Proposed Thesis Research

Xiaoning Bian

Dalhousie University

Abstract

This is a written report for my specialist exam outlining my proposed thesis research. First I introduce some background on quantum computing (including circuits), lambda calculus and Curry-Howard isomorphism, intuituitionistic linear logic, quantum lambda calculus, and Proto-Quipper. Then I list some questions which I plan to address in the thesis research.

1 Quantum computing

1.1 Some facts about Hilbert spaces, operator, and tensor product

We are only interested in finite dimensional complex Hilbert spaces, operators between them, and their tensor product. So the definitions are adapted accordingly, and the properties irrelevant to the following sections are omitted. Let $\mathbb{C}$ be the set of complex numbers, and let $c^\dagger$ be the complex conjugate of $c \in \mathbb{C}$.

Definition 1.1. A finite *dimensional Hilbert space* $\mathbf{H}$ is a finite dimensional complex vector space $\mathbf{H}$, equipped with *complex inner product*, i.e., a function $\langle \_|\_ \rangle : \mathbf{H} \times \mathbf{H} \to \mathbb{C}$ satisfying (for all $c_1, c_2 \in \mathbb{C}$ and $v_1, v_2, v_3 \in \mathbf{H}$)

- linear in the second component, i.e., $\langle v_3|(c_1v_1 + c_2v_2)\rangle = c_1 \langle v_3|v_1\rangle + c_2 \langle v_3|v_2\rangle$.
- $\langle v_1|v_2\rangle = \langle v_2|v_1\rangle^\dagger$.
- $\langle v_1|v_1\rangle \geqslant 0$.

Remark 1.2. By the first two condition, $\langle \_|\_ \rangle$ is *conjugate linear* in the first component, i.e., $\langle (c_1v_1 + c_2v_2)|v_3\rangle = c_1^\dagger \langle v_1|v_3\rangle + c_2^\dagger \langle v_1|v_3\rangle$.

Definition 1.3. The *norm* on $\mathbf{H}$ induced by the complex inner product is $\|v\| = \sqrt{\langle v|v\rangle}$, for $v \in \mathbf{H}$.

Example 1.4. $\mathbb{C}^n$ is a finite dimensional Hilbert space equipped with the following complex inner product: Let $e_1, ..., e_n$ be the standard basis of $\mathbb{C}^n$, define the complex inner product between them $\langle e_i|e_j\rangle = 1$ if $i = j$, otherwise $\langle e_i|e_j\rangle = 0$, and then extend to all $v_1, v_2 \in \mathbb{C}^n$ uniquely using the linear and conjugate liner conditions. That is, for $v_1 = (c_1, c_2, ...c_n)$ and $v_2 = (d_1, d_2...d_n)$,

$$\langle v_1|v_2\rangle = c_1^\dagger d_1 + ... + c_n^\dagger d_n.$$

Definition 1.5. A linear map between Hilbert spaces is also called a (linear) *operator*.

Remark 1.6. Given an operator A from $\mathbf{H}$ to $\mathbf{H}'$, there is a unique operator $A^\dagger$ from $\mathbf{H}'$ to $\mathbf{H}$, such that for all vectors $v \in \mathbf{H}'$, $v \in \mathbf{H}$, $\langle v|Aw\rangle = \left\langle A^\dagger v|w \right\rangle$ (we omit the proof). We also note that $(A^\dagger)^\dagger = A$ without the proof. We call $A^\dagger$ the *adjoint* of A. An operator S on $\mathbf{H}$ (to itself) is *self-adjoint* if $S^\dagger = S$. An operator U on $\mathbf{H}$ is *unitary* if $UU^\dagger = U^\dagger U = I$, the identity operator.

Remark 1.7. When $\mathbf{H}$ is equipped with a chosen basis, the operators on $\mathbf{H}$ are in one-to-one correspondence with $n \times n$ complex matrices. We write $Mat(A)$ for the matrix representation of A. The matrix representation of $A^\dagger$ is the conjugate transpose of the matrix representation of A, i.e.,

$$Mat(A^\dagger) = Mat(A)^\dagger.$$

Definition 1.8. The *tensor product* of $\mathbb{H} \otimes \mathbb{H}'$ two complex vector spaces $\mathbb{H}$ and $\mathbb{H}'$ defined in the usual way. In particular, if $e_1, ..., e_n$ is a basis of $\mathbb{H}$ and $f_1, ..., f_m$ is a basis of $\mathbb{H}'$, then $\{e_i \otimes f_j\}_{i,j}$ is a basis of $\mathbb{H} \otimes \mathbb{H}'$. If $\mathbb{H}$ and $\mathbb{H}'$ are equipped with inner products, then there a unique inner product on $\mathbb{H} \otimes \mathbb{H}'$ satisfying

$$\langle e_i \otimes f_j | e_k \otimes f_l \rangle = \langle e_i | f_j \rangle \langle e_k | f_l \rangle .$$

We write $\mathbf{H} \otimes \mathbf{H}'$ for the Hilbert space equipped with this inner product. More generally, for $v_1, v_2 \in \mathbf{H}$, and $v_1', v_2' \in \mathbf{H}'$

$$\langle v_1 \otimes v_1' | v_2 \otimes v_2 \rangle = \langle v_1 | v_2 \rangle \langle v_1' | v_2' \rangle .$$

Remark 1.9. Not every vector in $\mathbb{H} \otimes \mathbb{H}'$ can be written in the form $v \otimes v'$. For example $e_1 \otimes e_1' + e_2 \otimes e_2'$ cannot be written in this form.

Definition 1.10. Given operators A on $\mathbf{H}$ and B on $\mathbf{H}'$, the *Kronecker tensor* of A and B is an operator on $\mathbf{H} \otimes \mathbf{H}$ defined by

$$(A \otimes B)(e_i \otimes e_j') = Ae_i \otimes Be_j'.$$

We usually omit the parentheses when applying an operator to a vector, i.e., $Av := A(v)$. We also write $AB = A \circ B$ for the composition of two operators. We note without proof that

$$(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger, \qquad (A \otimes B) \circ (A' \otimes B') = (A \circ A') \otimes (B \circ B').$$

Definition 1.11. Let **FdHilb** be the category of finite dimensional Hilbert spaces and linear operators, and let $\mathbf{FdHilb}_u$ be its subcategory of unitary operators. We are also interested in the following skeletal subcategories of **FdHilb** and $\mathbf{FdHilb}_u$: Let **C** be the category whose objets are natural numbers and whose morphisms $n \to m$ are linear maps $\mathbb{C}^n \to \mathbb{C}^m$. Let **U** be its subcategory of unitary maps.

Each category is equipped with a tensor product as above, making them into symmetric monoidal categories. Moreover, $\mathbf{FdHilb}_u$ and **U** are symmetric monoidal groupoids, and **C** and **U** are strict monoidal.

1.2 Mathematical framework

Definition 1.12. An *n-qubit state* ψ is an equivalence class of unit vectors in the 2^n dimensional complex Hilbert space $\mathbf{H}_n = (\mathbb{C}^{2^n}, \langle \_ | \_ \rangle)$, where v and v' are equivalent if they are collinear, i.e., $v = \lambda v'$, where λ is a unit scalar. Given an n-qubit state ψ and an m-qubit state φ, we can always get an $n+m$-qubit state $\psi \otimes \varphi$ by tensor product since

$$\langle \psi \otimes \varphi | \psi \otimes \varphi \rangle = \langle \psi | \psi \rangle \langle \varphi | \varphi \rangle = 1,$$

that is, $\psi \otimes \varphi$ is a unit vector.

In quantum computing, a state φ is written $|\varphi\rangle$ and called a *ket*. The adjoint of $|\varphi\rangle$ (think of $|\varphi\rangle$ as a linear operator from $\mathbb{C}$ to $\mathbf{H}_n$) is written as $\langle \psi|$ and called a *bra*. The *bracket* (same notation as the inner product) is

$$\langle \varphi | \, |\psi\rangle = \varphi^\dagger(\psi) = \left\langle 1 | \varphi^\dagger(\psi) \right\rangle = \langle \varphi(1) | \psi \rangle = \langle \varphi | \psi \rangle .$$

The *computational basis* of $\mathbf{H}_n$ is the standard bass of $\mathbb{C}^{2^n}$. We denote the elements of the computational basis by $\{|0\rangle, |1\rangle, ..., |2^n - 1\rangle\}$. In order to draw comparisons with boolean circuits from classical computer science, people sometimes write the number in 'ket' using binary string form. For example $|0\rangle$=$|0...0\rangle$, and $|2^n - 1\rangle$=$|1...1\rangle$, both of which have n digits. In this notation, $|b_1...b_m\rangle \otimes |b_1'...b_n'\rangle = |b_1...b_m b_1'...b_n'\rangle$. Note that for all the matrices (including vectors) in the following, we use the computational basis.

Example 1.13. A one qubit state is

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

where $|0\rangle$ and $|1\rangle$ are the computational basis of $\mathbf{H}_1$, and α, β are complex numbers satisfying $|\alpha|^2 + |\beta|^2 = 1$. Under this basis, $|0\rangle = (1, 0)^T$, and $|1\rangle = (0, 1)^T$. An n-qubit state is

$$|\psi\rangle = \Sigma_{i=0}^{2^n - 1} \alpha_i |i\rangle$$

where $|i\rangle$ is the computational basis of $\mathbf{H}_n$, and the α_i's satisfy $\Sigma |\alpha_i|^2 = 1$.

Example 1.14. Given a qubit $|0\rangle$ and another qubit $|1\rangle$, we can form a 2-qubit state $|0\rangle \otimes |1\rangle$. Note that, as already mentioned in Remark 1.9, not every 2-qubit state can be written as a tensor product of two 1-qubit states, for example

$$|\psi\rangle = \frac{1}{\sqrt{2}}(0, 1, -1, 0)^T = \frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle - |1\rangle \otimes |0\rangle)$$

cannot be written in this form. In this case, we say that $|\psi\rangle$ is an *entangled state.*

Definition 1.15. A *n-qubit gate* is a unitary operator U on $\mathbf{H}_n$.

Remark 1.16. Note that there doesn't exist a qubit copying gate, i.e., no unitary operator satisfies

$$U\,|\varphi\rangle \otimes |k\rangle = |\varphi\rangle \otimes |\varphi\rangle$$

where $|\varphi\rangle$ is any n-qubit state, and $|k\rangle$ is some fixed n-qubit state. Say there were a U satisfying this, pick two n-qubit states $|\varphi_1\rangle$ and $|\varphi_2\rangle$, then

$$U\,|\varphi_1\rangle \otimes |k\rangle = |\varphi_1\rangle \otimes |\varphi_1\rangle\,, \qquad U\,|\varphi_2\rangle \otimes |k\rangle = |\varphi_2\rangle \otimes |\varphi_2\rangle\,.$$

The inner product of two states in the left side of the equations should equal to the one in the right side. Therefore

$$\langle\varphi_1|\varphi_2\rangle = (\langle\varphi_1|\varphi_2\rangle)^2.$$

This only happens when $\langle\varphi_1|\varphi_2\rangle = \{0, 1\}$, i.e., $|\varphi_1\rangle$ and $|\varphi_2\rangle$ are the same or orthogonal. This is called the *no-cloning property.*

Example 1.17. The following are one qubit gates

$$I_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \omega = e^{i\pi/4}, \quad H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 0 \\ 0 & \omega \end{pmatrix}$$

By abuse of notation, we identify ω with ωI_1. Also, since we identify qubit states up to a unit scalar, this scalar has no effect when it acts on a state. The following are 2-qubit gates

$$I_1 \otimes X = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad Z_c = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, \quad \text{CNot} = X_c = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad U_c = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & U_{00} & U_{01} \\ 0 & 0 & U_{10} & U_{11} \end{pmatrix}$$

where $U = \begin{pmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{pmatrix}$ is a 1-qubit gate. U_c is called a *controlled U gate*, and it acts on 2-qubit states in the following way

$$U_c\,|0b\rangle = |0b\rangle\,, \qquad U_c\,|1b\rangle = (I \otimes U)\,|1b\rangle$$

where $b = 0, 1$. That is when the first digit is 0, U_c does nothing on the state, and when the first digit is 1, it applies $I \otimes U$ to the state, or informally it applies U on the second qubit. The CNot and Z_c gates are special cases of controlled U gates.

Definition 1.18. An n-qubit measurement M is given by a finite family $\{S_i\}_i$ of operators satisfying the *completeness equations* $\Sigma_{i\in I} S_i^\dagger S_i = I$. When performing the measurement M on a quantum state $|\varphi\rangle \in \mathbf{H}_n$, one of the *measurement outcomes* $i \in I$ will be observed and the state will change. Given an n-qubit state $|\varphi\rangle \in \mathbf{H}_n$, the probability of observing measurement outcome $i \in I$ is

$$P(i) = ||S_i\,|\varphi\rangle\,||^2 = \langle\varphi|\,S_i^\dagger S_i\,|\varphi\rangle\,.$$

Note that $\Sigma_{i\in I} P(i) = \Sigma_{i\in I}\,\langle\varphi|\,S_i^\dagger S_i\,|\varphi\rangle = \langle\varphi|\,I\,|\varphi\rangle = 1$. Moreover, the state after the measurement is

$$t(i) = \frac{S_i\,|\psi\rangle}{\sqrt{p(i)}},$$

provided that outcome i was observed.

Given an m-qubit measurement M (specified by $\{S_i\}_{i\in I}$), and an n-qubit measurement M' (specified by $\{S_i\}_{i\in I}$), we can get an $m + n$-qubit measurement $M \otimes M'$ determined by $\{S_i \otimes S'_j\}_{i\in I, j\in J}$, since they still satisfy the completeness equations. We call $M \otimes M'$ a *tensor measurement.*

Example 1.19. The trivial measurement, let $I = \{*\}$ and S_* be the identity operator I. For any n-qubit state $|\varphi\rangle$, the measurement outcome will be $*$ with probability 1 and the state is unchanged.

Example 1.20. Let $I = \{0, 1\}$, $S_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ and $S_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$. It is easy to verify that S_0 and S_1 satisfy the completeness equations. Applying this measurement to a state $\alpha |0\rangle + \beta |1\rangle$ will yield outcome 0 with probability $|\alpha|^2$, outcome 1 with probability $|\beta|^2$, and the state after the measurement is $|0\rangle$ or $|1\rangle$, respectively. This is called *computational basis measurement.*

Example 1.21. Given a 1-qubit measurement M, let I be the trivial one-qubit measurement as in Example 1.19. The tensor measurement $I \otimes M$ is also called the *measurement on the second qubit.* Similarly we can define measurement on any subset of qubits.

1.3 QRAM model

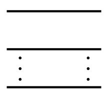
A useful abstract model of a quantum computer is the so-called *QRAM model* [11]. In this model, we assume to have access to N numbered qubits for some fixed, large enough N. That is the QRAM device holds an N-qubit state. The available operations are:

- prepare qubit i in state $|0\rangle$ or $|1\rangle$, as we instruct.
- apply an n-qubit gates U from a fixed finite set of gates to n distinct qubits $i_1, i_2, ..., i_n$.
- measure qubit i in the computational basis (see Example 1.20), and tell us the measurement result (0 or 1).

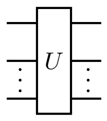
We symbolically write theses operations as $init_b(i)$, $U \langle i_1, i_2, ..., i_n\rangle$, and $meas(i)$, respectively.

1.4 Quantum circuit

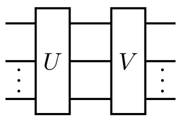
Definition 1.22. Recall that the strict symmetric monoidal groupoid **U** from Definition 1.11. Quantum circuit are string diagrams for symmetric monoidal groupoid [10]. Specifically, a quantum circuit is a graphical way to represent a morphisms in **U**. The object n is represented by n paralleled wires:



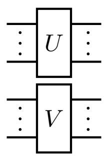
A morphism U from n to n is represented like this:



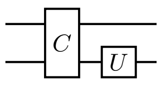
Composition is represented by connecting wires:



Tensor is represented by juxtaposing wires (for objects) and boxes (for morphisms):



Example 1.23. The following is a quantum circuit for the operator $(I \otimes U) \circ C$ (we read the circuit from left to right, and from top to bottom)



2 Lambda calculus

The lambda calculus was introduced by Church and Curry in the 1930's [2, 3]. We only summarize the most important notions, for a more detailed account see [1].

2.1 The untyped lambda calculus

2.1.1 Lambda terms

Definition 2.1. Let $\mathbb{V}$ be a countably infinite set of symbols which we call *variables.* Let $\mathbb{A}$ be the disjoint union of $\mathbb{V}$ and the set $\mathbb{S}$ of special symbols '(','$)$','λ', and '.'. Let $\mathbb{A}^*$ be the set of strings (finite sequences) over the alphabet $\mathbb{A}$. The set of *lambda terms* is the smallest subset $\mathbb{L} \subseteq \mathbb{A}^*$ such that:

- For any $x \in \mathbb{V}$, $x \in \mathbb{L}$
- For any $M, N \in \mathbb{L}$, $(MN) \in \mathbb{L}$
- For any $x \in \mathbb{V}$ and $M \in \mathbb{L}$, $(\lambda x.M) \in \mathbb{L}$

We can use the BNF (Backus-Naur Form) notation for convenience.

$$\text{Lambda terms} \quad M, N \quad ::= \quad x \quad | \quad (MN) \quad | \quad (\lambda x.M)$$

Terms of these forms are called *variables*, *applications*, and *lambda abstractions* respectively. An occurrence of a variable x inside a term of the form $\lambda x.N$ is said to be *bound.* The corresponding λx is called a *binder*, and we say that the subterm N is the *scope* of the binder. A variable occurrence that is not bound is *free.* Thus for example, in the term $(\lambda x.xy)(\lambda y.yz)$, x is bound, but z is free. The variable y has both a free and a bound occurrence. Let $FV(M)$ be the set of variables having free occurrences in M. If $FV(M)$ is empty, we say M is *closed.*

We can think of a lambda abstraction as a function $x \mapsto M$. We sometimes write $M(x)$ to emphasize that M contains a variable x. An application is like the function application — M is a function, and N is its argument.

Example 2.2. The following are lambda terms:

$$x \quad (xx) \quad (\lambda x.x) \quad (\lambda z.((yy)(xx)))$$

The last lambda term above contains many parentheses, and is not easy to read. We have the following conventions to make terms look simpler:

- The outermost parentheses are omitted, so $\lambda x.M$ is $(\lambda x.M)$.
- Applications associate to the left i.e. MNO stands for $(MN)O$.
- The body of a lambda abstraction extends as far to the right as possible, so $\lambda x.MN$ means $\lambda x.(MN)$.
- Multiple lambda abstractions can be contracted, so $\lambda x.\lambda y.\lambda z.M$ can be written as $\lambda xyz.M$.

With these conventions, the terms from Example 2.1 are written x, xx, $\lambda x.x$ and $\lambda z.yy(xx)$, respectively.

2.1.2 Alpha equivalence

Informally, the names of bound variables do not matter. For example $\lambda x.x$ and $\lambda y.y$ represent the same function, i.e., the identity function. This is made more precise in the following definition.

Definition 2.3. α-equivalence is the smallest equivalence relation that satisfies the following conditions:

$$\frac{M \sim M' \quad N \sim N'}{MN \sim M'N'} \qquad \frac{M \sim M'}{\lambda x.M \sim \lambda x.M'} \qquad \frac{y \notin M}{\lambda x.M \sim \lambda y.M[y/x]}.$$

Here $M[y/x]$ is the term with all the free occurrences of x being replaced by y, and $y \notin M$ means that the variable y doesn't appear (free or bound) in M. The horizontal line stands for 'implying' i.e. if the conditions above the line hold, then the conclusion below the line is valid. Comparing with the definition of lambda terms, we can see that α-equivalence is an equivalence relation compatible with the term forming rules. Later we will introduce more term forming rules, and we can extend α-equivalence accordingly.

From now on, we only consider terms up to α-equivalence, i.e., terms different only in the choice of bound variables are considered the same.

2.1.3 Beta reduction

To make the application MN behave like function application, we need to put the argument N into the term M. This is the purpose of β-reduction. For M to be a 'function', M must be a lambda abstraction, say $M = \lambda x.M'$. Then the application MN should equal to $M'[N/x]$, i.e., the term M' with all free variables x being replaced by N.

We must be careful in how to define $M'[N/x]$. For example, if $M' = \lambda x.y$, and $N = x$, it could be incorrect to define $M'[N/x] = \lambda x.x$, because the free variables x of N has been 'captured' by the binder λx, contrary to our principle that the names of bound variables should not matter. To avoid this this 'capturing' of free variables, it is necessary to first rename the bound variables of M' when necessary. The precise definition of capture avoiding substitution can be found in [1]; for our purposes, the above informal description will suffice.

Definition 2.4. The *single step β-reduction* is the smallest relation on lambda terms which is compatible with the term forming rules, and satisfies the following condition:

$$\frac{}{(\lambda x.M)N \to M[N/x]}.$$

By "compatible with the term forming rules", we mean that it satisfies the following conditions:

$$\frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'} \qquad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}.$$

For a term M, if no reduction rule applies, we say it is in *normal form*. Note that later when we introduce new term forming rules, we require that β-reduction is compatible with the new rules. Also, we want to reduce the new terms and hence we will need new reduction rules.

The following theorem states some interesting properties of the untyped lambda calculus.

Theorem 2.5 (Church-Rosser). *Let $\twoheadrightarrow$ be the transitive closure of $\to$. Suppose M, N, and P are lambda terms such that $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there exists a lambda term Z such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.*

Corollary 2.6 (Normal form is unique). *For a term M, if there is a reduction $M \twoheadrightarrow N$ with N in normal form, then N is unique (up to α-equivalence).*

2.2 The simply typed lambda calculus

In the untpyed lambda calculus, unlike in mathematics, functions do not have a domain or a codomain; for example xx is a literally well-formed untyped lambda term, where x is both the function and the argument. To rule out such terms, we can introduce type system. Informally, a type system assigns a domain and codomain to each function. A λ-abstractions (function) from a type (domain) A to a type (codomain) B has a type $A \to B$. (We also have other reasons to introduce types for the lambda calculus such as we will get a correspondence with logic, and such as well typed terms enjoy some good properties and so on, see Section 3.)

Definition 2.7. Let $\mathbb{B}$ be a finite set of symbols which we call the *basic types* denoted by $\{A, B, C, ...\}$. The *simple types* $\mathbb{T}$ consists of types of the following form:

$$S, T \quad ::= \quad B \quad | \quad S \to T$$

where B ranges over $\mathbb{B}$. Here we have used a BNF definition, which can be translated to an equivalent mathematical definition like in Definition 2.1. For example, if A, B are basic types, the following are types: A, $A \to B$, $(A \to B) \to A, \ldots$. We call the type of the form $S \to T$ a *function type.*

We need to assign types to lambda terms in a meaningful way. Recall that we identity terms up to α-equivalence; for convenience, we will always assume without loss of generality that all bound variables are chosen so that they don't clash with anything.

Definition 2.8. An *assignment of types* is a function from some finite subset of $\mathbb{V}$ to $\mathbb{T}$. We write Γ as $\{x : A, y : B, ...\}$ to denote that x has type A etc. We write $x : A \in \Gamma$ to denote x is assigned the type A. We write $\Gamma, x : A$ to denote the new assignment with $x : A$ added to Γ. A *typing judgement* is a expression of the form $\Gamma \vdash M : A$ (formally a triple (Γ, M, A) of an assignment, a term, and a type). A typing judgement is *valid* if it follows from the following *typing rules*:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

Informally $\Gamma \vdash M : A$ means that the assignment Γ implies M has type A, and if $\Gamma \vdash M : A$ is a valid, we say M is *well-typed* under Γ. Note that, for the application MN to have a type, M and N must have the right types. This amounts to requiring the function M must be applied to a correctly typed argument N.

Under these rules, not all lambda terms can be assigned a type. For example xx. It is an application but not satisfying the conditions of the application typing rule since x cannot have types $A \to B$ and A at the same time. The well-typed terms enjoy some good properties such as:

Theorem 2.9 (Subject Reduction)**.** *If* $\Gamma \vdash M : A$ *and* $M \to M'$*, then* $\Gamma \vdash M' : A$.

Definition 2.10. A term M is *weakly normalizing* if there exists a finite sequence of reductions $M \to M' \to M''... \to N$, such that N is in normal form, and *strongly normalizing* if every sequence of reductions is finite.

Theorem 2.11 (Strong normalization)**.** *If* $\Gamma \vdash M : A$*, then* M *is strong normalizing which implies weak normalizing.*

And of course, the well-typed lambda terms enjoy the Church-Rosser property, and the uniqueness theorem.

Another perspective to look at well-typed lambda terms is that they are picked up from all lambda terms and they behave as we want them to (i.e., lambda abstraction works as a function, and application works as a function application), but the remaining terms ('ill-typed lambda terms') are not well-behaved, and hence being dropped. We can think of the typing system as providing a way to eliminate 'ill-typed lambda terms'. We can also thinks of a type T as the set of all lambda terms of type T, which makes the 'types' more concrete, rather than just string of symbols.

3 The Curry-Howard Correspondence

We give a very brief overview of the Curry-Howard correspondence (CH correspondence). For more details see [6].

3.1 Intuitionistic propositional logic

Informally, an atomic proposition is a statement or assertion that must be true or false. Propositional formulas (only the implication fragment) are constructed from atomic propositions by using the logical connective $\to$ (called *implication*).

Definition 3.1. Let $\mathbb{B}$ be countable set of *atomic propositions*, denoted by $\{A, B, ...\}$. Let $\mathbb{T}$ be the set of *propositional formulas*, given via the BNF ($B \in \mathbb{B}$)

$$\text{F}ormulas: \quad S, T \quad ::= \quad B \quad | \quad S \to T.$$

Note that this is the same as Definition 2.7, if we identify the atomic formulas and basic types.

The *natural deduction* system (Gentzen[4, 5] and Prawitz [12]) formalizes how to make a proof. Here we introduce the implication fragment of intuitionistic natural deduction. A *sequent* is an expression $\Gamma \vdash A$ where Γ is a finite multiset of formulas, called a *context*, and A is a formula. The intended interpretation is that the assumptions in Γ imply A. We have three natural ways (derivations) to prove a formula.

$$\frac{A \in \Gamma}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}.$$

Here Γ, A is a new multiset with A added in the end. Comparing with the typing rules of lambda terms, we find that they are the same if we drop all the lambda terms and variables. This is the first part of the so-called Curry-Howard correspondence (Curry [3] and Howard [9]).

3.2 Proofs as terms

The second part of Curry-Howard correspondence is that a well-typed lambda term can be seen as a proof (using the three derivation rules and finite assumptions) of its type (formula), more precisely:

Theorem 3.2 (proofs-as-terms)**.** *The well-typed lambda terms of type* A *under assignment* Γ *are in one-to-one correspondence with the proofs of* A *under assumptions* Γ*. As a special case, well-typed closed lambda terms of type* A *are proofs of* A *(from no assumptions).*

Sketch of the proof. Given a derivation of $\Gamma \vdash A$, we define a typing judgement $\Gamma^L \vdash M : A$, and then prove it is valid using induction.

- Γ^L is obtained by labeling the assumptions in $\Gamma = \{A_1, A_2, ..., A_n\}$ using distinct variables $x_1, x_2, ..., x_n$ from $\mathbb{V}$.
- $M = \text{Top}(\Gamma \vdash A)$ is obtained by recursion on the derivation of $\Gamma \vdash A$. 'Top' is short for Term Of Proof, and is defined as follows:
    1. $\text{Top}(\frac{A \in \Gamma}{\Gamma \vdash A}) = x$, and x is the label of A in Γ^L.
    2. $\text{Top}(\frac{\Gamma \vdash B \to A \quad \Gamma \vdash B}{\Gamma \vdash A}) = M_1 M_2$, where $M_1 = \text{Top}(\Gamma \vdash B \to A)$, and $M_2 = \text{Top}(\Gamma \vdash B)$.
    3. $\text{Top}(\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}) = \lambda x.N$, where $N = \text{Top}(\Gamma, A \vdash B)$, and x is the label of A in $(\Gamma, A)^L$.

Now we prove $\Gamma^L \vdash M : A$ is valid. If $M = x$, then it is valid trivially. If $M = M_1 M_2$, then by induction hypothesis, $\Gamma^L \vdash M_1 : B \to A$ and $\Gamma^L \vdash M_2 : B$ are valid, so $\Gamma^L \vdash M : A$ is valid by the second typing rule. Similarly for the case $M = \lambda x.N$.

Conversely, given well-typed term M, fix a Γ under which M is well-typed, then there is only one typing derivation for the typing judgement $\Gamma \vdash M : A$ since the last typing rule used is unique. We define a derivation of $\Gamma^F \vdash A$ by recursion on the typing derivation of $\Gamma \vdash M : A$. Here, Γ^F is the assumptions dropping all the variables, and Pot is short for Proof Of Terms.

1. $\text{Pot}(\frac{x:A \in \Gamma}{\Gamma \vdash x:A}) = \frac{A \in \Gamma^F}{\Gamma^F \vdash A}$.
2. $\text{Pot}(\frac{\Gamma \vdash M_1 : B \to A \quad \Gamma \vdash M_2 : B}{\Gamma \vdash M_1 M_2 : A}) = \frac{D_1 \quad D_2}{\Gamma^F \vdash A}$, where $D_1 = \text{Pot}(\Gamma \vdash M_1 : B \to A)$, and $D_2 = \text{Pot}(\Gamma \vdash M_2 : B)$.
3. $\text{Pot}(\frac{\Gamma, x:A \vdash N:B}{\Gamma \vdash \lambda x.N : A \to B}) = \frac{D'}{\Gamma^F \vdash A \to B}$, where $D' = \text{Pot}(\Gamma, x : A \vdash N : B)$.

What Pot does is essentially dropping all the lambda terms and variables in the type derivation. Similarly by induction, the derivation we got complies with the natural deduction rules, i.e., it is valid. □

The *Curry-Howard Correspondence* refers to the correspondence between types and formulas, and between terms and proofs.

4 Intuitionistic propositional linear logic

As we noted before, a context of intuitionistic logic is a multiset. The rules of intuitionistic logic permit us to treat it as a set, i.e., to permute the elements and to identify equal elements.

Lemma 4.1 (Exchange). *If* $\Gamma, A, B, \Delta \vdash C$ *is valid, then* $\Gamma, B, A, \Delta \vdash C$ *is valid.*

Lemma 4.2 (Contraction). *If* $\Gamma, A, A, \Delta \vdash C$ *is valid, then* $\Gamma, A, \Delta \vdash C$ *is valid.*

Lemma 4.3 (Weakening). *If* $\Gamma, A, \Delta \vdash C$ *is valid, then* $\Gamma, A, B, \Delta \vdash C$ *is valid.*

In other words, the following so-called structural rules are derived rules:

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} ex, \qquad \frac{\Gamma, A, A, \Delta \vdash C}{\Gamma, A, \Delta \vdash C} contr, \qquad \frac{\Gamma, A, \Delta \vdash C}{\Gamma, A, B, \Delta \vdash C} w$$

What happens if we disallow the contraction and weakening rules? (We will always allow the exchange rule). To disallow the contraction rule means if we use A twice to prove C, we may not be able to prove C using A only once. To disallow the weakening rule means if we use one A to prove C, we may not be able to prove C given A and additional assumptions. If we think of assumptions and conclusion as resources, things becomes clear. If it is possible to produce C using two A's, then it may not be possible to produce C using only one A. If consuming one A produces a C, no one likes to spend more resources B.

Linear logic is a formalization of this idea.

Definition 4.4. Formulas for intuitionistic propositional linear logic are defined via BNF:

$$Formulas: \quad S, T \quad ::= \quad A \quad | \quad S \otimes T \quad | \quad S \multimap T \quad | \quad S \& T \quad | \quad S \oplus T \quad | \quad !S$$

where A ranges over a countable set of atomic propositions $\mathbb{A}$. $\otimes$, $\multimap$, &, and $\oplus$ are (binary) *connectives*. ! reads as 'bang'.

The natural deduction derivation rules:

$$\frac{}{A \vdash A}id, \quad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}contr, \quad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}w$$

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}!_I, \quad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}cut$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \multimap_I, \quad \frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B} \multimap_E$$

$$\frac{\Gamma \vdash A, \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\otimes_I, \quad \frac{\Gamma \vdash A \otimes B \quad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C}\otimes_E$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\oplus_{I1}, \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\oplus_{I2}, \quad \frac{\Gamma \vdash A \oplus B \quad \Delta, A \vdash C \quad \Delta, B \vdash C}{\Gamma, \Delta \vdash C}\oplus_E$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B}\&_I, \quad \frac{\Gamma \vdash A \& B}{\Gamma \vdash A}\&_{E1}, \quad \frac{\Gamma \vdash A \& B}{\Gamma \vdash B}\&_{E2}$$

Note that the rule (ex) continues to be a derived rule because contexts are multisets. One way to understand the difference between rules $\otimes$, &, and $\oplus$ is via the resource interpretation:

- if given resources A and B, then we can produce $A \otimes B$, and if given a resource $A \otimes B$, we can produce both A and B;
- if using resources Γ we can choose to produce A or B, but only one of them, then we can produce $A\&B$, and if using resources Γ we can produce $A\&B$, then we have the right to choose to produce A or B.
- if using resources Γ we can produce A or B, then we can produce $A \oplus B$, and if using resources Γ we can produce $A \oplus B$, then we can produce A or B (we don't know which one). $\oplus$ is the 'or' connective but in a linear logic background.

$\multimap$ is called *linear implication*. It is like the the usual implication but with a linear logic background. $A \multimap B$ can also be explained by resource interpretation: consuming A, producing B, and having resources $A \multimap B$ and A, we can produce B.

$!A$ is some resource that can be used any number of times (including 0). So the contraction and weakening rules for 'banged' resources still hold. $!\Gamma$ in rule $!_I$ stands for a context with all formulas of the form $!A$. The rule $!_I$ means that if all the resources used to produce A can be used many times, then we can produce A many times.

5 Quantum Lambda Calculus

There are several versions of Quantum Lambda Calculus (QLC) (see [20, 14, 15, 16, 17, 21, 18], and [13]). Here we introduce the version [21] with a slightly change in formalization. The main change is that we use *quantum names*, instead of variables in the formalization of the operational semantics.

5.1 Types, terms, and values

Definition 5.1. The types of QLC are defined by

$$Type \quad A, B \quad ::= \quad \top \quad | \quad bit \quad | \quad qubit \quad | \quad A \multimap B \quad | \quad A \otimes B \quad | \quad !A.$$

The intending meaning of types is as follows: $\top$ is a singleton type containing a unique element $*$, which we can think of as the 0-tuple; *bit* is a two-element type containing only two constants 0 and 1; *qubit* is the type of quantum names, i.e., pointers to a qubit in the QRAM; $A \multimap B$ is the function type ($\multimap$ comes form linear logic, see section 4); $A \otimes B$ is the type of pairs $\langle a, b \rangle$ of elements a of type A and b of type B; $!A$ the subset of elements of type A that can be used many times (including 0 times). We use $\mathbb{T}$ to denote the set of all types. We use the following convention when writing a type:

$$!^n A := !!...!A \qquad A^{\otimes n} := A \otimes A \otimes ... \otimes A$$

Definition 5.2. The terms of QLC are defined by

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

where U ranges over a given set $\mathbb{U}$ of term constants, q ranges over a given infinite set of *quantum names*, and x ranges over $\mathbb{V}$ (as before, the variables).

The intended meaning of abstraction and application is the same as before. The term $\langle M,N\rangle$ means a pair of terms M and N; the $*$ is the unique 0-tuple; the term $let\,\langle x,y\rangle = M\ in\ N$ is the program which first evaluates M to compute a pair $\langle V,W\rangle$, then assigns V to x and W to y before executing N; the term $let * \ = M\ in\ N$ is a syntax for reduction of M and returning nothing; the terms 0 and 1 mean 'false' and 'true', and the term $if\ P\ then\ M\ else\ N$ stands for a case distinction. When $P = 0$ (respectively 1), this term will be reduced to M (respectively N).

new is a function for state preparation. It inputs a classical bit, i.e., $0,1$, and outputs a quantum bit prepared in state $|0\rangle$, and $|1\rangle$ respectively. *meas* is a function for measurement. It inputs a quantum bit, measures it in computational basis $\{|0\rangle, |1\rangle\}$, and outputs the result as a classical bit. The term constants in $\mathbb{U}$ stand for unitary gates of varying arities. For example, we probably require $\mathbb{U}$ to contain $H, S, T, Cnot$, and $Swap$ etc. q should be considered as a label of a qubit in QRAM.

The operational semantics of QLC uses a call-by-value strategy, i.e., when doing a β-reduction, one first reduces the argument to a value. A value is informally a term that cannot be reduced.

Definition 5.3. A *value* of QLC is a term defined by

$$\begin{array}{lcl} \text{Value } V,W & ::= & x \ \mid\ \lambda x.M \\ & \mid & * \ \mid\ \langle V,W\rangle \\ & \mid & 0 \ \mid\ 1 \\ & \mid & new \ \mid\ meas \ \mid\ U \ \mid\ q \end{array}$$

5.2 Type system

5.2.1 Subtyping rules

We say that A is a subtype of B, in symbols $A <: B$, if every term that has type A also has type B. For example, $!A$ is a subtype of A, because a term that can be duplicated doesn't have to be duplicated. Also, $!!A$ and $!A$ are subtypes of each other, and in this case, we say that the types are *equivalent*. More formally, subtyping is defined as follows.

Definition 5.4. We define the subtyping relation $<:$ to be the smallest relation on types satisfying the following rules:

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

The $\multimap$-rule is because a function that takes arguments of type A can be applied to arguments of type A' (seen as subset of A). The function of output type B' can be seen as having output type B (seen as a superset of B'). We write $A \equiv B$ if $A <: B$ and $B <: A$.

5.2.2 Typing rules

Typing judgements are slightly different from Definition 2.8.

Definition 5.5. A typing judgement is a quadruple

$$\Gamma; Q \vdash M : T$$

where Γ is a type assignment (a function from a finite subset of $\mathbb{V}$ to types), Q is a finite subset of Q called a *quantum context*, M is a term and T is a type. A typing judgement is valid if it can be inferred using the rules in Table 1.

$$\frac{!\Delta, \Gamma_1; Q_1 \vdash M : A \multimap B \quad !\Delta, \Gamma_2; Q_2 \vdash N : A}{!\Delta, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash MN : B} app$$

$$\frac{\Gamma, x : A; Q \vdash M : B}{\Gamma; Q \vdash \lambda x.M : A \multimap B} \lambda_1 \qquad \frac{!\Delta, x : A; \emptyset \vdash M : B}{!\Delta; \emptyset \vdash \lambda x.M : !(A \multimap B)} \lambda_2$$

$$\frac{!\Delta, \Gamma_1; Q_1 \vdash M : !^n A \qquad !\Delta, \Gamma_2; Q_2 \vdash N : !^n B}{!\Delta, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash \langle M, N \rangle : !^n (A \otimes B)} \otimes .I \qquad \frac{}{!\Delta; \emptyset \vdash * : \top} \top$$

$$\frac{!\Delta, \Gamma_1; Q_1 \vdash M : !^n (A \otimes B) \quad !\Delta, \Gamma_2, x : !^n A, y : !^n B; Q_2 \vdash N : C}{!\Delta, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash let \ \langle x, y \rangle = M \ in \ N : C} \otimes .E$$

$$\frac{!\Delta, \Gamma_1; Q_1 \vdash P : \ bit \quad !\Delta, \Gamma_2; Q_2 \vdash M : A \quad !\Delta, \Gamma_2; Q_2 \vdash N : A}{!\Delta, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash if \ P \ then \ M \ else \ N : A} if$$

$$\frac{A <: B}{!\Delta, x : A; \emptyset \vdash x : B} ax_v \qquad \frac{}{!\Delta; \{q\} \vdash q : qubit} ax_q \qquad \frac{}{!\Delta; \emptyset \vdash c : A_c} ax_c$$

Table 1: Typing rules for QLC

There is an implicit condition for all rules that the domain of $!\Delta, \Gamma_1, \Gamma_2$ are disjoint from each other and Q_1 and Q_2 are disjoint. In all rules, $n \geqslant 0$. In the rule ax_c,

$$A_0, A_1 = bit \qquad A_U = !(qubit^{\otimes m} \multimap qubit^{\otimes m})$$
$$A_{new} = !(bit \multimap qubit) \qquad A_{meas} = !(qubit \multimap bit).$$

Note that using these rules, we cannot get a term of type $qubit \to qubit \otimes qubit$. In this sense, our typing system enforces the 'non-cloning' theorem.

5.3 Operational semantics

In lambda calculus, we only have one reduction rule — β-reduction (together with the congruence rules). In QLC, we have more terms, so we need to introduce new rules and new congruence rules. Moreover as the example in [18] shows, different reduction strategies can affect the computation result. For example, in an application $(\lambda x.M)N$, one strategy is to reduce N first and then do a substitution. This is call the *call-by-value* strategy. Another strategy is to always do the substitution first, before reducing N. This is called the *call-by-name* strategy. Since different evaluation strategies give different results, we need to choose a strategy, which needs more rules to describe.

The reduction rules are defined on closures. This is because a quantum name is intended to point to a qubit in the QRAM, and the name itself only serves as a reference constant and has no computational ability. Informally speaking, a closure consists of a *quantum state*, a QLC term (*classical control*), and a linking list assigning quantum names to specific qubit in the quantum state.

Definition 5.6. A *quantum closure* is a triple $[Q, L, M]$ where

- Q is a n-qubit state.
- L is a list of n distinct quantum names, written as $|q_1 q_2 ... q_n\rangle$.
- M is a QLC term.

If $L = |q_1 ... q_n\rangle$, we say that q_i points to the ith qubit of the quantum state.

Example 5.7. The quantum closure

$$[\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), |pq\rangle, \lambda x.xpq]$$

denotes a term $\lambda x.xpq$ with two embedded qubits p, q in the entangled state $|pq\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

Definition 5.8. The reduction rules are shown in Tables 2–4. We write $[Q, L, M] \to_p [Q', L', M']$ for a single-step reduction of quantum closures that takes place with probability p. In the rules for $\lambda x.M$, *let*, and *if*, $M[V/x]$ denotes the term M where the free variable x has been replaced by V (renaming bound variables if necessary). In the rule

$$[Q, L, (\lambda x.M)V] \to_1 [Q, L, M[V/x]]$$

$$[Q, L, let\ \langle x, y\rangle = \langle V, W\rangle\ in\ M] \to_1 [Q, L, M[V/x, W/y]]$$

$$[Q, L, let * = *\ in\ N] \to_1 [Q, L, N]$$

$$[Q, L, if\ 0\ then\ M\ else\ N)] \to_1 [Q, L, N]$$

$$[Q, L, if\ 1\ then\ M\ else\ N)] \to_1 [Q, L, M]$$

Table 2: Reduction rules: classical control

$$[Q, |q_1, ...q_n\rangle, U\langle q_{j_1}, ..., q_{j_n}\rangle] \to_1 [Q', |q_1, ...q_n\rangle, \langle q_{j_1}, ..., q_{j_n}\rangle]$$

$$[\alpha|Q_0\rangle + \beta|Q_1\rangle, |q_1, ...q_n\rangle,\ meas\ q_i] \to_{|\alpha|^2} [Q_0, |q_1, ...q_n\rangle, 0]$$

$$[\alpha|Q_0\rangle + \beta|Q_1\rangle, |q_1, ...q_n\rangle,\ meas\ q_i] \to_{|\beta|^2} [Q_1, |q_1, ...q_n\rangle, 1]$$

$$[Q, |q_1, ...q_n\rangle,\ new\ 0] \to_1 [Q \otimes |0\rangle, |q_1, ...q_n, q_{n+1}\rangle, q_{n+1}]$$

$$[Q, |q_1, ...q_n\rangle,\ new\ 1] \to_1 [Q \otimes |1\rangle, |q_1, ...q_n, q_{n+1}\rangle, q_{n+1}]$$

Table 3: Reduction rules: quantum data

for reducing the term $U\langle x_{j_1}, ..., x_{j_n}\rangle$, U is an n-ary built-in unitary gate, $j_1, ..., j_n$ are pairwise distinct, $\langle x_{j_1}, ..., x_{j_n}\rangle$ is short for $\langle x_{j_1}, \langle x_{j_2}, ...\rangle\rangle$ and Q' is the quantum state obtained from Q by applying this gate to qubits $j_1, ..., j_n$. In the rule for measurement, $|Q_0\rangle$ and $|Q_1\rangle$ are states of the form

$$|Q_0\rangle = \Sigma_j \alpha_j \left|\varphi_j^0\right\rangle \otimes |0\rangle \otimes \left|\psi_j^0\right\rangle, \quad |Q_1\rangle = \Sigma_j \beta_j \left|\varphi_j^1\right\rangle \otimes |1\rangle \otimes \left|\psi_j^1\right\rangle$$

where φ_j^0 and φ_j^1 are $i-1$-qubit states (so that the measured qubit is the one pointed to by x_i). In the rule for *new*, Q is an n-qubit state, so that $Q \otimes |i\rangle$ is an $(n+1)$-qubit state.

5.4 Safety properties

Definition 5.9. We say $[Q, |x_1, ..., x_n\rangle, M]$ is *well-typed* of type C, written as $[Q, |x_1, ..., x_n\rangle, M] : C$, if $\emptyset; \{x_1, ..., x_n\} \vdash M : C$ is a valid typing judgement.

Quantum lambda calculus enjoys the following safety properties.

Theorem 5.10 (Subject reduction)**.** *Suppose $[Q, L, M] : A$, and $[Q, L, M] \to_p [Q', L', M']$, then $[Q', L', M'] : A$.*

Theorem 5.11 (Progress)**.** *Suppose $[Q, L, M] : A$, then either $[Q, L, M]$ is a value state (i.e., M is a value), or there exists a closure $[Q', L', M']$ such that then $[Q, L, M] \to_p [Q', L', M']$. Moreover the total probability of all possible single-step reduction from $[Q, L, M]$ is 1.*

6 Proto-Quipper

Quipper is a programming language for quantum computation (see [19, 7, 8], and [22]). Quipper provides a syntax to express and manipulate quantum circuits as data. However Quipper is implemented as an embedded programming language and is not type-safe. Proto-Quipper (PQ) is a typed lambda calculus intended as a mathematical formalization of a fragment of Quipper. In some sense, PQ is an extension of QLC with a circuit-as-data feature (but without state preparation and measurement). This section follows the Proto-Quipper section in Ross's Ph.D. thesis [13].

6.1 Types, terms, and values

Definition 6.1. The types of PQ are defined by

$$Type\quad A, B \quad ::= \quad \top \quad | \quad bit \quad | \quad qubit \quad | \quad A \multimap B \quad | \quad A \otimes B \quad | \quad !A \quad | \quad Circ(T, U).$$

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

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

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

$$\frac{[Q,L,M_1] \to_p [Q',L',M_1']}{[Q,L,\langle M_1,V\rangle] \to_p [Q',L',\langle M_1',V\rangle]}$$

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

$$\frac{[Q,L,M] \to_p [Q',L',M']}{[Q,L,let\ \langle x,y\rangle = M\ in\ N] \to_p [Q',L',let\ \langle x,y\rangle = M'\ in\ N]}$$

$$\frac{[Q,L,M] \to_p [Q',L',M']}{[Q,L,let * = M\ in\ N] \to_p [Q',L',let * = M'\ in\ N]}$$

Table 4: Reduction rules: congruence rules

where T, U are *quantum data types* defined by

$$T, U \quad ::= \quad \top \quad | \quad qubit \quad | \quad T \otimes U.$$

All the types except $Circ(T, U)$ are inherited from QLC. In QLC, an element of type *qubit* is seen as the name of a qubit in the QRAM. In PQ, it is seen as the name of a wire in a quantum circuit. Elements of quantum data types are tuples of wire names, and work as the interfaces of circuits. The type $Circ(T, U)$ is the set of all circuits with input interface of type T and output interface of type U.

Definition 6.2. The terms of PQ are defined by

$$\begin{array}{lllllllllll} Terms: & M,N,P & ::= & x & | & \lambda x.M & | & MN \\ & & | & * & | & \langle M,N\rangle & | & let\,\langle x,y\rangle = M\ in\ N & | & let\, * \ = M\ in\ N \\ & & | & 0 & | & 1 & | & if\ P\ then\ M\ else\ N \\ & & | & (t,C,M) & | & rev & | & unbox & | & box^T & | & q \end{array}$$

where t is a *quantum data term* defined by

$$t, u \quad ::= \quad * \quad | \quad q \quad | \quad \langle t, u\rangle\,.$$

Here, C ranges over a set $\mathbb{C}$ of *circuit constants*, whose structure is described in more detail below.

The first three lines are inherited from QLC. In the fourth line, we don't have $new, meas$ and U's, since PQ extends a minimal version of QLC which doesn't have measurement and state preparation, and also since unitaries U's are replaced by circuit constants C.

A circuit constant C represents a quantum circuit, whose inputs and outputs are labelled by elements of $\mathbb{Q}$. We assume there exists a constant for every possible quantum circuit. Let $FSD(\mathbb{Q})$ is the set of finite sequences of distinct elements of $\mathbb{Q}$. Each C is equipped with two sequences $In(C), Out(C) \in FSD(\mathbb{Q})$, representing the labels on the input and output wires of C, respectively.

The intended meaning of the terms in the last line are: In (t, C, M), t is a tuple of quantum names that works as an interface to the circuit constant C, and M is any term. Our intention is that C is a circuit constant currently being constructed, and the purpose of M is to add more gate to C.

rev is a function on the set of terms of the form (t, C, t'), which reverses the reversible circuit C, and interchanges the interfaces t, t'. *unbox* will be assigned types $Circ(T, U) \to (T \to U)$, and transforms a quantum circuit to a corresponding term of function type for future computation. box^T will be assigned types $(T \to U) \to Circ(T, U)$, and transforms a term of function type to quantum circuit for future manipulation. It is indexed by a quantum type T, which will be useful when defining the operational semantics.

Definition 6.3. A *value* of PQ is a term defined by

$$\begin{array}{lll} \text{Value } V, W & ::= & x \mid \lambda x.M \\ & \mid & * \mid \langle V, W\rangle \\ & \mid & 0 \mid 1 \\ & \mid & (t, C, u) \mid rev \mid unbox \mid box^T \mid unbox\ V \mid q \end{array}$$

unbox V is a function waiting for an argument, hence is considered as a value.

6.2 Typing system

The type system of PQ extends the one of QLC by adding types for the new constants and one more typing rule for the new term introduced.

Remark 6.4. If T, U are quantum data types, then $T <: U$ implies $T = U$. This is easily shown by induction. Therefore we do not need a subtyping rule of the form

$$\frac{A <: A' \quad B <: B'}{Circ(A', B) <: Circ(A, B')}Circ$$

Definition 6.5. *rev*, *unbox*, and box^T are assigned a series of types indexed by quantum data types (T, U):

$$\begin{aligned} A_{rev} &= !(Circ(T, U) \multimap Circ(U, T)) \\ A_{unbox} &= !(Circ(T, U) \multimap !(T \multimap U)) \\ A_{box^T} &= !(\ !(T \multimap U) \multimap Circ(T, U)) \end{aligned}$$

Definition 6.6. Typing judgements for PQ are the same as for QLC. The typing rules of PQ are inherited from QLC with a notice that the constant rule ax_c now applies to the new constants — $box^T, unbox, rev, 0, 1$ (and $new, meas, U$ have been removed). We also need a new rule for terms (t, C, M):

$$\frac{!\Delta; Q_1 \vdash t : T \quad !\Delta; Q_2 \vdash M : U \quad In(C) = Q_1 \quad Out(C) = Q_2}{!\Delta, \emptyset \vdash (t, C, M) : Circ(T, U)}circ$$

Note that we see Q_1 in a typing judgement as a set by dropping the sequence structure.

6.3 Operational semantics

The reduction rules of PQ are defined on closures $[C, M]$ of a *labelled circuit* (definitions later) and a term. To define the reduction rules, we must assume some additional structure on circuit constants, which we now describe.

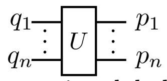
6.3.1 Circuit constructor

Recall that in Section 1.4, we introduced quantum circuits. Now we add changeable labels to these circuits.

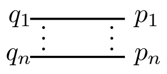
Definition 6.7. Consider the category $\mathbb{LC}$ with objects finite sequences of distinct elements from $\mathbb{Q}$, and morphisms between $\langle q_1, q_2, ..., q_n\rangle$ and $\langle p_1, p_2, ..., p_n\rangle$ are quantum circuits defined in 1.4 with n inputs and n outputs. We say that the i-th input wire has label q_i, and the j-th output wire has label p_j. We call the morphisms *labelled quantum circuits.*

- For any object $\langle q_1, q_2, ..., q_m\rangle$, there is a unique identity circuit.
- Composition $D \circ C$ is defined by connecting the output wires of C to the input wires of D with the same labels (only defined when the source of D and target of C are the same).
- Associativity is satisfied since morphisms are quantum circuits.

We use *dom* and *cod* as the maps sending a morphism to its domain and codomain.

Example 6.8. For example 

is a labelled quantum circuit. Any quantum circuit with input and output labelled by quantum names is a labelled quantum circuit.

Definition 6.9. A labelled identity circuit such as 

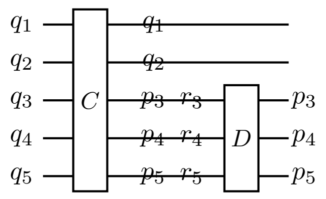
is called a *renaming* (or *binding* as in Ross's thesis [13]). Equivalently, a binding is the unique order-preserving (the sequence order) bijection between two finite sequences of $\mathbb{Q}$.

Definition 6.10. We define a 'partial tensor' on the category $\mathbb{LC}$. If two objects s_1, s_2 are disjoint (seen as sets), we define $s_1 \otimes s_2$ as the sequence obtained by appending s_2 to s_1. The tensor product of two morphisms is just the tensor product of two labelled quantum circuits.

Definition 6.11. Given two morphisms C, D of $\mathbb{LC}$, if $dom(D) \subset cod(C)$ (seen as sets) and $cod(D) \cap (cod(C) \setminus dom(D)) = \emptyset$ as set, then the 'partial composition' $D \circ' C$ is defined as follows:

- First extend D to D' such that $cod(C) = dom(D')$ as sets. Specifically, let $r = cod(C) \setminus dom(D)$ as a set. Make a sequence $\overrightarrow{r}$ from the set r. Let $D' = D \otimes I_{\overrightarrow{r}}$, where $I_{\overrightarrow{r}}$ is the identity on $\overrightarrow{r}$.
- Then rename $cod(C)$ such that $cod(C) = dom(D')$ as sequences. The renaming $b : cod(C) \to dom(D')$, i.e. the unique order-preserving bijection, will do.
- Finally compose $D' \circ b \circ C$ using the composition of $\mathbb{LC}$.

Example 6.12. The purpose of partial composition $D \circ' C$ is to append circuit D to C, when $dom(D)$ and $cod(C)$ are not the same. In this example, $r = \{q_1, q_2\}$, and b is the bijection renaming $\langle q_1, q_2, p_3, p_4, p_5\rangle$ to $\langle q_1, q_2, r_3, r_4, r_5\rangle$



Definition 6.13. A *circuit constructor* is the category $\mathbb{LC}$ equipped with the partial composition $\circ'$.

With a circuit constructor, we let $\mathbb{C} = Obj(\mathbb{LC})$, $In = dom$, and $Out = cod$.

6.3.2 Reduction rules

First we introduce some technical definitions used to transform tuples in lambda calculus to finite sequences in the circuit constructor, and the other way around.

Definition 6.14. If u is a quantum data term, define $Seq(u)$ to be the unique sequence that forgets the pair structure and keeps the quantum names and their occurrence order. Equivalently, we can use the following recursive definition:

$$\begin{aligned} Seq(\langle a, b\rangle) &= Seq(a), Seq(b) \text{ (the concatenation of two sequences)} \\ Seq(q) &= q \text{ (q is a quantum name)} \\ Seq(*) &= \emptyset \text{ (empty sequence)} \end{aligned}$$

Definition 6.15. If T is a quantum data type, define the *length* of T to be the number of occurrence of type *qubit* written as $len(T)$. Equivalently, $len(T)$ is defined by the following recursive definition:

$$\begin{aligned} len(T \otimes U) &= len(T) + len(U) \\ len(qubit) &= 1 \\ len(*) &= 0 \end{aligned}$$

Definition 6.16. If T is a quantum data type, and s is a sequence of length $len(T)$ with distinct element from $\mathbb{Q}$, define $Term(s, T)$ to be the unique quantum data term u of type T that keeps the quantum names in s and the sequence order (i.e. such that $Seq(u) = s$). Equivalently, we can use the following recursive definition:

$$\begin{aligned} Term(s, T \otimes U) &= \langle Term(s_1, T), Term(s_2, U)\rangle \\ Term(q, qubit) &= q \text{ (q is a quantum name)} \\ Term(\emptyset, \top) &= * \text{ (the empty pair)} \end{aligned}$$

where s_1 and s_2 are subsequences of length $len(T)$ and $len(U)$ such that $s_1, s_2 = s$.

Definition 6.17. If two objects r and s in $\mathbb{LC}$ are of the same length, define $bind(r, s)$ to be the unique renaming from r to s. If two quantum data terms u and t are of the same quantum data type, define $bind(u, t)$ to be the unique renaming from $Seq(u)$ to $Seq(t)$.

Definition 6.18. Any occurrence of a quantum name in a term (t, C, M) is called a *bound occurrence* (to circuit C); any other occurrence is called a *free occurrence.* If M is a term, define $FQ(M)$ to be the set of quantum names that occur freely in M. If b is a renaming such that $dom(b) = \{q_1, ..., q_n\} \subset FQ(u)$ as set, the $b(u)$ is a new term with quantum names renamed

$$b(u) = u[b(q_1)/q_1, ..., b(q_n)/q_n]$$

Definition 6.19. A *circuit closure* is a pair $[C, M]$ where

- C is a labelled circuit from the circuit constructor $\mathbb{LC}$.
- M is a lambda term.

Definition 6.20. The *one-step reduction relation*, written as $\to$, is defined on circuit closures by the rules given in Table 5, 6, and 7.

The following example illustrates how to do reductions in PQ.

Example 6.21. We assume that the circuit constructor contains the following gate $H, S, T, Cnot$. Let $H' = unbox(p, H, p)$, $S' = unbox(q, S, q)$, and $Cnot' = unbox(\langle r, t\rangle, Cnot, \langle r, t\rangle)$. Let $F = \lambda z.(let\ \langle x, y\rangle = z\ in\ Cnot'\ \langle H'x, S'y\rangle)$.

Now consider the closure

$$[-, box^{qubit\otimes qubit} F]. \tag{1}$$

where '$-$' is any circuit. The *box* rule applies. Pick a sequence s of length $len(T) = 2$. Say $s = \langle q_1, q_2\rangle$, and then $t = \langle q_1, q_2\rangle$. Then (1) reduces to

$$[-, (\langle q_1, q_2\rangle, \begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, F\,\langle q_1, q_2\rangle)]. \tag{2}$$

Since $F\,\langle q_1, q_2\rangle$ is not a value, rule *circ* applies. We need to reduce the closure

$$[\begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, F\,\langle q_1, q_2\rangle)]. \tag{3}$$

to a value, i.e., the when the term in it is a value. According to the reduction rules, in an application, reduce the argument first. In a pair, reduce the second component first.

$$\begin{aligned} 3 &\to [\begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, F\,\langle q_1, q_2\rangle)] \\ &\to [\begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, let\ \langle x, y\rangle = \langle q_1, q_2\rangle\ in\ Cnot'\,\langle H'x, S'y\rangle] \\ &... \\ &\to [\begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, Cnot'\,\langle H'q_1, S'q_2\rangle] \\ &= [\begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, Cnot'\,\langle H'q_1, unbox(q, S, q)q_2\rangle]. \end{aligned}$$

Now, the *unbox* rule applies. Specifically, $C = \begin{array}{l} q_1 \text{———} q_1 \\ q_2 \text{———} q_2 \end{array}, u = q, D = q\text{—}\boxed{S}\text{—}q, u' = q, V = q_2$, and $b = bind(V, u)$ is the renaming from q_2 to q. $b' = bind(dom(D), cod(D))$ is the renaming from q to q, and $b'(u') = q$. So it reduces by *unbox* and congruence rule to

$$[C, (\lambda x.M)V] \to [C, M[V/x]]$$

$$[C, let\ \langle x, y\rangle = \langle V, W\rangle\ \ in\ M] \to [C, M[V/x, W/y]]$$

$$[C, let * = *\ in\ N] \to [C, N]$$

$$[C, if\ 0\ then\ M\ else\ N)] \to [C, N]$$

$$[C, if\ 1\ then\ M\ else\ N)] \to [C, M]$$

Table 5: Reduction rules: classical control

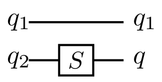
$$[C, rev\ (t, D, t')] \to [C, (t, D^{-1}, t')]$$

$$\frac{[D, M] \to [D', M']}{[C, (t, D, M)] \to [C, (t, D', M')]}\,circ$$

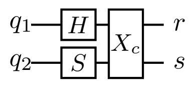
$$\frac{s \in Obj(\mathbb{LC}) \quad len(s) = len(T) \quad t = Term(s, T)}{[C, box^T(M)] \to [C, (t, Id_s, Mt)]}\,box$$

$$\frac{b = bind(V, u) \quad b' = bind(dom(D), cod(D))}{[C, (unbox(u, D, u'))V] \to [D \circ' (b \circ' C), b'(u')]}\,unbox$$

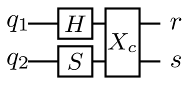
Table 6: Reduction rules: circuit operations

[ , $Cnot'\ \langle H' q_1, q\rangle$]

By repeatedly using *unbox* and congruence rule, we get

[ , $\langle r, s\rangle$].

Back to (2), it reduces by *circ* rule to

$[-, (\langle q_1, q_2\rangle$,  , $\langle r, s\rangle$].

6.4 Safety properties

Definition 6.22. We say $[C, M]$ is *well-typed* of type A, written as $[C, M] : A$, if $\emptyset; FQ(M) \vdash M : A$ is a valid typing judgement.

PQ enjoys the following safety property.

Theorem 6.23 (Subject reduction)**.** *Suppose $[C, M] : A$, and $[C, M] \to [C', M']$, then $[C', M'] : A$.*

Theorem 6.24 (Progress)**.** *Suppose $[C, M] : A$, then either M is a value, or there exists a closure $[C', M']$ such that then $[C, M] \to [C', M']$.*

$$\frac{[C,N] \to [C',N']}{[C,MN] \to [C',MN']}$$

$$\frac{[C,M] \to [C',M']}{[C,MV] \to [C',M'V]}$$

$$\frac{[C,M_2] \to [C',M_2']}{[C,\langle M_1,M_2\rangle] \to [C',\langle M_1,M_2'\rangle]}$$

$$\frac{[C,M_1] \to [C',M_1']}{[C,\langle M_1,V\rangle] \to [C',\langle M_1',V\rangle]}$$

$$\frac{[C,P] \to [C',P']}{[C, if\ P\ then\ M\ else\ N] \to [C', if\ P'\ then\ M\ else\ N]}$$

$$\frac{[C,M] \to [C',M']}{[C, let\ \langle x,y\rangle = M\ in\ N] \to [C', let\ \langle x,y\rangle = M'\ in\ N]}$$

$$\frac{[C,M] \to [C',M']}{[C, let * = M\ in\ N] \to [C', let * = M'\ in\ N]}$$

Table 7: Reduction rules: congruence rules

7 Problems to address

In my thesis research, I plan to address the following problems: extending Proto-Quipper with additional features that are present in quantum lambda calculus; devising an efficient type inference algorithm for Proto-Quipper; extending Proto-Quipper with imperative-style features; addressing problems of ancilla mangagement; and investigating the possible use of dependent types to address the distinction between parameters and state. I will briefly comment on each of these problems in the following sections.

7.1 Extension of PQ

Proto-Quipper extends a minimal version of quantum lambda calculus. One of the first problems I want to address is how to incorporate additional features of the quantum lambda calculus in Proto-Quipper, such as coproducts, recursion, and measurement.

To add measurements to Proto-Quipper, we will take the same approach as in Quipper: we will extend the notion of quantum circuit to also include classical wires, which hold a classical bit at circuit execution time. Measurement can then be treated as simply another kind of gate, namely, a gate that inputs a qubit and outputs a classical bit. It will be given the type $meas\ :\ qubit \to bit$.

Note that there is a difference between the type "bool", which holds a boolean whose value is known at circuit generation time, and the type "bit", which holds a boolean whose value is only known at circuit execution time. Values that are known at circuit generation time are also called "parameters" (we think of a program as describing a family of circuits, indexed by parameters). Values that are only known at circuit execution time are called "state". The interplay between parameters and state is subtle, and we will revisit it in Section 7.4 below.

Introducing classical bits and measurements into our circuits introduces a further complication: it is no longer the case that all circuits are reversible. This means that the "rev" operation of Proto-Quipper is no longer totally defined. In order to retain type-safety, it will therefore be necessary to extend the type system to be aware of the fact that certain circuits are reversible, while others are not. How best to do this is another open problem that I hope to address.

7.2 Type inference

Valiron described [20] a type inference algorithm for QLC. The input is an untyped QLC program. The output is a typing derivation, if it exists, or otherwise failure. Type inference is useful because it is tedious for programmers to write type annotations. But it is not known whether it can be done *efficiently*, i.e., in polynomial time. An interesting open problem is to find efficient type inference algorithm for QLC and/or PQ.

7.3 Imperative style

A quantum program typically contains section of code that just describe circuits in a sequential way, e.g.,

$$\begin{aligned}&let\ (x, y) = Cnot(x, y)\ in\\ &let\ y = Hy\ in\\ &let\ (y, x) = subroutine(y, x)\ in\\ &...\end{aligned}$$

In Quipper, it is possible to use a simpler 'imperative style' syntax like this

$$\begin{aligned}&Cnot(x, y);\\ &Hy;\\ &subroutine(y, x);\\ &...\end{aligned}$$

There is no theoretical foundation for such a syntax yet. It is still a research problem to design one. The problem is not as simple as it seems. It gets complicated when

- functions produce 'garbage' (ancilla qubits to hold intermediate results of the computation).
- function have some imperative and some non-imperative arguments. For example,

$$let\ x' = Cnot(x, y).$$

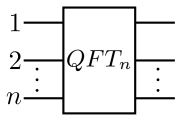
- functions are in the body of loops. For example,

$$let\ \langle x, \langle y, z\rangle\rangle = loop\ 100\ (\lambda\,\langle x, \langle y, z\rangle\rangle\ .let\ x = Hx\ in\ \langle x, \langle y, z\rangle\rangle)\ \langle x, \langle y, z\rangle\rangle\ .$$

It is still a open problem how to do this in a type-safe way.

7.4 Parameter-state distinction

In PQ, we have $box^T(M)$. Note that the annotation T is a quantum data type, which has a fixed length n. So when we run this program, it will start with an identity circuit on n qubits. In general, we want our programming language to describe *families* of circuits, not just single circuits. For example, consider the Fourier transform



There is a QFT for every size n. It would be natural to define a circuit family as a lambda term

$$QFT:\ (n:\ Nat) \multimap \otimes^n qubit \multimap \otimes^n qubit.$$

But this requires *dependent types*, i.e., the type $\otimes^n qubit$ depends on an earlier input (here n). When talking about a family of circuits C_n, we say that n is a parameter, and that the circuit family is parameterized by n. Parameters (such as n) are very different from state (such as qubits) because parameters must be known when a circuit is *generated*, whereas state is only known when a circuit is *executed.* Designing a type system (and semantics) to distinguish parameters and state is an open problem.

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Sole Distributors for the U.S.A. And Canada, Elsevier Science Pub. Co., 1984.

[2] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[3] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[4] G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[5] G. Gentzen. Untersuchungen über das logische schließen. ii. *Mathematische Zeitschrift*, 39(1):405–431, 1935.

[6] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989.

[7] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in quipper. *CoRR*, abs/1304.5485, 2013.

[8] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. *CoRR*, abs/1304.3390, 2013.

[9] W. A. Howard. The formulae-as-types notion of construction. 1995.

[10] A. Joyal and R. Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991.

[11] E. Knill. Conventions for quantum pseudocode. Technical report, Citeseer, 1996.

[12] D. Prawitz. *Natural Deduction: A Proof-theoretical Study.* Dover books on mathematics. Dover Publications, 2006.

[13] N. J. Ross. Algebraic and logical methods in quantum computation. *arXiv preprint arXiv:1510.02198*, 2015.

[14] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications, TLCA 2005, Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2005.

[15] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

[16] P. Selinger and B. Valiron. A linear-non-linear model for a computational call-by-value lambda calculus. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2008, Budapest*, volume 4962 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2008.

[17] P. Selinger and B. Valiron. On a fully abstract model for a quantum linear functional language. *Electronic Notes in Theoretical Computer Science*, 210:123–137, 2008.

[18] P. Selinger, B. Valiron, et al. Quantum lambda calculus. *Semantic Techniques in Quantum Computation*, pages 135–172, 2009.

[19] J. M. Smith, N. J. Ross, P. Selinger, and B. Valiron. Quipper: Concrete resource estimation in quantum algorithms. *CoRR*, abs/1412.0625, 2014.

[20] B. Valiron. *A functional programming language for quantum computation with classical control.* PhD thesis, University of Ottawa (Canada), 2004.

[21] B. Valiron. *Semantics for a higher order functional programming language for quantum computation.* PhD thesis, University of Ottawa, 2008.

[22] B. Valiron, N. J. Ross, P. Selinger, D. S. Alexander, and J. M. Smith. Programming the quantum future. *Communications of the ACM*, 58(8):52–61, 2015.