# 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 **FQC** of finite quantum computations, which provides a constructive semantics of irreversible quantum computations realizable 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 [10] and Grover [2] 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 **FQC**, which we introduce here. The **FQC** 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:

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

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

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

We can read $H$ as an operation which, depending on its input qbit $x$, returns one of two superpositions of a qbit. We can also easily calculate that applying $H$ 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 following algorithm to determine whether to bits, represented as qbits, are equal, which is based on Deutsch' algorithm (see [6], pp.32):

$$
\begin{aligned}
&\mathit{eq} : \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \\
&\mathit{eq}\ a\ b = \mathbf{let}\ (x, y) = \mathbf{if}^\circ \{\mathrm{qfalse} \mid \mathrm{qtrue}\} \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ (\mathrm{qtrue}, \mathbf{if}^\circ\ a \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ (\{\mathrm{qfalse} \mid (-1)\ \mathrm{qtrue}\}, (\mathrm{qtrue}, b)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\{(-1)\ \mathrm{qfalse} \mid \mathrm{qtrue}\}, (\mathrm{qfalse}, b))) \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathrm{qfalse}, \mathbf{if}^\circ\ b \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ (\{(-1)\ \mathrm{qfalse} \mid \mathrm{qtrue}\}, (a, \mathrm{qtrue})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\{\mathrm{qfalse} \mid (-1)\ \mathrm{qtrue}\}, (a, \mathrm{qfalse}))) \\
&\qquad\qquad \mathbf{in}\ \mathit{had}\ x
\end{aligned}
$$

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 and by offering different if-then-else (or, generally, case) operators, one that measures the qbit, **if**, and a second, $\mathbf{if}^\circ$, that doesn't – but which can only be used in certain situations.

## 2 Related work

Peter Selinger's influential paper [8] 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. Our approach can be summarized as *quantum control* and *quantum data*, and goes beyond Selinger's approach by offering high level structures for operating on quantum data. It is conceivable that the two approaches may be combined leading to a hybrid language which uses QML expressions for operations on quantum terms.

There are a number of papers on simulating or integrating quantum programming within conventional functional programming, e.g. [5] and [7]. The paper by Sabry introduces an elegant approach to structure the access to quantum data within a functional language. This is also useful when simulating quantum programs in a classical setting. However, it also follows in principle the paradigm of *classical control* and *quantum data*.

Yet another approach was suggested by Sanders and Zuliani [13], which extends the probabilistic guarded command language [4] by quantum registers and operations on quantum registers. Yet like all the other approaches suggested so far, it doesn't offer high level operations on quantum data.

Andre van Tonder has proposed a quantum $\lambda$-calculus incorporating higher order [11, 12] programs, however measurements are not treated explicitely and he also follows the *classical control* and *quantum data* paradigm.

## 3 Finite classical and quantum computation

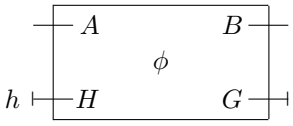
It is frequently emphasised that quantum computation relies on reversibility because quantum physics models reversible processes. This is true, but the same holds for classical computation — whether we base our notion of computation on Newtonian physics or Maxwellian electrodynamics, the underlying physical processes are reversible. Hence we should explain irreversible classical computation based on a reversible mechanism. 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 **FQC** of finite quantum computations and, for purposes of comparison, the category **FCC** of finite classical computations[1]. We will interpret QML programs by **FQC** 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 $A$. A reversible finite computation, that is a closed computational system, is modelled by a reversible operation $\phi$, which is a bijection of finite sets in the classical case, and a unitary operator on the Hilbert

[1] **FCC** may be viewed as a categorical account of a finite version of Bennet's results [1]

spaces in the quantum case. We write $A \multimap_{\text{unitary}} B$ for unitary operators, 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:



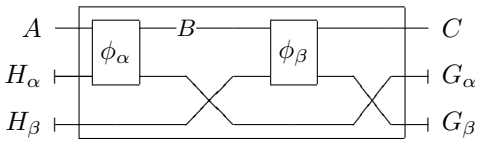

Note that in the above diagram heap inputs are initialised with a $\vdash$, and garbage outputs are terminated with a $\dashv$. To summarise, given finite sets $A, B$ a morphism $(H, h, G, \phi) \in \mathbf{FCC}\, A\, B$ is given by:

- 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 $(H, h, G, \phi) \in \mathbf{FQC}\, A\, B$ 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,
- 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 $\alpha$ and $\beta$:

$$\alpha = (H_\alpha, h_\alpha, G_\alpha, \phi_\alpha) \in \mathbf{FCC}\, A\, B$$
$$\beta = (H_\beta, h_\beta, G_\beta, \phi_\beta) \in \mathbf{FCC}\, B\, C$$

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

$$H = H_\alpha \times H_\beta$$
$$h = (h_\alpha, h_\beta)$$
$$G = G_\alpha \times G_\beta$$
$$\phi = (G_\alpha \times \phi_\beta) \circ (H_\beta \times \phi_\alpha)$$

Note that we have omitted some obvious symmetric monoidal isomorphisms for $\times$ from the definition of $\phi$.

We leave it to the reader to construct the identity computation.

Analogously, given morphisms

$$\alpha = (H_\alpha, h_\alpha, G_\alpha, \phi_\alpha) \in \mathbf{FQC}\, A\, B$$
$$\beta = (H_\beta, h_\beta, G_\beta, \phi_\beta) \in \mathbf{FQC}\, B\, C$$

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 **FCC**, a morphism $\alpha = (H, h, G, \phi) \in \mathbf{FCC}\, A\, B$ gives rise to a function on finite sets $\mathrm{U}_{\mathbf{FCC}}\, \alpha \in A \to B$ by

$$\begin{array}{ccc} A \times H & \xrightarrow[\phi]{} & B \times G \\ \uparrow{\scriptstyle (-,h)} & & \downarrow{\scriptstyle \pi_1} \\ A & \xrightarrow[\mathrm{U}_{\mathbf{FCC}}\, \alpha]{} & B \end{array}$$

How do we do this for **FQC**? 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 self-adjoint 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 [3], pp. 136 or [8] for details. Given $\alpha = (H, h, G, \phi) \in \mathbf{FQC}\, A\, B$ we write $\widehat{\phi} \in A \otimes H \multimap_{\mathrm{super}} B \otimes G$ 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 $\widetilde{h} \in \mathrm{Dens}\, H$ by $\widetilde{h} = |h\rangle \langle h|$. Combining this with the partial trace operator $\mathrm{tr}_G \in B \otimes G \multimap_{\mathrm{super}} B$ we obtain $\mathrm{U}_{\mathbf{FQC}}\, \alpha \in A \multimap_{\mathrm{super}} B$ by

$$\begin{array}{ccc} A \otimes H & \xrightarrow[\widehat{\phi}]{} & B \otimes G \\ \uparrow{\scriptstyle -\otimes \widetilde{h}} & & \downarrow{\scriptstyle \mathrm{tr}_G} \\ A & \xrightarrow[\mathrm{U}_{\mathbf{FQC}}\, \alpha]{} & B \end{array}$$

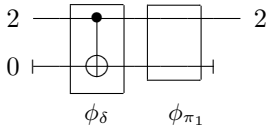
in the category of superoperators.

We say that two computations $\alpha, \beta \in F\,A\,B$ are extensionally equal ($\alpha =_{\text{ext}} \beta$), if the induced maps are equal; $\mathrm{U}_F\,\alpha = \mathrm{U}_F\,\beta$ where $F \in \{\mathbf{FCC}, \mathbf{FQC}\}$. We redefine the homsets of $\mathbf{FCC}, \mathbf{FQC}$ as the quotients of the underlying representation by extensional equality. We have verified that composition respects extensional equality but omit the proof for reasons of space.

As a consequence of our definition we obtain that the assignment of maps to computations gives rise to forgetful functors $\mathrm{U}_{\mathbf{FCC}} \in \mathbf{FCC} \to \mathbf{FinSet}$ and $\mathrm{U}_{\mathbf{FQC}} \in \mathbf{FQC} \to \mathbf{Super}$. Both functors are full and faithful. Hence, our categories **FCC** 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 $\mathbf{FCC}^\circ, \mathbf{FQC}^\circ$ of *strict* morphisms. We have shown that the image of $\mathbf{FCC}^\circ$ under $\mathrm{U}_{\mathbf{FCC}}$ are the injective maps, however $\mathbf{FQC}^\circ$ does not classify monos in **Super** [2].

While **FQC** and **FCC** are very similar indeed, the fact that **FQC** 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 $\delta \in 2 \to 2 \times 2$ (this becomes $\mathcal{Q}_2 \multimap \mathcal{Q}_2 \otimes \mathcal{Q}_2$ in the quantum case) be defined as $\delta\, x = (x, x)$; which can be easily realised by a CNOT gate (either classically or quantum). We can compose this with $\pi_1 \in 2 \times 2$ which leads to the following picture:




Clearly, classically we have just defined an inefficient version of the identity $\pi_1 \circ \delta = \mathrm{I}$; 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 qbit. 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 $\delta$ copies a qbit, 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 $\delta$. We claim that this is a natural explanation of contraction because it is completely uniform in both the classical

[2] Thanks to Peter Selinger for this observation.

and the quantum case. Indeed, classical functional languages do not implement contraction by copying data either. $\delta$ maps pure states to pure states, indeed it shares this behaviour with any computation which doesn't produce garbage; i.e. where $G = 1$. In contrast, operations like $\pi_1$ 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.

### 4.1 Typing rules

We will only present the typed syntax of QML, which is based on strict linear logic. 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 *dynamic semantics*, introduced in the next section. 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 qbit 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 $\{(\mathrm{qtrue}, \mathrm{qtrue}) \mid (\mathrm{qfalse}, \mathrm{qfalse})\}$, 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 $\Gamma \vdash t : \sigma$ and $\Gamma \vdash^\circ t : \sigma$ for decoherence-free terms, we embed $\vdash$ in $\vdash^\circ$:

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

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

$\Gamma$ is a context, i.e. a function from a finite set of variables $\mathrm{dom}\,\Gamma$ 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 $\Gamma$ extended by $x : \tau$. This operation is only defined if $\Gamma$ does not already assign a type to $x$. $t$ is a term of our language, and we introduce the syntax when we present the typing rules, and $\sigma$ is a type.

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

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

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

### 4.2 Denotational semantics

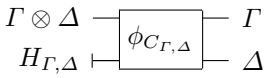
We assign to every type $\sigma$ the number $|\sigma|$ which is the size of a quantum register needed to store elements of $\sigma$, we also interpret expressions of the form $\sigma \sqcup \tau$:

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

The interpretation of a type is the **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 **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. Let $\Gamma \vdash t : \sigma$ and $\Delta \vdash u : \sigma$. Then the interpretation of $t \perp u$ is $[\![t \perp u]\!] = (S, \phi)$ where $S$ is an object of $\mathbf{FQC}$, $\phi \in \mathcal{Q}_2 \otimes S \multimap_{\text{unitary}} [\![\sigma]\!]$, such that there are $f \in \mathbf{FQC}\,[\![\Gamma]\!]\,S$ and $g \in \mathbf{FQC}\,[\![\Gamma]\!]\,S$ such that $[\![t]\!] = \phi \circ (\mathtt{qtrue} \otimes -) \circ f$ and $[\![u]\!] = \phi \circ (\mathtt{qfalse} \otimes -) \circ g$.

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

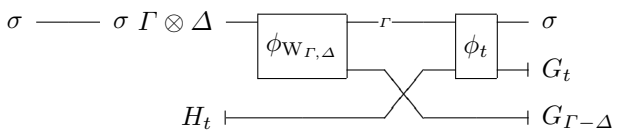



by induction over the definition of $\Gamma \otimes \Delta$: If a variable $x : \sigma$ appears in both contexts we have to use $\delta_\sigma \in \mathbf{FQC}^\circ\,[\![\sigma]\!]\,([\![\sigma]\!] \otimes [\![\sigma]\!])$ which generalises $\delta_2$, discussed earlier, by applying it in parallel to all qbits. All the other cases can be dealt with by applying monoidal isomorphisms. Similarily, we define an explicit weakening operator $\mathrm{W}_{\Gamma,\Delta} \in \mathbf{FQC}\,[\![\Gamma \otimes \Delta]\!]\,([\![\Gamma]\!])$.

### 4.3 Structural rules

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

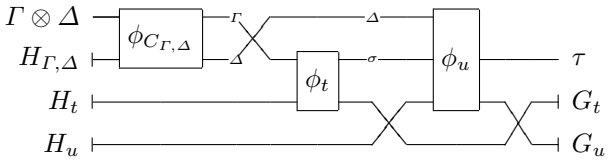
$$\frac{}{x : \sigma \vdash^\circ x : \sigma}\ \text{var} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \otimes \Delta \vdash t^{\mathrm{dom}\,\Delta} : \sigma}\ \text{weak}$$



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

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

$\circ \sqcap \circ = \circ$ 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:

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

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

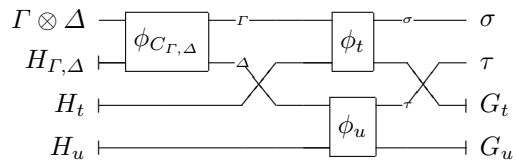
### 4.4 Rules for $\otimes$

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:

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

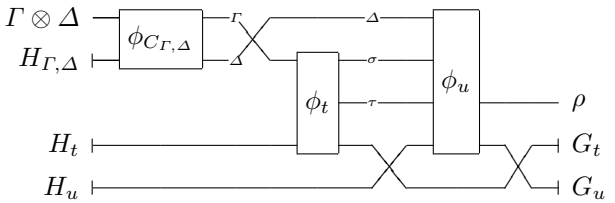
The interpretation of the rules for 1 in terms of circuits is invisible, since 1 doesn't carry any information. The interpretation of the rules for $\otimes$ is more interesting — the introduction rule simply merges the components

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



The interpretation of the elimination rule is similar to the let-rule:

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




As an example, here is a simple program which swaps two qbits:

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

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

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

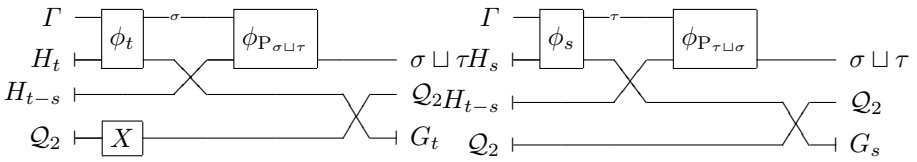
would measure the qbits while swapping them.

### 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.

$$\frac{\Gamma \vdash^a t : \sigma}{\Gamma \vdash^a \texttt{inl}\ \ t : \sigma \oplus \tau} + \text{intro}_1 \qquad \frac{\Gamma \vdash^a t : \tau}{\Gamma \vdash^a \texttt{inr}\ \ t : \sigma \oplus \tau} + \text{intro}_2$$




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

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

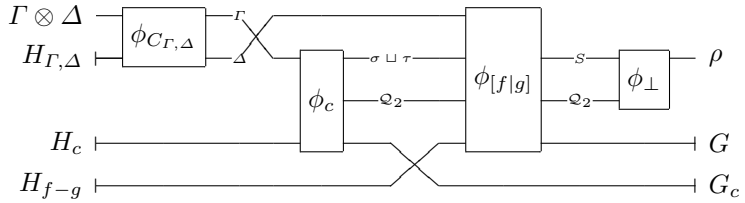
by the following matrix

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

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.

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

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 $\lceil [\![t]\!] \rceil, \lceil [\![u]\!] \rceil \in \mathbf{FQC}\,[\![\Delta \otimes (\sigma \sqcup \tau)]\!]\,[\![\rho]\!]$. Note, we have, for the underlying unitary operators, that the heaps $H$ and garbage $G$ are equal, because unitary operators exist only between spaces with isomorphic bases. Hence we can apply the choice operator to construct $\phi_{[t|u]} = [\phi_{\lceil [\![t]\!] \rceil} | \phi_{\lceil [\![u]\!] \rceil}]$, 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:

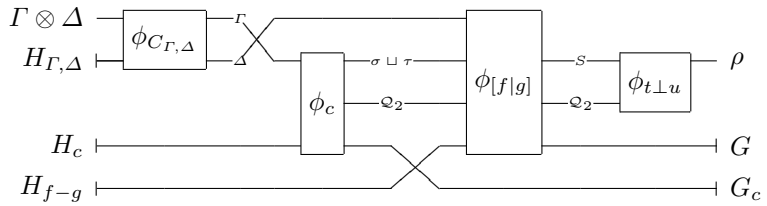
$mnot : \mathbf{Q_2} \multimap \mathbf{Q_2}$

$mnot\ x =$ **if** $x$ **then** qfalse **else** 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 orthogonality judgement later. Intuitively, $t \perp u$ holds if the outputs $t$ and $u$ are always orthogonal, e.g. we will be able to derive $\mathtt{qtrue}^{\{\}} \perp \mathtt{qfalse}^{\{\}}$. Hence, we introduce the strict case by:

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

We have to exploit the data from the orthogonality judgement $[\![t \perp u]\!] = (S, \phi)$ where $\phi \in S \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} [\![\rho]\!]$. Applying the same padding technique as before we obtain $\lceil f \rceil, \lceil g \rceil \in \mathbf{FQC}\,([\![\Delta]\!] \otimes [\![\sigma \sqcup \tau]\!])\,S$ and from this $\phi_{[f|g]} = [\phi_{\lceil f \rceil} | \phi_{\lceil g \rceil}]$. Now, the main observation is that we just have to apply the unitary operator $\phi_{t \perp u}$ to make the qbit 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 qbit 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*:

$$\begin{aligned}
&\mathit{qnot} : \mathbf{Q_2} \multimap \mathbf{Q_2} \\
&\mathit{qnot}\ x = \mathbf{if}^\circ\ x \\
&\qquad\qquad \mathbf{then}\ \text{qfalse} \\
&\qquad\qquad \mathbf{else}\ \text{qtrue}
\end{aligned}$$

and the conditional not *cnot*:

$$\begin{aligned}
&\mathit{cnot} : \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \otimes \mathbf{Q_2} \\
&\mathit{cnot}\ c\ x = \mathbf{if}^\circ\ c \\
&\qquad\qquad\quad \mathbf{then}\ (\text{qtrue}, \mathit{qnot}\ x) \\
&\qquad\qquad\quad \mathbf{else}\ (\text{qfalse},\ x)
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{toff} : \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \multimap \mathbf{Q_2} \otimes (\mathbf{Q_2} \otimes \mathbf{Q_2}) \\
&\mathit{toff}\ c\ x\ y = \mathbf{if}^\circ\ c \\
&\qquad\qquad\quad \mathbf{then}\ (\text{qtrue},\ \mathit{cnot}\ x\ y) \\
&\qquad\qquad\quad \mathbf{else}\ \ (\text{qfalse}, (x, y))
\end{aligned}$$

### 4.6 Superpositions

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

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

The algorithm for the preparation of the one-qbit state to a given degree of precision (which is a parameter of the compilation) can be obtained from the one-qbit case of the Kitaev-Solovay theorem, see [6], page 616-624.

### 4.7 Orthogonality

The idea of $t \perp u$ is that there is a boolean observation which tells the two terms apart in every environment. Given $\Gamma \vdash t, u : \rho$, the interpretation $[\![t \perp u]\!] = (S_{t\perp u}, \phi_{t\perp u})$, where $\phi_{t\perp u} \in S_{t\perp u} \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} [\![\rho]\!]$ is defined by induction over the derivation.

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

Given $\Gamma \vdash^\circ t : \sigma$ and $\Gamma \vdash^\circ u : \tau$, we define $S = \sigma \sqcup \tau$ and $\phi_{t\perp u}$ is simply the identity in the first case but negates the qbit in the second.

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

Just consider the first rule: We set $S_{\texttt{inl}\ t\perp\texttt{inl}\ u} = S_{t\perp u} \otimes \mathcal{Q}_2$, to derive $\phi_{\texttt{inl}\ t\perp\texttt{inl}\ u} \in (S_{t\perp u} \otimes \mathcal{Q}_2) \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} \mathcal{Q}_2 \otimes [\![\sigma \sqcup \tau]\!]$ from $\phi_{t\perp u} \in S_{t\perp u} \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} [\![\rho]\!]$ we just have to swap two qbits. The other case proceeds analogously.

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

Again we only consider the first rule: Given $\Gamma \vdash^\circ t, u : \sigma$ and $\Gamma \vdash^\circ v, w : \tau$. We set $S_{(t,v)\perp(u,w)} = S_{t\perp u} \otimes \tau$, to derive $\phi_{(t,v)\perp(u,w)} \in S_{t\perp u} \otimes \tau \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} [\![\sigma \otimes \tau]\!]$ from $\phi_{t\perp u} \in S_{t\perp u} \otimes \mathcal{Q}_2 \multimap_{\text{unitary}} [\![\rho]\!]$ requires just basic rewiring.

$$\frac{t_0 \perp u_1 \quad t_1 \perp u_0 \quad \lambda_0^* \kappa_1 = -\lambda_1^* \kappa_0}{\{(\lambda_0) t_0 \mid (\lambda_1) t_1\} \perp \{(\kappa_0) u_0 \mid (\kappa_1) u_1\}}$$

We reduce this case to the situation where we have two orthogonal qbits, i.e.

$$\{(\lambda_0)\texttt{qtrue} \mid (\lambda_1)\texttt{qfalse}\} \perp \{(\kappa_0)\texttt{qtrue} \mid (\kappa_1)\texttt{qfalse}\}$$

where $\lambda_0^* \kappa_1 = -\lambda_1^* \kappa_0$, in which case the matrix

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

gives rise to $\phi_{\{(\lambda_0)\texttt{qtrue}\mid(\lambda_1)\texttt{qfalse}\}\perp\{(\kappa_0)\texttt{qtrue}\ \mid\ (\kappa_1)\texttt{qfalse}\}}$. The general situation can be reduced to this via some plumbing.

### 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 $\Sigma$ to be a sequence of function definitions of the form $F\,\Gamma = t : \sigma$, we have to parameterise every judgement by $\Sigma$ and require that $\Gamma \vdash_\Sigma t : \sigma$ for the definition to be a wellformed extension of $\Sigma$. 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 because it introduces a natural way to compile functional terms into space efficient reversible circuits, as we avoid creating 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 (inefficently, 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 reasonable quantum programming language is the *control of decoherence*.

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

There are obvious directions in which to continue this work. 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 [9] and it seems that currently no canonical higher order structure on **Super** is known. Using our principle of uniformity with classical computing the space generated by the function space of the bases appears to be a natural candidate to model higher order types. We leave the investigation of this proposal to future work.

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. This may be guided from an investigation of the categorical infrastructure of the categories involved — in particular we would like to make the analogy of **FCC** and **FQC** precise. A first observation in this direction may be that that the category of bijections on finite sets is the free symmetric monoidal category over one generator, while it seems possible to describe the corresponding quantum category, the category of unitary operators on finite dimensional Hilbert spaces[3], as freely generated by a collection of one-qbit quantum gates.

It has been suggested to introduce infinite quantum data structures to model computations of arbitrary size. Another direction which seems more feasible would be to index quantum structures by classical values at compile time. We have some doubts whether the understanding of recursion in quantum programming is essential since it is not clear how to observe the termination of a quantum program of unknown runtime without disturbing the computation.

## Acknowledgements

We would like to acknowledge interesting discussions on the subject of this paper with John Baez, Slava Belavkin, Martin Hofmann, Conor McBride and Thomas Streicher. Amr Sabry and Juliana Vizotti provided extensive feedback on previous drafts of this paper. Peter Selinger pointed out a serious flaw in the definition of $\mathbf{case}^\circ$ and refuted our conjecture that strict maps classify monos in **Super**.

## References

1. C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.
2. L. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physics Review Letters*, 79(2):325–328, 1997.
3. M. Hirvensalo. *Quantum Computating*. Springer-Verlag NewYork, Inc., 2001.
4. C. Morgan and A. McIver. pgcl: Formal reasoning for random algorithms. *South African Computer Journal*, 1999.
5. S.-C. Mu and R. S. Bird. Quantum functional programming. In *2nd Asian Workshop on Programming Languages and Systems*, 2001.
6. M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.
7. A. Sabry. Modeling quantum computing in haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 39–49. ACM Press, 2003.
8. P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 2004.
9. P. Selinger. Towards a semantics for higher-order quantum computation. *Proceedings of the 2nd International Workshop on Quantum Programming Languages*, 2004.
10. P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*. CA: IEEE Press, 1994.

[3] To be precise: a dense subspace of the uncountable homsets.

11. A. van Tonder. A lambda calculus for quantum computation. quant-ph/0307150, 2003. to appear in SIAM Journal of Computing.
12. A. van Tonder. Quantum computation, categorical semantics and linear logic. quant-ph/0312174, 2003.
13. P. Zuliani. *Quantum Programming*. PhD thesis, Oxford University, 2001.