A functional quantum programming language

Thorsten Altenkirch and Jonathan Grattage
School of Computer Science and IT, Nottingham University
email: {txa,jjg}@cs.nott.ac.uk

Abstract

We introduce the language QML, a functional language for quantum computations on finite types. Its design is guided by its categorical semantics: QML programs are interpreted by morphisms in the category $\mathbf{F\,QC}$ of finite quantum computations, which provides a constructive semantics of irreversible quantum computations realisable as quantum gates. QML integrates reversible and irreversible quantum computations in one language, using first order strict linear logic to make weakenings explicit. Strict programs are free from decoherence and hence preserve superpositions and entanglement – which is essential for quantum parallelism.

1 Introduction

The discovery of efficient quantum algorithms by Shor [18] and Grover [6] has triggered much interest in the field of quantum programming. However, it is still a very hard task to find new quantum algorithms. One of the reasons for this situation might be that quantum programs are very low level: they are usually represented as quantum circuits, or in some combinator language which gives rise to circuits. Here we attempt to remedy this situation by introducing the quantum programming language QML, which is based on high-level constructs known from conventional functional programming. Though functional (programs are expressions), our language is first order and finitary; all datatypes are finite. We will discuss possible extensions in the conclusions, but we believe that the approach presented here represents a significant progress towards the goal of a natural quantum programming language.

We present a semantics of our language by interpreting terms as morphisms in the category of finite quantum computations $\mathbf{F\,QC}$ which we introduce here. The $\mathbf{F\,QC}$ semantics gives rise to a denotational semantics in terms of superoperators, the accepted domain of irreversible quantum computation, and at the same time to a compiler into quantum circuits, an accepted operational semantics for quantum programs.

As an illustration, one of the basic quantum circuits is the Hadamard gate, which is usually defined by presenting its matrix:

$$\mathrm{had} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

But what does this mean in programming terms? In QML this operation is implemented by the following program

$$had : \mathbf{Q_2} \multimap \mathbf{Q_2}$$
$$had\ x = \mathbf{if}^\circ\ x$$
$$\qquad\quad \mathbf{then}\ \{\mathrm{qfalse} \mid (-1)\ \mathrm{qtrue}\}$$
$$\qquad\quad \mathbf{else}\ \{\mathrm{qfalse} \mid \mathrm{qtrue}\}$$

We can read had as an operation which, depending on its input qubit x, returns one of two superpositions of a qubit. We can also easily calculate that applying had twice gets us back where we started by cancelling out amplitudes.

An important feature of quantum programming is the possibility to create superpositions which have non-local effects. A simple application of this idea is the algorithm in figure 1 to determine whether two classical bits, represented as qubits, are equal, which is based on Deutsch's algorithm (see [12], pp.32). It exploits quantum parallelism by querying both inputs at the same time; this corresponds to the fact that the expressions $\mathbf{if}^\circ\ a$ and $\mathbf{if}^\circ\ b$ in our program are not nested. The famous algorithms by Shor and Grover rely on a more subtle exploitation of this effect.

The reader may have noticed that we do not insist on quantum programs being reversible. We will discuss this further in section 3, by comparing classical and quantum computation. It turns out that in both cases irreversible computations can be reduced to reversible ones in a similar fashion. However, reversibility plays a more central role in quantum computation due to the fact that forgetting information leads to decoherence, which destroys entanglement, and hence negatively affects quantum parallelism. Thus one of the central features of our language is *control of decoherence*, which is achieved by keeping track of weakening through the use of strict linear logic (or just strict logic) and by offering different if-then-else (or, generally, case) operators, one that measures the qubit, $\mathbf{if}$, and a second, $\mathbf{if}^\circ$, that doesn't – but which can only be used in certain situations.

$$eq : \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2}$$

$$
eq\ a\ b = \mathbf{let}\ (x, y) = \mathbf{if}^\circ \{\, \mathrm{qfalse} \mid \mathrm{qtrue} \,\}
$$

$$
\mathbf{then}\ (\mathrm{qtrue}, \mathbf{if}^\circ\ a
$$

$$
\mathbf{then}\ (\{\, \mathrm{qfalse} \mid (-1)\, \mathrm{qtrue} \,\}, (\mathrm{qtrue}, b))
$$

$$
\mathbf{else}\ (\{\, (-1)\, \mathrm{qfalse} \mid \mathrm{qtrue} \,\}, (\mathrm{qfalse}, b)))
$$

$$
\mathbf{else}\ (\mathrm{qfalse},\ \mathbf{if}^\circ\ b
$$

$$
\mathbf{then}\ (\{\, (-1)\, \mathrm{qfalse} \mid \mathrm{qtrue} \,\}, (a, \mathrm{qtrue}))
$$

$$
\mathbf{else}\ (\{\, \mathrm{qfalse} \mid (-1)\, \mathrm{qtrue} \,\}, (a, \mathrm{qfalse})))
$$

$$
\mathbf{in}\ had\ x
$$

Figure 1. A variant of Deutsch's algorithm

We hasten to add that this *intrinsic decoherence* is not related to the decoherence which is caused by thermal noise in a hypothetical quantum computer. As one of the referees has pointed out, *control of decoherence* is in spirit similar to Reynold's *control of interference* [13].

2 Related work

There are a number of papers on simulating or integrating quantum programming within conventional functional programming, e.g. Mu and Bird's proposal on modelling quantum programming in a functional language [11], Karczmarczuk's use of functional programming to model quantum systems [9] and Sabry's proposal to structure embedded quantum programs using *virtual values* [14]. Yet another approach was suggested by Sanders and Zuliani [22], which extends the probabilistic guarded command language [10] by quantum registers and operations on quantum registers.

Peter Selinger's influential paper [15] 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 [17] have presented a functional language based on the *classical control* and *quantum data* paradigm.

Selinger and Valiron's approach is in some sense complementary to ours: they use an affine type system (no contraction), while we use a strict system (no weakening). The lack of contraction is justified by the no-cloning property of quantum states. However, this does not apply to our approach as we model contraction by sharing, not by copying - this is also used in [4]. Indeed, classical programming languages do not implement contraction by copying data, but by sharing via pointers.

Andre van Tonder has proposed a quantum λ-calculus

incorporating higher order [20, 19] programs, however he is not considering measurements as part of his language. In [19] he suggests a semantics for a strict higher order quantum language based on vector bundles. At the current time it is not clear, to us, whether the details of this construction work out.

Abramsky and Coecke [1] have investigated a categorical semantics for quantum protocols using the compact closed structure of the category of finite dimensional Hilbert spaces. They suggest that their semantics may be relevant for type systems for quantum programming language. It remains to be seen how this relates to our work, since our approach does not exploit compact closure.

All the previous approaches adopt a basically combinatory approach to quantum data: operations on quantum data are given by combinators implementing unitary operators. We believe that our work is novel in that we are proposing high-level quantum control structures, i.e. we are aiming at quantum control and quantum data.

## 3 Finite classical and quantum computation

It is frequently emphasised that quantum computation relies on reversibility because quantum physics models reversible processes. This is true, but the same holds for classical computation — whether we base our notion of computation on Newtonian physics or Maxwellian electrodynamics, the underlying physical processes are reversible for a closed system. Hence we should explain irreversible classical computation based on a reversible mechanism. Here, we will develop a picture which applies to classical and quantum computation. This makes it easy to identify the essential differences and also guides the design of QML which realises structures common to both computational paradigms by syntactic constructs established in classical functional programming.

We introduce the category $\mathbf{F\,QC}$ of finite quantum computations and, for purposes of comparison, the category $\mathbf{F\,CC}$ of finite classical computations[1]. We will interpret

---

[1]$\mathbf{FCC}$ may be viewed as a categorical account of a finite version of

QML programs by **F QC**morphisms. It is straightforward to identify a classical sublanguage of QML which can be interpreted in **FCC**; however we will not carry this out in detail.

Objects of both categories are finite sets, for which we use the letters A, B, C. While classical computations are carried out on the elements of those sets, quantum computations take place in finite dimensional Hilbert spaces; we write $\mathbb{C}^A$ for the space generated by A, whose elements are functions [2]. A reversible finite computation, that is a closed computational system, is modelled by a reversible operation ϕ, which is a bijection of finite sets in the classical case, and a unitary operator on the Hilbert spaces in the quantum case. We write $A \multimap_{\text{unitary}} B$ for the set of unitary operators from the space generated by A to the space generated by B, which in the finite-dimensional case correspond exactly to norm-preserving linear isomorphisms. The initial state of a computation is divided into the input A and the initial heap H, and the final state into the output B and garbage G; using cartesian product ($\times$) in the classical and tensor product ($\otimes$) in the quantum case. To actually perform a computation we also need a heap initialisation constant h, which intuitively sets all memory cells in a defined state, e.g. 0. In the classical case this is just an element of the set $h \in H$, while in the quantum case it is an element of the vector space $h \in \mathbb{C}^H$. Such a computational system can be visualised by the following diagram:
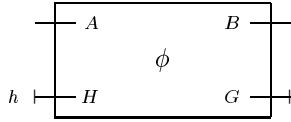
Note that in the above diagram heap inputs are initialised with a $\vdash$, and garbage outputs are terminated with a $\dashv$. To summarise, given finite sets A, B a morphism $(H, h, G, \phi) \in \textbf{FCC}\,A\,B$ is given by:

- a finite set of initial heaps H,

- an initial heap $h \in H$,

- a finite set of garbage states G,

- a bijection $\phi \in A \times H \simeq B \times G$,

while a morphism $(H, h, G, \phi) \in \textbf{F QC}\,A\,B$ is given by

- a finite set H, the basis of the space of initial heaps,

- a heap initialisation vector $h \in \mathbb{C}^H$,

- a finite set G, the basis of the space of garbage states,

Bennet's results [5].

[2] $\mathbb{C}^A$ gives rise to a *Kleisli structure*, [2], here bind is realised by matrix multiplication. Its Kleisli category is the category of finite dimensional vector spaces.

- a unitary operator $\phi \in A \otimes H \multimap_{\text{unitary}} B \otimes G$.

Given two computational systems we can compose them by combining initial and final heaps:

$$\phi_{\beta \circ \alpha}$$
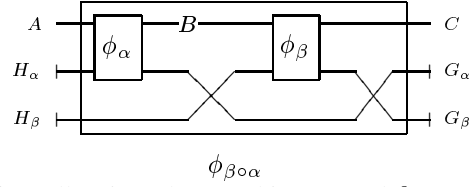
More formally, given the morphisms α and β:

$$\begin{aligned}
\alpha &= (H_\alpha, h_\alpha, G_\alpha, \phi_\alpha) \in \textbf{F CC}\,A\,B \\
\beta &= (H_\beta, h_\beta, G_\beta, \phi_\beta) \in \textbf{FCC}\,B\,C
\end{aligned}$$

the composite morphism $\beta \circ \alpha = (H, h, G, \phi)$ is given by:

$$\begin{aligned}
H &= H_\alpha \times H_\beta \\
h &= (h_\alpha, h_\beta) \\
G &= G_\alpha \times G_\beta \\
\phi &= (G_\alpha \times \phi_\beta) \circ (H_\beta \times \phi_\alpha)
\end{aligned}$$

Note that we have omitted some obvious symmetric monoidal isomorphisms for $\times$ from the definition of ϕ. We leave it to the reader to construct the identity computation.

Analogously, given morphisms

$$\begin{aligned}
\alpha &= (H_\alpha, h_\alpha, G_\alpha, \phi_\alpha) \in \textbf{F QC}\,A\,B \\
\beta &= (H_\beta, h_\beta, G_\beta, \phi_\beta) \in \textbf{FQC}\,B\,C
\end{aligned}$$

the composite $\beta \circ \alpha = (H, h, G, \phi)$ is given by

$$\begin{aligned}
H &= H_\alpha \otimes H_\beta \\
h &= h_\alpha \otimes h_\beta \\
G &= G_\alpha \otimes G_\beta \\
\phi &= (G_\alpha \otimes \phi_\beta) \circ (H_\beta \otimes \phi_\alpha)
\end{aligned}$$

Note that $\otimes$ is actually $\times$ on the underlying finite sets, since $\mathbb{C}^A \otimes \mathbb{C}^B \simeq \mathbb{C}^{A \times B}$. However, we shall use the tensor symbol because we interpret the constructed set as the basis of the tensor product of the associated vector spaces. As in the classical case we omit symmetric monoidal isomorphisms for $\otimes$.

We consider two computational systems as extensionally equal if they map the same inputs to the same outputs. That is, for **F CC** a morphism $\alpha = (H, h, G, \phi) \in \textbf{F CC}\,A\,B$ gives rise to a function on finite sets $U_{\textbf{F CC}}\alpha \in A \to B$ by

How do we do this for **F QC**? There is no sensible projection operation on tensor products. Indeed, forgetting a part of a pure state (i.e. a vector of the Hilbert space) leads to a mixed state, which is modelled by a density operator $\delta \in A \multimap A$. This is a positive operator, whose eigenvalues are interpreted as the probability that the system is in the corresponding eigenstate. Extensionally, quantum computations give rise to completely positive mappings, also called superoperators, see [7], pp. 136 or [15] for details. Given $\alpha = (H, h, G, \phi) \in$ **F QC** $A\,B$ we write $\widehat{\phi} \in A \otimes H \multimap_{\text{super}} B \otimes G$ for the associated superoperator $\widehat{\phi}\,\rho = \phi \circ \rho \circ \phi^{\dagger}$. The heap initialisation vector $h \in \mathbb{C}^H$ can be lifted to a density matrix $\widetilde{h} \in \text{Dens}\,H$ by $\widetilde{h} = |h\rangle \langle h|$. Combining this with the partial trace operator $\text{tr}_G \in B \otimes G \multimap_{\text{super}} B$ we obtain $\text{U}_{\textbf{F QC}}\alpha \in A \multimap_{\text{super}} B$ by

$$
\begin{array}{ccc}
A \otimes H & \xrightarrow{\;\widehat{\phi}\;} & B \otimes G \\[2pt]
{\scriptstyle -\otimes \widetilde{h}}\uparrow & & \downarrow {\scriptstyle \text{tr}_G} \\[2pt]
A & \xrightarrow[\;\text{U}_{\textbf{FQC}}\alpha\;]{} & B
\end{array}
$$

in the category of superoperators.

We say that two computations $\alpha, \beta \in F\,A\,B$ are extensionally equal ($\alpha =_{\text{ext}} \beta$), if the induced maps are equal; $\text{U}_F\,\alpha = \text{U}_F\,\beta$ where $F \in \{$**F CC**, **F QC**$\}$. We define the homsets of **F CC**, **F QC** as the quotients of the underlying representation by extensional equality. It is straightforward to verify that composition respects extensional equality.

As a consequence of our definition we obtain that the assignment of maps to computations gives rise to forgetful functors $\text{U}_{\textbf{FCC}} \in$ **F CC** $\to$ **FinSet** and $\text{U}_{\textbf{F QC}} \in$ **FQC** $\to$ **Super**. Both functors are full [3] and faithful. Hence, our categories **F CC** and **FQC** can be viewed just as different presentations of **FinSet** and **Super**. However, going via **FCC** and **FQC** has the benefit that we get an implementation of our programs as reversible circuits in the classical case and quantum circuits in the quantum case.

An important class of morphisms are the ones which do not produce garbage, i.e. where $G = 1$, they give rise subcategories **F CC°**, **F QC°** of *strict* morphisms. All strict maps are isometries, i.e. linear maps such that $\langle f\,v | f\,w \rangle = \langle v | w \rangle$. However, not all isometries arise from strict computations. [4]

While **F QC** and **FCC** are very similar indeed, the fact that **F QC** is based on wave mechanics enables non-local interaction which is exploited in quantum programming. However, there is also a new challenge: the possibility of decoherence. Let $\delta \in 2 \to 2 \times 2$ where $2 = \{0,1\}$

---

[3] In the case of **FQC** fullness is a consequence of Kraus' decomposition theorem.

[4] This is only due to dimensional reasons, indeed in the domain of our interpretation where all spaces are of a size 2^n the functor is full.

---

be defined as $\delta\,x = (x, x)$; which can be easily realised by a CNOT gate. The same implementation gives rise to $\mathcal{Q}_2 \multimap \mathcal{Q}_2 \otimes \mathcal{Q}_2$, writing $\mathcal{Q}_2$ for the object 2 in **F QC** In either case, we can compose this with $\pi_1 \in 2 \times 2 \to 2$ (or $\pi_1 \in \mathcal{Q}_2 \otimes \mathcal{Q}_2 \multimap_{\text{super}} \mathcal{Q}_2$) which leads to the following picture:
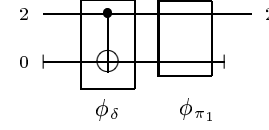
Clearly, classically we have just defined an inefficient version of the identity $\pi_1 \circ \delta = \text{I}$; we copy a bit and then throw the copy away. However, the situation is quite different in the quantum case: while the implementation is given by the same diagram by replacing classical reversible circuits with quantum circuits, the composition is not the identity, it is a *measurement* operation. That is, if we input a pure state like $R = \{\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\}$ the output is a mixed state $\frac{1}{2}\{|0\rangle\} + \frac{1}{2}\{|1\rangle\}$ corresponding to a random qubit. We have lost the advantage of quantum computation and are back in the world of classical probabilistic computations.

As a consequence of this observation we draw the conclusion that one of the main issues a quantum programming language has to address is the *control of decoherence*. This is somehow the opposite of the common view which insists that the *no cloning theorem* outlaws contraction. We observe that the implementation of δ shares a qubit, but it doesn't clone it; considering R again we obtain the EPR state $\{\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle\}$ after executing only δ. We claim that this is a natural explanation of contraction because it is completely uniform in both the classical and the quantum case. Indeed, classical functional languages do not implement contraction by copying data either. δ is strict and therefor maps pure states to pure states. In contrast, operations like π_1 are interpreted by a non-trivial partial trace which introduces decoherence. Hence it is *weakening* which deserves our attention, not *contraction*.

4 QML: Rules and semantics

We introduce here the typing rules and the denotational semantics of QML, the latter gives rise to a compilation of QML programs to quantum circuits. The compilation is presented diagramatically, implementing it requires some care to make sure that the wires generated by subcomputations match as intended.

4.1 Typing rules

We will only present the typed syntax of QML, which is based on strict linear logic, the untyped syntax is implicit in the typed one. We do allow explicit weakenings

annotating a term by by a context. This leads to an unambiguous type assignment. Any weakening will be translated into the use of a non-trivial partial trace, and hence decoherence in the *denotational semantics*. Another source of decoherence is the use of *case*, or its special instance *if-then-else*. We make this explicit by introducing two different case-operators: one which observes a qubit and thus leads to decoherence; and another which is free of decoherence but requires that we derive that the two alternatives live in orthogonal spaces. For this purpose we introduce a judgement $t \perp u$. Another novelty of our language is a *term–former* to create superpositions; we can, for example, write $\{ (\text{qtrue}, \text{qtrue}) \mid (\text{qfalse}, \text{qfalse}) \}$, to create an EPR state. Note that we are ignoring the factor $\frac{1}{\sqrt{2}}$ which can be automatically inserted by the compiler. The construction of a superposition also requires to show that the participating terms are orthogonal.

Our basic typing judgements are $\Gamma \vdash t : \sigma$ meaning that t has type σ under context Γ. and $\Gamma \vdash^\circ t : \sigma$ for strict terms. We embed $\vdash^\circ$ in $\vdash$:

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

To avoid repetition, we also use the schematic judgements $\Gamma \vdash^a t : \sigma$ where $a \in \{-, \circ\}$. We use σ, τ and ρ to quantify over types, which are generated by $1, \sigma \oplus \tau, \sigma \otimes \tau$. Qubits are defined as $\mathbf{Q_2} = 1 \oplus 1$.

Γ is a context, i.e. a function from a finite set of variables $\text{dom}\,\Gamma$ into the set of types. We write contexts as $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and use $\bullet$ for the empty context. $\Gamma, x : \tau$ is the context Γ extended by $x : \tau$. This operation is only defined if Γ does not already assign a type to x.

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

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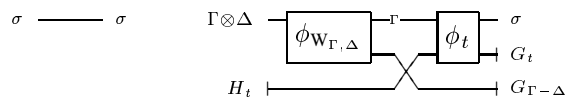\bullet \otimes \Delta &= \Delta
\end{aligned}
$$

This operation is partial – it is only well-defined if the two contexts do not assign different types to the same variable.

4.2   Denotational semantics

We assign to every type σ the number $|\sigma|$ which is the size of a quantum register needed to store elements of σ, we also interpret expressions of the form $\sigma \sqcup \tau$:

$$
\begin{aligned}
|1| &= 0 \\
|\sigma \sqcup \tau| &= \max\{|\sigma|, |\tau|\} \\
|\sigma \oplus \tau| &= |\sigma \sqcup \tau| + 1 \\
|\sigma \otimes \tau| &= |\sigma| + |\tau|
\end{aligned}
$$

The interpretation of a type is the $\mathbf{FQC}$ object of quantum registers of the right size: $[\![\sigma]\!] = \mathcal{Q}_2{}^{|\sigma|}$. Contexts $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ are interpreted as the tensor product of their components $[\![\Gamma]\!] = [\![\tau_1]\!] \otimes [\![\tau_2]\!] \otimes \ldots \otimes [\![\tau_n]\!]$. A typing derivation $\Gamma \vdash t : \sigma$ is interpreted as an $\mathbf{FQC}$ morphism $[\![t]\!] \in \mathbf{FQC}\,[\![\Gamma]\!]\,[\![\sigma]\!]$, correspondingly, $\Gamma \vdash^\circ t : \sigma$ is interpreted as $[\![t]\!] \in \mathbf{FQC}^\circ\,[\![\Gamma]\!]\,[\![\sigma]\!]$.

The interpretation of orthogonality is more involved. Given $\Gamma \vdash^\circ t : \sigma$ and $\Gamma' \vdash^\circ u : \sigma$ where $|\Gamma| = |\Gamma'|$ we interpret a derivation $t \perp u$ as a structure (S, f, g, ψ) where S is an object of $\mathbf{FQC}$, $l \in \mathbf{FQC}\,[\![\Gamma]\!]\,S$, $g \in \mathbf{FQC}\,[\![\Gamma']\!]\,S$ such that $[\![t]\!] = \phi \circ (\text{qtrue} \otimes -) \circ f$ and $[\![u]\!] = \phi \circ (\text{qfalse} \otimes -) \circ g$.

To interpret the operator $\otimes$ on contexts we define an $\mathbf{FQC}^\circ$ morphism $C_{\Gamma, \Delta} \in \mathbf{FQC}^\circ\,[\![\Gamma \otimes \Delta]\!]\,([\![\Gamma]\!] \otimes [\![\Delta]\!])$

by induction over the definition of $\Gamma \otimes \Delta$: If a variable $x : \sigma$ appears in both contexts we have to use $\delta_\sigma \in \mathbf{FQC}^\circ\,[\![\sigma]\!]\,([\![\sigma]\!] \otimes [\![\sigma]\!])$ which generalises δ_2, discussed earlier, by applying it in parallel to all qubits. All the other cases can be dealt with by applying monoidal isomorphisms. Similarly, we define an explicit weakening operator $W_{\Gamma, \Delta} \in \mathbf{FQC}\,[\![\Gamma \otimes \Delta]\!]\,[\![\Gamma]\!]$.

4.3   Structural rules

We start with the strict variable rule and the non-strict weakening and their interpretations

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

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

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

$\circ \sqcap \circ = \circ$ and $-$ otherwise. We leave the condition that $\Gamma \otimes \Delta$ is defined as an implicit precondition of this and subsequent rules using $\otimes$. The interpretation of the let-rule is given by the following circuit:

Weakenings can affect the meaning of a program. As an example consider:

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

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

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

This program is interpreted by a circuit equivalent to the one corresponding to $\pi_1 \circ \delta$ shown earlier; hence it introduces a measurement.

4.4 Rules for $\otimes$

The rules for 1, $\otimes$ are the standard rules from linear logic. In the case of 1 instead of an explicit elimination rule we allow implicit weakening:

$$\frac{}{\bullet \vdash^\circ () : 1}\ 1 - \mathrm{intro} \qquad \frac{\Gamma, x : 1 \vdash^a t : \sigma}{\Gamma \vdash^a t : \sigma}\ 1 - \mathrm{weak}$$

The interpretation of the rules for 1 in terms of circuits is invisible, since 1 doesn't carry any information. The interpretation of the rules for $\otimes$ is more interesting — the introduction rule simply merges the components

$$\frac{\Gamma \vdash^a t : \sigma \quad \Delta \vdash^a u : \tau}{\Gamma \otimes \Delta \vdash^a (t, u) : \sigma \otimes \tau}\ \otimes - \mathrm{intro}$$
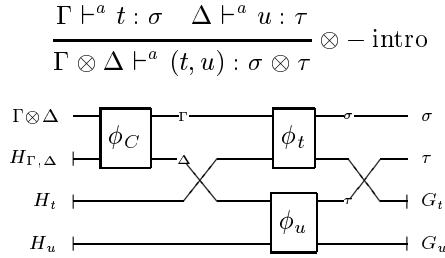
The interpretation of the elimination rule is similar to the let-rule:

$$\frac{\begin{array}{c}\Gamma \vdash^a t : \sigma \otimes \tau \\ \Delta, x : \sigma, y : \tau \vdash^b u : \rho\end{array}}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \mathtt{let}\ (x, y) = t\ \mathtt{in}\ u : \rho}\ \otimes - \mathrm{elim}$$
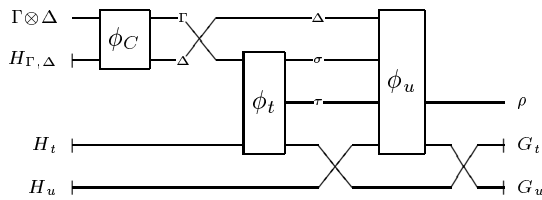
As an example, here is a simple program which swaps two qubits:

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

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

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

would measure the qubits while swapping them.

4.5 Rules for $\oplus$

We represent values in $\sigma \oplus \tau$ as words of fixed length, as in classical computing. Unfolding our type interpretation we have that $[\![\sigma \oplus \tau]\!] = \mathcal{Q}_2 \otimes [\![\sigma \sqcup \tau]\!]$ where $[\![\sigma \sqcup \tau]\!]$ can store a value either of $[\![\sigma]\!]$ or $[\![\tau]\!]$. To adjust the size we use an easily definable padding operator $\mathrm{P}_{\sigma \sqcup \tau} \in \mathbf{FQC}\ [\![\sigma]\!]\ [\![\sigma \sqcup \tau]\!]$, which simply sets unused bits to 0.

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

$$\frac{\Gamma \vdash^a s : \sigma}{\Gamma \vdash^a \mathtt{inl}\ s : \sigma \oplus \tau}\ + \mathrm{intro}_1$$

$$\frac{\Gamma \vdash^a t : \tau}{\Gamma \vdash^a \mathtt{inr}\ t : \sigma \oplus \tau}\ + \mathrm{intro}_2$$
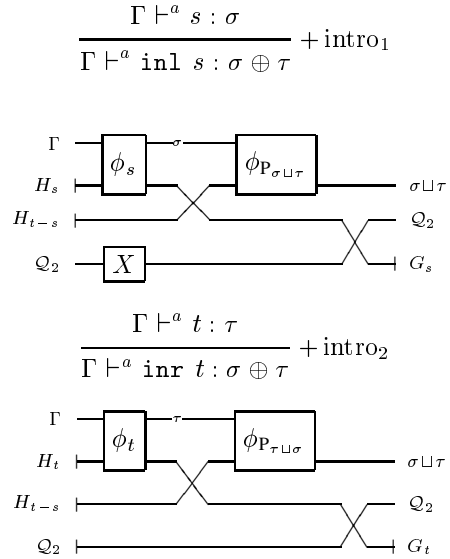
where X is negation.

We define $\mathtt{qtrue}^X = \mathtt{inl}\ ()^X : \mathcal{Q}_2$ and $\mathtt{qfalse}^X = \mathtt{inr}\ ()^X : \mathcal{Q}_2$. To be able to interpret case expressions we introduce a biconditional operation on unitary operators. Given $\phi, \psi \in A \multimap_{\mathrm{unitary}} B$ we construct

$$\phi | \psi \in \mathcal{Q}_2 \otimes A \multimap_{\mathrm{unitary}} \mathcal{Q}_2 \otimes B$$

by the following matrix

$$\begin{array}{llll}
\phi | \psi & (\mathrm{true}, a)\ (\mathrm{true}, b) & = & \phi\, a\, b \\
\phi | \psi & (\mathrm{false}, a)\ (\mathrm{false}, b) & = & \psi\, a\, b \\
\phi | \psi & (x, a)\ (y, b) & = & 0 \quad \mathrm{everywhere\ else}
\end{array}$$

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

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

We have $[\![t]\!] \in \mathbf{FQC}\,[\![\Delta \otimes \sigma]\!]\,[\![\rho]\!]$ and $[\![u]\!] \in \mathbf{FQC}\,[\![\Delta \otimes \tau]\!]\,[\![\rho]\!]$. By padding the input we turn them into $\lceil[\![t]\!]\rceil, \lceil[\![u]\!]\rceil \in \mathbf{F\,QC}[\![\Delta \otimes (\sigma \sqcup \tau)]\!]\,[\![\rho]\!]$. There is no reason why the size of the associated heap and garbage should be the same, however, we have that $H_t + G_u = H_u + G_t$ and hence we can stretch both maps uniformly to $H = H_t \sqcup H_u$ and $G = G_t \sqcup G_u$ giving rise to $\phi_{\lceil[\![t]\!]\rceil}$ and $\phi_{\lceil[\![u]\!]\rceil}$ of identical dimensions. Hence we can apply the choice operator to construct $\psi = \phi_{\lceil[\![t]\!]\rceil}|\phi_{\lceil[\![u]\!]\rceil}$, and with some plumbing we obtain:
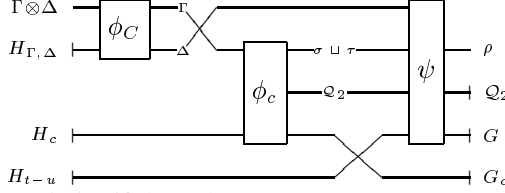
We can derive if-then-else as
   if b **then** t **else** $u =$
      case b **of** $\{\,inl\,\_ \Rightarrow t \mid inr\,\_ \Rightarrow u\,\}$
and use this to implement a form of negation:
   $mnot : \mathbf{Q_2} \multimap \mathbf{Q_2}$
   $mnot\ x = \mathbf{if}\ x\ \mathbf{then}\ \mathrm{qfalse}\ \mathbf{else}\ \mathrm{qtrue}$
However, this program will measure the qubit 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 $\mathrm{qtrue}^{\{\}} \perp \mathrm{qfalse}^{\{\}}$. Hence, we introduce the strict case by:

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

It turns out that there is no sensible way to define $\mathtt{case}^\circ$ if σ and τ have different sizes. Hence we define the orthogonality judgement in a way that it only succeeds, if $|\sigma| = |\tau|$ and hence $[\![\sigma]\!] = [\![\tau]\!]$.

To define the interpretation, we have to exploit the data from the orthogonality judgement $[\![t \perp u]\!] = (S, f, g, \psi)$ where $\psi \in S \otimes \mathcal{Q}_2 \multimap_{\mathrm{unitary}} [\![\rho]\!]$ and $f, g \in \mathbf{F\,QC}\,([\![\Delta]\!] \otimes [\![\sigma]\!])\,S$. We note that both morphisms must have the same heap and hence we can construct

$$\phi_f|\phi_g \in \mathbf{F\,QC}^\circ\,(\mathcal{Q}_2 \otimes [\![\Delta]\!] \otimes [\![\sigma]\!])\,(\mathcal{Q}_2 \otimes S).$$

Now, the main observation is that we just have to apply the unitary operator $\phi_{t \perp u}$ to make the qubit disappear, leading

to the following diagram:
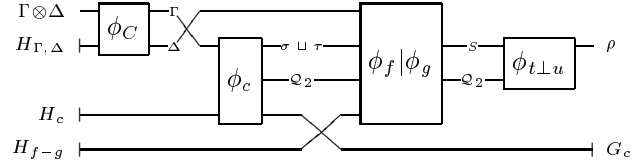
Note that we only allow strict terms in the branches of a strict case. In a previous draft of this paper we tried to be more liberal, however, this causes problems because the qubit we are branching over can be indirectly measured by the garbage. This problem was pointed out by Peter Selinger.

Using the decoherence-free version $\mathbf{if}^\circ$ we can implement standard reversible and hence quantum operations such as $qnot$:
   $qnot : \mathbf{Q_2} \multimap \mathbf{Q_2}$
   $qnot\ x = \mathbf{if}^\circ\ x$
         then qfalse
         else qtrue
and the conditional not $cnot$:
   $cnot : \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \otimes \mathbf{Q_2}$
   $cnot\ c\ x = \mathbf{if}^\circ\ c$
         then $(\mathrm{qtrue}, qnot\ x)$
         else $(\mathrm{qfalse},\ x)$
and finally the Toffolli operator which is basically a conditional $cnot$:
   $toff : \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \otimes (\mathbf{Q_2} \otimes \mathbf{Q_2})$
   $toff\ c\ x\ y = \mathbf{if}^\circ\ c$
         then $(\mathrm{qtrue}, cnot\ x\ y)$
         else $(\mathrm{qfalse}, (x, y))$

4.6 Superpositions

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

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

The algorithm for the preparation of the one-qubit state to a given degree of precision (which is a parameter of the compilation) can be obtained from the one-qubit case of the Kitaev-Solovay theorem, see [12], page 616-624.

4.7 Orthogonality

Given $\Gamma \vdash t : \sigma$ and $\Delta \vdash u : \sigma$ where $|\Delta| = |\Gamma|$ we define $t \perp u$ by the following rules. The idea of

$t \perp u$ is that there is a boolean observation which tells the two terms apart in every environment. The interpretation $[\![t \perp u]\!] = (S, f, g, \psi)$ is defined by induction over the derivations. We present here a sound but incomplete formalisation of orthogonality, achieving completeness is subject of further work.

$$\frac{\Gamma \vdash^\circ t : \sigma \qquad \Gamma \vdash^\circ u : \tau}{\mathtt{inl}\, t \perp \mathtt{inr}\, u \qquad \mathtt{inr}\, t \perp \mathtt{inl}\, u}$$

Here $\rho = \sigma \oplus \tau$, we set $S = \sigma \sqcup \tau$. In both cases f 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:

$$\frac{t \perp u}{\mathtt{inl}\, t \perp \mathtt{inl}\, u \qquad \mathtt{inr}\, t \perp \mathtt{inr}\, u}$$

Let $\Gamma \vdash^\circ \mathtt{inl}\, t, \mathtt{inl}\, u : \sigma \oplus \tau$ and let (S, f, g, ψ) be the interpretation of $t \perp u$. From this data we are constructing the interpretation of $\mathtt{inl}\, t \perp \mathtt{inl}\, u$ as (S, f', g', ψ'). We set $S' = S \otimes Q_2 \otimes H$ where H is the heap needed by inl. We construct f' and g' by applying inl to l, r on the level of semantics using the appropriate part of S' as the heap. ψ is given by the following diagram:
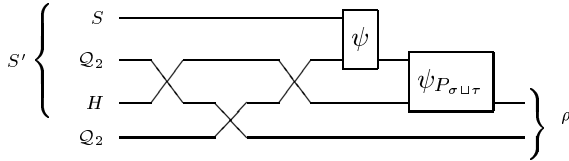
The second rule for inr is done symmetrically.

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

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

The definition of ψ' is given by the following diagram:
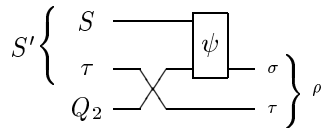
$$\frac{t \perp u \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\{(\lambda_0)t \mid (\lambda_1)u\} \perp \{(\kappa_0)t \mid (\kappa_1)u\}}$$

As before, assume as given the interpretation of $t \perp u$ as (S, f, g, ψ). We construct the interpretation of the conclusion as (S, f, g, ψ') where ψ' is given as
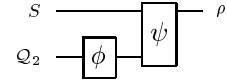
using the rotation $\phi \in Q_2 \multimap_{\mathrm{unitary}} Q_2$ given by

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

4.8 Programs

So far we have introduced a language of expressions. It is straightforward to extend this to a notion of first order programs. E.g. we consider a program Σ to be a sequence of function definitions of the form $F\,\Gamma = t : \sigma$, we have to parameterise every judgement by Σ and require that $\Gamma \vdash_\Sigma t : \sigma$ for the definition to be a wellformed extension of Σ. We also have to introduce a rule for function-application which can just be translated into an iterated let-expression.

5 Conclusions and further work

We have introduced a language for finite quantum programs which uniformly extends a finitary classical language. The classical part of our language may be of interest for its own sake, as it introduces a natural way to compile functional terms into space efficient reversible circuits, due to no unnecessary garbage. This uniformity is one of the main design principles of our language, which, we hope, makes it a natural vehicle to express quantum programming and to develop *quantum thinking*.

We are currently implementing a compiler for QML in Haskell. The compiler produces a representation of quantum circuits which can be simulated (inefficiently, of course) by our own simulator or by using a standard simulator for quantum gates.

There are other design ideas for quantum programming languages. A potential criticism of our approach is that we leave contractions implicit, which is an operation which depends on the choice of basis. However, our type assignment system clearly fixes the places where contractions have to happen, and moreover, and we believe more importantly, it fixes the places where projections, or *tracing*, is happening. A central feature of any quantum programming language seems to be *control of decoherence*.

Having noted this, it seems that decoherence is something you always want to minimise. It is straightforward to design an inference algorithm which infers weakenings $t^{\operatorname{dom}\Gamma}$ such that decoherence is minimised. Maybe this should be the default, which can be overridden if the programmer wants to enforce measurement.

We would like to have an orthogonality judgement which is complete with respect to the denotational semantics. One of the referees commented that we would need an inner product judgement to achieve this. We plan to explore this proposal in future work.

The restriction that $\mathbf{case}^\circ$ is only allowed for balanced coproducts is a direct reaction to the comments of the same referee who pointed out that our previous approach, which involved padding the data, is problematic. Indeed, this problem seems unfixable; if we branch over $Q_1 \otimes Q_2$ the garbage which is created by padding may indirectly measure the qubit we are branching over. Consequently, this approach would not be compositional, and hence should be rejected. The inability to deal with quantum control over arbitrary coproducts is a consequence of the fact that while we deal with quantum data and control, the structure, i.e. the memory allocation, of our data is classical. One way to overcome this limitation would be to use an operational semantics which employs a quantum memory allocation. Such a semantics would have to exploit an infinite state space, and it is questionable whether such a system is physically plausible. Another direction, which seems more feasible, would be to index quantum structures by classical values at compile time.

We have some doubts as to whether the understanding of general recursion and partiality in quantum programming is essential, because partiality is only interesting for systems with infinite state spaces. Moreover, it is not clear how to observe the termination of such a hypthothetical quantum system of unknown runtime without disturbing the computation.

Higher order programming would be a worthwhile addition to reflect the way many quantum algorithms are presented: e.g. the Quantum Fourier Transform can be parameterised by a function on quantum words. Recently, Selinger investigated this problem [16] and it seems that currently no canonical higher order structure on $\mathbf{Super}$ is known. We are investigating whether the category of presheaves over $\mathbf{Super}$ would provide a sound denotational model for higher order quantum computation. This semantics would employ Day's construction to interpret tensor products.

Another line of work is to reap the benefits of the fact that our language uses high level constructs, and develop high level reasoning principles for QML programs. To achieve this, our next goal is to give a direct translation of QML to superoperators which factors through the $\mathbf{F}\,\mathbf{QC}$ semantics presented here. This translation will be based on the implementation of superoperators using arrows [8] in Haskell [21]. A direct consequence of this construction is that the translation presented here is compositional with respect to the extensional equality.

In joint work with Sabry and Vizzotto we are currently developing an equational theory for QML, an algebra for quantum programming, which is sound and complete, with repect to the denotational semantics suggested here. Since the completeness proof relies on *inverting evaluation*, such a proof also gives rise to normalisation; exploiting the approach developed in [3] for a classical system.

Acknowledgements

References

[1] S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th IEEE conference on Logic in Computer Science (LiCS'04)*, 2004.

[2] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, number 1683 in LNCS, pages 453–468, 1999.

[3] T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda^{\to 2}$. In *Functional and Logic Programming*, number 2998 in LNCS, pages 260 – 275, 2004.

[4] P. Arrighi and G. Dowek. Operational semantics for a formal tensorial calculus. *Proceedings of the International Workshop on Quantum Programming Languages*, pages 21–38, 2004.

[5] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[6] L. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physics Review Letters*, 79(2):325–328, 1997.

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

[8] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[9] 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.

[10] C. Morgan and A. McIver. pgcl: Formal reasoning for random algorithms. *South African Computer Journal*, 1999.

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

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

[13] J. C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM Press, 1978.

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

[15] P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[16] P. Selinger. Towards a semantics for higher-order quantum computation. *Proceedings of the International Workshop on Quantum Programming Languages*, pages 127–143, 2004.

[17] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. To appear in Proceedings of the International Conference on Typed Lambda Calculi and Applications, 2005.

[18] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*. CA: IEEE Press, 1994.

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

[20] A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal of Computing*, 33:1109–1135, 2004.

[21] J. K. Vizzotto, T.Altenkirch, and A. Sabry. Structuring quantum effects: Superoperators as arrows. Submitted for publication, 2005.

[22] P. Zuliani. *Quantum Programming*. PhD thesis, Oxford University, 2001.