Proc. QPL 2004, pp. 163–178

Quantum typing

Benoît Valiron
University of Ottawa, Canada

Abstract

The objective of this paper is to develop a functional programming language for quantum computers. We develop a lambda-calculus for the *QRAM* model, following the work of P. Selinger (2003) on quantum flow-charts. We define a call-by-value operational semantics, and we develop a type system using affine intuitionistic linear logic. The main result of this preprint is the subject-reduction of the language.

1 Introduction

The objective of this paper is to develop a functional programming language for quantum computers. Quantum computing has become a fast growing research area in recent years. The object is to study what would happen if one encodes data using quantum particles instead of classical ones. For a good introduction, see e.g. [7, 6].

The laws of quantum physics restrict the kinds of operations that one can perform on a quantum state: there are only two kinds of basic operations, namely *unitary transformations* and *measurements*. Many existing models of 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. 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 [2, 4], 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 [10, 11], which is a higher-order, purely quantum language without an explicit measurement operation.

On the other hand, one might imagine a more realistic model of a quantum computer where unitary operations and measurements can be interleaved. One example is the so-called *QRAM model* of Knill [5], which is also described by Bettelli, Calarco and Serafini [3]. Here, a quantum computer consists of a classical computer with a quantum device attached to it. 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. This situation is summarized by the slogan "quantum data, classical control" [8]. Several programming languages have been proposed to deal with such a model [3, 9], but the one on which this paper is based is the work of Selinger [8].

In this paper, we propose a *higher-order* quantum programming language, i.e., one in which functions can be considered as data. In our language, 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 in a complete different manner. 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 [7, 6]. However, quantum data types are very powerful, due to the phenomena of quantum superposition and entanglement. Due to entanglement, quantum bits cannot be directly encoded in the lambda-term. However one can define it as a free variable and set an external function to link this variable to the actual quantum bit in the quantum device.

The semantic described in this paper is operational: a program is an abstract machine with reductions rules. The machine embbed a representation for the quantum device as a vector in a suitable Hilbert space, a lambda-term and a function that links the free variables of the term to the quantum bits in the quantum device. The reduction rules are probablistic, and one problem we solve is to describe the behavior of the program with respect to this probabilistic reduction. One part of the challenge is that we don't want to consider the quantum feature as a side effect.

Difficulties arise during manipulation of functions: the duplicability feature of a function is independant of the one of its argument and of its value: A non-duplicable function may accept only duplicable argument, and a duplicable function may return non-duplicable values. We describe a type system to handle all these cases, based on affine intuitionistic linear logic. This paper expose how the validity of a program is linked to the reduction procedure, and then induce the choice of the typing rules to make them verify subject reduction.

2 Terms

2.1 Quantum States

The quantum data in the *QRAM* can be modelized as a vector in a Hilbert space [7, 6]. Formally we define a quantum bit, or **qubit**, as a normalized vector of the 2-dimensional Hilbert space $\mathbb{C}^2$. We denote the orthonormal basis as $\{|0\rangle, |1\rangle\}$. A quantum superposition of several qubits is a normalized vector in $\otimes_{i=1}^{n}\mathbb{C}^2$. We call it a n-quantum bit, and we denote it as $|\phi\rangle = \sum_{i=0}^{2^n} |bin_n(i)\rangle$ with $bin_n(i)$ being the binary representation of i in n digits. We will refer to the k-th qubit of this n-qubit to refer to the k-th 0 or 1 in $|bin_n(i)\rangle$.

We would like to extend the lambda calculus with the ability to manipulate quantum data. We first need a syntax to express quantum states in the lambda calculus. In simple cases, we might simply insert quantum states into a lambda term, such as $(\lambda x.(\alpha|0\rangle + \beta|1\rangle))$. However, in the general case, such a syntax is insufficient. Consider for instance the lambda term $(\lambda y.\lambda f.fpy)(q)$, where p and q are quantum bits which are jointly in the entangled state $|pq\rangle = \alpha|00\rangle + \beta|11\rangle$. Such a state cannot be represented locally by

replacing p and q with some constant expressions of type qubit. The non-local nature of quantum states thus forces to introduce a level of indirection into the representation of a state. Thus, to represent a program, we should have a lambda-term M to encode the operations, but also an exterior n-qubit state Q to store the quantum data of the program. Further, to link both parts, we need a third element, which is a function Q_f from $FV(M)$ to $\{0, \ldots, n-1\}$, such that if $Q_f(x) = i$, x would be a representation of the i-th qubit in Q.

We provide several built-in operations for quantum bits. The operator new represents a function that takes a bit (0 or 1) and allocates a new qubit of the corresponding value. We also need to be able to act on qubits via unitary operations; thus, we will assume a given set $\mathcal{U}^1$ of unary unitary gates. Currently, our language has no way of representing tuples, so we will restrict ourselves to unary quantum gates for now; tuples and n-ary gates will be considered in Section 5.

In the following examples, we will often use the Hadamard gate H, which we assume to be an element of $\mathcal{U}^1$:

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

Finally, we equip the language with a measurement operation. Let $meas$ be the term for it. It takes a quantum bit, performs a measurement, and returns the classical bit 0 or 1 which is the result of the measurement. Of course, the outcome of this operation is probabilistic. If U ranges over $\mathcal{U}^1$, we define a **term** by the following Backus-Naur Form:

$$\begin{array}{c} RawTerm\ M, N, P\ ::=\ x \mid (MN) \mid \lambda x.M \mid\ if(M; N; P) \\ \mid c \mid 0 \mid 1 \mid\ meas\ \mid\ new\ \mid U \end{array}$$

with x ranging over $\mathcal{V}$ a countable set of variables, and c ranging over $\mathcal{C}$ a set of constants. The set of constants should at least contain the truth values 0 and 1. $\lambda x.M$ represents a function of variable x, and (MN) is the application M applied to N. We also call $\lambda x.M$ an **abstraction**. The term $if(M; N; P)$ denotes the conditional expression "if M then N else P". Here, only one of N or P is evaluated, depending on the truth value of M. Finally, the set of constants should also contain the terms $meas$ and new, to be able to measure and create qubits.

The notion of free and bound variable is defined as usual [1], just as the notion of α-equivalence, substitution and β-reduction.

As usual, terms are identified up to α-equivalence. In that sense we will write $\lambda x.x = \lambda y.y$.

Definition. A **quantum state** is a triple $[Q, Q_f, M]$ where Q is a normalized vector of $\otimes_{i=0}^{n-1}\mathbb{C}^2$, M is a lambda-term, and Q_f is a function from W to $\{0, \ldots, n-1\}$, where $FV(M) \subseteq W \subseteq \mathcal{V}$. Q_f is also called the **linking function**. We denote the set of quantum states by $\mathbb{S}$. If $n = 0$, then we denote $Q = 1 \in \mathbb{C}$ by $Q = |\rangle$.

The notion of α-equivalence extends naturally to quantum states, for instance, the states

$$[|1\rangle, \{x \mapsto 0\}, \lambda y.x] \text{ and } [|1\rangle, \{z \mapsto 0\}, \lambda y.z]$$

are equivalent. More formally, the α-**equivalence** on quantum states is the smallest equivalence relation such that if $x \in FV(M)$ and $z \notin FV(M)$, then

$$[Q, Q_f \cup \{x \mapsto i\}, M] =_\alpha [Q, Q_f \cup \{z \mapsto i\}, M[z/x]]$$

We will work under this equivalence when speaking of quantum states.

In order to simplify the notation, we will often use the following trick: we use p_i to denote the free variable x such that $Q_f(x) = i$. A quantum state is abreviated by $[Q, M']$ with $M' = M[p_{i_1}/x_1] \dots [p_{i_n}/x_n]$ if the domain of Q_f is $\{x_1, \dots, x_n\}$, where $i_k = Q_f(x_k)$.

Reduction of the quantum state We should now address the question of how a quantum state should be reduced. One restriction is that it is forbidden to duplicate a quantum bit, due to the no-cloning property of quantum physics. Let us illustrate this with an example, using a call-by-value reduction procedure. Let us define a binary **and** in our language in that way: $\mathbf{and} = \lambda xy.\, \mathit{if}(x; \mathit{if}(y; 1; 0); 0)$. Now consider the term $(\lambda x.\mathbf{and}(meas(x))(meas(H\ x))\ (|0\rangle)$. Naïvely, we expect this to reduce to

$$\mathbf{and}(meas(|0\rangle))(meas(H\ |0\rangle)),$$

then to measure the right argument $H\ |0\rangle$, then the left argument which reduce to 0 with probability 1, and then apply the **and** function. We expect to obtain the result 0 with probability 1. Using the quantum state notation, let us reduce this term more formally:

$$\begin{array}{c}[|0\rangle, (\lambda x.\mathbf{and}(meas(x))(meas(H\ x))\ (p_0)] \\ \longrightarrow_{CBV} [|0\rangle, \mathbf{and}(meas(p_0))(meas(H\ p_0))]\end{array}$$

In the QRAM, applying H to a qbit is modifying the actual state of the qbit. Let us reduce the right argument $(H\ p_0)$:

$$\longrightarrow_{CBV} [\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right), \mathbf{and}(meas(p_0))(meas(p_0))].$$

Reducing the right argument again, we obtain, assuming that the measurement is non-destructive (Indeed, if we used destructive measurement, the program would not even be well-defined, since we would have a p_0 alone):

$$\longrightarrow_{CBV} \begin{cases} [|0\rangle, \mathbf{and}(meas(p_0))(0)] \text{ with prob. } 0.5 \\ [|1\rangle, \mathbf{and}(meas(p_0))(1)] \text{ with prob. } 0.5. \end{cases}$$

This reduces to $[|0\rangle, 0]$ with probability 0.5 and to $[|1\rangle, 1]$ with probability 0.5. Clearly, this is not the intended result.

The program is unpredictable due to the duplication of p_0. The problem derives from the fact that a value such as p_0 does not represent a constant, as in the classical lambda calculus, but rather it is a *pointer* into the quantum state. We never act *on* p_0, we act on the value it points to. To ensure the predictability of programs, it is necessary to disallow the duplication of terms that contain p_i's, since we don't want to allow side-effects. We will

call an abstraction $\lambda x.M$ **linear** if x appears at most once as a free variable in M. We also say that M is **linear in** x in this case.

Another problem can occur: let us call **plus** the function which acts as the addition modulo 2 on classical bits. We can easily construct such a function in our language: $\mathbf{plus} = \lambda xy.\, if\,(x; if\,(y; 0; 1); if\,(y; 1; 0))$. Consider the state

$$[\,|\rangle, (\lambda x.\mathbf{plus}\ x\ x)(meas(H(new\ \ 0)))].$$

Now reduce that system using the call-by-value reduction system. Intuitively one shall get:

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

and then with probability 0.5:

$$\begin{array}{c} [\,|0\rangle, (\lambda x.\mathbf{plus}\ x\ x)(0)] \ \text{ or } \ [\,|1\rangle, (\lambda x.\mathbf{plus}\ x\ x)(1)] \\ [\,|0\rangle, \mathbf{plus}\ 0\ 0] \ \text{ or } \ [\,|1\rangle, \mathbf{plus}\ 1\ 1] \end{array}$$

which evaluate with probability 0.5 to $[\,|0\rangle, 0\,]$ or to $[\,|1\rangle, 0\,]$.

Had we reduced the same term under a call-by-name strategy, we would have obained in the first step

$$[\,|\rangle, \mathbf{plus}\ (meas(H(new\ \ 0)))\ (meas(H(new\ \ 0))))],$$

and then a quantum state with an array of four qubits instead of two.

Moreover, if we had mixed the call-by-value and call-by-name strategies, the program could have led to an ill-defined result: reducing by call-by-value until

$$[\frac{\sqrt{2}}{2}(|0\rangle + |1\rangle), (\lambda x.\mathbf{plus}\ x\ x)(meas\ \ p_0)]$$

and then changing to call-by-name, we would obtain in one step:

$$[\frac{\sqrt{2}}{2}(|0\rangle + |1\rangle), (\mathbf{plus}\ (meas\ \ p_0)\ (meas\ \ p_0)],$$

which is not even a valid program since there are 2 occurences of p_0.

In other words, it does not make sense to speak of a general β-reduction procedure for the whole quantum state. If we define a reduction procedure, we have to choose a reduction procedure to before writing programs.

2.2 Probabilistic reduction system

We define a **probabilistic reduction system** as 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, and $\sum_{x' \in R_x} prob(x, x') \leq 1$. We call $prob$ the one-step reduction, and we note $x \longrightarrow_p y$ for $prob(x, y) = p$.

Let us extend $prob$ to the n-step reduction:

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

We use the notation $x \longrightarrow_p^n y$ for $prob^n(x,y) = p$.

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 $x \in X$, $prob_U(x) + prob_\infty(x) \leq 1$.*

We define the **error probability of** x to be the number

$$prob_{err}(x) = 1 - prob_U(x) - prob_\infty(x)$$

We can define a notion of equivalence in X:

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

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 $prob$,

$$x \longrightarrow_{>0} y \quad \text{implies} \quad x \longrightarrow y$$

with $x \longrightarrow y$ for xRy. Let us denote by $\longrightarrow^*$ the relation such that

$$x \longrightarrow^* y \quad \text{iff} \quad \exists n \quad xR^n y$$

with R^n define as the n-th composition of R. Similarly,

$$x \longrightarrow_{>0}^* y \quad \text{implies} \quad x \longrightarrow^* y$$

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 $x \longrightarrow^* e$

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

Note that the converse is false.

We need this notion of weak reachability, 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. What will happen if we measure this qubit and get 1? We need to be sure that even in this case the program will not crash. Hence we separate in a sense the null probability of getting a certain result, and the computational impossibility.

2.3 Quantum reduction

We need a deterministic decision procedure to choose which redex to reduce. Let us analyse a call by value procedure, since this is the most intuitive one.

A useful subset of $\mathbb{S}$ is the subspace $\mathbb{V}$ of **value states**:

$$\mathbb{V} = \{ \quad [Q, Q_f, V] \in \mathbb{S} \quad | \quad V \text{ is a value} \quad \}$$

A **value** is a term of the following form:

$$\mathit{Value}\ V \ ::= \ x \mid \lambda x.M \mid c \mid 0 \mid 1 \mid \ \mathit{meas} \ \mid \ \mathit{new} \ \mid U$$

We define a probabilistic call-by-value reduction procedure. We write $M \longrightarrow_{CBV\,p} N$ if M reduces to N with probability p, or $M \longrightarrow_p N$ for short.

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

Let $Q = \alpha|Q_0\rangle + \beta|Q_1\rangle$ being normalized, with

$$|Q_0\rangle = \sum_i \alpha_i |\phi_i^0\rangle \otimes |0\rangle \otimes |\psi_i^0\rangle \qquad |Q_1\rangle = \sum_i \beta_i |\phi_i^1\rangle \otimes |1\rangle \otimes |\psi_i^1\rangle$$

with $|0\rangle$ and $|1\rangle$ being the i-th qubit. Let $\mu_0 = |\alpha|^2$ and $\mu_1 = |\beta|^2$

$$\begin{array}{l} [\alpha|Q_0\rangle + \beta|Q_1\rangle, \mathit{meas}\ p_i] \longrightarrow_{\mu_0} [|Q_0\rangle, 0] \\ [\alpha|Q_0\rangle + \beta|Q_1\rangle, \mathit{meas}\ p_i] \longrightarrow_{\mu_1} [|Q_1\rangle, 1] \end{array}$$

If Q is in a space of dimension 2^{n-1},

$$\frac{}{[Q, \mathit{new}\ 0] \longrightarrow_1 [Q \otimes |0\rangle, p_n]} \quad \frac{}{[Q, \mathit{new}\ 1] \longrightarrow_1 [Q \otimes |1\rangle, p_n]}$$

If Q is in a space of dimension 2^{n-1}, let $Q' = (I_j \otimes H \otimes I_{n-j-2})(Q)$

$$\frac{}{[Q, H\ p_j] \longrightarrow_1 [Q', p_j]}$$

In any case (V a value):

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

The cases for *if* are:

$$\frac{}{[Q, \mathit{if}(0; M; N)] \longrightarrow_1 [Q, N]} \quad \frac{}{[Q, \mathit{if}(1; M; N)] \longrightarrow_1 [Q, M]}$$

$$\frac{[Q, P] \longrightarrow_p [Q', P']}{[Q, \mathit{if}(P; M; N)] \longrightarrow_p [Q', \mathit{if}(P'; M; N)]}$$

We define a weaker relation $\rightsquigarrow$. This relation modelize the transformations that can happen due to decoherence and imprecision of physical operations. It is supposed to be at least the one above, even when $p = 0$ in the definition, plus the following one, if Q and Q' are in the same vector space: $[Q, M] \rightsquigarrow [Q', M]$.

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

3 Types

The notion of lambda-term is a powerful way of representing functions and programs. But we need a way to prevent run-time errors as much as possible. The usual way to do that is to use a **type system**.

3.1 Subtyping

Let us define a type system. We are going to define it together with an ordering relation $<:$. We need constant types and types for abstractions (the functions). Moreover, we need a notion of duplicability of term. We want to be able to say whether or not a term can be duplicated. So let us define in BNF:

$$qType\ A, B\ ::=\ \alpha \mid X \mid\ !A \mid (A \multimap B)$$

where α spans a poset of type constants ordered by $\leq$ and X spans a countable set of type variables. $A \multimap B$ stand for "function with argument of type A which returns a result of type B". The notation "!" is a flag to state that the typed term is duplicable. We will call a type "exponential" if it is written "$!A$".

Let us extend the ordering relation $\leq$ on types by $<:$.

$$\frac{\alpha_i \leq \alpha_j}{\alpha_i <: \alpha_j}\ (ax) \qquad \frac{A <: B}{!A <: B}\ (D) \qquad \frac{}{X <: X}\ (var)$$

$$\frac{!A <: B}{!A <: !B}\ (!) \qquad \frac{A <: A' \quad B <: B'}{A' \multimap B <: A \multimap B'}\ (\multimap)$$

Lemma 4 *For any type A and B, if $A <: B$ and $(m = 0) \vee (n \geq 1)$, then $(n)(A) <: (m)(B)$*

Proof by double induction on m and n.

Let us notice that one can rewrite types using the notation:

$$qType\ A, B\ ::=\ (n)(\alpha_i) \mid (n)(X), (n)(Y) \ldots \mid (n)(A \multimap B)$$

with $n \in \mathbb{N}$. $(n)(A)$ standing for $\underbrace{!!!\ldots!!!}_{n \text{ times}} A$

The rules can be re-written:

$$\frac{(m=0) \vee (n \geq 1)}{(n)(X) <: (m)(X)}\ (var_2) \quad \frac{\alpha_i \leq \alpha_j \quad (m=0) \vee (n \geq 1)}{(n)(\alpha_i) <: (m)(\alpha_j)}\ (\alpha)$$

$$\frac{A <: A' \quad B <: B' \quad (m=0) \vee (n \geq 1)}{(n)(A' \multimap B) <: (m)(A \multimap B')}\ (\multimap_2)$$

The two sets of rules are equivalent.

Proof Set (2) implies set (1): (var) and (α) follow directly from the previous lemma, $(\multimap_2)$ comes from the fact that we know that $A <: A'$ and $B <: B'$. So by $(\multimap)$ we have $A' \multimap B <: A \multimap B'$. And by Lemma 4, we have obtained the desired result. The proof that set (1) implies set (2) is done by induction on the proof that $A <: B$. □

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

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

The proof is done using the second set of rules, and the transitivity of the implication in the equivalence $(m=0) \vee (n \geq 1)$ iff $(m \geq 1) \Rightarrow (n \geq 1)$ □

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

Proof Using the first set of rules, $A <: !B$ can only come from (D) or $(!)$. In both cases, A is of the form $!C$. □

3.2 Typing rules

We need to define what it means for a quantum state $[Q, Q_f, M]$ to be well-typed. It turns out that the typing does not depend on Q and Q_f, but only on M. Given a term M, we need to be able to say whether or not it is well-typed. As usual, we introduce typing judgements to deal with terms that may have free variables. Note that the free variables of M which are in the domain of Q_f have to be of type $qbit$.

A **typing judgement** is a tuple $\Delta \rhd M : B$ where M is a term, B is a $qType$, and Δ is a set of variables $|\Delta| = \{x_1, \ldots x_n\}$ together with a function Δ_f from $|\Delta|$ to $qType$. We usually denote Δ by $\{x_1 : A_1, \ldots x_n : A_n\}$, with $A_i = \Delta_f(x_i)$. Δ is called a **typing context**.

A **program** is defined as a closed quantum state $[Q, Q_f, M]$, where there exists a type B such that $(\Delta \rhd M{:}B)$ with $\Delta = \{\ x{:}qbit \mid x \in FV(M)\ \}$ is well-typed.

For A and B in $qType$:

$$\text{The axioms:}\quad \frac{A <: B}{\Delta, x : A \rhd x : B}\ (ax_1) \qquad \text{For } c \text{ a constant term,}\quad \frac{A_c <: B}{\Delta \rhd c : B}\ (ax_2)$$

$$\text{For the } \textit{if} \text{ term,}\quad \frac{\Gamma_1, !\Delta \rhd P : \mathit{bit} \quad \Gamma_2, !\Delta \rhd M : A \quad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd \mathit{if}(P; M; N) : A}\ (if)$$

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

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

Table 1: TYPING RULES

We know what type the constant terms should have. Let us assign a given type A_c for each constant term c, from the set of constant terms to $qType$:

$A_0 = !\mathit{bit}$ $\quad A_1 = !\mathit{bit}$ $\quad A_{new} = !(\mathit{bit} \multimap \mathit{qbit})$
$A_U = !(\mathit{qbit} \multimap \mathit{qbit})$ $\quad A_{meas} = !(\mathit{qbit} \multimap !\mathit{bit})$

Remark: we set $new : !(\mathit{bit} \multimap \mathit{qbit})$. We could also have put $!bit$ in place of bit, since we want a bit to be always duplicable. However, this will be a corollary of the typing rules, and we therefore put the most general type for the constant.

For contexts Δ, Δ_1 and Δ_2: $\Delta = \Delta_1, \Delta_2$ means $|\Delta_1| \cap |\Delta_2| = \emptyset$ and $\Delta = \Delta_1 \cup \Delta_2$. $\Delta = \Delta_1, x : A$ means $\Delta = \Delta_1, \{x : A\}$. $\Delta = !\Delta_1$ means $\Delta = \{x_1 : !A_1, \dots x_n : !A_n\}$ if $\Delta_1 = \{x_1 : A_1, \dots x_n : A_n\}$. The rules for constructing **valid typing judgements** are shown in Table 1.

Lemma 8

1. *If* $x \notin FV(M)$, $(\Delta, x{:}A \rhd M{:}A)$ *implies* $(\Delta \rhd M{:}A)$.
2. *If* A *is in* $qType$, $(\Delta \rhd M{:}A)$ *implies* $(\Gamma, \Delta \rhd M{:}A)$.

Proof by induction on the proof of $\Delta \rhd M : A$.

4 Subject reduction

We **define** a subtyping relation between contexts by: $\Delta <: \Delta'$ iff $|\Delta'| = |\Delta|$ and for all $x \in |\Delta'|$, $\Delta_f(x) <: \Delta'_f(x)$. This relation is an ordering relation.

Lemma 9 *If* $(\Gamma <: \Delta)$, $(\Delta \rhd N{:}A)$ *and* $(A <: B)$, *then* $(\Gamma \rhd N{:}B)$.

Proof by structural induction on N.

Lemma 10 *If* V *is a value such that* $(\Delta \rhd V{:}!A)$, *then for all* $x \in FV(V)$, *there is* $U \in qType$ *such that* $\Delta_f(x) = !U$.

Proof

- If V is a variable x, Δ splits in $(\Delta', x : B)$ and the typing judgement comes from $\dfrac{B <: !A}{\Delta', x : B \rhd x :!A}$. Since $B <: !A$, by Lemma 7, B needs to be exponential. Hence the lemma is verified.
- If V is a constant c: The term is closed, hence by vacuity we have the result.
- If $V = \lambda x.M$, the only rule that applies is (λ_2), and Δ splits into $(\Delta_1, !\Delta_2)$ with $FV(M) \cap |\Delta_1| = \emptyset$. So every free variable y except maybe x in M is exponential. Since $FV(\lambda x.M) = (FV(M) \setminus \{x\})$, the lemma is also true in this case.

□

Lemma 11 *For* A *and* B $qType$, *and* V *a value, if* $!\Delta, \Gamma_1 \rhd V{:}A$ *and* $!\Delta, \Gamma_2, x{:}A \rhd M{:}B$, *then* $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x]{:}B$.

Let ω be a proof-tree for $!\Delta, \Gamma_2, x : A \rhd M : B$. The proof is done by structural induction on ω.

Lemma 12 *If* $\Gamma_1, !\Delta, x{:}A \rhd M{:}B$ *and* $\Gamma_2, !\Delta \rhd V{:}(\tau)A$, *then* $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x]{:}B$.

This is a corollary from Lemmas 11 and 9.

Theorem 1 *(Subject reduction)*
The typed lambda-calculus defined is preserved by $\rightsquigarrow$.

Proof We are going to restrict the study to the reduction rules, it extends easily to $\rightsquigarrow$. If $[Q, M] \longrightarrow_p [Q', M']$, we want to verify that $(\Delta \rhd M{:}B)$ implies $(\Delta \rhd M'{:}B)$ Since it is a relation defined by induction, we are going to do it by induction on the definition

- For the rule $[Q, (\lambda x.M)V] \longrightarrow_1 [Q, M[V/x]]$. If $\Phi \rhd (\lambda x.M)V : B$, we have the following typing tree:

$$\dfrac{!\Delta, \Gamma_1 \rhd V : A \quad \dfrac{!\Delta, \Gamma_2, x : A \rhd M : B}{!\Delta, \Gamma_2 \rhd \lambda x.M : A \multimap B}}{!\Delta, \Gamma_1, \Gamma_2 \rhd (\lambda x.M)V : B}$$

  with Φ splitting in $(!\Delta, \Gamma_1, \Gamma_2)$. Using Lemma 12, $M[V/x]$ is of type B, since $(!\Delta, \Gamma_1 \rhd V{:}A)$ and $(!\Delta, \Gamma_2, x{:}A \rhd M{:}B)$.

- The rules for $meas$ are

$$[\alpha|Q_0\rangle + \beta|Q_1\rangle, meas\ \ p_i] \xrightarrow{\mu_0} [|Q_0\rangle, 0]$$
$$[\alpha|Q_0\rangle + \beta|Q_1\rangle, meas\ \ p_i] \xrightarrow{\mu_1} [|Q_1\rangle, 1]$$

If $\Gamma, !\Delta, x : qbit \rhd meas\ \ x : B$ is valid it must come from:

$$\frac{\omega_1 \quad \omega_2}{\Gamma, !\Delta, x : qbit \rhd meas\ \ x : B}\ (app)$$

with $\Gamma = (\Gamma_1, \Gamma_2)$, and ω_1 and ω_2 to be:

$$\omega_1 = \left\{ \frac{!(qbit \multimap bit) <: A \multimap B}{\Gamma_1 !\Delta \rhd meas : A \multimap B}\ (ax) \right.$$
$$\omega_2 = \left\{ \frac{qbit <: A}{\Gamma_2, !\Delta, x : qbit \rhd x : A}\ (ax_1) \right.$$

From the subtyping rule $(\multimap_2)$, (ω_1) implies that $bit <: B$ and $A <: qtype$. Hence $A = qtype$, and $\dfrac{bit <: B}{\Gamma_1 !\Delta \rhd 0 : B}$ is a valid proof. In the same way $\dfrac{bit <: B}{\Gamma_1 !\Delta \rhd 1 : B}$, hence we have subject reduction.

The idea is the same for *new* and H. For the derivation rules of terms of the form $if(M; N; P)$ and (MN), the proof only use the induction hypothesis. □

Theorem 2 *If $[Q, M]$ is such that there exists a type A such that $(\Delta \rhd M{:}A)$ with $|\Delta|$ being the domain of Q_f and $Q_f(x) = qbit$ for all x in $|\Delta|$, then it is consistent. Hence any closed well-typed term either converges to a value, or diverges.*

Proof We prove that for all well-typed closed term $[Q, M]$, either it is a value, or there exists at least one M' such that $M \longrightarrow M'$. We do it by induction on the proof of validity of the typing judgement. There is two cases. Either it is a value, in which case there is nothing to do, or it is not, and the only 2 rules that apply are (app) and (if).

(app) $M = PQ$ and the typing tree starts with $\dfrac{\Delta_1 \rhd P{:}B \multimap A \quad \Delta_2 \rhd Q{:}B}{\Delta \rhd PQ : A}$ with $\Delta = (\Delta_1, \Delta_2) = \{x{:}qbit \mid x \in FV(M)\}$. Since $FV(M) = FV(P) \cup FV(Q)$, and there are disjoint, the two typing judgements we have are of the form required by the theorem. So by induction hypothesis, either we can reduce Q, and we are done, or it is a value. If it is a value, let us study P: P is also either reducible, and then we are done, or it is a value. If it is a value, then it is an abstraction and PQ is reducible, either it is a constant function, new, $meas$ or H. Since the typing judgement is valid, we are done, we can reduce in this last case.

(if) The if statement is the same: $M = if(P; Q; R)$, and either we can reduce P, or it is a value, so 0 or 1 and we can reduce M in Q or R.

So by induction any closed well-typed term is consistent. □

RawTerm	M, N, P	::=	$x \mid MN \mid \lambda x.M \mid \mu f x.M \mid if(M;N;P)$ $\mid 0 \mid 1 \mid \ meas \ \mid new \mid U \mid \ * \ \mid \langle M_1, \ldots, M_k \rangle$ $\mid$ let $\langle x_1, \ldots, x_k \rangle = M$ in N
Value	V	::=	$x \mid \lambda x.M \mid \mu f x.M \ 0 \mid 1$ $\mid \ meas \ \mid new \mid U \mid \ * \ \mid \langle V_1, \ldots, V_k \rangle$
qType	A, B	::=	$(n)(\alpha) \mid (n)(\top) \mid (n)(X) \mid (n)(A \multimap B)$ $\mid (n)(A_1 \otimes \ldots \otimes A_k)$

The subtyping relation is extended to

$$\frac{}{\top <: \top}\ (\top) \qquad \frac{\forall i \quad A_i <: B_i}{A_1 \otimes \ldots \otimes A_k <: B_1 \otimes \ldots \otimes B_k}\ (\otimes)$$

Table 2: EXTENDED TERMS AND TYPES

5 Extension of the language

Let us extend the language with product types and recursion. Extended terms and types are defined in Table 2. In this case we allow the U to be unitary operations of more than one qubit. For example if U is a binary unitary gate, we use it in that way: U : *qbit* $\otimes$ *qbit* $\multimap$ *qbit* $\otimes$ *qbit*. We add to the previous definition a notion of tuples: we will denote a k-tuple by $\langle M_1, \ldots, M_k \rangle$.

Product versus Tensor We use in our language the tensor product instead of a cartesian product. The reason is the following: If we define our product as cartesian, we need two projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. Then there has to be a bijection $\langle \pi_1(M), \pi_2(M) \rangle \leftrightarrow M$. But such a projection cannot exists: if M is not duplicable, we do not have the right to write $\langle \pi_1(M), \pi_2(M) \rangle$. This is not linear in M.

Thus, we have to take care of the fact that we can have non-duplicable terms in a tuple. Let us take an example. $\left[\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \langle p_0, p_1 \rangle\right]$ is a perfectly valid quantum state. In the term $M = \langle p_0, p_1 \rangle$ we have stored two qubits. Let us say we want to apply the H gate on p_1 and then the *CNOT* gate on both of them. The *CNOT* gate is $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ Using projections π_1 and π_2, we would have to write this as $CNOT\langle H(\pi_1 M), \pi_2 M \rangle$, but that is not a valid program since we are duplicating M. If we want to stay linear, we have either to forget p_1 in reducing π_2 or to forget p_2 in reducing π_1. So we cannot use cartesian products.

With tensor product, the linearity is kept: we can retreive information in both A and B of a product $A \otimes B$ in a linear manner using $(let \ \ \langle x, y \rangle = M \ \ in \ \ N)$. In that way, we

First let us define the type of the new term constant:

$$* \quad \mapsto \quad \top$$

For the μ abstraction, one need $FV(M) \cap |\Gamma| = \emptyset$ and $A' \multimap B' <: A \multimap B$:

$$\frac{\Gamma, !\Delta, f :!(A \multimap B), x : A' \rhd M : B'}{\Gamma, !\Delta \rhd \mu f x.M : (n+1)(A \multimap B)} \ (\infty)$$

$$\frac{!\Delta, \Gamma_1 \rhd M_1 : (n)(A_1) \quad \dots \quad !\Delta, \Gamma_k \rhd M_k : (n)(A_k)}{!\Delta, \Gamma_1, \dots, \Gamma_k \rhd \langle M_1, \dots, M_k \rangle : (n)(A_1 \otimes \dots \otimes A_k)} \otimes.I$$

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

Table 3: TYPING RULES FOR TUPLES, UNIT AND RECURSION

have obtained a monoidal category: we can define the linear functions:

$$\begin{array}{lccc|lccc} \sigma : & A \otimes B & \multimap & B \otimes A & \alpha : & (A \otimes B) \otimes C & \multimap & A \otimes (B \otimes C) \\ \lambda : & A \otimes \top & \multimap & A & \rho : & A & \multimap & A \otimes \top \end{array}$$

as following:

$$\begin{array}{r|l} \alpha = \lambda p.(let \ \langle x, y \rangle = p \ in & \sigma = \lambda p.(let \ \langle x, y \rangle = p \ in \ \langle y, x \rangle) \\ let \ \langle z, t \rangle = x \ in & \lambda = \lambda p.(let \ \langle x, y \rangle = p \ in \ x) \\ \langle z, \langle t, y \rangle \rangle) & \rho = \lambda x.\langle x, * \rangle \end{array}$$

Moreover, given $f : A \multimap B$ and $g : C \multimap D$, one can define $f \otimes g : A \otimes C \multimap B \otimes D$ by $f \otimes g = \lambda p.(let \ \ \langle x, y \rangle = p \ \ in \langle fx, gy \rangle)$. The above problem has this solution: (let $\langle x, y \rangle = M$ in (*CNOT* $\langle (Hx), y \rangle$) since linearity of the product's elements is preserved.

The typing rules to add are in Table 3. The reduction procedure for these new terms is found in Table 4.

Compatibility with the previous results All the previous lemmas still hold in the extended language, and the subject reduction still holds. For the subject reduction, we have to check that the new structures added, to know the recursion and the tuples, have rules that are compatible. The tuple rules are just an extension of the application rules, so using a similar method, it is working. For the recursion, suppose we have, with V a value,

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

with $[Q, (\mu f x.M)V]$ a program. Then one can find B such that
$\Delta \rhd (\mu f x.M)V : B$ and $\Delta = \{x{:}qbit | x \in FV((\mu f x.M)V)\}$. The only typing tree is

$$\frac{\dfrac{\Delta_1, f :!(A \multimap B), x : A' \rhd M : B'}{\Delta_1 \rhd \mu f x.M :!(A \multimap B)} \qquad \Delta_2 \rhd V : A}{\Delta_1, \Delta_2 \rhd (\mu f x.M)V : B}$$

If V is a value:
$$[Q, (\mu f x.M)V] \longrightarrow_1 [Q, M[\mu f x.M/f, V/x]]$$

If for all i, V_i is a value and N is linear in x_i if V_i is non-duplicable
$$[Q, \text{let } \langle x_1, \ldots, x_k\rangle = \langle V_1, \ldots, V_k\rangle \text{ in } N] \longrightarrow_1 [N[V_1/x_1]\ldots[V_n/x_n]]$$

One reduces a tuple from left to right:
$$\frac{[Q, M_1] \longrightarrow_p [Q', M_1']}{[Q, \langle M_1, \ldots, M_k\rangle] \longrightarrow_p [Q', \langle M_1', \ldots, M_k\rangle]}$$
$$\frac{[Q, M_2] \longrightarrow_p [Q', M_2']}{[Q, \langle M_1, M_2, \ldots, M_k\rangle] \longrightarrow_p [Q', \langle M_1, M_2', \ldots, M_k\rangle]}$$
$$\cdots$$
$$\frac{[Q, M_k] \longrightarrow_p [Q', M_k']}{[Q, \langle M_1, \ldots, M_k\rangle] \longrightarrow_p [Q', \langle M_1, \ldots, M_k'\rangle]}$$

Table 4: EXTENDED REDUCTION

with $|\Delta_1| = FV(\mu f x.M) \subseteq FV(M) \cup \{f, x\}$, $|\Delta_2| = FV(V)$ and $A' \multimap B' <: A \multimap B$. By the rule (∞), since there is no duplicable variable in Δ_1, $|\Delta_1| \cap FV(M) = \emptyset$. Since $A <: A'$ and $B' <: B$,

$$\Delta_1, f :!(A \multimap B), x : A \rhd M : B$$

is valid, and one can apply V to M from Lemma 11 and Lemma 12. Applying them to replace $\mu f x.M$ for f gives

$$\Delta \rhd M[\mu f x.M/f, V/x] : B$$

valid. □

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 [10, 11], our language is characterized by the fact that it combines classical as well as quantum features; thus, we have classical data types as well as quantum ones. We also provide both unitary operations and measurements as primitive features of our language.

As the language shows, linearity constraints do not just exist at base types, but also at higher-order types, due to the fact that higher-order functions are represented as closures which may in turns contain embedded quantum data. We have shown that affine intuitionistic linear logic provides precisely the right type system to deal with this situation.

There are many open problems for further work. It is an interesting question whether one can have an automatic type inference algorithm for the language described here; this question will be answered in the affirmative in the author's forthcoming M.Sc. thesis.

Another interesting question is whether the syntax of this language can be extended to include the additive types of linear logic, in addition to the multiplicative ones discussed here.

A very important open problem is to find a satisfactory denotational semantics for a higher order quantum programming language. One approach for finding such a semantics is to extend the framework of the work of Selinger [8] and to identify an appropriate higher-order version of the notion of a superoperator.

7 Acknowledgments

I would like to thank Peter Selinger for his supervision and his feedback.

References

[1] H. P. Barendregt, *The Lambda-Calculus, its Syntax and Semantics*, North Holland, 2nd edition, 1984.

[2] 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

[3] S. Bettelli, T. Calarco and L. Serafini: Toward an architecture for quantum programming, arXiv:cs.PL/0103009 v3, 2003

[4] D. Deutsch: Quantum theory, the Church-Turing principle and the universal quantum computer, *Proceedings of the Royal Society of London A 400*, 97-117, 1985.

[5] E. Knill: Convention for quantum pseudocode, *LANL report LAUR-96-2724*, 1996

[6] M. A. Nielsen and I. L. Chuang: *Quantum Computation and Quantum Information*, Cambridge University Press, 2002.

[7] J. Preskill. Lecture Notes for Physics 229: Quantum Information and Computation, 1998. Available at `http://www.theory.caltech.edu/people/preskill/ph229/#lecture`.

[8] P. Selinger: Toward a Quantum programming language. To appear in *Mathematical Structures in Computer Science*, 2004.

[9] J.W. Sanders and P. Zuliani: *Q*uantum Programming, Mathematics of Program Construction, Springer LNCS 1837, 80-99, 2000.

[10] A. van Tonder: A lambda calculus for quantum computation, arXiv:quant-ph/0307150 v5 (2004). To appear in *SIAM Journal of Computing*.

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