Types and Quantum Programming Languages

Quantum algorithms can provide asymptotic improvement over classical algorithms in key problems like integer factorization. However, they require a very different set of abstractions and tools than classical computing. Quantum algorithms typically reason explicitly about quantum gates and wires. If used or composed incorrectly, these elements will not form circuits that can actually be implemented by any quantum computers. The simple act of forking a wire is invalid in the language of quantum circuits. This has led to a rich vein of programming languages research to use types to statically ensure properties of quantum programs, as well as allow abstraction, code reuse and code verification. This paper introduces three, typed quantum programming languages, the Quantum Lambda Calculus, Qwire, and Quipper. To give the reader the proper context to understand these languages, core concepts of quantum computing, like qubits, quantum circuits, and entanglement are explained. This paper will also discuss efforts to apply formal methods to each of these languages. A researcher with a strong background in programming languages theory and is familiar with undergraduate level linear algebra should come away from this paper understanding these three languages and with enough background to read quantum programming languages papers.

1 INTRODUCTION

Quantum computers are quickly becoming larger and more practical[TODO: cite]. Practical quantum computation promises to revolutionize many fields of computer science, including cryptography [TODO: cite] and computational chemistry [TODO: cite]. However, quantum programs are hard to write and even harder to write correctly. Quantum physics concepts, like quantum superposition, entanglement, and the quantum no cloning theorem, don't map cleanly onto our intuitions of computation, and quantum algorithms rely on leveraging these concepts to outperform classical algorithm. There is a large gap between the way quantum algorithms are specified in papers and the way they can actually be written directly on real quantum computers.

This is exactly the kind of situation that programming languages research is intended to address. The tools that come out of programming languages research are designed to help us manage complex concerns by providing us the proper abstractions and sometimes preventing us from shooting ourselves in the foot. And there is an impressive body of work in the space of programming languages for quantum computing. However, this research can be difficult to approach for a programming languages researcher who does not already know a lot about quantum computation. Broadly speaking, this paper is written to help such programming languages researchers. In particular, this paper will cover the Quantum Lambda Calculus and Qwire programming languages. Both of these languages are linearly typed, functional programming languages that are designed to write quantum algorithms.

This paper aims to teach readers about three things:

(1) the basics of quantum computation, in particular enough to understand a significant abstraction in real quantum algorithms,
(2) the Quantum Lambda Calculus, how its syntax and semantics work and how to write programs in it,
(3) and Qwire, how its syntax and semantics work and how to write programs in it.

Author's address:

2022. XXXX-XXXX/2022/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

2.1 Qubits and Quantum State Systems

The most basic unit of data in quantum computing is called the qubit. As the name suggests, a qubit is a quantum computing generalization of the classical bit. A bit is some object, typically a wire, that can be in one of two states labelled 0 and 1. A qubit is some object that can be in any of the states, 0, 1, or a linear combination of 0 and 1 with possibly complex valued weights. In other words, a bit is an element of some finite type of size two, while a qubit is an element of the type of complex valued vectors of size two. Qubits are actually even more constrained. The norm of the the vector must also be one. To show some concrete examples, we must first introduce a little bit of notation. In this paper, we will use the bra-ket notation, a way of denoting vectors that came out of the quantum mechanics literature and is standard in this field.

Single Qubit Systems. $|0\rangle$ is the vector $\binom{1}{0}$, the vector with a 1 in the 0th element, and 0's everywhere else. $|1\rangle$ is the vector $\binom{0}{1}$, the vector with a 1 in the 1st element, and 0's everywhere else. Note that $|0\rangle$ and $|1\rangle$ form a basis for the vector space $\mathbb{C}^2$. Given a Greek letter, for example ϕ, $|\phi\rangle$ refers to an arbitrary state of a qubit. So any element of that space can be written as $\alpha |0\rangle + \beta |1\rangle$ for some α and β where $\alpha^2 + \beta^2 = 1$. The complex numbers α and β are called the amplitudes of the states 0 and 1. We call any state with nonzero α and β a superposition of the classical states 0 and 1. One example is the vector $\begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$, which represents an equal superposition of the states 0 and 1.

Quantum Measurement. Unlike with classical bits, we cannot directly observe what precise state a qubit is in. There is no operation we can perform on a qubit that will tell us the amplitudes of that state. The only operation we can perform that gives us some information about the state is quantum measurement. If you measure a qubit in the state $|\phi\rangle = \alpha |0\rangle + \beta |1\rangle$, then you will observe either the classical state 0 with probability α^2 or the classical state 1 with probability β^2. Note that is a valid probability space precisely because the norm of $|\phi\rangle$ is required to be 1. This measurement operation affects the state as well. After measurement, the state becomes the classical state that was just measured. So if you measure a state twice in immediate succession, then you will observe the same state each time. For example, if you measure the state $|0\rangle$, you will observe 0 and the state of the qubit will remain $|0\rangle$. But if you measure $|\phi\rangle = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$, then there is a 1/2 chance that you will observe 0 and the state will change to be $|0\rangle$, and there is a 1/2 chance that you will observe 1 and the state will change to be $|1\rangle$.

Note that qubits carry more information than is encoded in the measured distribution. The states $|0\rangle$, $-|0\rangle$ and $i|0\rangle$ will all be measured to be in state 0 with probability 1, but they are distinct states. And this distinction is essential in quantum algorithms.

We can extend this notion of quantum states from a single qubit to a system of n qubits. The state of an n qubit system is represented by a vector in $\mathbb{C}^{2^n}$ whose norm is 1. Just as in the single qubit system, if x is a sequence of n bits, and therefore a classical state in an n bit system, then $|x\rangle$ is the quantum state where the xth index is a 1 and all other indices are 0. Once again, the classical states form a basis for the quantum states.

It is important to be able to distinguish between the mathematical denotation of an n qubit system and the physical implementation of it. From a mathematical standpoint, an n qubit state is a unit vector in $\mathbb{C}^{2^n}$. From a physical standpoint, it is n specially prepared objects that each could be measured to be in one of two states. Here, we can begin to see where some of the power of quantum computing comes from. Viewed classically, such a system has only 2^n states. Viewed through quantum mechanics, it can encode exponentially more states. However, our inability to directly read quantum states the way that we can read classical states is still a major limitation.

Once again, we can only measure the state of an n qubit system and observe a single classical state of the system with a probability equal to the squared norm of the amplitude of that state. The state can also be partially measured, measuring the state of some qubits but leaving others unobserved. Modeling this is important for giving complete semantics to quantum programs, but no examples in this paper will require knowledge in that depth. [LS: Should I explain the math of such observations or leave that out?] [LS: I feel like this is too much straightline text, more of the math should be factored into easily readable and eyecatching lines]

[LS: bring up further]

Tensor Product. The tensor product is a matrix operation that is not universally covered in linear algebra courses, but is used extensively in quantum programming, so we briefly introduce it here.

Definition 2.1. Suppose A is an $m \times n$ matrix, and B is a $p \times q$ matrix, then $A \otimes B$ is the $mp \times nq$ matrix defined by the below block matrix.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}$$

Intuitively, we scale B separately by each index of A and then flatten this matrix of matrices into a single matrix of larger dimension. For example we take the tensor product of $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|0\rangle$ below.

$$\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \\ 0 \end{bmatrix}$$

2.2 Quantum Entanglement

In multiple qubit systems, the complicated phenomenon of quantum entanglement arises. For a concrete example, let us consider a 2 qubit system, whose state can be modeled by unit vectors in $\mathbb{C}^4$. This system has the classical states $|00\rangle, |01\rangle, |10\rangle$, *and* $|11\rangle$ along with the equal superposition state. It also has states like $|\phi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. This state is unique in two important ways. First, look at the distribution we get out if we measure $|\phi\rangle$. With 1/2 chance, both of the qubits are observed to be 0, and with 1/2 chance, both of the qubits are observed to be 1. These qubits are constrained to be observed to have the same value. $|\phi\rangle$ is also unique because it cannot be decomposed into the tensor product of two states from a 1 qubit system. (Maybe just focus on the first part) [LS: focus on what entanglement allows us to do and how it makes all operations "global" in a sense and unfold H ket 0 sometimes]

Entanglement allows us, in certain cases, to take a set of useful classical states and increase the probability of measuring them at the cost of probability of measuring useless classical states. [LS: I am not sure I have covered this topic enough, I am also not sure how much to foreground it]

2.3 Quantum Gates And Circuits

Now that we have a basic understanding of what a qubit is, we can talk more about the operations that we can perform on them. We have already discussed measurement, but measuring a state is only useful if we have carefully prepared it to have some useful properties already. How can we go from a normal classical state to some sort of usefully entangled one? And how do we know what information these measurements can actually tell us?

Asides from measurement, we directly manipulate quantum state systems by running them through quantum circuits. Quantum circuits are a straightforward analogue to standard boolean circuits. Quantum circuits are composed of quantum gates connected by a series of wires. Think of these wires as carrying qubits from the output of one gate to the input of another. Just like boolean circuit gates, these gates implement some function on their inputs and output the results.

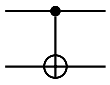
However, all quantum gates, and therefore quantum circuits as well, implement reversible, linear functions. In fact the full restriction is that an n qubit quantum circuit is denoted by unitary matrix from $\mathbb{C}^{2^n \times 2^n}$. square matrix is unitary if its inverse is its complex conjugate. Three important facts are implied by this constraint. First, unitary matrices preserve the norm of vectors they are applied to. This means that quantum circuits never take a qubit to some non unit norm vector. Second, unitary matrices have inverses, so any quantum circuit is reversible. And third, matrices are all linear functions. Since the classical states form a basis for any quantum state system, this means that the behavior of quantum circuits is fully determined by its behavior on classical states.

One important 1 qubit gate is the hadamard gate, H. It is denoted by the matrix $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. If you consider applying the hadamard gate to $|0\rangle$ and $|1\rangle$, you will see that $H|0\rangle = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$ and $H|1\rangle = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$. This shows that the hadamard gate takes classical states to states in equal superposition.

Another important gate is the CNOT, or controlled-not, gate. It is a 2 qubit gate. One input is the control qubit and the other is the target qubit. Focusing on its affect on classical states, CNOT checks the control qubit, if it is $|0\rangle$ it leaves the target qubit alone, and if it is $|1\rangle$ then it flips the target qubit. Knowing this, we can write the 4×4 matrix that it implements.

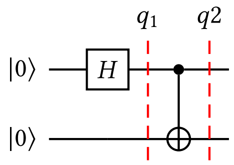
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

This is a simple permutation matrix that flips the 3rd and 4th index of a vector. In circuit diagrams, we represent applying CNOT to two wires with the following diagram fragment.



The wire with the filled in circle is the control qubit, and the wire with the open circle wire a target is the target qubit.[LS: that is worded a bit confusingly]

With the hadamard and CNOT gates, we can create an entangled state. But in order to do that, we need to understand how to combine quantum state systems. Given two quantum state systems, we can combine them using the tensor product.[LS: Should I introduce the tensor product or is that assumed] So combining an n qubit system, represented by unit vectors in $\mathbb{C}^{2^n}$, with an m qubit system, represented by unit vectors in $\mathbb{C}^{2^m}$, we get an $n + m$ qubit system, represent by unit vectors in $\mathbb{C}^{2^{n+m}}$. If the n qubit system is in state $|\phi_n\rangle$ and the m qubit system is in state $|\phi_m\rangle$, then the combined system is in state $|\phi_n \otimes \phi_m\rangle$. However, while we can also combine state systems with the tensor product, we cannot always decompose the combined system back into the components. These states that cannot be decomposed are the entangled states. [LS: this paragraph is awkward]. Consider the below circuit.



We begin with two qubits, both in the $|0\rangle$ state. Then we use a hadamard gate to change the state of the top qubit into into the equal superposition $\begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$. By taking the tensor product, we can see the total state at this point q_1 is $\begin{pmatrix} 1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \\ 0 \end{pmatrix}$. We can also write this state as $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$, an equal superposition between the 00 and 10 states. Then, q_1 is subjected to the CNOT gate, which swaps the 3rd and 4th indices creating the state $q_2 = \begin{pmatrix} 1/\sqrt{2} \\ 0 \\ 0 \\ 1/\sqrt{2} \end{pmatrix}$. This state is one of the entangled states that we previously discussed and can be written as $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$.

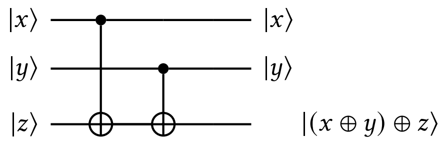
Circuits that create useful entangled states often start out with quantum states that put all classical states in equal superposition. We can accomplish this with a series of hadamard gates. Given an n qubit system, applying a single hadamard gate to each wire will have the effect of taking the state $|0^n\rangle$ to an equal superposition of all classical states. This circuit is called H_n.

2.4 Quantum Oracles

Suppose we have a boolean valued function f from 2^n to 2^m for some $n \geq m$. We know that we can create a classical circuit that computes this function. However, all quantum circuits are invertible, and f might not be an invertible function. However, we can (often? need to get clarification on this) embed f in a quantum circuit as what is called a quantum oracle. A quantum oracle for a function $f : 2^n \longrightarrow 2^m$ is an $n + m$ qubit circuit whose behavior on classical states is the below two equations.

The second equation is a straightforward corollary of the first, but it is important to point it out because it gets to the heart of the purpose of quantum oracles. You can think of the bottom wire as a kind of m qubit workspace. If you have some classical state that you want to apply f to, you can feed it into the top n qubits, and feed 0 into the bottom m qubits and then have that unchanged state along the top wire and the actual function output will be written onto the workspace.

For a concrete example, we can show the whole circuit for a quantum oracle for the exclusive-or function. $\oplus$ is a function from 2^2 to 2^1, and its quantum oracle is the below 3 qubit circuit.



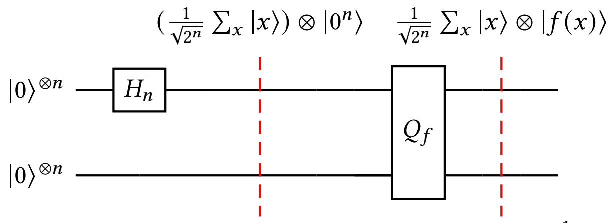
To understand this circuit, we can analyze how it transforms classical states. The first thing to notice about this circuit is that the upper two wires, x and y, remain unchanged. The second thing to notice is that the third wire, initially carrying z, gets flipped up to two times, one for each of x and y that is in a 1 state. As noted in the diagram, this is equal to $(x \oplus y) \oplus z$. This means that the circuit satisfies the equations of the quantum oracle for exclusive-or.

2.5 QRAM Model of Quantum Computing

We now have discussed quantum state systems and how to transform them using quantum circuits. However, these tools do not tell we can effectively use these tools. The primary point of quantum programs is to take problems that come up in classical settings, like integer factorization for example, and use quantum circuits to perform key parts of an algorithm to solve the problem.[LS: This is a subtle and complicated idea I am not currently getting across well]. This can even happen in a loop, with a classical computer repeatedly designing a circuit, sending it to a quantum computer computer to be run, and then recording some information and designing new circuits based on that. The QRAM model is an abstraction for quantum computing that maintains a distinction between a quantum computer, which can realize a specification for a quantum circuit and then execute it, and a classical computer, which can perform all of the normal functions of a computer, including writing specifications for quantum circuits.[LS: That programs can write other programs should be a familiar idea for PL people] Each language discussed in this paper realizes this model of computation in some way.

2.6 Running Example: Functional Entanglement

Consider a fixed positive number n, along with a fixed quantum oracle Q_f where $f : 2^n \longrightarrow 2^n$. We can construct the below $2n$ qubit quantum circuit whose inputs are fixed to be the classical state $|0\rangle$ scaled up to n qubits.


$(\frac{1}{\sqrt{2^n}} \sum_x |x\rangle) \otimes |0^n\rangle$
$\frac{1}{\sqrt{2^n}} \sum_x |x\rangle \otimes |f(x)\rangle$
$|0\rangle^{\otimes n}$
H_n
Q_f
$|0\rangle^{\otimes n}$

The circuit diagram marks the intermediate state $(\frac{1}{\sqrt{2^n}} \sum_x |x\rangle) \otimes |0^n\rangle$. The state at this point in the circuit is the tensor product of the state of the upper wire, which is an equal superposition of all classical states, and the state of the lower wire, which is just the classical state of all 0's. We derive this from the behavior from the top row of the extended hadamard gate H_n. The following reasoning shows how this intermediate state transforms into the final state in the diagram.

$$Q_f((\frac{1}{\sqrt{2^n}} \sum_x |x\rangle) \otimes |0^n\rangle) = \tag{1}$$

$$Q_f(\frac{1}{\sqrt{2^n}} \sum_x (|x\rangle \otimes |0^n\rangle)) = \tag{2}$$

$$\frac{1}{\sqrt{2^n}} \sum_x Q_f(|x\rangle) \otimes |0^n\rangle) = \tag{3}$$

$$\frac{1}{\sqrt{2^n}} \sum_x (|x\rangle \otimes |f(x)\rangle) \tag{4}$$

The first two lines are equal because the tensor product is linear in both of its arguments. The next two lines are equal because Q_f is also a linear transformation. And the final two lines are equal by the definition of a quantum oracle.

The final state, $\frac{1}{\sqrt{2^n}} \sum_x (|x\rangle \otimes |f(x)\rangle)$, has the property that that the $x \times y$th coordinate is $\frac{1}{\sqrt{2^n}}$ if $y = f(x)$ and 0 otherwise. Preparing this state is part of several quantum algorithms, including Shor's factorization algorithm. Intuitively, this state is useful because it turns this sequence of qubits into a search space of every input-output pair in f. Even though this system only requires $2n$ qubits to represent, it still encodes all 2^{2n} possible input-output pairs. From this state, we can solve problems by carefully designing quantum circuits to have "good" input-output pairs reinforce each other and "bad" input-output pairs interfere with each other, leading to a final state with a high probability of measuring a "good" input-output pair. This paper will use this circuit as a running example, to see how the different languages each encode this.

2.7 Quantum No Cloning Theorem

One tricky property about quantum computing is the quantum no cloning theorem. Intuitively, it says that there is no quantum circuit that can take a quantum state $|\phi\rangle$ and output it twice along two separate wires. This continues to hold if you introduce some other fixed input We can state it formally as well.

Theorem 2.2 (Quantum No Cloning). *Given any positive n, there is no unitary matrix* $U : \mathbb{C}^{2^{2n} \times 2^{2n}}$ *such that given any quantum state* $|\phi\rangle$, $U(|\phi\rangle \otimes |0^n\rangle) = |\phi\rangle \otimes |\phi\rangle$.

The proof of this theorem is surprisingly simple. Any U that could clone a state would not be linear. Suppose such a U existed, then we could derive a contradiction with the following valid equations.

$$\begin{aligned} U(|\phi\rangle \otimes |0^n\rangle) &= \\ 2U(\frac{1}{2}|\phi\rangle \otimes |0^n\rangle) &= \\ 2(\frac{1}{2}|\phi\rangle \otimes \frac{1}{2}|\phi\rangle) &= \\ \frac{1}{2}|\phi\rangle \otimes |\phi\rangle &\neq |\phi\rangle \otimes |\phi\rangle \end{aligned}$$

2.8 Static and Dynamic Lifting

2.9 Density Matrix Representation

One reason that quantum computing is difficult is that it mixes classical probability with quantum state systems. As soon as you measure a wire, you need to jump from the scary world of qubits to the downright terrifying world of distributions over qubits. Density matrices provide us with a compact representation of distributions over quantum states that cleanly interacts with linear transformations over quantum states. [TODO: remember this is from tseng and chuang while the rest is from the other textbook]

Definition 2.3 (Density Matrix). A square matrix $\rho : \mathbb{C}^{2^n \times 2^n}$ is a density matrix over $\mathbb{C}^{2^n}$ if there is some set of pairs (p_i, ϕ_i) where $\phi_i : \mathbb{C}^{2^n}$, $p_i \geq 0$ and $\sum_i p_i = 1$ such that $M = \sum_i p_i |\phi\rangle |\phi\rangle^*$. Given a quantum state space H, H^* is used to denote the space of density matrices over H.

Consider a single qubit system. Three example density matrices are

$$\left[\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right] = |0\rangle |0\rangle^\dagger \tag{5}$$

$$\left[\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right] = |1\rangle |1\rangle^\dagger \tag{6}$$

$$\left[\begin{smallmatrix} 3/4 & 0 \\ 0 & 1/4 \end{smallmatrix}\right] = \frac{3}{4}|0\rangle |0\rangle^\dagger + \frac{1}{4}|1\rangle |1\rangle^* \tag{7}$$

To measure the probability of measuring a state $|\phi\rangle$ given a density matrix ρ, you compute $|\phi\rangle^* M |\phi\rangle$. So, for example, the probability of $|0\rangle$ in $\left[\begin{smallmatrix} 3/4 & 0 \\ 0 & 1/4 \end{smallmatrix}\right]$ is $\frac{3}{4}$. If ρ is defined with a set of orthogonal vectors ϕ_i, then the probability of ϕ_i must be the corresponding p_i. Note that even if a state $|\phi\rangle$ is not one of the states that M is defined it, it can still have nonzero probability. For example, the given the matrix $M = \frac{3}{4}|0\rangle |0\rangle^* + \frac{1}{4}|1\rangle |1\rangle^*$, the probability of measuring $\left(\begin{smallmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{smallmatrix}\right)$ is $\frac{1}{2}$.

For the purposes of this paper, we are primarily interested in measuring with respect to classical states, but it is worth noting that measuring with respect to superpositions is a sensible thing to do.

Given a density matrix ρ, and an operator between quantum state spaces U, we can compute the transformed density matrix as $U^*(M) = UMU^\dagger$. This transforms U, which is a linear transformation over quantum state spaces, into a superoperator, a linear transformation over density matrices. So in order to denote circuits as transformations over density matrices, we can reuse all that we have already learned about quantum circuits as unitary matrices.

Moreover, superoperators over density matrices allow us to give matrix representations to the non-reversible operations essential to quantum computations, and even transformations between different quantum state spaces. This is very useful when reasoning about operations like measurement and preparation of new states. We saw above how to lift a pure quantum to the corresponding density matrix. We can also give a superoperator for the measurement of a 1 qubit system with respect to the classical states $|0\rangle$ and $|1\rangle$ as

$$(|0\rangle\,|0\rangle^\dagger)^* + (|1\rangle\,|1\rangle^\dagger)^*$$

When reading this definition be careful to note that the + here is elementwise addition over functions, not matrix addition. In particular, $f^*+g^* \neq (f+g)^*$. As such, the measurement superoperator defined above is not equal to the identity matrix, even though the sum of the two matrices is the identity matrix. If you do the arithmetic, you will find that applying the measurement superoperatory to the pure state $\begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}^\dagger$ yields the mixed state $\frac{1}{2}\,|0\rangle\,|0\rangle^\dagger + \frac{1}{2}\,|1\rangle\,|1\rangle^\dagger$. In other words measuring the equal superposition of $|0\rangle$ and $|1\rangle$ has even odds of yielding the state $|0\rangle$ and the state $|1\rangle$, as we have previously discussed.

We still need to define a method of taking the tensor product over superoperators. We will define this tensor product over a decomposition of superoperators.

Lemma 2.4. *Given any superoperator f, there exists a set X of matrices such that $f(\rho) = \sum_{M\in X} M^*\rho$.*

From here, we can define $(f\otimes g)(\rho) := \sum_{(M_x,M_y)\in X\times Y}(M_x\otimes M_y)^*\rho$ where X is the decomposition of f and Y is the decomposition of g.

3 QUANTUM LAMBDA CALCULUS

3.1 Introduction to Quantum Lambda Calculus

For the most part, the Quantum Lambda Calculus acts just like a linearly typed lambda calculus. We will discuss more the the specifics of that in Section [TODO: reference section], but for now, what is important to know is that linear types allow static analysis of resource consumption, inlcuding usage of qubits. The Quantum Lambda Calculus contains the basic syntax structures that you would expect in a simply typed lambda calculus.

[TODO: create the figure here]

In addition to these standard constructs, the Quantum Lambda Calculus has several specific constructs.

- The `new` keyword prepares a new qubit starting in either the state $|0\rangle$ or $|1\rangle$.
- The `meas` keyword measures a qubit, consumes it, and produces a boolean.
- The application of one of a preset collection of quantum gates `U` like the hadamard gate `H`.

For types, we have a standard collection of linear types. [TODO: create figure] Note that the $\multimap$ operator indicates a linear function type and the !, pronounced bang, operator indicates that the value is resuseable. The Quantum Lambda Calculus also provides a `qubit` type. Quantum circuits are given types of linear functions over qubits.

```
let entangle : qubit  ⊗  qubit :=
    let x := new 0 in
    let y := new 0 in
    let x := H x in
    CNOT (x, y)
```

The Quantum Lambda Calculus allows programmers to freely mix gate applications and classical control flow. We can demonstrate this in the below program which takes two single qubit function nad two qubits as arguments, and decides which function to actually apply based on the measurement of the first qubit.

```
let controlled : !(qubit ⊸ qubit) ⊸ !(qubit ⊸ qubit) ⊸ qubit ⊸ qubit ⊸ qubit
     :=
    λ g1 g2 x y ⇒
        let b : bool := meas x in
        if b then g1 y else g2 y
```

Note that each function has a ! type. Without it, the above function would be ill-typed. Because the Quantum Lambda Calculus is linearly typed, values can only be used exactly once by default. We need that to allow the function to not be applied in certain cases.

The function that we just wrote requires dynamic lifting, even though nothing in the syntax or types directly makes that clear. On the one hand, this is problematic because it gives no direct way to separate out the expensive dynamic lifts from the relatively inexpensive static lifts. On the other hand, this simple approach allows the Quantum Lambda Calculus to function as a strongly typed quantum programming language while only needing the syntax of a standard lambda calculus.

3.2 Quantum Lambda Calculus Type System

The Quantum Lambda Calculus type system relies on a subtyping relation in order to lessen the burden of using exponential types where linear ones are required. $\tau_1 <: \tau_2$ should be read as τ_1 is a subtype of τ_2, or "we can use a τ_1 anywhere that a τ_2 required".

First, we will see how subtyping works with the interaction of qubits and ! types.

$$\frac{n = 0 \rightarrow m = 0}{!^n qubit <:!^m qubit}\, qubit$$

The arithmetic condition in this rule just enforces that if the left type is not resusable, then the right type must also not be reusable. Each subtyping rule has that condition. For example, this inference rule lets us infer that $!qubit <: qubit$. This makes sense, as a resusable qubit should be able to be used in any context that requires a qubit. However, you **cannot** prove $qubit <:!qubit$. This makes sense, as a context which requires a reusable qubit might use it twice, something that is not valid to do for a nonreusable qubit. The rule for products and sums has a similar condition, and is otherwise covariant in its arguments. [TODO: maybe boolean or something would be a better choice for this example, !qubit should not be an inhabitable type unless you have some weird constants]

$$\frac{n = 0 \rightarrow m = 0 \qquad A_1 <: B_1 \qquad A_2 <: B_2}{!^n(A_1 \otimes A_2) <:!^m(B_1 \otimes B_2)}\otimes \qquad \frac{n = 0 \rightarrow m = 0 \qquad A_1 <: B_1 \qquad A_2 <: B_2}{!^n(A_1 \oplus A_2) <:!^m(B_1 \otimes B_2)}\oplus$$

Finally, there is a subtyping rule for functions.

$$\frac{n = 0 \rightarrow m = 0 \qquad B_1 <: A_1 \qquad A_2 <: B_2}{!^n(A_1 \multimap A_2) <:!^m(B_1 \multimap B_2)}\multimap$$

For readers familiar with subtyping function types, this is a straightforward, contravariant condition. For those unfamiliar, recall that the type on the left is the more specific type, and therefore enforces a stronger condition of the code. Functions that can relax restrictions on their input or strengthen conditions on their output satisfy a stronger condition than the supertype requires.

This subtyping relation is used in the general type system when we need to type variables or constants. The typing judgement for the Quantum Lambda Calculus has the following shape, typical of typing derivations, $\Omega \vdash e : A$. Ω is a typing context, e is a term, and A is a type. In these rules, we will denote a typing context which contains only reusable variables as $!\Omega$, and will denote the set of variable names used in a context as $|\Omega|$. The variable typing derivation makes use of this.

$$\frac{A <: B}{\Omega, x : A \vdash x : B}\ \mathrm{VAR}$$

Note the presence of the Ω in the conclusion of this rule. This shows that we can actually throw away variables without using them. In the rules dealing the the product type, we will see how the type system prevents us from reusing nonreusable variables.

$$\frac{!\Omega_1, \Omega_2 \vdash e_1 :!^n A_1 \qquad !\Omega_1, \Omega_3 \vdash e_2 :!^n A_2}{!\Omega_1, \Omega_2, \Omega_3 \vdash (e_1, e_2) :!^n (A_1 \otimes A_2)}\ \otimes I$$

$$\frac{!\Omega_1, \Omega_2 \vdash e_1 :!^n (A_1 \otimes A_2) \qquad !\Omega_1, \Omega_3, x_1 :!^n A_1, x_2 :!^n A_2 \vdash e_2 : B}{!\Omega_1, \Omega_2, \Omega_3 \vdash let(x, y) = e_1 ine_2 : B}\ \otimes E$$

Note how the context of reusable variables, $!\Omega_1$, itself gets to be reused, while the nonreusable contexts are each only used once in the hypotheses. We can view the $\otimes I$ rule as saying if you can construct $e_1 :!^n A_1$ by using $!\Omega_1$ and consuming the values in Ω_2, and you can construct $e_2 :!^n A_2$ by using $!\Omega_1$ and consuming the values in Ω_2, then you can construct the product and it will have type $!^n (A_1 \otimes A_2)$. This means that variables bound in Ω_2 cannot be used in the construction of e_2 and the variables bound in Ω_3 cannot be used in the construction of e_1.

With this rule, we can derive the following typing $x : \texttt{qubit}, y : \texttt{qubit} \vdash (x, y) : \texttt{qubit} \otimes \texttt{qubit}$. We cannot derive $x : \texttt{qubit} \vdash (x, x) : \texttt{qubit} \otimes \texttt{qubit}$, because you cannot separate the context into two disjoint pieces that each have x bound. The $\otimes E$ rule maintains a similar distinction, where Ω_2 can be used only in the construction of e_1, Ω_3 can be used only in the construction of e_2 and $!\Omega_1$ can be used in both.

The function application rule also works similarly.

$$\frac{!\Omega_1, \Omega_2 \vdash e_1 : A \multimap B \qquad !\Omega_1, \Omega_3 \vdash e_2 : A}{!\Omega_1, \Omega_2, \Omega_3 \vdash e_1\ e_2 : B}\ \otimes I$$

The construction of functions is slightly more complicated. The type system needs to make sure that reusable functions don't capture nonreusable variables. As such, the rule for nonreusable functions is very simple.

$$\frac{\Omega, x : A \vdash e : B}{\Omega \vdash \lambda x : A.e : A \multimap B}\ \lambda$$

For reusable functions, we add an extra condition that no nonreusable variables are captured by the function.

$$\frac{!\Omega_1, \Omega_2, x : A \vdash e : B \qquad FV(e) \cap |\Omega_2| = \emptyset}{!\Omega_1, \Omega_2 \vdash \lambda x : A.e :!^n (A \multimap B)}\ !\lambda$$

[TODO: introduce the list type constructor] [TODO: a bit more transition here]

3.3 Running example

To implement the functional entanglement in full generality, we need to use lists of qubits to represent the n qubit input and output wires. Given this, we can define the extended hadamard gate as a function from lists of qubits to lists of qubits, and the initialization of an n qubit wire in the 0 state as a function from positive integers to lists of qubits. To stay within the types formalized in the Quantum Lambda Calculus paper, we represent positive integers as lists of the unit type, where `nil` is 1 and `Cons (_,y)` is $1 + y$.

```
type qlist := list qubit

let rec extended_had (l : qlist) : qlist :=
    match unfold x with
    | inl _ ⇒ Cons (hadamard x, nil)
    | inr (x,l') ⇒ Cons (hadamard x, extended_had l)
    end

let rec extended_zero (list unit) : list qubit :=
    match unfold x with
    | inl _ ⇒ Cons (hadamard x, nil)
    | inr (_,l') ⇒ Cons (new 0, extended_zero l')
    end
```

Given the above functions, we create a functional entanglement state given a number n encoded as a list of units, and a quantum oracle represented as a function that takes and returns pairs of lists of qubits.

```
let functional_entanglement (n : list unit) (Qf : qlist⊗qlist ⊸ qlist⊗ qlist) :
     qlist ⊗ qlist :=
    let x = extended_zero n in
    let y = extended_zero n in
    let x = extended_had x in
    Qf (x,y)
```

While the above function does allow us to create a functional entanglement state, the type system has no way of ensuring that `Qf` actually is written to take in two n length lists of qubits. Alternatively, we could have chosen to represent multi-qubit wires as tuples of qubits, but this would have forced us to write a different functional entanglement function for each size. However, most practical languages lack support for dependent types and programmers are used to dealing with bugs that come from unmet size expectactions.

3.4 Quantum Lambda Calculus Type Operational Semantics

When not dealing directly with quantum computation primitives, the Quantum Lambda Calculus behaves just like any other lambda calculus. It has a step relation that behaves as you would expect when it comes to operations like function application. However, the semantics needs to be able to handle the allocation of qubits, gate applications and measurements. This requires the semantics to handle state and probability. The Quantum Lambda Calculus keeps track of this using *quantum closures.*

Definition 3.1. A quantum closure is a three element tuple $[Q, X, e]$ where, with some fixed natural number n,

- Q is an n qubit quantum state
- X is a collection of distinct variables $x_1, ..., x_n$, written as $|x_1, ..., x_n\rangle$
- e is a term in the Quantum Lambda Calculus

The variables in X represent the names of the qubits that make of the quantum state system Q.

The small step semantics is a relation over quantum closures. Intuitively, $[Q, X, e] \Rightarrow_p [Q', X', e']$ means that the state Q, variable collection X, and term e evolve to the state Q', variable collection X' and term e' with probability p.

$$[Q, X, G\left|x_{j_1}, ..., x_{j_k}\right\rangle] \Rightarrow_1 [Q', X, \left|x_{j_1}, ..., x_{j_k}\right\rangle]$$

where $x_{j_1}, ..., x_{j_n}$ are a sequence of qubit variables bound in X, and Q' is the resulting quantum state from applying G to the specified qubits. To evaluate the `new` keyword, we extend the quantum state Q with a new qubit using the tensor product. We also extend the list of bound qubit variables with a new variable name, and return that identifier for the program to pass around.

$$[Q, X, new0] \Rightarrow_1 [Q \otimes |0\rangle, |x_1, ..., x_n, x_{n+1}\rangle, x_{n+1}]$$
$$[Q, X, new1] \Rightarrow_1 [Q \otimes |1\rangle, |x_1, ..., x_n, x_{n+1}\rangle, x_{n+1}]$$

The `meas` keyword is the only probabilistic operation in the language. To evaluate `meas x_i`, we first split the quantum state Q into $aQ_0 + bQ_1$, where the Q_0 is the state Q evolves to if the measurement returns 0 and Q_1 is the state Q evolves to if the measurement returns 1.

$$[aQ_0 + bQ_1, X, measx_i] \Rightarrow_{a^2} [Q_0, X, 0]$$
$$[aQ_0 + bQ_1, X, measx_i] \Rightarrow_{b^2} [Q_1, X, 0]$$

3.5 Type System Guarantees

In order to discuss the theoretical guarantees of this language, we first must introduce a generalization of the multistep relation for probabilistic programs, as well as a way of typing quantum closures, and a notion of values. Intuitively, this reachability relation captures the set of possibly reachable quantum closures from a given start, and a quantum closure can be given the type of e.

Definition 3.2. The *reachability* relation $\rightsquigarrow$ is defined as the reflexive transitive closure of the set of pairs of quantum closures $([Q, X, e], [Q'.X', e'])$ such that there exists some number p where $[Q, X, e] \Rightarrow_p [Q'.X', e']$. This includes when the number p is 0, meaning that this is an overapproximation of the states that the quantum closure has some probability of evolving to.

Definition 3.3. The quantum closure $[Q, |x_1, ..., x_n\rangle, e]$ has type τ, typeset as $[Q, |x_1, ..., x_n\rangle, e] : \tau$, if $x_1 : qubit, ..., x_n : qubit \vdash e : \tau$.

Definition 3.4. A term v is a value if it is either an abstraction $\lambda x.e$, a constant, a variable, a pair (v_1, v_2) where both elements are values, or $inl(v')$ or $inr(v')$ where v' is a value.

Theorem 3.5 (Preservation). *If $[Q, X, e] : \tau$ and $[Q, X, e] \rightsquigarrow [Q', X', e']$, then $[Q', X', e'] : \tau$.*

Theorem 3.6 (Progress). *If $[Q, X, e] : \tau$, then either e is a value, or there exists some set of quantum closures $\{[Q_i, X_i, e_i]\}$ such that for each i, $[Q, X, e] \Rightarrow_{p_i} [Q_i, X_i, e_i]$ and $\sum_i p_i = 1$.*

4 QWIRE

The Qwire language is parameterized by a host language. In this sense, Qwire really describes a family of languages and not a single one. However, with only a few assumptions about the host language, namely that it has booleans, product types, and is strongly normalizing [LS: big assumption can't brush past this easily], we can give a formal semantics to Qwire.

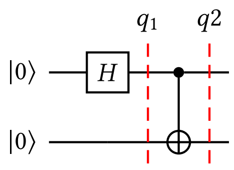
4.1 Introduction to Qwire

Qwire breaks up reasoning about quantum programs into three levels. At the top, we have the host language capable of everything that a strongly normalizing functional programming language can do. We require that this host level have boolean and pair types, and we extend the host language with a new type of circuits. Then we have the circuit layer, which is isolated into a special modality. It is only in that modality that we can directly manipulate the final level, wires. Much like in our original introduction of quantum programming, circuits are made up of quantum gates and wires. Unlike in the quantum lambda calculus, wires and qubits are not standard terms that can be manipulated anywhere in the language. We cannot create functions from qubits to qubits in Qwire.

However, a circuit can be parameterized on initial input wires, acting like a function over qubits. With these tools, we can easily define a a circuit that creates the entanglement of the states $|00\rangle$ and $|11\rangle$.

[TODO: make up some kind of syntax highlighting?] [TODO: put these next to each other]

```
x ← gate new0 ();
y ← gate new0 ();
x ← gate hadamard x;
o ← gate control not (x,y);
output o
```



[LS: maybe I need a version of this circuit with different annotations]

Inside this circuit, the gate keyword is used to both to create new qubits and apply gates. On the first two lines, we introduce new qubits initialized to the $|0\rangle$ state with the `new0` gate. Then we apply the `hadamard` gate to the `x` qubit. Note that we reuse the variable name `x` with variable shadowing. Because Qwire's linear type system prevents us from using a wire value twice, this is fine[LS: very weaselly worded]. Qwire is parameterized on a collection of base gates. Each gate needs to be given a gate type, which specifies the input and output wires of the gate. The notion of gate here is broader than the notion of gate in the background section. In that section, gates only referred to reversible computations that perform no measurement and preserve superposition information. In Qwire these are called unitary gates, while standard gates include measurment and the initial prepration of classical states. Unitary gates can be inverted or controlled, producing new unitary gates.[LS: previous sentence not great] `not` is a unitary gate, so we can construct a controlled version `not` gate that behaves the same as the CNOT gate in the background section.

4.2 Circuit Host Language Interactions

There are two ways to take a circuit like that and embed it into the top level host language. The first way is to specify its inputs with and delimit it with the box keyword. The boxed circuit then becomes a term with a circuit type in the host language. The boxed circuit can then be unboxed and used in a circuit. The second way is to take a circuit with no undefined inputs and running it with the runkeyword. Intuitively, running a circuit first yields some quantum state, which is then implicitly measured. For example, consider running the code from above.

```
run
    x ← gate new0 ();
    y ← gate new0 ();
```

[TODO: find better way to format this]

$$W ::= \texttt{qubit} \quad | \quad \texttt{bit} \quad | \quad W_1 \otimes W_2$$
$$\tau ::= \ldots \quad | \quad \texttt{bool} \quad | \quad \tau_1 \times \tau_2 \quad | \quad \texttt{Circ}(W_1, W_2)$$

```
x ← gate hadamard x;
o ← gate control not (x,y);
output o
```

This program would produce the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, and then measure both wires. This has an equal likelihood of producing the pair (`true`,`true`) and the pair (`false`,`false`). Note that this means that you cannot preserve any part of the quantum system when finished running a circuit.

Only having static lifting is too restrictive for some quantum algorithms[TODO: at least have a citation, ideally a bit more of an explanation]. Sometimes, algorithms require certain wires to be measured while the rest of the state is preserved, and then determine what transformations to make to the remaining state from state based on the measured values. Qwire enables this with the **`lift`** keyword.

Consider the following example program with input wires `x` and `y`.

```
b ← lift x;
y ← unbox (if b then c1 else c2) y;
output y
```

This circuit observes the state in `x` and uses that value to decide which circuit to apply to `y`. This operation is very different from previous ones that we have seen in Qwire. At first, this might appear similar to the **`control`** gate. However, **`control`** only works on unitary gates, where the above program can work with arbitrary circuits. The extra power comes from the fact that the **`lift`** keyword creates a host language boolean value, not a circuit level wire. This gives us access to all of the tools of a full programming language to turn this measured value into a circuit.

4.3 Qwire Type System

Qwire uses a mixture of linear and non-linear types in order to prevent violations of quantum circuit rules with minimal extra burden placed on programmers. Linear types are used to prevent reuse of wire values inside of circuits. Non-linear types are used everywhere else in the host language[LS: bad parallelism]. To maintain this distinction, Qwire has three different typing derivations. The lowest level derivation is the wire typing derivation, which has the below shape.

$$\Omega \Rightarrow p : W$$

Ω refers to a context of wire variables. Intuitively, p is some arrangement of the wire variables in Ω with type W. You can also read this derivation as saying that Ω is a collection of wires that have been permuted into value p with type W. All of the inference rules for this derivation are given in Figure 1. These typing rules preserve the invariant that every variable bound in Ω is used in p exactly once.

At the next level is the circuit typing derivation which has the below shape.

$$\Gamma; \Omega \vdash C : W$$

Ω, once again, refers to the linear context of wire variables. Γ refers to the nonlinear context of host language variables. So while Ω contains exclusively tensor products of bits and qubits, Γ can contain natural numbers, functions, and circuits. And while variables in Ω must all be used exactly once, variables in Γ can be used arbitrarily many times or even not at all.

$$\frac{}{\cdot \Rightarrow () : \texttt{Unit}} \qquad \frac{}{x : W \Rightarrow x : W} \qquad \frac{\Omega_1 \Rightarrow p_1 : W_1 \qquad \Omega_2 \Rightarrow p_2 : W_2}{\Omega_1, \Omega_2 \Rightarrow (p_1, p_2) : W_1 \otimes W_2}$$

Fig. 1. Inference rules for wire typing derivation

[TODO: it might be better to inline the typing rules here]

All of the circuit typing rules are presented in Figure 2 The OUTPUT rule provides a way to terminate a circuit with some collection of wires. Note that this rule provides no direct way have values bound in Γ influence p. [LS: how useful is this observation?]

The GATE rule ensures that inputs consumed by a gate cannot be used in the resulting circuit, unless of course the gate leaves those wires unchanged. It accomplishes this by having the typing assumption for the continuation C replace the context with the single pattern output by the gate. So none of the inputs to the gate are in scope when typing the continuation.

The UNBOX rule proves one way to have the host context Γ influence a circuit. The UNBOX rule requires a circuit c to be typed in the host typing derivation. Unlike wire values, this circuit could come as the input of some host level function or be the direct result of some complex host level computation. It also requires and consumes an input pattern, preventing wire variables from being forgotten or reused.

The LIFT rule provides another, more subtle, way for Γ to influence a circuit. As previously mentioned the `lift` keyword is used for dynamic lifting. [LS: not sure the reader remembers what that is] Note that unlike the GATE rule, the LIFT rule binds a variable by adding it to the host context Γ. It uses the $|\cdot|$ function to map wire types onto products of booleans. This allows us to use host language control flow operators on this value, selecting between one of many different circuits to apply to the remaining quantum wires in Ω_2.

The COMPOSE rule lets you take the output wires of some circuit and use it in defining some continuation circuit. This is particularly useful in connecting unboxed circuits with gates.

Finally, we have the host level typing derivation, with the below shape.

$$\Gamma \vdash e : \tau$$

Most of this typing derivation is going to be dependent on the host language. There are only two rules that are specific to Qwire. First we have the BOX rule, which allows us to take an open circuit, give a type for its input, and store it as a host term.

$$\frac{\Omega \Rightarrow p : W_1 \qquad \Gamma; \Omega \vdash C : W_2}{\Gamma \vdash \texttt{box}\ \ (\texttt{p} : W_1) \Rightarrow C : \texttt{Circ}(W_1, W_2)}\ \text{BOX}$$

Second we have the RUN rule, which allows us to take a circuit with no inputs, execute it and return the measured results.

$$\frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \texttt{run}\ \ C : |W|} RUN$$

This typing rule forces the argument of `run` to have no inputs by forcing it to be typed with an empty wire context.

4.4 Functional Entanglement Implementation

Dependent Types. In this presentation we assume that the host language has dependent types. In the background section we talked extensively about n by n quantum circuits for abstract n.

$$\frac{\Omega \Rightarrow p : W}{\Gamma; \Omega \vdash \mathtt{output}(p)}\ \textsc{Output} \qquad \frac{\Omega_1 \Rightarrow p : W_1 \qquad \mathtt{g} \in \mathtt{gate}(W_1, W_2) \qquad \Gamma; x : W_2, \Omega_2 \vdash C : W}{\Gamma; \Omega_1, \Omega_2 \vdash x \leftarrow \mathtt{gate\ \ g\ \ p}; C : W}\ \textsc{Gate}$$

$$\frac{\Omega \Rightarrow p : W_1 \qquad \Gamma \vdash \mathtt{c} : \mathtt{Circ}(W_1, W_2)}{\Gamma; \Omega \vdash \mathtt{unbox\ \ c\ \ p} : W_2}\ \textsc{Unbox} \qquad \frac{\Gamma; \Omega_1 \vdash C : W_1 \qquad \Omega \Rightarrow p : W \qquad \Gamma; \Omega, \Omega_2 \vdash C' : W_2}{\Gamma; \Omega_1, \Omega_1 \vdash p \leftarrow C; C' : W_2}\ \textsc{Compose}$$

$$\frac{\Omega_1 \Rightarrow p : W_1 \qquad \Gamma, x : |W_1|; \Omega_2 \vdash C : W_2}{\Gamma; \Omega_1, \Omega_2 \vdash x \Leftarrow \mathtt{lift}\ \ p; C : W_2}\ \textsc{Lift}$$

Fig. 2. Inference rules for circuit typing derivation

Dependent types allow us to parameterize the types of wires and circuits with sizes. This allows us to fully formalize the example circuit in full generality.

To begin, we can write the following host level function from positive integers to wire types that implements sized qubit vectors.

```
Fixpoint qubits (n : pos) : WireType :=
    match n with
    | 1 ⇒ qubit
    | S m ⇒ qubit ⊗ (qubits m)
    end.
```

Then we can formalize the extended hadarmard circuit, as well as an n qubit ancillary 0 state.

```
Fixpoint extended_had (n : pos) : Circ (qubits n, qubits n) :=
    match n with
    | 1 ⇒ box (x : qubit) ⇒ x  ← gate hadamard x; output x
    | S m ⇒ box ((x, y) : (qubit ⊗ qubits m)) ⇒
              x  ← gate hadamard x;
              y  ← unbox (extended_had m) y;
              output (x,y)
    end.

Fixpoint extended_zero (n : pos) : Circ (Unit , qubits n) :=
    match n with
    | 1 ⇒ box (_ : Unit ) ⇒ x  ← gate new0 (); output x
    | S m ⇒ box (_ : Unit ) ⇒
              x  ← gate new0 ();
              y  ← unbox extend_zero m;
              output (x,y)
    end.
```

With these programs written, we can write a function that takes in a size n, a $2n$ by $2n$ circuit that we assume is the quantum oracle of some function, and produce the functional entanglement circuit.

```
Definition functional_entanglement (n : pos)
    (Qf : Circ(qubits (2 * n), qubits (2 * n) )) : Circ (Unit , qubits (2 * n) ) :=
        box (_ : Unit ) ⇒
            x  ← unbox extended_zero n;
            y  ← unbox extended_zero n;
            x  ← unbox (extended_had n) x
            unbox Qf (x,y).
```

This program precisely matches the circuit description. It is important to note that while this section is written mostly in Coq syntax for the host language, actually using Coq's dependent types for this purpose would be more complicated than it appears here.

4.5 Qwire Operational Semantics

Qwire has an operational semantics based on the operational semantics for the host language. Given the relations $\to_H$, which is a small step semantics for host terms, and $\to_b$, which is a small step semantics for boxed circuits, we can define $\Rightarrow$, the small step semantics for circuits. We also assume that the host language has some way of simulating quantum circuits in order to simulate the run keyword. [LS: here I can point a pointer to a figure that just includes all of the rules]

Let us assume that $\to_H$ satisfies progress, preservation and strong normalization. Then we can prove the corresponding properties of Qwire. In these theorem statements, the variable $\mathcal{Q}$ ranges over wire contexts where every variable either has type qubit or bit. An open circuit C, where $\cdot; \mathcal{Q} \vdash C : W$, is normal if it consists only of output statements, gate applications, and dynamic lift statements. As the name normal suggests, such circuits cannot reduce any further.

Theorem 4.1 (Preservation).

(1) If $\cdot \vdash t : \tau$ and $t \to t'$, then $\cdot \vdash t' : \tau$.
(2) If $\cdot; \mathcal{Q} \vdash C : W$ and $C \Rightarrow C'$ then $\cdot; \mathcal{Q} \vdash C' : W$.

Theorem 4.2 (Progress).

(1) If $\cdot \vdash t : \tau$, then either there exists some term t' such that $t \to t'$, or t is a value.
(2) If $\cdot; \mathcal{Q} \vdash C : W$ then either C is normal or there exists some term C' such that $C \Rightarrow C'$.

Theorem 4.3 (Strong Normalization).

(1) If $\cdot \vdash t : \tau$ then there exists some value v such that $t \to^* v$.
(2) If $\cdot; \mathcal{Q} \vdash C : W$ then there exists some normal circuit N such that $C \Rightarrow^* N$.

4.6 Qwire Denotational Semantics

While progress, preservation, and strong normalization are all very useful properties for programming languages, they don't directly relate to some of the core issues of quantum programming. In particular, these theorems do not neccessarily imply that inhabitants of circuit types are implementable even by idealized quantum computers. We could have relaxed the constraints of the linear types, and ended up with a language that could have been useful for reasoning about classical circuits that has those properties as well.

Qwire also comes with a denotational semantics specifically for circuits. This denotational semantics assigns every well typed circuit to matrices that map density matrices over the input wire denotation to density matrices over the output wire denotation. This wire type denotation is given below.

$$[bit] = [qubit] = \mathbb{C}^2$$
$$[W_1 \otimes W_2] = [W_1] \otimes [W_2]$$
$$[\mathtt{Unit}] = \mathbb{C}$$

Take as a given that all of the above vector spaces are restricted to unit norm vectors. We can denote $[\cdot; \Omega \vdash C : W]$ as a superoperator from $[\Omega]^*$ to $[W]^*$. At a base level, this requires denoting every gate as a superoperator as well. For unitary matrices, this is done as normal.[TODO: fill in more when I understand more] Here are some of the denotations of other mentioned gates [LS: bad sentence].

$$[init0] = (|0\rangle\, |0\rangle^{\dagger})^{*}$$
$$[init1] = (|1\rangle\, |1\rangle^{\dagger})^{*}$$
$$[meas] = (|0\rangle\, |0\rangle^{\dagger})^{*} + (|1\rangle\, |1\rangle^{\dagger})^{*}$$

With all of these pieces in place, the denotational semantics satisfies the following theorem.

Theorem 4.4. *If* $\cdot; \mathcal{Q} \vdash C : W$ *and* $C \Rightarrow C'$ *then* $[\cdot; \mathcal{Q} \vdash C : W] = \cdot; \mathcal{Q} \vdash C' : W$.

This theorem assures us that, assuming all of the given gates are realizable by our quantum computer, then every circuit specified in a Qwire program is realizable on our idealized quantum computer.

5 DISCUSSION

5.1 Advantages of the Quantum Lambda Calculus

The most impressive aspect of the Quantum Lambda Calculus is how it demonstrates that existing programming languages constructs are enough to capture quantum programming abstractions in full generality. The abstract view of quantum computation requires us to write algorithms and circuits that abstract over different parameter algorithms or circuits. This is precisely the kind of abstraction that lambda calculi are well-suited to deal with. The primary complications that the Quantum Lambda Calculus adds are the use of affine types to control which values are allowed to be reused, and the extension of the operational semantics to handle quantum state and probabilistic effects. All of this was well understood previously, making it simpler to apply to the application of quantum computing. [TODO: this is awkward]

5.2 Limitations of the Quantum Lambda Calculus

While it is useful to see how quantum programming can be modelled using with few extensions, the language is not organized along the same lines as the abstractions of quantum computing. In the Quantum Lambda Calculus, there is no clean distinction between the classical and quantum parts of the any algorithm. Every function is inherently stateful and probabilistic and the effects are not tracked by the type system at all.

Furthermore, the Quantum Lambda Calculus makes no attempt to separate out static from dynamic lifting. Classical control flow and controlled quantum gates can be switched between with no constraints. This is problematic because static lifting is much cheaper to perform then dynamic lifting. The Quantum Lambda Calculus paper provides no static analysis to distinguish these different kinds of lifting, giving no way to accomplish the computations more efficiently. In fact it gives no compilation strategy to any lower level quantum computer, making it unclear precisely how to use this language to program a quantum computer rather than just to analyze or classically simulate a quantum computation.

5.3 Advantages of Qwire

Qwire makes a clean distinction between the classical and quantum parts of the language. Classical computations are performed by the host language and quantum computations are performed by the circuit language. The host language can only pass along boxed circuits as values and not qubits. Static and dynamic lifting correspond directly to the `run` and `lift` keywords directly, making it clear when an actual quantum computer would need to incur the costs and dynamic lifting and when it would not need to.

In [TODO: cite], Rand and Paykin contributed a verified compiler from Qwire's circuit language to QASM [TODO: cite] a popular low level quantum language that can be run directly on some modern quantum computers [TODO: check that and edit accordingly]. This demonstrates that Qwire is not only a useful abstraction for thinking about quantum programming, but can be used as a practical, high level language for quantum computing.

The denotational semantics given to Qwire circuits are closely related to the underlying mathematics of quantum computation, making reasoning about the correctness of Qwire circuits similar to reasoning about the correctness of abstract circuits [TODO: confusing terminology].

The inclusion of dependent types in Qwire's host language allows programmers to fully specify sized quantum circuits, statically ruling out a pernicious class of potential programming errors.

5.4 Limitations of Qwire

While dependent types can be useful, in practice they are difficult to work with. Qwire leaves all details of the dependent types to the host language, giving no extra help to the programmer. Also, although superoperators over density matrices are a valid mathematical model for quantum circuits, most of the quantum computing resources tend to only consider measurement at the end of a circuit, allowing the circuit to be viewed as a unitary matrix, a simpler mathematical object. You could imagine separating out Qwire's circuit description language into two parts, one for circuits purely consisting of unitary gates, and one for circuits that include measurement. The purely unitary circuits could then be denoted as unitary matrices, making them easier to reason about.

5.5 Related Work

There are many more quantum programming languages being used. Much like QWIRE, Quipper[TODO: cite] and LIQUi|⟩[TODO: cite] both embed circuit descriptions in a functional language, representing circuits as datastructures in the host language. Quipper is embedded in Haskell, and LIQUi|⟩is embedded in F#. The Qiskit [TODO: cite], Circ, and Q# languages are all embedded in Python and represent quantump circuits as sequences of imperative operations on qubits. They are the primary quantum programming languages of IBM, Google, and Microsoft respectively None of the listed languages use linear types, instead opting either for running static analyses at compile time or throwing exceptions at runtime if a circuit attempts to clone a qubit. The QASM language [TODO: cites] is a low level circuit description language that seeks to be an intermediate language representation for higher level languages to target.

There are also many applications of programming languages techniques to quantum programming beyond language design. In Qwire particularly, there is a body of work on program and program transformation verification [TODO: cite]. Researchers also use rich type systems to capture complex, quantum specific program properties. [TODO: cite Twist] designed a type system for keeping track of purity and entanglement, and [TODO: cite gottesman types] developed Gottesman types as a way to characterize quantum circuits as transformations between eigenvector bases which also allows you to track entanglement.

6 CONCLUSION

This paper provides an introduction to quantum programming and an overview of two strongly typed quantum programming langugaes.

The Quantum Lambda Calculus is a linearly typed functional programming language for quantum programming that is structured like a normal lambda calculus.