tau \mid \sigma \oplus \tau$$

We assume an infinite supply of concrete variable names, and use x, y, z to vary over names. Typing contexts (Γ, Δ) are given by

$$\Gamma = \bullet \mid \Gamma, x : \sigma$$

where $\bullet$ stands for the empty context, but is omitted if the context is non-empty. For simplicity we assume that every variable appears at most once. Contexts correspond to functions from a finite set of variables to types.

To define the syntax of expressions we also use complex number constants $\kappa, \iota \in \mathbb{C}$ and function variables to refer to a previously defined QML program:

$$\begin{array}{ll} t = x & \mid \mathbf{let}\ x = t\ \mathbf{in}\ u \\ \quad \mid x^{\vec{y}} & \mid () \\ \quad \mid (t, u) & \mid \mathbf{let}\ (x, y) = t\ \mathbf{in}\ u \\ \quad \mid \mathrm{qinl}\ t \mid \mathrm{qinr}\ u & \\ \quad \mid \mathbf{case}\ \ t\ \mathbf{of}\ \{\mathrm{qinl}\ x \Rightarrow u \mid \mathrm{qinr}\ y \Rightarrow u'\} & \\ \quad \mid \mathbf{case}^\circ\ t\ \mathbf{of}\ \{\mathrm{qinl}\ x \Rightarrow u \mid \mathrm{qinr}\ y \Rightarrow u'\} & \\ \quad \mid \{(\kappa)\ t \mid (\iota)\ u\} & \\ \quad \mid f\ \vec{t} & \end{array}$$

Here, the vector notation $\vec{a}$ is used for sequences of syntactic objects. Formally, it corresponds to the following meta notation:

$$\vec{a} = \epsilon \mid a\ \vec{a}$$

A QML program is a sequence of function definitions $\vec{d}$, where a function definition d is given by $f\ \Gamma = t : \tau$. However, we shall use a Haskell style syntax to present program examples, using $\multimap$ instead of $\to$ in the definition. That is we write

$$f : \sigma_1 \multimap \sigma_2 \cdots \multimap \sigma_n \multimap \tau$$
$$f\ x_1\ x_2 \ldots x_n = t$$

for

$$f\ (x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n) = t : \tau$$

Our basic typing judgements are

Typing of terms

$$\vec{d}; \Gamma \vdash t : \sigma$$

Typing of strict terms

$$\vec{d}; \Gamma \vdash^{\circ} t : \sigma$$

Orthogonality

$$t \perp u$$

Well-typed programs

$$\vdash \vec{d}$$

Since all of the rules just pass $\vec{d}$ we omit $\vec{d}$ from rule definitions, with the exception of the application rule (app), and just write $\Gamma \vdash t : \sigma$ (or $\Gamma \vdash^{\circ} t : \sigma$) instead.To avoid repetition, we also use the schematic judgements $\Gamma \vdash^{a} t : \sigma$ where $a \in \{-, \circ\}$.

For the additive rules, we introduce the operator $\otimes$, mapping pairs of contexts to contexts:

$$\begin{array}{lcll} \Gamma, x : \sigma \otimes \Delta, x : \sigma & = & (\Gamma \otimes \Delta), x : \sigma & \\ \Gamma, x : \sigma \otimes \Delta & = & (\Gamma \otimes \Delta), x : \sigma & \text{if } x \notin \operatorname{dom} \Delta \\ \bullet \otimes \Delta & = & \Delta & \end{array}$$

This operation is partial – it is only well-defined if the two contexts do not assign different types to the same variable. Whenever, we use this operator we omit the implicit assumption that it is well-defined.

2.1 Structural rules

We embed strict terms into terms by

$$\frac{\Gamma \vdash^{\circ} t : \sigma}{\Gamma \vdash t : \sigma}\,\text{emb}$$

The variable rule is strict and hence requires the context to contain only the variable used. We also introduce the explicit weakening rules, which is non-strict: a term can be marked by a set of variables over which it is weakened:

$$\frac{}{x : \sigma \vdash^{\circ} x : \sigma}\,\text{var} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \otimes \Delta \vdash t^{\operatorname{dom} \Delta} : \sigma}\,\text{weak}$$

$\operatorname{dom} \Delta$ is the set of variables defined in Δ, this corresponds to the functional reading of contexts. Combining the two rules we can derive a non-strict variable rule:

$$\frac{}{\Gamma, x : \sigma \vdash x^{\operatorname{dom} \Gamma} : \sigma}$$

However, having only this rule is not sufficient since we need to use non-atomic weakening when constructing **case**$^{\circ}$-expressions, because we cannot use weakenings in the strict branches.

Next, we introduce a let-rule which is also the basic vehicle to define first order programs.

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \\ \Delta,\, x : \sigma \vdash^a u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^a \texttt{let}\ x = t\ \texttt{in}\ u : \tau}\ \text{let}$$

By combining (let) and (emb) we can derive a more convenient scheme:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \\ \Delta,\, x : \sigma \vdash^b u : \tau\end{array}}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \texttt{let}\ x = t\ \texttt{in}\ u : \tau}$$

where $\circ \sqcap \circ = \circ$ and $-$ otherwise.
Weakenings can affect the meaning of a program. As an example consider:

$$y : \mathcal{Q}_2 \vdash \texttt{let}\ x = y\ \texttt{in}\ x^{\{\}} : \mathcal{Q}_2$$

This program will be interpreted as the identity circuit, in particular it is decoherence-free. However, consider

$$y : \mathcal{Q}_2 \vdash \texttt{let}\ x = y\ \texttt{in}\ x^{\{y\}} : \mathcal{Q}_2$$

This program uses weakening which will be interpreted as a measurement which causes decoherence.

2.2 Products ($\otimes$)

The rules for Q_1, $\otimes$ are the standard rules from linear logic. In the case of Q_1 instead of an explicit elimination rule we allow implicit weakening:

$$\frac{}{\bullet \vdash^{\circ} () : \mathcal{Q}_1}\ \mathrm{Q}_1 - \text{intro} \qquad \frac{\Gamma, x : \mathcal{Q}_1 \vdash^a t : \sigma}{\Gamma \vdash^a t : \sigma}\ \mathcal{Q}_1 - \text{weak}$$

Note that $\mathcal{Q}_1 -$ weak preserves strictness, because no actual piece of information is eliminated. Using weakening we can derive non-strict version of the introduction rule:

$$\frac{}{\Gamma \vdash ()^{\mathrm{dom}\,\Gamma} : \mathcal{Q}_1}$$

The product introduction rule is standard:

$$\frac{\Gamma \vdash^a t : \sigma \quad \Delta \vdash^a u : \tau}{\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau}\ \otimes - \text{intro}$$

The elimination rule is the usual pattern matching variant of the let-rule:

$$\frac{\begin{array}{l}\Gamma \vdash^a t : \sigma \otimes \tau \\ \Delta,\, x : \sigma, y : \tau \vdash^a u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^a \texttt{let}\ (x,y) = t\ \texttt{in}\ u : \rho}\ \otimes - \mathrm{elim}$$

As before for the let-rule, using (emb) we can derive

$$\frac{\begin{array}{l}\Gamma \vdash^a t : \sigma \otimes \tau \\ \Delta,\, x : \sigma, y : \tau \vdash^b u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \texttt{let}\ (x,y) = t\ \texttt{in}\ u : \rho}$$

As an example, here is *swap* again:

$$p : \mathcal{Q}_2 \otimes \mathcal{Q}_2 \vdash \texttt{let}\ \ (x,y) = p\ \texttt{in}\ (y^{\{\}}, x^{\{\}}) : \mathcal{Q}_2 \otimes \mathcal{Q}_2$$

It is important to mark the variables with the empty set of variables. The alternative program

$$p : \mathcal{Q}_2 \otimes \mathcal{Q}_2 \vdash \texttt{let}\ \ (x,y) = p\ \texttt{in}\ (y^{\{p\}}, x^{\{p\}}) : \mathcal{Q}_2 \otimes \mathcal{Q}_2$$

would measure the qbits while swapping them — this corresponds to the first version of *swap* in the introduction, based on the multiplicative rules.

2.3 Coproducts ($\oplus$)

The introduction rules for $\oplus$ are the usual classical rules for $+$; note that they preserve strictness.

$$\frac{\Gamma \vdash^a s : \sigma}{\Gamma \vdash^a \texttt{inl}\ \ s : \sigma \oplus \tau}\ \oplus\!-\!\mathrm{intro}_1 \qquad \frac{\Gamma \vdash^a t : \tau}{\Gamma \vdash^a \texttt{inr}\ \ t : \sigma \oplus \tau}\ \oplus\!-\!\mathrm{intro}_2$$

We define qbits $\mathcal{Q}_\mathbf{2} = \mathcal{Q}_\mathbf{1} \oplus \mathcal{Q}_\mathbf{1}$ and abbreviate $\mathrm{qtrue}^{\vec{x}} = \mathrm{qinl}\ ()^{\vec{x}}$ and $\mathrm{qfalse}^{\vec{x}} = \mathrm{qinr}\ ()^{\vec{x}}$. We are not restricted to $\mathcal{Q}_\mathbf{2}$ but can represent any finite type directly. As an example consider qtrits $\mathcal{Q}_\mathbf{3} = \mathcal{Q}_\mathbf{1} \oplus (\mathcal{Q}_\mathbf{1} \otimes \mathcal{Q}_\mathbf{1})$ whose constructors $0, 1, 2$ can be defined anagolously to qtrue and qfalse.

As already indicated, we have two different elimination rules. We begin with the one which measures a qbit, since it is basically the classical rule modulo additivity of contexts.

$$\frac{\begin{array}{l}\Gamma \vdash c : \sigma \oplus \tau \\ \Delta,\, x : \sigma \vdash t : \rho \\ \Delta,\, y : \tau \vdash u : \rho\end{array}}{\Gamma \otimes \Delta \vdash \texttt{case}\ \ c\ \texttt{of}\ \ \{\texttt{inl}\ x \Rightarrow t \mid \texttt{inr}\ y \Rightarrow u\} : \rho}\ \oplus\!-\!\mathrm{elim}$$

We can derive if-then-else as

$$\textbf{if } b \textbf{ then } t \textbf{ else } u = \textbf{case } b \textbf{ of } \{ \mathit{inl}\ \_ \Rightarrow t \mid \mathit{inr}\ \_ \Rightarrow u \}$$

and use this to implement a form of negation:

$\mathit{mnot} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$

$\mathit{mnot}\ x = \textbf{if } x \textbf{ then } \text{qfalse} \textbf{ else } \text{qtrue}$

However, this program will measure the qbit before negating it. If we want to avoid this we have to use the decoherence-free version of case, which relies on the orthogonality judgement: $t \perp u$, which is defined for terms in the same type and context $\Gamma \vdash t, u : A$. We will introduce the rules for orthogonality later. Intuitively, $t \perp u$ holds if the outputs t and u are always orthogonal, e.g. we will be able to derive $\texttt{qtrue}^{\{\}} \perp \texttt{qfalse}^{\{\}}$. Hence, we introduce the strict case by:

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

Using the decoherence-free version $\textbf{if}^\circ$ we can implement standard reversible and hence quantum operations such as *qnot*:

```
$\mathit{qnot} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$
$\mathit{qnot}\ x = \textbf{if}^\circ\ x$
            $\textbf{then}$ qfalse
            $\textbf{else}$  qtrue
```

and the conditional not *cnot*:

```
$\mathit{cnot} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2}$
$\mathit{cnot}\ c\ x = \textbf{if}^\circ\ c$
              $\textbf{then}$ (qtrue, $\mathit{qnot}\ x$)
              $\textbf{else}$  (qfalse, $x$)
```

and finally the Toffolli operator which is basically a conditional *cnot*:

```
$\mathit{toff} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2} \otimes (\mathcal{Q}_\mathbf{2} \otimes \mathcal{Q}_\mathbf{2})$
$\mathit{toff}\ \ c\ x\ y = \textbf{if}^\circ\ c$
                $\textbf{then}$ (qtrue, $\mathit{cnot}\ x\ y$)
                $\textbf{else}$  (qfalse, $(x, y)$)
```

2.4 Superpositions

To be able to exploit quantum parallelism we have to be able to create superpositions like $\{\text{qtrue} \mid \text{qfalse}\}$, which is actually a shorthand for $\{(\frac{1}{\sqrt{2}})\texttt{qtrue}|(\frac{1}{\sqrt{2}})\texttt{qfalse}\}$.

$$\frac{\begin{array}{l}\Gamma \vdash^\circ t, u : \sigma \qquad t \perp u \\ |\lambda|^2 + |\lambda'|^2 = 1\end{array}}{\Gamma \vdash^\circ \{(\lambda)t \mid (\lambda')u\} : \sigma} \ \text{sup}$$

As an example we can implement the Hadamard operator in QML:

$had \in \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$

$had\ x = \mathbf{if}^\circ\ x\ \mathbf{then}\ \{(-1)\ \mathrm{qtrue} \mid \mathrm{qfalse}\}$
$\qquad\qquad \mathbf{else}\ \ \{\mathrm{qtrue} \mid \mathrm{qfalse}\}$

As already indicated earlier we omit the normalisation factors, which can be inferred. As we will see below the two alternatives are actually orthogonal, hence the the use of **if**$^\circ$ is permitted.

If one of the coefficents is 0 it may be omitted, e.g. we write $\{(-1)\ \mathrm{qtrue}\}$ to construct a qbit which behaves like qtrue, if measured, but which has a different phase. There is no need to introduce an n-ary superposition operator, since this can be simulated by nested uses of the binary operator, e.g.

$\{0 \mid 1 \mid 2\} : \mathcal{Q}_\mathbf{3}$

can be encoded as

$\{(\frac{1}{3})\ 0 \mid (\frac{2}{3})\{1 \mid 2\}\} : \mathcal{Q}_\mathbf{3}$

2.5 Orthogonality

The idea of $t \perp u$ is that we have an observation which tells the two terms apart. We will see that the information obtained by interpreting the derivations of this judgement is essential to compile **case**$^\circ$ and superpositions.

Different injections are orthogonal, we leave the typing premises for t, u implicit:

$$\frac{\Gamma \vdash^\circ t : \sigma \qquad \Gamma \vdash^\circ u : \tau}{\texttt{inl}\ t \perp \texttt{inr}\ u \qquad \texttt{inr}\ t \perp \texttt{inl}\ u}\ \mathrm{Oinlr, Oinrl}$$

Constructors preserve orthogonality:

$$\frac{t \perp u}{\texttt{inl}\ t \perp \texttt{inl}\ u \quad \texttt{inr}\ t \perp \texttt{inr}\ u}\ \mathrm{Oinl, Oinr}$$

$$\frac{t \perp u}{(t, v) \perp (u, w) \quad (v, t) \perp (w, u)}\ \mathrm{Opairl, Opairr}$$

Finally, superpositions can be orthogonal:

$$\frac{t \perp u \quad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\{(\lambda_0)t \mid (\lambda_1)u\} \perp \{(\kappa_0)t \mid (\kappa_1)u\}}\ \mathrm{Osup}$$

An instance of this rule is used when defining the Hadamard operator: we have that

$$\{(-1)\texttt{qtrue}|\texttt{qfalse}\}|\{\texttt{qtrue}|\texttt{qfalse}\}$$

which is just short for

$$\{(-\frac{1}{\sqrt{2}})\texttt{qtrue}|(\frac{1}{\sqrt{2}})\texttt{qfalse}\} \perp \{(\frac{1}{\sqrt{2}})\texttt{qtrue}|(\frac{1}{\sqrt{2}})\texttt{qfalse}\}$$

The rules for orthogonality are incomplete, e.g. we never allow **case**°-expressions to be orthogonal, even though semantically this may be the case. Hence, we may in future add additional rules.

2.6 Programs

Programs are definitions of terms in contexts. We have omitted passing the programs explicitely through the rules and also the requirement that the axioms require that the program is well-typed. Well-typed programs can be constructed by the following obvious rule:

$$\frac{\vdash \vec{d} \qquad \vec{d}; \Gamma \vdash t : \sigma}{\vdash \vec{d}, f\Gamma = t : \sigma}\,\text{def}$$

Note that unlike [Sel04], we do not allow any recursion. Previously defined functions can be used:

$$\frac{\vec{d} = \vec{d'}, f(x_1 : \sigma_1, .., x_n : \sigma_n) = t : \tau \quad \vec{d}; \Gamma \vdash t_1 : \sigma_1, \ldots t_n : \sigma_n}{\vec{d}; \Gamma \vdash f\vec{t} : \tau}\,\text{app}$$

2.7 Examples

We present two small QML programs: a variant of Deutsch's algorithm and an encoding of quantum teleportation in QML.

2.7.1 Deutsch's algorithm

Deutsch's algorithm is usually presented as the problem to find out whether a classical function on Booleans is constant by querying the function only once. To avoid to have to resort to higher order, we solve here the morally equivalent problem to decide whether two qbits, which are assumed to be classical, are equal, with the property that each branch of the program only queries one of the input bits. We arrive at the following QML program:

```
deutsch : $\mathcal{Q}_2 \multimap \mathcal{Q}_2 \multimap \mathcal{Q}_2$
deutsch a b = let (x, y) = if° {qfalse | qtrue}
                           then (qtrue, if° a
                                        then ({qfalse | (−1) qtrue}, (qtrue, b))
                                        else ({(−1) qfalse | qtrue}, (qfalse, b)))
                           else (qfalse, if° b
                                        then ({(−1) qfalse | qtrue}, (a, qtrue))
                                        else ({qfalse | (−1) qtrue}, (a, qfalse)))
              in had x
```

We observe that the need to store both input qbits in the temporary structure computed by **if**° is actually unnecessary since we can assume that these bits are classical and hence can be used without further measuring. If we had access to classical bits we could simplify this program to:

```
$\mathit{deutsch} : 2 \multimap 2 \multimap \mathcal{Q}_\mathbf{2}$
$\mathit{deutsch}\ a\ b = \mathbf{let}\ (x, y) = \mathbf{if}^\circ \{\mathrm{qfalse} \mid \mathrm{qtrue}\}$
                             $\mathbf{then}\ (\mathrm{qtrue}, \mathbf{if}\ a$
                                     $\mathbf{then}\ \{\mathrm{qfalse} \mid (-1)\ \mathrm{qtrue}\}$
                                     $\mathbf{else}\ \{(-1)\ \mathrm{qfalse} \mid \mathrm{qtrue}\}$
                             $\mathbf{else}\ (\mathrm{qfalse}, \mathbf{if}\ b$
                                     $\mathbf{then}\ \{(-1)\ \mathrm{qfalse} \mid \mathrm{qtrue}\}$
                                     $\mathbf{else}\ \{\mathrm{qfalse} \mid (-1)\ \mathrm{qtrue}\}$
                     $\mathbf{in}\ \mathit{had}\ x$
```

We plan to incorporate classical bits in future versions of QML, very much along the lines of Selinger's QPL [Sel04].

2.7.2 Quantum teleportation

The quantum teleport protocoll allows us to *teleport* a qbit to a partner with whom we share a EPR pair, using only two bits of classical information. We cannot formalize the separation of the partners or their classical computation in QML, but we can implement a function *tel*, which encodes what happens to the teleported qbit. The correctness of the teleport protocol can be verified by showing that *tel* is extensionally equivalent to the identity function.

Auxilliary, we define Pauli's Z-function:

```
$z : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$
$z\ x = \mathbf{if}\ x\ \mathbf{then}\ \{(-1)\ \mathrm{qtrue}\}\ \mathbf{else}\ \mathrm{qfalse}$
```

and we implement *tel* as:

```
$\mathit{tel} : \mathcal{Q}_\mathbf{2} \multimap \mathcal{Q}_\mathbf{2}$
$\mathit{tel}\ x = \mathbf{let}\ (a, b) = \{(\mathrm{qfalse}, \mathrm{qfalse}), (\mathrm{qtrue}, \mathrm{qtrue})\}$
          $(a', x') = \mathit{cnot}\ a\ x$
          $b' = \mathbf{if}\ a'\ \mathbf{then}\ \mathit{qnot}\ b\ \mathbf{else}\ b$
          $b'' = \mathbf{if}\ \mathit{had}\ x'\ \mathbf{then}\ z\ b'\ \mathbf{else}\ b'$
      $\mathbf{in}\ b''$
```

3 Operational semantics

We define an operational semantics of QML by presenting a translation of QML derivations to quantum computations representable as circuits. We will show that the semantics doesn't depend on the choice of derivations, but only on the term up to extensional equality; introduced in the next section. We represent irreversible computations as a reversible computation, which may use additional quantum registers, or wires, which are initialised, and which can dispose of some registers at the end of the computation. We have used this construction to define the category of finite quantum computations, FQC [AG04]. Here we will use the same notion for QML's operational semantics, but we will not identify computations up to extensional equality here.

We use the size of a register, or equivalently the number of wires, as the types of our computation. We use Q_1 for 0 because it can represent just one state, and Q_2 for 1, which is the type of qbits. We define $a \otimes b = a + b$, because addition of wires corresponds to the tensor product in the structures we consider. We identify finite sets with natural numbers, i.e. any number $a \in \mathbb{N}$ is identified with its initial segment, e.g. $2 = \{0, 1\}$.

3.1 Reversible computations

We define the set if reversible quantum computations (or circuits) of size $a \in \mathbb{N}$, $\mathbf{FQC}^{\simeq}\ a$ inductively:

rotation rot $u \in \mathbf{FQC}^{\simeq}\ \mathcal{Q}_\mathbf{2}$, where $u \in 2 \to 2 \to \mathbb{C}$ is a unitary matrix

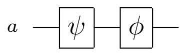
$$\begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix}$$

with $u_{00}u_{10} + u_{01}u_{11} = 0$. Note that negation $qnot \in \mathbf{FQC}^{\simeq}\ \mathcal{Q}_\mathbf{2}$ is a special rotation given by $qnot =$ rot $unot$ where

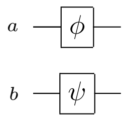
$$unot = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

wires wires $\phi \in \mathbf{FQC}^{\simeq}\ a$ where $\phi : a \simeq a$ is a bijection. This represents any rewiring, including the identity $id_a = \text{wires}\, id$

sequential composition Given $\phi \in \mathbf{FQC}^{\simeq}\ a$ and $\psi \in \mathbf{FQC}^{\simeq}\ a$ we construct $\phi \circ \psi \in \mathbf{FQC}^{\simeq}\ a$.

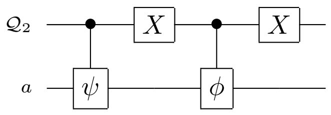


parallel composition Given $\phi \in \mathbf{FQC}^{\simeq}\ a$ and $\psi \in \mathbf{FQC}^{\simeq}\ b$ we construct $\phi \otimes \psi : \mathbf{FQC}^{\simeq}\ (a \otimes b)$.



conditional Given $\phi, \psi \in \mathbf{FQC}^{\simeq}\ a$ we construct $\phi \mid \psi \in \mathbf{FQC}^{\simeq}\ (\mathcal{Q}_\mathbf{2} \otimes a)$.

In the literature the unary conditional is used. However, it is straightforward to reduce our binary conditional to the usual unary:



To be able to interpret circuit diagrams unambiguously we assume that $(\mathbf{FQC}^{\simeq}, \mathrm{id}, \circ, \mathcal{Q}_{\mathbf{1}}, \otimes)$ is a strict monoidal category where $\mathbf{FQC}^{\simeq}\ a\ a = \mathbf{FQC}^{\simeq}\ a$ and $\mathbf{FQC}^{\simeq}\ a\ b = \{\}$, if $a \not\equiv b$ and that wires are a strict monoidal functor. That is, we consider $\mathbf{FQC}^{\simeq}$ as a quotient — this could have been avoided by using a notion of monoidal normal form, but this seems hardly relevant here.

In the literature, it is common to consider only a finite set of gates, this can be achieved by using only a finite base of rotations from which any rotation can be approximated, see [NC00], pp. 188.

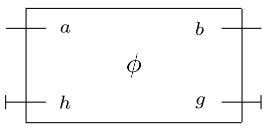
3.2 Irreversible computations

Given $a, b \in \mathbb{N}$ standing for the size of inputs and output registers of the computation, a finite quantum computation $\alpha \in \mathbf{FQC}\ a\ b$ is given by $\alpha = (h, g, \phi)$ where

- $h \in \mathbb{N}$ is the size of the initial heap,
- $g \in \mathbb{N}$ is the size of the garbage to be disposed at the end of the computation,
- $c = a \otimes h = b \otimes g$, otherwise we wouldn't be able to find a reversible computation,
- $\phi \in \mathbf{FQC}^{\simeq}\ c$ is a reversible computation.

This is different from our presentation of **FQC** in [AG04]: we only consider finite sets of powers of 2, i.e. bit vectors, and we omit the heap initialisation vector, as a vector of 0's is always used for the initialisation.

Diagrammatically, we represent such a computation as:

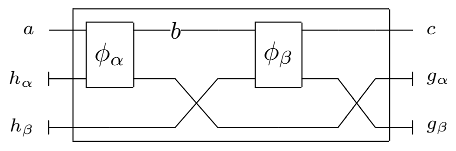


Note that, in the above diagram, heap inputs are initialised with a $\vdash$, and garbage outputs are terminated with a $\dashv$.

Given $a \in \mathbb{N}$ we define $\mathrm{id}_a = (Q_1, Q_1, \mathrm{id}_a)$ and the sequential composition of computations $\alpha = (h_\alpha, g_\alpha, \phi_\alpha) \in \mathbf{FQC}\, a\, b$ and $\beta = (h_\beta, g_\beta, \phi_\beta) \in \mathbf{FQC}\, a\, b$ as $\beta \circ \alpha = (h, g, \phi) \in \mathbf{FQC}\, a\, c$ where

$$\begin{aligned} h &= h_\alpha \otimes h_\beta \\ g &= g_\alpha \otimes g_\beta \\ \phi_{\beta\circ\alpha} &= (g_\alpha \otimes \phi_\beta) \circ (g_\beta \otimes \phi_\alpha) \end{aligned}$$

omitting some obvious monoidal isomorphisms from the definition of $\phi_{\beta\circ\alpha}$. Diagrammatically this construction is given by:

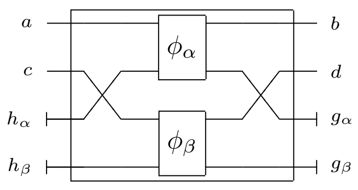


$\phi_{\beta\circ\alpha}$

We also define parallel composition for $\alpha \in \mathbf{FQC}\, a\, b$ and $\beta \in \mathbf{FQC}\, c\, d$ as $\alpha \otimes \beta = (h, g, \phi) \in \mathbf{FQC}\,(a \otimes c)\,(b \otimes d)$ where

$$\begin{array}{rcl} h & = & h_\alpha \otimes h_\beta \\ g & = & g_\alpha \otimes g_\beta \\ \phi_{\alpha\otimes\beta} & = & \phi_\alpha \otimes \phi_\beta \end{array}$$

again omotting monoidal isomorphisms. Diagrammatically, this is simply:



$\phi_{\alpha\otimes\beta}$

We notice that $(\mathbf{FQC}, \mathrm{id}, \circ, \mathcal{Q}_{\mathbf{1}}, \otimes)$ is a strict monoidal category. However, we will not exploit this fact but always construct $\mathbf{FQC}^{\simeq}$ diagrams directly.

We define strict computations $\alpha = (h, \phi) \in \mathbf{FQC}^{\circ}\ a\ b$ as one where $g = \mathcal{Q}_{\mathbf{1}}$, hence we are omitting it. We observe that this is a monoidal subcategory of $\mathbf{FQC}$.

3.3 Interpretation of judgements

First, we have to calculate the size of types and contexts, indeed this is the same as for classical circuits:

$$\begin{array}{rcl} |1| & = & \mathcal{Q}_{\mathbf{1}} \\ & = & 0 \\ |\sigma \oplus \tau| & = & \mathcal{Q}_{\mathbf{2}} + |\sigma| \sqcup |\tau| \\ & = & 1 + |\sigma| \sqcup |\tau| \\ |\sigma \otimes \tau| & = & |\sigma| \otimes |\tau| \\ & = & |\sigma| + |\tau| \end{array}$$

We use $a \sqcup b$ for the maximum of two numbers. Contexts correspond to the tensor product of their component types, hence:

$$\begin{array}{rcl} |\bullet| & = & \mathcal{Q}_{\mathbf{1}} \\ |\Gamma, x : \sigma| & = & |\Gamma| \otimes |\sigma| \end{array}$$

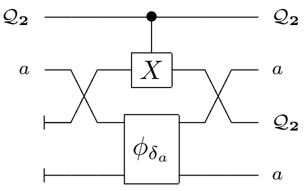
We will frequently omit the size function and just write Γ for $|\Gamma|$ and σ for $|\sigma|$. We interpret derivations $\frac{d}{\Gamma \vdash t:\sigma}$ as $[\![d]\!]_{\mathrm{op}} \in \mathbf{FQC}\ \Gamma\ \sigma$ and $\frac{d}{\Gamma \vdash^{\circ} t:\sigma}$ as $[\![d]\!]_{\mathrm{op}} \in \mathbf{FQC}^{\circ}\ \Gamma\ \sigma$. Given $\Gamma \vdash^{\circ} t : \sigma$ and $\Gamma' \vdash^{\circ} u : \sigma$ we interpret a derivation $\frac{d}{t \perp u}$ as a structure $[\![d]\!]_{\perp} = (c, l, r, \psi)$ where

- $c \in \mathbb{N}$,
- $l \in \mathbf{FQC}^{\circ}\ \Gamma\ c$
- $r \in \mathbf{FQC}^{\circ}\ \Gamma'\ c$
- $\psi \in \mathbf{FQC}^{\simeq}\ \sigma\ (\mathcal{Q}_{\mathbf{2}} \otimes c)$

The semantics of programs $\vdash \vec{d}$, is given by an assignement of circuits to function names. We will not study this in detail, since this it is straightforward and standard.

3.4 Operations on contexts

Using $\Gamma \otimes \Delta$ we can use a variable several times, allowing contraction. This is interpreted using $\delta = (\mathcal{Q}_2, \phi_\delta) \in \mathbf{FQC}^{\circ}\ \mathcal{Q}_2\ (\mathcal{Q}_2 \otimes \mathcal{Q}_2)$ where $\phi_\delta = \mathrm{id}|qnot$ is conditional negation. First, we note that we can iterate δ to contract registers of any size: given $a \in \mathbb{N}$ we define $\delta_a \in \mathbf{FQC}^{\circ}\, a\, (a \otimes a)$ by $\delta_0 =$ wires id and $\delta_{a \otimes \mathcal{Q}_2}$ can be constructed from δ_a by



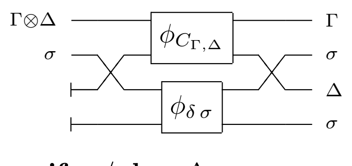
$\mathcal{Q}_{\mathbf{2}}$
a
X
ϕ_{δ_a}
$\mathcal{Q}_{\mathbf{2}}$
a
$\mathcal{Q}_{\mathbf{2}}$
a

Given Γ, Δ such that $\Gamma \otimes \Delta$ is well-defined, we construct

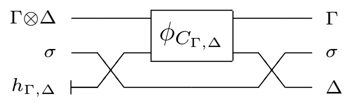
$$\mathrm{C}_{\Gamma,\Delta} \in \mathbf{FQC}^{\circ}\ |\Gamma \otimes \Delta|\ (|\Gamma| \otimes |\Delta|)$$

by induction over the definition of $\Gamma \otimes \Delta$. Note that the explicit use of $|\ldots|$ in this formula is essential, since $\otimes$ is defined differently for contexts and for their size.

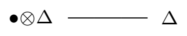
$\Gamma, x : \sigma \otimes \Delta, x : \sigma = (\Gamma \otimes \Delta), x : \sigma$



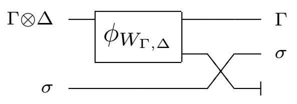
$\Gamma, x : \sigma \otimes \Delta = (\Gamma \otimes \Delta), x : \sigma$**, if** $x \notin \mathbf{dom}\,\Delta$



$\bullet \otimes \Delta = \Delta$

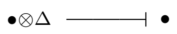


For the weakening rule we need an operator which forgets all variables which only occur in the second context, $\mathrm{W}_{\Gamma,\Delta} \in \mathbf{FQC}\,|\Gamma \otimes \Delta|\,|\Gamma|$. Again, this is defined by induction over the definition of $\Gamma \otimes \Delta$: in the first two cases we have

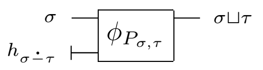


and in the final case where, Γ is empty, we simply put Δ into the garbage.

$$\bullet \otimes \Delta = \Delta$$



To interpret injections, it is necessary to increase the size of a morphism so it is the same size as another. To adjust the value of smaller size we use a padding operator $\mathrm{P}_{\sigma,\tau} \in \mathbf{FQC}^{\circ}\,\sigma\,(\sigma \sqcup \tau)$, which simply sets unused qbits to 0 using extra heap space, equal to the difference.

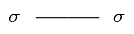


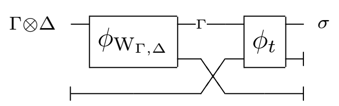
Here $a \dot{-} b$ is cutoff subtraction, i.e. $a \dot{-} b = a - b$, if $b \leqslant a$ and 0, otherwise.

3.5 Structural rules

We present the compilation of rules using circuit diagrams. The interpretation of (emb) is invisible, since $\mathbf{FQC}^{\circ}\,a\,b \subseteq \mathbf{FQC}\,a\,b$. Below are the circuits arising from the variable and the weakening rule. (weak) uses the interpretation of its premise, we omit explicit references to h and g but refer to the reversible circuit ϕ_t arising from the interpretation of $\frac{d}{\Gamma \vdash t:\sigma}$ and we will continue to use this convention in the subsequent diagrams.

$$\frac{}{x : \sigma \vdash^{\circ} x : \sigma}\,\text{var} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \otimes \Delta \vdash t^{\mathrm{dom}\,\Delta} : \sigma}\,\text{weak}$$

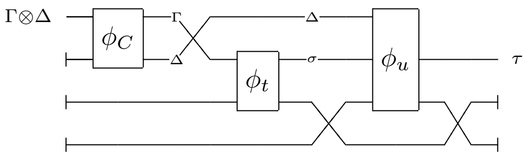




Note that the variable rule is interpreted by a strict morphism. The let-rule is actually a scheme of two rules depending on a:

$$\frac{\Gamma \vdash^{a} t : \sigma \qquad \Delta, x : \sigma \vdash^{a} u : \tau}{\Gamma \otimes \Delta \vdash^{a} \texttt{let}\ x = t\ \texttt{in}\ u : \tau}\,\text{let}$$

Its circuit interpretation uses the copy circuit C and is uniform in a:

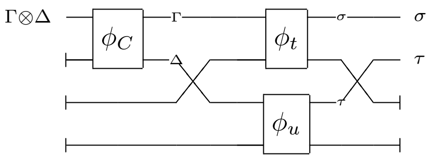


We notice that the result is strict, if both subderivations were strict.

3.6 Products (⊗)

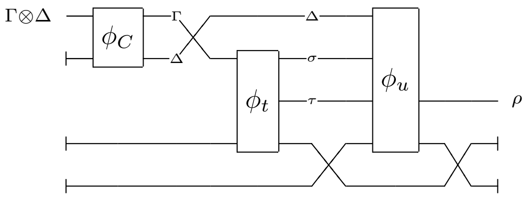
The interpretation of the rules for $\mathcal{Q}_\mathbf{1}$ in terms of circuits is invisible, since $\mathcal{Q}_\mathbf{1}$ doesn't carry any information. The interpretation of the rules for ⊗ is slightly more interesting — the introduction rule simply merges the components:

$$\frac{\Gamma \vdash^a t : \sigma \quad \Delta \vdash^a u : \tau}{\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau} \otimes - \text{intro}$$



Using the fact that $\sigma \otimes \tau$ is just interpreted by concatenating wires, the interpretation of the elimination rule is basically identical to the let-rule:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \otimes \tau \\ \Delta, x : \sigma, y : \tau \vdash^a u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^a \texttt{let } (x, y) = t \texttt{ in } u : \rho} \otimes - \text{elim}$$

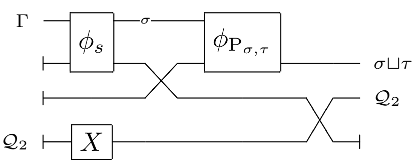


3.7 Coproducts (⊕)

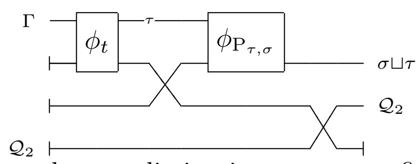
As in classical computing, we represent values of $\sigma \oplus \tau$ as a register which is big enough to hold a value of either σ or τ and an extra bit for tagging. To adjust the size of the value of smaller size we use the padding operator:

The introduction rules are compiled into circuits which pad and set the tag appropriately, we use $\sigma \sqcup \tau$ to mark a quantum register which is large enough to hold values of either type.

$$\frac{\Gamma \vdash^a s : \sigma}{\Gamma \vdash^a \texttt{inl}\ \ s : \sigma \oplus \tau} \oplus \text{intro}_1$$



$$\frac{\Gamma \vdash^a t : \tau}{\Gamma \vdash^a \texttt{inr}\ \ t : \sigma \oplus \tau} \oplus \text{intro}_2$$



We turn our attention to the two elimination operators: first the non-strict case rule

$$\frac{\begin{array}{l}\Gamma \vdash c : \sigma \oplus \tau \\ \Delta,\, x : \sigma \vdash t : \rho \\ \Delta,\, y : \tau \vdash u : \rho\end{array}}{\Gamma \otimes \Delta \vdash \texttt{case}\ \ c\ \texttt{of}\ \ \{\texttt{inl}\ x \Rightarrow t \mid \texttt{inr}\ y \Rightarrow u\} : \rho} \oplus - \text{elim}$$

We want to use the biconditional on the tagging qbit which arises from the interpretation of the derivation of c. However, there is no reason, why the reversible computations arising from the derivations of t and u:

$$\begin{array}{rcl}\phi_t & \in & \mathbf{FQC}^{\simeq}\, s_t \\ \phi_u & \in & \mathbf{FQC}^{\simeq}\, s_u\end{array}$$

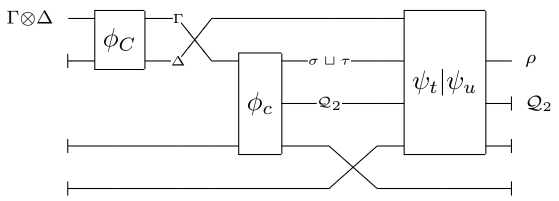
where

$$\begin{array}{rcl}s_t & = & \Gamma \otimes \sigma \otimes h_t \\ & = & \rho \otimes g_t \\ s_u & = & \Gamma \otimes \tau \otimes h_u \\ & = & \rho \otimes g_u\end{array}$$

should have the same size. However, we can pad the smaller computation: We obtain

$$\begin{array}{rcl}\psi_t & = & \phi_t \otimes (s_u \dot{-} s_t) \\ & \in & \mathbf{FQC}^{\simeq}\, (s_t \sqcup s_u) \\ \psi_u & = & \phi_t \otimes (s_t \dot{-} s_u) \\ & \in & \mathbf{FQC}^{\simeq}\, (s_t \sqcup s_u)\end{array}$$

We arrive at the following circuit:

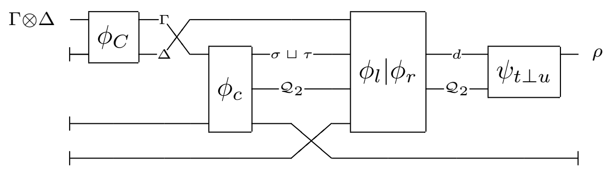


Note that the non-strict case always introduce an extra qbit of garbage, which is the qbit being measured.

To compile the strict case rule:

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

we have to exploit the data $(d, l, r, \psi_{t\perp u})$ which arises from interpreting the orthogonality judgement $t \perp u$. However, no further procrustenation is needed, because both $\phi_l, \phi_r \in \mathbf{FQC}^{\simeq}d$. We arrive at the following circuit:



We observe that this rule preserve strictness, no garbage is added to the one already produced by interpreting the derivation of c.

3.8 Superpositions

There is a simple syntactic translation we use to reduce the superposition operator to the problem of creating an arbitrary 1-qbit state:

$$\frac{\begin{array}{ll}\Gamma \vdash^\circ t, u : \sigma & t \perp u \\ |\lambda|^2 + |\lambda'|^2 = 1 & \lambda, \lambda' \neq 0\end{array}}{\begin{array}{lll}\Gamma & \vdash^\circ & \{(\lambda)t \,|\, (\lambda')u\} : \sigma \\ & \equiv & \texttt{if}^\circ\ \ \{(\lambda)\texttt{qtrue} \,|\, (\lambda')\texttt{qfalse}\} \\ & & \ \texttt{then}\ \ t\ \ \texttt{else}\ \ u\end{array}}$$

We can use our rotation primitive to produce any 1-qbit superposition simply by rotating 0, the heap initialisation, to the intended position, i.e. we use

$$\begin{pmatrix} \lambda & \lambda' \\ \lambda' & -\lambda \end{pmatrix}$$

Note that we have some freedom how to rotate the rest of the sphere, but any choice is as good as any other.

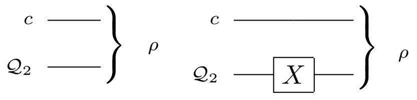
3.9 Orthogonality

We assume that $\Gamma \vdash^{\circ} t, u : \rho$ and interpret $\frac{d}{t \perp u}$ as $[\![d]\!]^{\circ}_{\perp} = (c, l, r, \psi)$ by induction over the derivation:

(Oinlr,Oinrl)

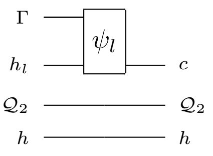
$$\frac{\Gamma \vdash^{\circ} t : \sigma \qquad \Gamma \vdash^{\circ} u : \tau}{\texttt{inl}\ t \perp \texttt{inr}\ u \qquad \texttt{inr}\ t \perp \texttt{inl}\ u}\ \text{Oinlr, Oinrl}$$

Here $\rho = \sigma \oplus \tau$, we set $c = |\sigma| \sqcup |\tau|$. In both cases l is obtained by interpreting t combined with padding and r is given by the interpretation of u and padding. The circuits for ψ for these rules are given by:

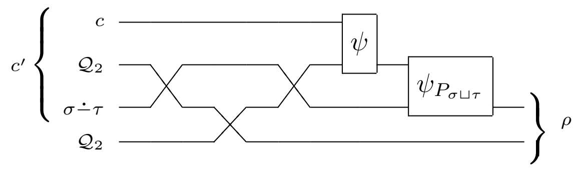


$$\frac{t \perp u}{\texttt{inl}\ t \perp \texttt{inl}\ u \quad \texttt{inr}\ t \perp \texttt{inr}\ u}$$

Let $\Gamma \vdash^{\circ} \texttt{inl}\ t, \texttt{inl}\ u : \sigma \oplus \tau$ and let (c, l, r, ψ) be the interpretation of $t \perp u$. From this data we are constructing the interpretation of $\texttt{inl}\ t \perp \texttt{inl}\ u$ as (c', l', r', ψ'). We set $c' = c \otimes \mathcal{Q}_2 \otimes h$ where $h = |\sigma| \dot{-} |\tau|$, which is the heap needed by *inl*. We construct l' and r' by applying *inl* to l, r on the level of semantics but by using the appropriate part of c' as the heap:



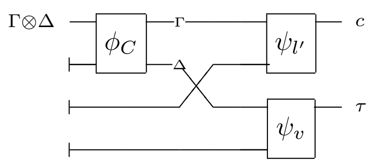
ψ is given by the following diagram:



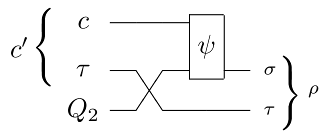
The second rule for *inr* is done symmetrically.

$$\frac{t \perp u}{(t,v) \perp (u,w) \quad (v,t) \perp (w,u)}$$

As above, let $\Gamma \vdash^{\circ} (t,v),(u,w) : \sigma \otimes \tau$ and let (c,l,r,ψ) be the interpretation of $t \perp u$ to construct the interpretation of $(t,v) \perp (u,w)$ as (c',l',r',ψ'). We set $c' = c \otimes |\tau|$ and construct l' and r' by pairing with v,w, semantically:



The definition of ψ' is given by the following diagram:



$$\frac{t \perp u \quad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\{(\lambda_0)t \mid (\lambda_1)u\} \perp \{(\kappa_0)t \mid (\kappa_1)u\}}$$

Let (c,l,r,ψ) be the interpretation of $t \perp u$ we construct the interpretation of the conclusion as (c,l,r,ψ). We set $c = c'$ and define $\psi \in \mathbf{FQC}^{\simeq}\ Q_2$ as

$$\psi = \begin{pmatrix} \lambda_0 & \lambda_1 \\ \kappa_0 & \kappa_1 \end{pmatrix}$$

4 Denotational semantics

The denotational semantics of QML is based on a high level view of quantum mechanics inspired by Selinger's denotational semantics of QPL [Sel04]: we model finite-dimensional quantum states as vectors in a complex vector space, i.e. as functions from the finite classical state space to complex numbers. We can perform measurements on those states which have probabilistic outcomes related to the complex amplitude and which collapses the part of the state which is measured. Hence we require that the sum of probabilities of all possible outcomes of a measurement add up to 1. Reversible quantum computations can be modelled by unitary operators, these are linear isomorphisms which preserve the probabilistic interpretation of amplitudes. Irreversible programs, involve measurements because we cannot dispose of a quantum bit without measuring it, and hence lead to mixed states, i.e. probabilistic distribution of pure states. We use density operators to model mixed states and super operators (aka completely positive operators) to interpret irreversible programs acting on pure states.

4.1 Linear algebra

In this section we review some basic notions from linear algebra. We also introduce the notation we use, which is based on functional programming idioms.

We use natural numbers as objects of the category of finite sets **FinSet** whose homsets are given by **FinSet** $a\ b = a \to b$, as before we identify natural numbers with their initial segements. Given $a \in \mathbb{N}$ we define $\mathbf{C}\ a = a \to \mathbb{C}$. This function on objects $\mathbf{C} \in \mathbf{FinSet} \to \mathbf{Set}$ is monadic, i.e. it gives rise to a Kleisli structure, see [AR99], with

$$
\begin{aligned}
&\mathit{return} \in a \to \mathbf{C}\ a\\
&\mathit{return}\ a = \lambda b \to \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&(\gg\!=) \in (\mathbf{C}\ a) \to (a \to \mathbf{C}\ b) \to \mathbf{C}\ b\\
&v \gg\!= f = \lambda b \to \Sigma\ a.(v\ a)(f\ a\ b)
\end{aligned}
$$

We are using an approximation to Haskell here, the proper definition is a bit more involved, since we have to take into account that a, b are finite. See [VAS04] for a full development of superoperators as an instance of the arrow class in Haskell.

The associated Kleisli category is the category of finite dimensional complex vector spaces **FinVec**, its homsets are given by **FinVec** $a\ b = a \to \mathbf{C}\ b$, where $a, b \in \mathbb{N}$, and hence correspond to $a \times b$ complex matrices. Since we are working with bit-vectors most of the time we define $\mathbf{C_2}\ a = \mathbf{C}\ (2 \to a)$ and $\mathbf{FinVec_2}\ a\ b = \mathbf{FinVec}\ (2 \to a)\ (2 \to b)$.

The cartesian product on finite sets (the numeric product on natural numbers) defines the tensor product on **FinVec**. I.e., on objects, $a \otimes b = ab$; and on morphisms, given $f \in \mathbf{FinVec}\ a\ b$, $g \in \mathbf{FinVec}\ c\ d$ we define $f \otimes g \in \mathbf{FinVec}\ (a \otimes c)\ (b \otimes d)$ as $f \otimes g = \lambda(a, c) \to \lambda(b, d) \to (f\ a\ b)(g\ c\ d)$. The unit of the tensor is $I = 1$, and $(\mathbf{FinVec}, \otimes, I)$ is a strict monoidal category. The tensor product in $\mathbf{FinVec_2}$ is given by $+$.

For vectors $v, w \in \mathbf{C}\ a$ we define their inner product $\langle v|w\rangle \in \mathbb{C}$ as $\langle v|w\rangle = \Sigma a.(va)^*\ (w\ a)$, where $(x + yi)^* = x - yi$ is the complex conjugate. The norm of a vector $\|v\| \in \mathbb{R}^+$ is defined as $\|v\| = \langle v|v\rangle$. Two vectors are orthogonal, $v \perp w$, if $\langle v|w\rangle = 0$. A base of a vectorspace is orthonormal, if any two different base vectors are orthogonal.

Given $f \in \mathbf{FinVec}\ a\ b$ the adjoint of f is given by $f^\dagger = \lambda b\ a \to (f\ a\ b)^*$, with the defining property $\langle v|fw\rangle = \langle f^\dagger v|w\rangle$. A map $u \in \mathbf{FinVec}\ a\ b$ is unitary, if its adjoint is its inverse $u \circ u^\dagger = id$, this implies that u is an isomorphism, and hence also $a = b$. Unitary maps are isometric, i.e. they preserve the inner product, $\langle v|w\rangle = \langle u\ v|u\ w\rangle$. However, not all isometric maps are unitary, e.g. the diagonal maps $\delta_a \in \mathbf{FinVec}\ a\ (a \otimes a)$, which are given by $\delta\ a\ (b, c) = \mathbf{if}\ a \equiv b\ \&\ b \equiv c\ \mathbf{then}\ 1\ \mathbf{else}\ 0$ are isometric but not unitary.

A linear map $f \in \mathbf{FinVec}\ a\ a$ is self-adjoint, if $f = f^\dagger$. A self-adjoint map has only real eigenvalues; $f\ v = \lambda v$ implies $\lambda \in \mathbb{R}$. The map is positive if all eigenvalues are positive, that is $\lambda \geqslant 0$. The trace of a map $||f||$ is the sum of all eigenvalues, which can be directly calculated as $||f|| = \Sigma\ a.f\ a\ a$.

4.2 Reversible and strict computations

We model a quantum state as a vector with norm 1. The probability that a measurement will result in a is $(va)^*(va) \in \mathbb{R}^+$. The condition on the norm guarantees that the measurements are a probability distribution. Reversible and strict computations can be modelled as linear functions on states, while general irreversible computations involve probabilities, and are modelled as linear functions on density matrices, called superoperators.

We introduce the categories $\mathbf{Q}^\circ$, to model strict, and $\mathbf{Q}^\simeq$ to model reversible, computations: their objects are natural numbers which correspond to the size of a quantum register, or the number of wires in a circuit, and the homsets $\mathbf{Q}^\simeq\ a\ b$ and $\mathbf{Q}^\circ\ a\ b$ are the linear maps $u \in \mathbf{FinVec_2}\ a\ b$, which we require to be isometric in the case of $\mathbf{Q}^\circ$ and strict in the case of $\mathbf{Q}^\simeq$. Since every unitary map is also isometric, $\mathbf{Q}^\simeq$ is a subcategory of $\mathbf{Q}^\circ$. Since $\mathbf{Q}^\simeq\ a\ b$ is only non-empty, if $a = b$ we abbreviate $\mathbf{Q}^\simeq\ a\ a$ as $\mathbf{Q}^\simeq\ a$.

We interpret all $\phi \in \mathbf{FQC}^\simeq\ a$ as $[\![\phi]\!] \in \mathbf{Q}^\simeq\ a$, by induction over the inductive definition of $\mathbf{FQC}^\simeq$:

rotation $[\![\text{rot}\ u]\!] = u$

wires $[\![\text{wires}\ \phi]\!] = f$ where $f\ a\ b = \textbf{if}\ \phi\ a \equiv b\ \textbf{then}\ 1\ \textbf{else}\ 0$.

sequential composition $[\![\phi \circ \psi]\!] = [\![\phi]\!] \circ [\![\psi]\!]$

parallel composition $[\![\phi \otimes \psi]\!] = [\![\phi]\!] \otimes [\![\psi]\!]$, here we exploit $2^{ab} = 2^a + 2^b$.

conditional $[\![\phi \mid \psi]\!] = [\![\phi]\!] | [\![\psi]\!]$

$$\phi|\psi\ (0,a)\ (0,b) = \phi\ a\ b$$
$$\phi|\psi\ (1,a)\ (1,b) = \psi\ a\ b$$

Note that $\phi|\psi$ is isometric (unitary) if both ϕ and ψ are isometric (unitary).

We note that $\mathbf{Q}^\simeq$ is a monoidal category and wires extends to a monoidal functor, hence $[\![\cdot]\!]$ respects the equality on computations. However, it certainly does identify many more computations.

Indeed, this interpretation is full; every $\phi \in \mathbf{Q}^\simeq\ a$ can be generated by the appropriate reversible computation. This is a consequence of the Solovay-Kitaev theorem.

Given $z \in \mathbf{C_2}\ h$ we use $\otimes z \in \mathbf{FinVec_2}\ a\ (a \otimes h)$ for the map which initialises the 2^{nd} part of the product. We note that it is isometric; $\otimes z \in \mathbf{Q}^\circ\ a\ (a \otimes h)$. To any $(h, \phi) \in \mathbf{FQC}^\circ\ a\ b$ with $\phi \in \mathbf{FQC}^\simeq\ (a \otimes h)\ b$, we assign $[\![h, \phi]\!] \in \mathbf{Q}^\circ\ a\ b$ by $[\![h, \phi]\!] = \phi\ (\otimes 0^h)$, where $0^h \in \mathbf{C_2}\ h$ is the constant zero vector.

For orthogonal maps, that is $f, g \in \mathbf{Q}^\circ\ a\ b$ such that for all $\vec{a} \in \mathbf{C_2}\ a$ we have $f \perp g$ we can define another form of the conditional $f|^\circ g \in \mathbf{Q}^\circ\ (\mathcal{Q}_\mathbf{2} \otimes a)\ b$ as

$f|^\circ g\ (0, a)\ b = f\ a\ b$
$f|^\circ g\ (1, a)\ b = g\ a\ b$

Clearly, we have $f|g = 0 \otimes f|^\circ 1 \otimes g$.

4.3 Irreversible computations

To interpret irreversible computations we have to model mixed states, which arise as the result of a measurement. A mixed state of size a is represented as a positive map $\rho \in \mathbf{FinVec_2}\ a\ a$, such that $||\rho|| = 1$. This is called a density matrix. The idea is that the probability that ρ is in state v is λ, if $\rho\ v = \lambda v$, i.e. v is an eigenvector with eigenvalue λ, and 0 otherwise. The trace condition ensures that this is a probability distribution on the vectors in any orthonormal base.

A linear map $f \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$ can be interpreted as an operator on density matrices by using that $\mathbf{FinVec_2}\ a\ a \simeq \mathbf{C_2}\ (a \otimes a)$ [2] . We say that such an operator is positive if it preserves positivity. It is completely positive if $f \otimes (c \otimes c) \in \mathbf{FinVec_2}\ ((a \otimes c) \otimes (a \otimes c))\ ((b \otimes c) \otimes (b \otimes c))$, is positive for any $c \in \mathbb{N}$. It is a superoperator if it is completely positive, and trace-preserving. We define $\mathbf{Q}$ as the category of superoperators, i.e. its objects are natural numbers and its morphisms are superoperators, that is $\mathbf{Q}\ a\ b$ is given by $f \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$, which are completely positive and norm-preserving. The tensor product on super operators is given by the tensor product of the underlying vector space.

Isometric maps give rise to superoperators. Given $f \in \mathbf{Q}^\circ\ a\ b$, we define $\overline{f} \in \mathbf{Q}\ a\ b$ as follows: given a density matrix $\rho \in \mathbf{FinVec_2}\ a\ a$, we construct $\overline{f}\ \rho = f \circ \rho \circ (f^\dagger) \in \mathbf{FinVec_2}\ b\ b$, using the isomorphism $\mathbf{FinVec}\ a\ a \simeq \mathbf{C}\ (a \otimes a)$. This gives rise to $\overline{f} \in \mathbf{FinVec_2}\ (a \otimes a)\ (b \otimes b)$, which is completely positive and trace preserving.

We interpret measurements as partial trace, defining $\mathrm{Tr}_{ab} \in \mathbf{Q}\ (a \otimes b)\ a$ as

$\mathrm{Tr}_{ab} \in \mathbf{FinVec_2}\ ((a \otimes b) \otimes (a \otimes b))\ (a \otimes a)$

```
Tr_ab = λ(a, b) (a′, b′) → if   b ≡ b′
                           then return (a, a′)
                           else λ(_, _) → 0
```

It can be verified that Tr_{ab} is completely positive and norm preserving.

Given the above, we can interpret $(h, g, \phi) \in \mathbf{FQC}\ a\ b$ as $[\![(h, g, \phi)]\!]_\mathbf{FQC} \in \mathbf{Q}\ a\ b$ using $[\![(h, \phi)]\!]^\circ \in \mathbf{Q}^\circ\ a\ (b \otimes g)$, which can be embedded in $\mathbf{FQC}$: $(\overline{[\![h, \phi]\!]^\circ_\mathbf{FQC}}) \in \mathbf{Q}\ a\ (b \otimes g)$ and finally using the partial trace, we obtain

$[\![(h, g, \phi)]\!]_\mathbf{FQC} \in \mathbf{Q}\ a\ b$

$[\![(h, g, \phi)]\!]_\mathbf{FQC} = \mathrm{Tr}_{bg}\ ((\overline{[\![(h, \phi)]\!]}))$

[2] Correct would be $\mathbf{C}_2\ (a^\perp \otimes a)$, but since our objects are natural numbers, this doesn't really matter.

We can use Kraus' representation theorem to show that this interpretation is full, i.e. all superoperators can be realised by an **FQC**-morphism.

To see that $[\![\cdot]\!]_{\mathbf{FQC}}$ is actually functorial, we have to show that the composition in **FQC** as defined in section 3.2 is preserved by $[\![\cdot]\!]$.

Proposition 1 $[\![\cdot]\!] \in \mathbf{FQC} \to \mathbf{Q}$ *is a strict monoidal functor, that is*

1. $[\![\mathrm{id}]\!] = \mathrm{id}$

2. $[\![f \circ g]\!] = [\![f]\!] \circ [\![g]\!]$

3. $[\![f \otimes g]\!] = [\![f]\!] \otimes [\![g]\!]$

Proof: Both 1. and 3. follow directly from monoidal identities. The only interesting case is 2. which follows from the fact that the following diagram commutes:

$$\begin{tikzcd}
& a \otimes h_f \otimes h_g \arrow[rr, "\overline{\phi_f} \otimes h_g"] & & b \otimes g_f \otimes h_g \arrow[rr, "\overline{\phi_g} \otimes g_f"] \arrow[dr, "tr_{g_f}"] & & c \otimes g_f \otimes g_g \arrow[dd, "tr_{g_g}"] \arrow[dr, "tr_{g_f g_g}"] & \\
a \arrow[ur, "z_f \otimes z_g"] \arrow[dr, "z_f"'] & & & & & & c \\
& a \otimes h_f \arrow[uu, "z_g"] \arrow[r, "\overline{\phi_f}"'] & b \otimes g_f \arrow[uur, "z_g"] \arrow[r, "tr_{g_f}"'] & b \arrow[r, "z_g"'] & b \otimes h_g \arrow[r, "\overline{\phi_g}"'] & c \otimes g_g \arrow[ur, "tr_{g_g}"'] &
\end{tikzcd}$$

□

By combining the operational semantics and the denotational semantics of computations we obtain an interpretation of derivations as isometric maps or superoperators, that is given a derivation of $\frac{d}{\Gamma \vdash t:\sigma}$ we get $[\![d]\!] \in \mathbf{Q}\ \Gamma\ \sigma$ by $[\![d]\!] = [\![[\![d]\!]_{\mathrm{op}}]\!]_{\mathbf{FQC}}$ and given $\frac{d}{\Gamma \vdash^{\circ} t:\sigma}$ we get $[\![d]\!]^{\circ} \in \mathbf{Q}^{\circ}\ \Gamma\ \sigma$ by $[\![d]\!] = [\![[\![d]\!]^{\circ}_{\mathrm{op}}]\!]^{\circ}_{\mathbf{FQC}}$.

5 Conclusions and further work

We have introduced a language for finite quantum programs which features quantum control and quantum data. We have identified the fact that weakenings affect the behaviour of a quantum program as one of the main structural differences between quantum and classical programming, and consequently use a strict linear type system where weakenings are explicit. The fact that forgetting information may affect other parts of the computation also necessitates the orthogonality judgement, which witnesses the fact that our quantum control operator **case**° does not irreversibly disposes information.

We have given an operational semantics of the language in terms of reversible quantum circuits. These circuits can model irreversible computation by having access to initialised heap registers at the start of the computation, which disopse unusued data at the end. The operational semantics has been implemented by Grattage in Haskell, [GA05].

We also present a denotational semantics by interpreting the circuits arising from the operational semantics by superoperators — here we draw heavily on Selinger's work on QPL [Sel04].

The present paper is an extended version of our conference submission [AG04]. We also have tried to improve and reorganise the presentation by clearly separating operational and denotational semantics: While in [AG04] we collapsed morphisms in **FQC** upto extensional equality, we now only identify computations upto monoidal identities (upto isomorphic circuit diagrams), and leave the extensional equality to the denotational model, i.e. to the category **Q**.

Much remains to be done, which we have left out of the current paper for reasons of space and time: to show the denotational semantics is derivation independent, i.e. different typing derivations of the same term do not affect its interpretation upto extensional equality; and is also compositional, i.e. replacing extensionally equivalent subterms results in extensionally equivalent programs.

To show derivation independence, we observe that most of the rules are structural with the exception of (emb) and ($\oplus$−elim$^\circ$). (emb) doesn't cause any problems since it is interpreted by $\mathbf{FQC}^\circ\, a\, b \subseteq \mathbf{FQC}\, a\, b$ in the operational semantics. ($\oplus$−elim$^\circ$) is more interesting: we need to show that the interpretation of $t \perp u$ is semantically correct, i.e. that the interpretation of the terms can be obtained by composing the corresponding component of the orthogonality judgement with the unitary map ψ; that $[\![t]\!] = \psi^{-1} \circ (\mathtt{qtrue} \otimes l)$ and $[\![u]\!] = \psi^{-1} \circ (\mathtt{qfalse} \otimes r)$. We can use this to show that the interpretation of ($\oplus$−elim$^\circ$) does not depend on the derivation of orthogonality.

To show compositionality, it seems that the best way is to directly give an intepretation of QML programs in **Q** and then show that this interpretation factors through the operational semantics. This denotational semantics can be effectively computed, we plan to use the material in [VAS04] to implement the semantics in Haskell. In many cases compositionality follows from proposition 1 and the observation that we only use horizontal ($\circ$) and vertical ($\otimes$) composition to define the interpretation of terms from their components. The only exceptions are the elimination rules for $\oplus$: In the case of ($\oplus$−elim$^\circ$) we have to show that $f|^\circ g$ comutes with initialisations. ($\oplus$−elim) is slightly more involved since we also have to commute the partial traces. Indeed, this only works because we measure the qbit we are branching over.

More tentative is the extension of QML to higher types, and being able to incorporate infinite data structures.

Q doesn't seem to have a closed structure which would allow us to interpret higher order programs (see [**?**] for a discussion). However, this is not really necessary since we are only running higher order programs once they are fully applied. Semantically, this observation can be exploited by interpreting higher order programs in the presheaf category over **Q**, which has a tensor product by Day's construction and is automatically closed with respect to this tensor product. There is no clear candidate for $\oplus$, and, since it is not at all obvious how a coproduct of higher order quantum functions may be implemented, the best choice may be not to allow this, but to limit $\oplus$ to first order types.

Infinite data structures could be interpreted in infinite-dimensional vector spaces using the standard approaches from mathematical physics. An alternative, which is closer to potential implementations of quantum programs, is to allow quantum programs to be indexed by classical structures in a way akin to Dependent ML (DML) [**?**]. DML is a language with dependent types where index expressions and actual programs are clearly separated. In the case of DML, this separation is needed to deal with impurities in the actual programs, such as non-termination. In a dependently typed version of QML, the same approach would be used to separate the classical structure of the computation from quantum effects.

Finally, having a high level language with a clear semantics should lead to reasoning principles which could be expressed as an algebra of quantum programming. This algebra should enable us to give mathematically clear, formal correctness proofs of quantum programs. This algebra is related to Tonders quantum λ calculus, [vT03b], but we would aim to include measurements and justify the equations by showing that they are sound and complete with respect to the denotational semantics in $\mathbf{Q}$.

Acknowledgements

We would like to thank our colleagues and friends Slava Belavkin, Bob Coecke, Martin Hofmann, Graham Hutton, Conor McBride, Amr Sabry, Peter Selinger, Alex Simpson, Thomas Streicher, Janine Swarbrick and Juliana Vizotti for interesting discussions and feedback on our work. Amr Sabry and Juliana Vizotti provided extensive feedback on previous drafts of this paper. Peter Selinger pointed out a serious flaw in an earlier version of $\mathbf{case}^{\circ}$ and refuted our conjecture that strict maps classify monos in $\mathbf{Q}$.

References

[AD04] P. Arrighi and G. Dowek. Operational semantics for a formal tensorial calculus, 2004. Draft proceedings of the 2nd International Workshop on Quantum Programming Languages.

[AG04] T. Altenkirch and J. Grattage. A functional quantum programming language. quant-ph/0409065, November 2004.

[AR99] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, 1999.

[GA05] J. Grattage and T. Altenkirch. A compiler for a functional quantum programming language. Submitted for publication, January 2005.

[Gru99] J. Gruska. *Quantum Computing.* McGraw-Hill, Maidenhead, 1999.

[Hir01] M. Hirvensalo. *Quantum Computating.* Springer-Verlag NewYork, Inc., 2001.

[Hug00] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[Kar03] J. Karczmarczuk. Structure and interpretation of quantum mechanics: a functional framework. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 50–61. ACM Press, 2003.

[Kni96] E. Knill. Conventions for quantum pseudocode, 1996.

[MB01] S-C. Mu and R. S. Bird. Quantum functional programming. In *2nd Asian Workshop on Programming Languages and Systems*, 2001.

[NC00] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, 2000.

[Öme03] B. Ömer. *Structured Quantum Programming.* PhD thesis, Institute for Theoretical Physics, Technical University of Vienna, Vienna, 2003.

[Pit00] A. Pittenger. *An Introduction to Quantum Computing Algorithms*, volume 19 of *Progress in Computer Science and Applied Logic (PCS).* Berkhauser, 2000.

[Pre99] J. Preskill. Quantum computation course lecture notes, 1999.

[Sab03] A. Sabry. Modeling quantum computing in haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 39–49. ACM Press, 2003.

[Sel04a] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 2004.

[Sel04b] P. Selinger. Towards a semantics for higher-order quantum computation. *Draft proceedings of the 2nd International Workshop on Quantum Programming Languages*, 2004.

[SV05] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. To appear in the proceedings of TLCA05, 2005.

[SZ00] J. Sanders and P. Zuliani. Quantum programming. In *Mathematics of Program Construction*, volume 1837 of *Springer Lecture Notes in Computer Science*, pages 80–99. Springer-Verlag, 2000.

[VAS04] J. K. Vizzotto, T. Altenkirch, and A. Sabry. Structuring quantum effects: Superoperators as arrows. Submitted for publication, 2004.

[vT03a] A. van Tonder. A lambda calculus for quantum computation. quant-ph/0307150, 2003. to appear in SIAM Journal of Computing.

[vT03b] A. van Tonder. Quantum computation, categorical semantics and linear logic. quant-ph/0312174, 2003.

[Xi98] H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.