A lambda calculus for quantum computation with classical control

Peter Selinger, Benoît Valiron

Department of Mathematics and Statistics, University of Ottawa,
Ottawa, Ontario K1N 6N5, Canada

Abstract The objective of this paper is to develop a functional programming language for quantum computers. We develop a lambda calculus for the classical control model, following the first author's work on quantum flow-charts. We define a call-by-value operational semantics, and we give a type system using affine intuitionistic linear logic. The main results of this paper are the safety properties of the language and the development of a type inference algorithm.

1 Introduction

The objective of this paper is to develop a functional programming language for quantum computers. Quantum computing is a theory of computation based on the laws of quantum physics, rather than of classical physics. Quantum computing has become a fast growing research area in recent years. For a good introduction, see e.g. [9,10].

Due to the laws of quantum physics, there are only two kinds of basic operations that one can perform on a quantum state, namely *unitary transformations* and *measurements*. Many existing formalisms for quantum computation put an emphasis on the former, i.e., a computation is understood as the evolution of a quantum state by means of unitary gates. Measurements are usually performed at the end of the computation, and outside of the formalism. In these models, a quantum computer is considered as a purely quantum system, i.e., without any classical parts. One example of such a model is the quantum Turing machine [3,6], where the entire machine state, including the tape, the finite control, and the position of the head, is assumed to be in quantum superposition. Another example is the quantum lambda calculus of van Tonder [14,15], which is a higher-order, purely quantum language without an explicit measurement operation.

On the other hand, one might imagine a model of a quantum computer where unitary operations and measurements can be interleaved. One example is the so-called *QRAM model* of Knill [8], which is also described by Bettelli, Calarco and Serafini [4]. Here, a quantum computer consists of a classical computer connected to a quantum device. In this configuration, the operation of the machine is controlled by a classical program which emits a sequence of instructions to the quantum device for performing measurements and unitary operations. In such a model, the control structures of the machine are classical, and only the data being operated upon is quantum. This situation is summarized by the slogan "quantum data, classical control" [12]. Several programming languages have been proposed to deal with such a model [4,11]. The present paper is based on the work of [12].

In this paper, we propose a *higher-order* quantum programming language, i.e., one in which functions can be considered as data. A program is a lambda term, possibly with some quantum data embedded inside. The basic idea is that lambda terms encode the control structure of a program, and thus, they would be implemented classically, i.e., on the classical device of the QRAM machine. However, the data on which the lambda terms act is possibly quantum, and is stored on the QRAM quantum device.

Because our language combines classical and quantum features, it is natural to consider two distinct basic data types: a type of *classical bits* and a type of *quantum bits*. They behave very differently. For instance, a classical bit can be copied as many times as needed. On the other hand, a quantum bit cannot be duplicated, due to the well-known *no cloning property* of quantum states [9,10]. However, quantum data types are very powerful, due to the phenomena of quantum superposition and entanglement.

The semantics described in this paper is operational; a program is an abstract machine with reductions rules. The reduction rules are probabilistic.

Some care is needed when defining a type system for higher-order quantum functions. This is because the question of whether a function is duplicable or not cannot be directly seen from the types of its arguments or of its value, but rather it depends on the types of any free variables occurring in the function definition. As it turns out, the appropriate type system for higher-order quantum functions in our setting is affine intuitionistic linear logic.

We also address the question of finding a type inference algorithm. Using the remark that a linear type is a decoration of an intuitionistic one, we show that the question of deciding whether or not a program is valid can be reduced to the question of finding an intuitionistic type for it and to explore a finite number of linear decorations for the type.

This work is based on the second author's Master's thesis [13].

2 Quantum computing basics

We briefly recall the basic definitions of quantum computing; please see [9,10] for a complete introduction to the subject. The basic unit of information in quantum computation is a quantum bit or *qubit*. The state of a single qubit is a a normalized vector of the 2-dimensional Hilbert space $\mathbb{C}^2$. We denote the standard basis of $\mathbb{C}^2$ as $\{|0\rangle, |1\rangle\}$, so that the general state of a single qubit can be written as $\alpha|0\rangle + \beta|1\rangle$, where $|\alpha|^2 + |\beta|^2 = 1$.

The state of n qubits is a normalized vector in $\otimes_{i=1}^n \mathbb{C}^2 \cong \mathbb{C}^{2^n}$. We write $|xy\rangle = |x\rangle \otimes |y\rangle$, so that a standard basis vector of $\mathbb{C}^{2^n}$ can be denoted $|\ulcorner i \urcorner^n\rangle$, where $\ulcorner i \urcorner^n$ is the binary representation of i in n digits, for $0 \leqslant i < 2^n$. As a special case, if $n = 0$, we denote the unique standard basis vector in $\mathbb{C}^1$ by $|\rangle$.

The basic operations on quantum states are unitary operations and measurements. A unitary operation maps an n-qubit state to an n-qubit state, and is given by a unitary $2^n \times 2^n$-matrix. It is common to assume that the computational model provides a certain set of built-in unitary operations, including for example the *Hadamard gate* H and the *controlled not-gate* $CNOT$, among others:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad CNOT = \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix}.$$

The measurement acts as a projection. When a qubit $\alpha|0\rangle + \beta|1\rangle$ is measured, the observed outcome is a classical bit. The two possible outcomes 0 and 1 are observed with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. Moreover, the state of the qubit is affected by the measurement, and collapses to $|0\rangle$ if 0 was observed, and to $|1\rangle$ if 1 was observed. More generally, given an n-qubit state $|\phi\rangle = \alpha_0|0\rangle \otimes |\psi_0\rangle + \alpha_1|1\rangle \otimes |\psi_1\rangle$, where $|\psi_0\rangle$ and $|\psi_1\rangle$ are normalized $(n-1)$-qubit states, then measuring the leftmost qubit results in the answer i with probability $|\alpha_i|^2$, and the resulting state will be $|i\rangle \otimes |\psi_i\rangle$.

3 The untyped quantum lambda calculus

3.1 Terms

Our language uses the notation of the intuitionistic lambda calculus. For a detailed introduction to the lambda calculus, see e.g. [2]. We start from a standard lambda calculus with booleans and finite products. We extend this language with three special quantum operations, which are new, $meas$, and built-in unitary gates. new maps a classical bit to a quantum bit. $meas$ maps a quantum bit to a classical bit by performing a measurement operation; this is a probabilistic operation. Finally, we assume that there is a set $\mathcal{U}^n$ of built-in n-ary unitary gates for each n. We use the letter U to range over built-in unitary gates. Thus, the syntax of our language is as follows:

$$\begin{array}{llll} \mathit{Term} & M, N, P & ::= & x \mid MN \mid \lambda x.M \mid \mathit{if}\ M\ \mathit{then}\ N\ \mathit{else}\ P \mid 0 \mid 1 \mid \mathit{meas} \\ & & & \mid \mathit{new} \mid U \mid * \mid \langle M, N\rangle \mid \mathit{let}\ \langle x, y\rangle = M\ \mathit{in}\ N, \end{array}$$

We follow Barendregt's convention for identifying terms up to α-equivalence. We also sometimes use the shorthand notation $\langle M_1, \dots, M_n\rangle = \langle M_1, \langle M_2, \dots\rangle\rangle$.

3.2 Programs

The reader will have noticed that we have not provided a syntax for constant quantum states such as $\alpha|0\rangle + \beta|1\rangle$ in our language. One may ask why we did not allow the insertion of quantum states into a lambda term, such as $\lambda x.(\alpha|0\rangle + \beta|1\rangle)$. The reason is that, in the general case, such a syntax would be insufficient. Consider for instance the lambda term $(\lambda y.\lambda f.fpy)(q)$, where p and q are entangled quantum bits in the state $|pq\rangle = \alpha|00\rangle + \beta|11\rangle$. Such a state cannot be represented locally by replacing p and q with some constant qubit expressions. The non-local nature of quantum states thus forces us to introduce a level of indirection into the representation of a state of a quantum program.

Definition 1. A *program state* is represented by a triple $[Q, L, M]$, where

- Q is a normalized vector of $\otimes_{i=0}^{n-1}\mathbb{C}^2$, for some $n \geqslant 0$
- M is a lambda term,
- L is a function from W to $\{0, \dots, n-1\}$, where $FV(M) \subseteq W \subseteq \mathcal{V}_{term}$. L is also called the *linking function* or the *qubit environment*.

The purpose of the linking function is to assign specific free variables of M to specific quantum bits in Q. The notion of α-equivalence extends naturally to programs, for instance, the states $[|1\rangle, \{x \mapsto 0\}, \lambda y.x]$ and $[|1\rangle, \{z \mapsto 0\}, \lambda y.z]$ are equivalent. The set of program states, up to α-equivalence, is denoted by $\mathbb{S}$.

Convention 1. In order to simplify the notation, we will often use the following convention: we use p_i to denote the free variable x such that $L(x) = i$. A program $[Q, L, M]$ is abbreviated to $[Q, M']$ with $M' = M[p_{i_1}/x_1] \ldots [p_{i_n}/x_n]$, where $i_k = L(x_k)$.

3.3 Linearity

An important well-formedness property of quantum programs is that quantum bits should always be *uniquely referenced*: roughly, this means that no two variable occurrences should refer to the same physical quantum bit. The reason for this restriction is the well-known no-cloning property of quantum physics, which states that a quantum bit cannot be duplicated: there exists no physically meaningful operation which maps an arbitrary quantum bit $|\phi\rangle$ to $|\phi\rangle \otimes |\phi\rangle$.

Syntactically, the requirement of unique referencing translates into a *linearity condition*: A lambda abstraction $\lambda x.M$ is called *linear* if the variable x is used at most once during the evaluation of M. A well-formed program should be such that quantum data is only used linearly; however, classical data, such as ordinary bits, can of course be used non-linearly. Since the decision of which subterms must be used linearly depends on type information, we will not formally enforce any linearity constraints until we discuss a type system in Section 4; nevertheless, we will assume that all our untyped examples are well-formed in the above sense.

3.4 Evaluation strategy

As is usual in defining a programming language, we need to settle on a reduction strategy. The obvious candidates are call-by-name and call-by-value. Because of the probabilistic nature of measurement, the choice of reduction strategy affects the behavior of programs, not just in terms of efficiency, but in terms of the actual answer computed. We demonstrate this in an example. Let $\mathbf{plus}$ be the boolean addition function, which is definable as $\mathbf{plus} = \lambda xy.\, \mathit{if}\ x\ \mathit{then}\ (\mathit{if}\ y\ \mathit{then}\ 0\ \mathit{else}\ 1)\ \mathit{else}\ (\mathit{if}\ y\ \mathit{then}\ 1\ \mathit{else}\ 0)$. Consider the term $M = (\lambda x.\mathbf{plus}\ x\ x)(\mathit{meas}(H(\mathit{new}\ \ 0)))$.

Call-by-value. Reducing this in the empty environment, using the call-by-value reduction strategy, we obtain the following reductions:

$$\begin{array}{l}
\longrightarrow_{CBV} [|0\rangle, (\lambda x.\mathbf{plus}\ x\ x)(\mathit{meas}(H\ p_0))] \\
\longrightarrow_{CBV} [\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), (\lambda x.\mathbf{plus}\ x\ x)(\mathit{meas}\ \ p_0)] \\
\longrightarrow_{CBV} \left\{ \begin{array}{l} [\ |0\rangle, (\lambda x.\mathbf{plus}\ x\ x)(0)] \\ [\ |1\rangle, (\lambda x.\mathbf{plus}\ x\ x)(1)] \end{array} \right. \longrightarrow_{CBV} \left\{ \begin{array}{l} [\ |0\rangle, \mathbf{plus}\ 0\ 0] \\ [\ |1\rangle, \mathbf{plus}\ 1\ 1] \end{array} \right. \longrightarrow_{CBV} \left\{ \begin{array}{l} [\ |0\rangle, 0] \\ [\ |1\rangle, 0] \end{array} \right.
\end{array}$$

each with a probability of $1/2$. Thus, under call-by-value reduction, this program produces the boolean value 0 with probability 1. Note that we have used Convention 1 for writing these program states.

Call-by-name. Reducing the same term under the call-by-name strategy, we obtain in one step $[\ |\rangle, \mathbf{plus}\ (\mathit{meas}(H(\mathit{new}\ \ 0)))\ (\mathit{meas}(H(\mathit{new}\ \ 0)))]$, and then with probability $1/4$, $[\ |01\rangle, 1\ ]$, $[\ |10\rangle, 1\ ]$, $[\ |00\rangle, 0\ ]$ or $[\ |11\rangle, 0\ ]$. Therefore, the boolean output of this function is 0 or 1 with equal probability.

Mixed strategy. Moreover, if we mix the two reduction strategies, the program can even reduce to an ill-formed term. Namely, reducing by call-by-value until $[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), (\lambda x.\mathbf{plus}\ x\ x)(meas\ \ p_0)]$, and then changing to call-by-name, we obtain in one step the term $[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), (\mathbf{plus}\ (meas\ \ p_0)\ (meas\ \ p_0)]$, which is not a valid program since there are 2 occurrences of p_0.
In the remainder of this paper, we will only consider the call-by-value reduction strategy, which seems to us to be the most natural.

3.5 Probabilistic reduction systems

In order to formalize the operational semantics of the quantum lambda calculus, we need to introduce the notion of a probabilistic reduction system.

Definition 2. A *probabilistic reduction system* is a tuple $(X, U, R, prob)$ where X is a set of *states*, $U \subseteq X$ is a subset of *value states*, $R \subseteq (X \setminus U) \times X$ is a set of *reductions*, and $prob : R \to [0, 1]$ is a *probability function*, where $[0, 1]$ is the real unit interval. Moreover, we impose the following conditions:

- For any $x \in X$, $R_x = \{\ x' \mid (x, x') \in R\ \}$ is finite.
- $\sum_{x' \in R_x} prob(x, x') \leqslant 1$

We call $prob$ the one-step reduction, and denote $x \longrightarrow_p y$ to be $prob(x, y) = p$. Let us extend $prob$ to the n-step reduction

$$prob^0(x,y) = \begin{cases} 0 \text{ if } & x \neq y \\ 1 \text{ if } & x = y \end{cases}$$
$$prob^1(x,y) = \begin{cases} prob(x,y) \text{ if } & (x,y) \in R \\ \quad 0 & \text{else} \end{cases}$$
$$prob^{n+1}(x,y) = \sum_{z \in R_x} prob(x,z) prob^n(z,y),$$

and the notation is extended to $x \longrightarrow_p^n y$ to mean $prob^n(x, y) = p$.

We say that y is *reachable in one step with non-zero probability* from x, denoted $x \longrightarrow_{>0} y$ when $x \longrightarrow_p y$ with $p > 0$. We say that y is *reachable with non-zero probability* from x, denoted $x \longrightarrow^*_{>0} y$ when there exists n such that $x \longrightarrow_p^n y$ with $p > 0$.

We can then compute the probability to reach $u \in U$ from x: It is a function from $X \times U$ to $\mathbb{R}$ defined by $prob_U(x, u) = \sum_{n=0}^{\infty} prob^n(x, u)$. The total probability for reaching U from x is $prob_U(x) = \sum_{n=0}^{\infty} \sum_{u \in U} prob^n(x, u)$.

On the other hand, there is also the probability to *diverge* from x, or never reaching anything. This value is $prob_\infty(x) = \lim_{n \to \infty} \sum_{y \in X} prob^n(x, y)$.

Lemma 1. *For all* $x \in X$, $prob_U(x) + prob_\infty(x) \leqslant 1$.

We define the *error probability of* x to be the number $prob_{err}(x) = 1 - prob_U(x) - prob_\infty(x)$.

Definition 3. We can define a notion of equivalence in X:

$$x \approx y \quad \text{iff} \quad \forall u \in U \begin{cases} prob_U(x,u) = prob_U(y,u) \\ prob_\infty(x) = prob_\infty(y) \end{cases}$$

Definition 4. In addition to the notion of reachability with non-zero probability, there is also a weaker notion of reachability, given by R: We will say that y is *reachable* from x if xRy. By the properties of $prob$, $x \longrightarrow_{>0} y$ implies $x \rightsquigarrow y$ with $x \rightsquigarrow y$ for xRy. Let us denote by $\longrightarrow^*$ the relation such that $x \rightsquigarrow^* y$ iff there exists n such that $xR^n y$, with R^n defined as the n-th composition of R. Similarly, $x \longrightarrow^*{}_{>0}\, y$ implies $x \rightsquigarrow^* y$.

Definition 5. In a probabilistic reduction system, a state x is called an *error-state* if $x \notin U$ and $\sum_{x' \in X} prob(x, x') < 1$. An element $x \in X$ is *consistent* if there is no error-state e such that $x \rightsquigarrow^* e$.

Lemma 2. *If x is consistent, then $prob_{err}(x) = 0$. The converse is false.*

Remark 1. We need the weaker notion of reachability $x \rightsquigarrow^* y$, in addition to reachability with non-zero probability $x \longrightarrow_{>0}{}^* y$, because a null probability of getting a certain result is not an absolute warranty of its impossibility. In the QRAM, suppose we have a qubit in state $|0\rangle$. Measuring it cannot theoretically yield the value 1, but in practice, this might happen with small probability, due to imprecision of the physical operations and decoherence. Therefore, when we prove type safety (see Theorem 2), we will use the stronger notion. In short: a type-safe program should not crash, even in the event of random QRAM errors.

3.6 Operational semantics

We will define a probabilistic call-by-value reduction procedure for the quantum lambda calculus. Note that, although the reduction itself is probabilistic, the choice of which redex to reduce at each step is deterministic.

Definition 6. A *value* is a term of the following form:

$$\mathit{Value} \quad V, W \quad ::= \quad x \mid \lambda x.M \mid 0 \mid 1 \mid \mathit{meas} \mid \mathit{new} \mid U \mid * \mid \langle V, W \rangle.$$

The set of *value states* is $\mathbb{V} = \{[Q, L, V] \in \mathbb{S} \mid V \in \mathit{Value}\}$.

The reduction rules are shown in Table 1, where we have used Convention 1 to shorten the description of states. We write $[Q, L, M] \longrightarrow_p [Q', L', M']$ for a single-step reduction of states which takes place with probability p. In the rule for reducing the term $U\langle p_{j_1}, \ldots, p_{j_n}\rangle$, U is an n-ary built-in unitary gate, $j_1, \ldots, j_n$ are pairwise distinct, and Q' is the quantum state obtained from Q by applying this gate to qubits $j_1, \ldots, j_n$. In the rule for measurement, $|Q_0\rangle$ and $|Q_1\rangle$ are normalized states of the form $|Q_0\rangle = \sum_j \alpha_j |\phi_j^0\rangle \otimes |0\rangle \otimes |\psi_j^0\rangle$ and $|Q_1\rangle = \sum_j \beta_j |\phi_j^1\rangle \otimes |1\rangle \otimes |\psi_j^1\rangle$, where ϕ_j^0 and ϕ_j^1 is an i-qubit state (so that the measured qubit is the one pointed to by p_i). In the rule for for new, Q is an n-qubit state, so that $Q \otimes |i\rangle$ is an $(n+1)$-qubit state, and p_n refers to its rightmost qubit.

We define a weaker relation $\rightsquigarrow$. This relation models the transformations that can happen in the presence of decoherence and imprecision of physical operations. We define $[Q, M] \rightsquigarrow [Q', M']$ to be $[Q, M] \longrightarrow_p [Q', M']$, even when $p = 0$, plus the additional rule, if Q and Q' are vectors of equal dimensions: $[Q, M] \rightsquigarrow [Q', M]$.

$$[Q, (\lambda x.M)V] \longrightarrow_1 [Q, M[V/x]]$$

$$\frac{[Q,N] \longrightarrow_p [Q',N']}{[Q,MN] \longrightarrow_p [Q',MN']}$$

$$\frac{[Q,M] \longrightarrow_p [Q',M']}{[Q,MV] \longrightarrow_p [Q',M'V]}$$

$$\frac{[Q,M_1] \longrightarrow_p [Q',M_1']}{[Q,\langle M_1,M_2\rangle] \longrightarrow_p [Q',\langle M_1',M_2\rangle]}$$

$$\frac{[Q,M_2] \longrightarrow_p [Q',M_2']}{[Q,\langle V_1,M_2\rangle] \longrightarrow_p [Q',\langle V_1,M_2'\rangle]}$$

$$[Q, \textit{if } 0 \textit{ then } M \textit{ else } N] \longrightarrow_1 [Q, N]$$

$$[Q, \textit{if } 1 \textit{ then } M \textit{ else } N] \longrightarrow_1 [Q, M]$$

$$[Q, U\langle p_{j_1},\dots,p_{j_n}\rangle] \longrightarrow_1 [Q', \langle p_{j_1},\dots,p_{j_n}\rangle]$$

$$[\alpha|Q_0\rangle + \beta|Q_1\rangle, \textit{meas}\ \ p_i] \longrightarrow_{|\alpha|^2} [|Q_0\rangle, 0]$$

$$[\alpha|Q_0\rangle + \beta|Q_1\rangle, \textit{meas}\ \ p_i] \longrightarrow_{|\beta|^2} [|Q_1\rangle, 1]$$

$$[Q, \textit{new}\ \ 0] \longrightarrow_1 [Q \otimes |0\rangle, p_n]$$

$$[Q, \textit{new}\ \ 1] \longrightarrow_1 [Q \otimes |1\rangle, p_n]$$

$$\frac{[Q,P] \longrightarrow_p [Q',P']}{[Q, \textit{if } P \textit{ then } M \textit{ else } N] \longrightarrow_p [Q', \textit{if } P' \textit{ then } M \textit{ else } N]}$$

$$\frac{[Q,M] \longrightarrow_p [Q',M']}{[Q, \text{let } \langle x_1,x_2\rangle = M \text{ in } N] \longrightarrow_p [Q', \text{let } \langle x_1,x_2\rangle = M' \text{ in } N]}$$

$$[Q, \text{let } \langle x_1,x_2\rangle = \langle V_1,V_2\rangle \text{ in } N] \longrightarrow_1 [Q, N[V_1/x_1, V_2/x_2]]$$

Table 1. Reductions rules of the quantum lambda calculus

Lemma 3. *Let $prob$ be the function such that for $x, y \in \mathbb{S}$, $prob(x, y) = p$ if $x \longrightarrow_p y$ and 0 else. Then $(\mathbb{S}, \mathbb{V}, \rightsquigarrow, prob)$ is a probabilistic reduction system.* □

This probabilistic reduction system has error states, for example, $[Q, H(\lambda x.x)]$ or $[Q, U\langle p_0, p_0\rangle]$. Such error states correspond to run-time errors. In the next section, we introduce a type system designed to rule out such error states.

4 The typed quantum lambda-calculus

We will now define a type system designed to eliminate all run-time errors arising from the reduction system of the previous section. We need base types (such as bit and $qbit$), function types, and product types. In addition, we need the type system to capture a notion of duplicability, as discussed in Section 3.3. We follow the notation of linear logic [7]. By default, a term of type A is assumed to be non-duplicable, and duplicable terms are given the type $!A$ instead. Formally, the set of types is defined as follows, where α ranges over a set of type constants and X ranges over a countable set of type variables:

$$\textit{qType} \quad A, B \quad ::= \quad \alpha \mid X \mid !A \mid (A \multimap B) \mid \top \mid (A \otimes B)$$

Note that, because all terms are assumed to be non-duplicable by default, the language has a linear function type $A \multimap B$ and a linear product type $A \otimes B$. This reflects the fact that there is in general no canonical diagonal function $A \to A \otimes A$. Also, $\top$ is the linear unit type. This will be made more formal in the typing rules below. We write $!^n A$ for $!!!\dots!!A$, with n repetitions of !. We also write A^n for the n-fold tensor product $A \otimes \dots \otimes A$.

4.1 Subtyping

The typing rules will ensure that any value of type $!A$ is duplicable. However, there is no harm in using it only once; thus, such a value should also have type A. For this reason, we define a subtyping relation $<:$ as follows:

$$\frac{}{\alpha <: \alpha}\ (ax) \qquad \frac{}{X <: X}\ (var) \qquad \frac{}{\top <: \top}\ (\top) \qquad \frac{A <: B}{!A <: B}\ (D) \qquad \frac{!A <: B}{!A <: !B}\ (!)$$

$$\frac{A_1 <: B_1 \quad A_2 <: B_2}{A_1 \otimes A_2 <: B_1 \otimes B_2}\ (\otimes) \qquad \frac{A <: A' \quad B <: B'}{A' \multimap B <: A \multimap B'}\ (\multimap)$$

Lemma 4. *For types A and B, if $A <: B$ and $(m = 0) \vee (n \geqslant 1)$, then $!^n A <: !^m B$.* $\square$

Notice that one can rewrite types using the notation:

$$\mathit{qType} \quad A, B \quad ::= \quad !^n\alpha \mid !^n X \mid !^n(A \multimap B) \mid !^n\top \mid !^n(A \otimes B)$$

with $n \in \mathbb{N}$. Using the overall condition on n and m that $(m = 0) \vee (n \geqslant 1)$, the rules can be re-written as:

$$\frac{}{!^n X <: !^m X}\ (var_2) \qquad \frac{}{!^n\alpha <: !^m\alpha}\ (\alpha) \qquad \frac{}{!^n\top <: !^m\top}\ (\top)$$

$$\frac{A_1 <: B_1 \quad A_2 <: B_2}{!^n(A_1 \otimes A_2) <: !^m(B_1 \otimes B_2)}\ (\otimes) \qquad \frac{A <: A' \quad B <: B'}{!^n(A' \multimap B) <: !^m(A \multimap B')}\ (\multimap_2)$$

The two sets of rules are equivalent.

Lemma 5. *The rules of the second set are reversible.* $\square$

Lemma 6. *$(\mathit{qType}, <:)$ is reflexive and transitive. If we define an equivalence relation $\doteqdot$ by $A \doteqdot B$ iff $A <: B$ and $B <: A$, $(\mathit{qType}/\doteqdot, <:)$ is a poset.* $\square$

Lemma 7. *If $A <: !B$, then there exists C such that $A = !C$.* $\square$

Remark 2. The subtyping rules are a syntactic device, and are not intended to catch all plausible type isomorphisms. For instance, the types $!A \otimes !B$ and $!(A \otimes B)$ are not subtypes of each other, although an isomorphism between these types is easily definable in the language.

4.2 Typing rules

We need to define what it means for a quantum state $[Q, L, M]$ to be well-typed. It turns out that the typing does not depend on Q and L, but only on M. We introduce typing judgments of the form $\Delta \rhd M : B$. Here M is a term, B is a qType, and Δ is a typing context, i.e., a function from a set of variables to qType. As usual, we write $|\Delta|$ for the domain of Δ, and we denote typing contexts as $x_1{:}A_1, \ldots, x_n{:}A_n$. As usual, we write $\Delta, x{:}A$ for $\Delta \cup \{x{:}A\}$ if $x \notin |\Delta|$. Also, if $\Delta = x_1{:}A_1, \ldots, x_n{:}A_n$, we write $!\Delta = x_1{:}!A_1, \ldots, x_n{:}!A_n$. A typing judgement is called *valid* if it can be derived from the rules in Table 2.

$$\frac{A <: B}{\Delta, x{:}A \rhd x : B}\ (ax_1) \qquad \frac{A_c <: B}{\Delta \rhd c : B}\ (ax_2)$$

$$\frac{\Gamma_1, !\Delta \rhd P : bit \quad \Gamma_2, !\Delta \rhd M : A \quad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd \mathit{if}\ P\ \mathit{then}\ M\ \mathit{else}\ N : A}\ (\mathit{if})$$

$$\frac{\Gamma_1, !\Delta \rhd M : A \multimap B \quad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd MN : B}\ (app)$$

$$\frac{x{:}A, \Delta \rhd M : B}{\Delta \rhd \lambda x.M : A \multimap B}\ (\lambda_1) \qquad \frac{\text{If } FV(M) \cap |\Gamma| = \emptyset\text{:} \quad \Gamma, !\Delta, x{:}A \rhd M : B}{\Gamma, !\Delta \rhd \lambda x.M : !^{n+1}(A \multimap B)}\ (\lambda_2)$$

$$\frac{!\Delta, \Gamma_1 \rhd M_1 : !^n A_1 \quad !\Delta, \Gamma_2 \rhd M_2 : !^n A_2}{!\Delta, \Gamma_1, \Gamma_2 \rhd \langle M_1, M_2 \rangle : !^n(A_1 \otimes A_2)}\ (\otimes.I) \qquad \frac{}{\Delta \rhd * : !^n \top}\ (\top)$$

$$\frac{!\Delta, \Gamma_1 \rhd M : !^n(A_1 \otimes A_2) \quad !\Delta, \Gamma_2,\ x_1{:}!^n A_1,\ x_2{:}!^n A_2 \rhd N : A}{!\Delta, \Gamma_1, \Gamma_2 \rhd \text{let } \langle x_1, x_2 \rangle = M \text{ in } N : A}\ (\otimes.E)$$

Table 2. Typing rules

The typing rule (ax) assumes that to every constant c of the language, we have associated a fixed type A_c. The types A_c are defined as follows:

$$\begin{array}{lll} A_0 = !bit & A_{new} = !(bit \multimap qbit) & \\ A_1 = !bit & A_{meas} = !(qbit \multimap !bit) & A_U = !(qbit^n \multimap qbit^n) \end{array}$$

Note that we have given the type $!(bit \multimap qbit)$ to the term new. Another possible choice would have been $!(!bit \multimap qbit)$, which makes sense because all classical bits are duplicable. However, since $!(bit \multimap qbit) <: !(!bit \multimap qbit)$, the second type is less general, and can be inferred by the typing rules.

Note that, if $[Q, L, M]$ is a program state, the term M need not be closed; however, all of its free variables must be in the domain of L, and thus must be of type $qbit$. We therefore define:

Definition 7. A program state $[Q, L, M]$ is *well-typed of type* B if $\Delta \rhd M : B$ is derivable, where $\Delta = \{x{:}\ qbit \mid x \in FV(M)\}$. In this case, we write $[Q, L, M] : B$.

Note that the type system enforces that variables holding quantum data cannot be duplicated; thus, $\lambda x.\langle x, x \rangle$ is not a valid term of type $qbit \multimap qbit \otimes qbit$. On the other hand, we allow variables to be discarded freely. Other approaches are also possible, for instance, Altenkirch and Grattage [1] propose a syntax that allows duplication but restricts discarding of quantum values.

4.3 Example: quantum teleportation

Let us illustrate the quantum lambda calculus and the typing rules with an example. The following is an implementation of the well-known quantum teleportation protocol (see e.g. [9]). The purpose of the teleportation protocol is to send a qubit from location A to location B, using only classical communication and a pre-existing shared entangled quantum state. In fact, this can be achieved by communicating only the content of two classical bits.

In terms of functional programming, the teleportation procedure can be seen as the creation of two non-duplicable functions $f : qbit \multimap bit \otimes bit$ and $g : bit \otimes bit \multimap qbit$, such that $f \circ g(x) = x$ for an arbitrary qubit x.

We start by defining the following functions $\mathbf{EPR} : !(\top \multimap (qbit \otimes qbit))$, $\mathbf{BellMeasure} : !(qbit \multimap (qbit \multimap bit \otimes bit))$, and $\mathbf{U} : !(qbit \multimap (bit \otimes bit \multimap qbit))$:

$$\begin{aligned}
&\mathbf{EPR} &&= \lambda x. CNOT\langle H(new\,0), new\,0\rangle,\\
&\mathbf{BellMeasure} &&= \lambda q_2.\lambda q_1.(let\ \langle x, y\rangle = CNOT\langle q_1, q_2\rangle\ in\ \langle\, meas(Hx), meas\, y\rangle,\\
&\mathbf{U} &&= \lambda q.\lambda\langle x, y\rangle.\mathit{if}\ x\ \mathit{then}\ (if\ y\ then\ U_{11}q\ else\ U_{10}q)\\
& && \qquad\qquad\qquad \mathit{else}\ (if\ y\ then\ U_{01}q\ else\ U_{00}q),
\end{aligned}$$

where

$$U_{00} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, U_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, U_{10} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, U_{11} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.$$

The function $\mathbf{EPR}$ creates an entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. The function $\mathbf{BellMeasure}$ performs a so-called Bell measurement, and the function $\mathbf{U}$ performs a unitary correction on the qubit q depending on the value of two classical bits. We can now construct a pair of functions $f : qbit \multimap bit \otimes bit$ and $g : bit \otimes bit \multimap qbit$ with the above property by the following code:

$$\begin{aligned}
&let\ \langle x, y\rangle = \mathbf{EPR} *\\
&\ in\ let\ f\ = \mathbf{BellMeasure}\ x\\
&\quad in\ let\ g\ = \mathbf{U}\ y.\\
&\quad in\ \langle f, g\rangle.
\end{aligned}$$

The functions f and g thus created do indeed have the desired property that $f \circ g(x) = x$, where x is any qubit. Note that, since f and g depend on the state of the qubits x and y, respectively, these functions cannot be duplicated, which is reflected in the fact that the types of f and g do not contain a top-level "!".

4.4 Properties of the type system

We derive some basic properties of the type system.

Definition 8. We extend the subtyping relation to contexts by writing $\Delta <: \Delta'$ if $|\Delta'| = |\Delta|$ and for all x in $|\Delta'|$, $\Delta_f(x) <: \Delta'_f(x)$.

Lemma 8. *1. If $x \notin FV(M)$ and $\Delta, x{:}A \rhd M{:}B$, then $\Delta \rhd M{:}B$.*
2. If $\Delta \rhd M{:}A$, then $\Gamma, \Delta \rhd M{:}A$.
3. If $\Gamma <: \Delta$ and $\Delta \rhd N : A$ and $A <: B$, then $\Gamma \rhd N : B$.

The next lemma is crucial in the proof of the substitution lemma. Note that it is only true for a value V, and in general fails for an arbitrary term M.

Lemma 9. *If V is a value and $\Delta \rhd V : !A$, then for all $x \in FV(V)$, there exists some $U \in qType$ such that $\Delta(x) = !U$.*

Proof. By induction on V.

- If V is a variable x, then the last rule in the derivation was $\dfrac{B <: !A}{\Delta', x : B \rhd x : !A}$. Since $B <: !A$, B must be exponential by Lemma 7.
- If V is a constant c, then $FV(V) = \emptyset$, hence the result holds vacuously.
- If $V = \lambda x.M$, the only typing rule that applies is (λ_2), and $\Delta = \Gamma, !\Delta'$ with $FV(M) \cap |\Delta'| = \emptyset$. So every $y \in FV(M)$ except maybe x is exponential. Since $FV(\lambda x.M) = (FV(M) \setminus \{x\})$, this suffices.
- The remaining cases are similar. □

Lemma 10 (Substitution). *If V is a value such that $\Gamma_1, !\Delta, x{:}A \rhd M : B$ and $\Gamma_2, !\Delta \rhd V : A$, then $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x] : B$.*

Corollary 1. *If $\Gamma_1, !\Delta, x{:}A \rhd M : B$ and $\Gamma_2, !\Delta \rhd V : !^n A$, then $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x] : B$.*

Proof. From Lemma 10 and Lemma 8(3).

Remark 3. We note that all the usual rules of affine intuitionistic linear logic are derived rules of our type system, *except* for the general promotion rule. However, the promotion rule is derivable when V is a *value*:

$$\frac{!\Gamma \rhd V : A}{!\Gamma \rhd V : !A.}$$

4.5 Subject reduction and progress

Theorem 1 (Subject Reduction). *Given $[Q, L, M] : B$ and $[Q, L, M] \rightsquigarrow^* [Q', L', M']$, then $[Q', L', M'] : B$.*

Proof. It suffices to show this for $[Q, L, M] \longrightarrow_p [Q', L', M']$, and we proceed by induction on the rules in Table 1. The rule $[Q, (\lambda x.M)V] \longrightarrow_1 [Q, M[V/x]]$ and the rule for "let" use the substitution lemma. The remaining cases are direct applications of the induction hypothesis. □

Theorem 2 (Progress). *Let $[Q, L, M] : B$ be a well-typed program. Then $[Q, L, M]$ is not an error state in the sense of Definition 5. In particular, either $[Q, L, M]$ is a value, or else there exist some state $[Q', L', M']$ such that $[Q, L, M] \longrightarrow_p [Q', L', M']$. Moreover, the total probability of all possible single-step reductions from $[Q, L, M]$ is 1.*

Corollary 2. *Every sequence of reductions of a well-typed program either converges to a value, or diverges.*

The proof of the Progress Theorem is similar to the usual proof, with two small differences. The first is the presence of probabilities, and the second is the fact that M is not necessarily closed. However, all the free variables of M are of type $qbit$, and this property suffices to prove the following lemma, which generalizes the usual lemma on the shape of closed well-typed values:

Lemma 11. *Suppose $\Delta = x_1{:}qbit, \ldots, x_n{:}qbit$, and V is a value. If $\Delta \rhd V : A \multimap B$, then V is new, $meas$, U, or a lambda abstraction. If $\Delta \rhd V : A \otimes B$, then $V = \langle V_1, V_2 \rangle$. If $\Delta \rhd V : bit$, then $V = 0$ or $V = 1$.* □

Proof of the Progress Theorem. By induction on M. The claim follows immediately in the cases when M is a value, or when M is a left-hand-side of one of the rules in Table 1 that have no hypotheses. Otherwise, using Lemma 11, M is one of the following: PN, NV, $\langle N, P \rangle$, $\langle V, N \rangle$, $\mathit{if}\ N\ \mathit{then}\ P\ \mathit{else}\ Q$, $\mathit{let}\ \langle x, y \rangle {=} N\ \ \mathit{in}\ \ P$, where N is not a value. In this case, the free variables of N are still all of type $qbit$, and by induction hypothesis, the term $[Q, L, N]$ has reductions with total probability 1, and the rules in Table 1 ensure that the same is true for $[Q, L, M]$. □

5 Type inference algorithm

It is well-known that in the simply-typed lambda calculus, as well as in many programming languages, satisfy the *principal type property*: every untyped expression has a most general type, provided that it has any type at all. Since most principal types can usually be determined automatically, the programmer can be relieved from the need to write any types at all.

In the context of our quantum lambda calculus, it would be nice to have a type inference algorithm; however, the principal type property fails due to the presence of exponentials $!A$. Not only can an expression have several different types, but in general none of the types is "most general". For example, the term $M = \lambda xy.xy$ has possible types $T_1 = (A \multimap B) \multimap (A \multimap B)$ and $T_2 = !(A \multimap B) \multimap !(A \multimap B)$, among others. Neither of T_1 and T_2 is a substitution instance of the other, and in fact the most general type subsuming T_1 and T_2 is $X \multimap X$, which is not a valid type for M. Also, neither of T_1 and T_2 is a subtype of the other, and the most general type of which they are both subtypes is $(A \multimap B) \multimap !(A \multimap B)$, which is not a valid type for M.

In the absence of the principal type property, we need to design a type inference algorithm based on a different idea. The approach we follow is the one suggested by V. Danos, J.-B. Joinet and H. Schellinx [5]. The basic idea is to view a linear type as a "decoration" of an intuitionistic type. Our type inference algorithm is based on the following technical fact, given below: if a given term has an intuitionistic type derivation π, then it is linearly typable if and only if there exists a linear type derivation which is a decoration of π. Typability can therefore be decided by first doing intuitionistic type inference, and then checking finitely many possible linear decorations.

5.1 Skeletons and decorations

The class of *intuitionistic types* is

$$\mathit{iType} \quad U, V \quad ::= \quad \alpha \mid X \mid (U \Rightarrow V) \mid (U \times V) \mid \top$$

where α ranges over the type constants and X over the type variables.

To each $A \in \mathit{qType}$, we associate its *type skeleton* ${}^{\dagger}A \in \mathit{iType}$, which is obtained by removing all occurrences of "!". Conversely, every $U \in \mathit{iType}$ can be lifted to some ${}^{\clubsuit}U \in \mathit{qType}$ with no occurrences of "!". Formally:

Definition 9. Define functions $\dagger : \mathit{qType} \to \mathit{iType}$ and $\clubsuit : \mathit{iType} \to \mathit{qType}$ by:

$$\begin{array}{ll}
{}^{\dagger}!^n\alpha = \alpha, \quad {}^{\dagger}!^nX = X, \quad {}^{\dagger}!^n\top = \top, & {}^{\clubsuit}\alpha = \alpha, \quad {}^{\clubsuit}X = X, \quad {}^{\clubsuit}\top = \top, \\
{}^{\dagger}!^n(A \multimap B) = {}^{\dagger}A \Rightarrow {}^{\dagger}B, & {}^{\clubsuit}(U \Rightarrow V) = {}^{\clubsuit}U \multimap {}^{\clubsuit}V, \\
{}^{\dagger}!^n(A \otimes B) = {}^{\dagger}A \times {}^{\dagger}B, & {}^{\clubsuit}(U \times V) = {}^{\clubsuit}U \otimes {}^{\clubsuit}V.
\end{array}$$

Lemma 12. *If $A <: B$, then ${}^{\dagger}A = {}^{\dagger}B$. If $U \in \mathit{iType}$, then $U = {}^{\dagger\clubsuit}U$.*

Writing $\Delta \blacktriangleright M : U$ for a typing judgement of the simply-typed lambda calculus, we can extend the notion of skeleton to contexts, typing judgments, and derivations as follows:

$$\begin{array}{c}
{}^{\dagger}\{x_1{:}A_1, \ldots, x_n{:}A_n\} = \{x_1{:}{}^{\dagger}A_1, \ldots, x_n{:}{}^{\dagger}A_n\} \\
{}^{\dagger}(\Delta \rhd M : A) = ({}^{\dagger}\Delta \blacktriangleright M : {}^{\dagger}A).
\end{array}$$

From the rules in Table 2, it is immediate that if $\Delta \rhd M : A$ is a valid typing judgment in the quantum lambda-calculus, then ${}^{\dagger}(\Delta \rhd M : A) = ({}^{\dagger}\Delta \blacktriangleright M : {}^{\dagger}A)$ is a valid typing judgment in the simply-typed lambda-calculus.

We now turn to the question of how an intuitionistic typing derivation can be "decorated" with exponentials to yield a valid quantum typing derivation. These decorations are going to be the heart of the quantum type inference algorithm.

Definition 10. Given $A \in \mathit{qType}$ and $U \in \mathit{iType}$, we define the *decoration* $U \looparrowright A \in \mathit{qType}$ *of* U *along* A by

1. $U \looparrowright\, !^nA = !^n(U \looparrowright A)$,
2. $(U \Rightarrow V) \looparrowright (A \multimap B) = (U \looparrowright A \multimap V \looparrowright B)$,
3. $(U \times V) \looparrowright (A \otimes B) = (U \looparrowright A \otimes V \looparrowright B)$, and in all other cases:
4. $U \looparrowright A = {}^{\clubsuit}U$.

The following lemma is the key to the quantum type inference algorithm:

Lemma 13. *If M is well-typed in the quantum lambda-calculus with typing judgment $\Gamma \rhd M : A$, then for any valid typing judgment $\Delta \blacktriangleright M : U$ in simply-typed lambda-calculus with $|\Delta| = |\Gamma|$, the typing judgment $\Delta \looparrowright \Gamma \rhd M : U \looparrowright A$ is valid in the quantum lambda-calculus.*

5.2 Elimination of repeated exponentials

The type system in Section 4 allows types with repeated exponentials such as $!!A$. While this is useful for compositionality, it is not very convenient for type inference. We therefore consider a reformulation of the typing rules which only requires single exponentials.

Lemma 14. *The following are derived rules of the type system in Table 2, for all $\tau, \sigma \in \{0, 1\}$.*

$$\frac{!\Delta, \Gamma_1 \rhd M_1 : !A_1 \quad !\Delta, \Gamma_2 \rhd M_2 : !A_2}{!\Delta, \Gamma_1, \Gamma_2 \rhd \langle M_1, M_2\rangle : !(!^{\tau}A_1 \otimes !^{\sigma}A_2)}\ (\otimes.I')$$

$$\frac{!\Delta, \Gamma_1 \rhd M : !(!^{\tau}A_1 \otimes !^{\sigma}A_2) \quad !\Delta, \Gamma_2,\ x_1{:}!A_1,\ x_2{:}!A_2 \rhd N : A}{!\Delta, \Gamma_1, \Gamma_2 \rhd \mathit{let}\ \langle x_1, x_2\rangle = M\ \mathit{in}\ N : A}\ (\otimes.E')$$

Lemma 15. *If M is typable in the quantum lambda calculus by some derivation π, then M is typable in the system with the added rules $(\otimes.I')$ and $(\otimes.E')$, by a derivation π' using no repeated exponentials. Moreover, $^\dagger\pi' = {}^\dagger\pi$.* $\square$

5.3 Description of the type inference algorithm

To decide the typability of a given term M, first note the following: if M is not typable in simply-typed lambda calculus, then M is not quantum typable. On the other hand, suppose M admits a typing judgment $\Gamma \blacktriangleright M : U$ in the simply-typed lambda calculus, say with typing derivation π. Moreover, suppose without loss of generality that the derivation π uses no dummy variables, i.e., each sequent $\Gamma' \blacktriangleright M' : U'$ of π satisfies $|\Gamma'| = FV(M')$. Then by the proof of Lemma 13, M is quantum typable if and only if M has a quantum derivation whose skeleton is π. Thus we can perform type inference in the quantum lambda-calculus in two steps:

1. Find an intuitionistic typing derivation π, if any, using no dummy variables.
2. Find a decoration of π which is a valid quantum typing derivation, if any.

Step (1) is known to be decidable. For step (2), note that by Lemma 15, it suffices to consider decorations of π without repeated exponentials. Since there are only finitely many such decorations, the typability of M is clearly a decidable problem. Also note that if the algorithm succeeds, then it returns a possible type for M. However, it does not return a description of all possible types.

It should further be noted that the space of all decorations of π, while exponential in size, can be searched efficiently by solving a system of constraints. More precisely, if we create a boolean variable for each place in the type derivation which potentially can hold a "!", then the constraints imposed by the linear type system can all be written in the form of implications $x_1 \wedge \ldots \wedge x_n \Rightarrow y$, where $n \geqslant 0$, and negations $\neg z$. It is well-known that such a system can be solved in polynomial time in the number of variables and clauses, which is in turns polynomial in the size of the type derivation. Note, however, that the size of the type derivation need not be polynomial in the size of the term M, as the type of M can be of exponential size in the worst case.

6 Conclusion and further work

In this paper, we have defined a higher-order quantum programming language based on a linear typed lambda calculus. Compared to the quantum lambda calculus of van Tonder [14,15], our language is characterized by the fact that it contains classical as well as quantum features; for instance, we provide classical datatypes and measurements as a primitive feature of our language. Moreover, we provide a subject reduction result and a type inference algorithm. As the language shows, linearity constraints do not just exist at base types, but also at higher types, due to the fact that higher-order function are represented as closures which may in turns contain embedded quantum data. We have shown that affine intuitionistic linear logic provides the right type system to deal with this situation.

There are many open problems for further work. An interesting question is whether the syntax of this language can be extended to include recursion. Another question is to study extensions of the type system, for instance with additive types as in linear logic. One may also study alternative reduction strategies. In this paper, we have only considered the call-by-value case; it would be interesting to see if there is a call-by-name equivalent of this language. Finally, another important open problem is to find a good denotational semantics for a higher order quantum programming language. One approach for finding such a semantics is to extend the framework of Selinger [12] and to identify an appropriate higher-order version of the notion of a superoperator.

References

1. T. Altenkirch and J. Grattage. A functional quantum programming language. Available from arXiv:quant-ph/0409065, 2004.
2. H. P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North Holland, second edition, 1984.
3. P. Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22:563–591, 1980.
4. S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D*, 25(2):181–200, August 2003.
5. V. Danos, J.-B. Joinet, and H. Schellinx. On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic*, 33:387–412, 1995.
6. D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 400(1818):97–117, July 1985.
7. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
8. E. Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.
9. M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.
10. J. Preskill. Lecture notes for Physics 229, quantum computation. Available from http://www.theory.caltech.edu/people/preskill/ph229/#lecture, 1999.
11. J. W. Sanders and P. Zuliani. Quantum programming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction: 5th International Conference*, volume 1837 of *Lecture Notes in Computer Science*, pages 80–99, Ponte de Lima, Portugal, July 2000. Springer-Verlag.
12. P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.
13. Benoît Valiron. A functional programming language for quantum computation with classical control. Master's thesis, University of Ottawa, September 2004.
14. A. van Tonder. Quantum computation, categorical semantics and linear logic. On arXiv: quant-ph/0312174, 2003.
15. A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal of Computing*, 33(5):1109–1135, 2004. Available from arXiv:quant-ph/0307150.