Type Systems for Quantum Computation

Bill Zorn

May 1, 2014

Abstract

Type systems are useful to programmers for a variety of reasons. In the classical world, they help us to define interfaces between separate pieces of code, and catch bugs when inconsistencies arise. But type systems are more than just another tool for debugging. They let us reason formally about what is going on in the code at a higher level of abstraction. Types are, ultimately, just a description of computation.

Looking towards the quantum world, we would like to have a higher-level description of computation than the quantum gate or operator models that are used in quantum information theory. To be productive in the hands of programmers who are not intimately familiar with the mathematics behind quantum information, a quantum programming language must provide a very powerful abstraction. The type system of such a language will necessarily be very interesting, and by understanding the abstraction, we hope to learn something about what it is that makes quantum computation powerful.

1 Introduction

In this review paper, we will discuss some recent developments in the field of type systems for quantum programming languages, with an emphasis on Peter Selinger's quantum lambda calculus[6] and QML[1]. First we will provide a general overview of how quantum programming languages work to provide some background for the discussion of type systems. Then we will examine the type systems behind the quantum lambda calculus and QML, highlighting the differences between their methods for handling the quantumness of quantum data.

Along the way we will also touch upon QCL, or "Quantum Computation Language," another quantum programming language but one without an interesting type system. We can think of QCL as a quantum "C" code, and the quantum lambda calculus and QML as higher-level languages, much like Standard ML in the classical world. We will consider a few simple examples, specifically quantum teleportation, and see what a type-theoretic analysis can tell us about what is happening.

2 Quantum Information

First, let's take a small detour and discuss the special properties of quantum information that make it different from classical information. Quantum information is very similar to classical information, but instead of bits we have slightly more generalized "qubits." A qubit can be in one of two orthogonal basis states, which we call $|0\rangle$ and $|1\rangle$, employing Dirac notation from quantum physics. Compare this to a classical bit being either 0 or 1. However, a qubit can also be in a superposition of the two basis states, say $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. We call the coefficients the "probability amplitudes" of the states in the wave-function, and require that they are normalized so that the sum of the absolute values of their squares is exactly one. If a state in a superposition is measured, the measurement will yield each of the basis states with probability equal to its amplitude (absolute value) squared, and the state will collapse into that basis state, with amplitude 1.

So lets say we have a qubit in the state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and we measure it. Both amplitudes are $\frac{1}{\sqrt{2}}$, which when squared gives us $\frac{1}{2}$, so we can see that this is a good, normalized state, and our measurement will come out to be $|0\rangle$ or $|1\rangle$ with equal probability. Note that the outcome of the measurement is actually classical information, so we should say that our result is 0 or 1 without referring to a Dirac ket. Though they must always be normalized, the amplitudes for a given state can be any complex numbers, so we can construct a state with any probability distribution among the basis states, or indeed an infinite number of these states by changing the complex phase of the amplitudes.

2.1 Entanglement

We should put in a word about entanglement, which is often considered to be the unique feature of quantum computation that makes it different and powerful. Entanglement is different from a superposition: a single qubit can be in a superposition, but we need multiple qubits to be entangled. For a state with multiple qubits, we assign a different amplitude to each possibly assignment of all the qubits to the basis states: with two qubits, for example, we have amplitudes for $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, in general 2^b possibilities if we have b qubits.

Many of these states are boring, such as the situation of two qubits in the $|0\rangle$ basis state, in which case we'd simply have $|00\rangle$. But consider the state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. Say we measure one

of the qubits, but not the other. We have an equal probability of getting 0 or 1. If someone else measures the second qubit, they should also have an equal probability of getting 0 or 1—but not really. The two qubits could only have been in the states $|00\rangle$ or $|11\rangle$, so whatever we measured, they must measure as well. In other words, measuring one of the cubits tells you something about the state of the other. This is true even if the qubits are separated before the measurement, so you can say that by measuring one of the entangled qubits, you have learned something instantly about a distant system, without needing to wait for the speed of light delay to transmit information.

Fortunately (or perhaps unfortunately, depending on your perspective) we can't actually use this effect to transmit data. A full discussion of the nuances of quantum information is beyond the scope of this section, but a toy understanding should be sufficient for the discussion of quantum programming languages.

2.2 The No-Cloning Theorem

One very important result in quantum information theory is the so-called no-cloning theorem. In English, the theorem states that you cannot make an exact copy of a quantum state. More formally, there is no operation you can perform that will take some known proto-state $|e\rangle$ and an *arbitrary* state $|\psi\rangle$ and transform them from the state $|e\rangle\,|\psi\rangle$ to the state $|\psi\rangle\,|\psi\rangle$. (We're being a little loose with the notation here: we can write states of multiple particles or qubits as a concatenation either inside or outside the kets: $|0\rangle\,|1\rangle$ is the same as $|01\rangle$, and so on.)

We can provide a small proof of the theorem, filling in the necessary pieces from quantum mechanics as we go. First we should note that according to quantum information theory we can represent any operation, or any computation, if you will, as a unitary operator on some Hilbert space. So, our "cloning" would be accomplished by some unitary operator $\hat{U}$, with

$$\hat{U}\,|e\rangle\,|\psi\rangle = |\psi\rangle\,|\psi\rangle\,.$$

Recall that $|\psi\rangle$ was necessarily arbitrary, so we also have

$$\hat{U}\,|e\rangle\,|\phi\rangle = |\phi\rangle\,|\phi\rangle\,.$$

Now we can write an inner product, and use the fact that $\langle e|e\rangle = 1$, to get

$$(\langle e|\,\langle\psi|)(|e\rangle\,|\phi\rangle) = \langle\psi|\phi\rangle\,.$$

But, as it is a unitary operator, we have $\hat{U}^\dagger\hat{U} = 1$, so we can insert it to get

$$(\langle e|\,\langle\psi|)(|e\rangle\,|\phi\rangle) = (\langle e|\,\langle\psi|)\hat{U}^\dagger\hat{U}(|e\rangle\,|\phi\rangle).$$

Allowing $\hat{U}^\dagger$ to act to the left and $\hat{U}$ to act to the right, we can see that

$$(\langle e|\,\langle\psi|)\hat{U}^\dagger\hat{U}(|e\rangle\,|\phi\rangle) = (\langle\psi|\,\langle\psi|)(|\phi\rangle\,|\phi\rangle).$$

Putting all this together, we reach the conclusion that

$$(\langle e| \langle \psi|)(|e\rangle |\phi\rangle) = \langle \psi|\phi\rangle = \langle \psi|\phi\rangle^2 .$$

The inner product $\langle \psi|\phi\rangle$ is some real number, so the only way for it to be equal to its square is for it to be one or zero. If the inner product is one, then $|\psi\rangle = |\phi\rangle$. If it's zero, then $|\psi\rangle$ and $|\phi\rangle$ are orthogonal to each other. Either way, $|\psi\rangle$ and $|\phi\rangle$ are related, so no matter what we choose for $|e\rangle$ and $\hat{U}$ we can never clone a totally arbitrary state. If we claim we can clone one state, then we can only clone that exact state or other states orthogonal to it.

This result has profound philosophical implications. It means that you can't make an exact quantum copy of an object, so you can't for example have a machine that scans things (like people) and produces multiple perfect replicas, in exactly the right quantum state. For a type system, it means that you cannot in general let a programmer duplicate variables at will, a situation you never have with a classical programming language. Interestingly, the no-cloning theorem does not prevent us from moving a state, just from duplicating it. As we will see with quantum teleportation, it is quite possible to transfer quantum information from one location to another as long as the information at the source is appropriately destroyed in the process.

3 Quantum Programming

The idea of a quantum programming language is simple. It's like a classical programming language, but with the ability to describe quantum information and algorithms that manipulate it. The most common approach for building a quantum programming language is the so-called "classical control, quantum data" model, which is exactly as it sounds. Such a language contains a classical subset that is by itself a complete programming language, like C or Haskell, and extends it with the ability to interact with quantum data.

Essentially, the programming language describes a classical machine that has access to a quantum oracle. A given program executes classically, but one of the classical, deterministic operations it is allowed to perform is to make contact with the oracle.

Of course, the actual interface is usually much more fine-grained than sending a query to an oracle. The oracle needs to be implemented somehow, so most quantum programming languages provide a complete set of primitive quantum operations that the classical control can perform on the quantum data. Typically this means qubit initialization and measurement and a small number of one and two qubit gates. These primitive operations correspond exactly to operators familiar in quantum information theory, a sort of quantum assembly, if you will.

This model is very useful to programmers who are already familiar with quantum information theory, as they can take gate-level descriptions of their algorithms and implement them directly in code. The classical control language even makes it easy to specify circuits with input-dependent size, and to construct hybrid algorithms like Shor's factorization algorithm which make a variable, nondeterministic number of calls to a circuit. In other words, the classical aspect of quantum languages is quite well developed (as it should be) but there is very little support for abstracting the quantum aspect beyond the lowest level of assembly.

The perfect example of this paradigm is QCL, or Quantum Computation Language. QCL greatly resembles C, but with an extension to handle qubits in quantum registers. Any quantum data is represented strictly as pointers: the names of qubits, but never the qubits themselves, which avoids the problem of the no-cloning theorem. Users can define custom quantum operators as a

combination of other existing operators, but there is neither the possibility of interesting control flow like recursion, or an interesting type system beyond checking properties like register widths.

In a practical sense, QCL is immensely useful, as it allows gate-level descriptions of algorithms to be quickly and easily translated into code. Code is definitely useful: it can be compiled to test on a quantum computer emulator, or linked with other classical programs to give them access to efficient quantum algorithms. However, it is relatively uninteresting in that it does not provide any significant abstraction beyond the gate-level description from quantum information theory. The quantum portion of any QCL program can almost as compactly be written out as a series of gates or a chain of operators.

Ideally, we would like to do better than this. We would like to have a type system that does more than check that operators act on registers of the right size. We would like to ask what it means to have a quantum function, and if it makes sense for it to be recursive. To address these questions, we turn our attention to the typed quantum lambda calculus as proposed by Selinger and Valiron.

4 The Quantum Lambda Calculus

The quantum lambda calculus proposed by Peter Selinger and Benoît Valiron is a natural extension of the classical typed lambda calculus to handle quantum data [6]. The type system is based upon intuitionistic linear logic, with the key feature being a notion of duplicability. We reproduce the syntax of the types below:

$$A, B ::= qbit \mid !A \mid (A \multimap B) \mid \top \mid (A \otimes B) \mid (A \oplus B).$$

$A \otimes B$ and $A \oplus B$ are the typical sum and product types, and $\top$ is the unique unit type. Classical information can be represented with the derived type $bit = \top \oplus \top$. $A \multimap B$ is a function type, which takes an argument of type A and produces type B.

The only particularly interesting type is $!A$, or "duplicable" A, which Selinger and Valiron refer to as the exponential. It has exactly the expected behavior: if A is some type, then $!A$ is a duplicable version of A with the additional property that it can be reused. Types satisfy a subtyping relation $<:$ such that a duplicable type is a subtype of the non-duplicable (or not necessarily duplicable) version, i.e. $!A <: A$. For a full discussion, see [6, pp. 10].

For a program to be quantumly sensible under the no-cloning theorem, we require that the type $!qbit$ never occur in it. This is not established with any particular typing judgment, but rather by making it impossible to introduce a term with the type $!qbit$. The terms in the language are given by the syntax

$$\begin{aligned} M, N, P \quad ::= \quad & c \mid x \mid \lambda x.M \mid MN \mid \langle M, N \rangle \mid * \mid \\ & let\ \langle x, y \rangle = M\ in\ N \mid inj_l(M) \mid inj_r(M) \mid \\ & match\ P\ with\ (x \to M \mid y \to N) \mid let\ rec\ f\ x = M\ in\ N. \end{aligned}$$

Again, these are fairly typical terms, with constructors and destructors for the function, product and sum types. For a complete description, refer to [6, pp. 6]. The quantumness of the language is hidden in the mysterious c term, which is intended to range over a complete set of quantum primitives. Selinger and Valiron suggest

$$c ::= new \mid meas \mid U.$$

new takes in a classical bit and instantiates a new qubit in one of the basis states: *new* 0 evaluates to $|0\rangle$ and *new* 1 evaluates to $|1\rangle$. *meas* performs a measurement of a qubit in the standard basis and returns the result as a classical bit. Finally, U ranges over some universal set of unitary gates, typically one-qubit rotations and a CNOT or CPHASE gate to cause interaction between qubits.

The primitives are assigned the following (most general) types:

$$\begin{aligned} new &: bit \multimap qbit \\ U &: qbit^{\otimes n} \multimap qbit^{\otimes n} \\ meas &: qbit \multimap !bit. \end{aligned}$$

It's fairly straightforward to show that it is impossible to introduce any terms that should contain the type $!qbit$ given the rules of the subtyping relation. See [6, pp. 12]. *new* is the only term that can produce a qubit, and it can be typed with any of

$$!(bit \multimap qbit) <: !(!bit \multimap qbit) <: !bit \multimap qbit$$

in addition to the typical $bit \multimap qbit$. Though some of these types accept duplicable arguments, or are themselves duplicable, they never introduce anything that is duplicable. This should make sense: the *new* operation itself would be implemented as some state preparation method on the quantum memory: though it creates (non-duplicable) quantum data, the method itself is just an operation performed by a classical machine, and could be reused without any complications.

Since none of the well-formed terms allows introduction of the type $!qbit$, it follows that any well-typed program, composed of only well-formed terms, does not contain the type $!qbit$. So the type system ensures that all programs make quantum sense, without having to introduce an explicit check for cloning beyond the notion of duplicability.

Quantum programs as described by the quantum lambda calculus are closures of the form $[Q, L, M]$, where Q is a normalized vector of n amplitudes, effectively the full state of the quantum memory, L is a list of n distinct term variables, and M is a lambda term whose free variables all appear in L. This allows us to describe systems with quantum data embedded in them. Selinger and Valiron provide a set of reduction rules to allow evaluation of quantum programs: for the full details, see [6, pp. 16].

The quantum lambda calculus is interesting because it provides a fairly minimal extension to a classical lambda calculus-like language that is universal for quantum computation. It does this by adding a notion of duplicability, which comes almost directly from intuitionistic linear logic. This should provide some indication of the close relationship between quantum information and linear logic—we will return to this topic later. Although the formal, algebraic semantics the quantum lambda calculus provides is useful, it is in many ways still unenlightening. What Selinger and Valiron have created is essentially a type system for a QCL-like language that still leaves all of the interesting quantum interaction sealed away inside a set of opaque quantum primitives, giving no higher level understanding that the gate or operator models. Still, having even a limited type system will allow us to make some interesting observations when we discuss teleportation.

5 QML

QML is an alternative quantum programming language, prosed by Jonathan Grattage [3]. While the quantum lambda calculus is firmly a classical control, quantum data language, QML attempts to provide quantum control in addition to quantum data. This leads to an entirely different notion of types and a different approach to handling no-cloning.

The types of QML are very simple, given by

$$\sigma, \tau ::= \mathcal{Q}_1 \mid \mathcal{Q}_2 \mid \sigma \otimes \tau.$$

These correspond to a quantum unit type, a quantum binary type (for all intents and purposes a qubit) and a product type, to handle systems of multiple qubits. The syntax of QML terms is given by

$$\begin{array}{rrl}
(Variables) & & x, y, ... \\
(ProbabilityAmplitudes) & & k, l, ... \\
(Patterns)\ p, q & ::= & x \mid (x, y) \\
(Terms)\ t, u & ::= & x \mid x^{\vec{y}} \mid () \mid (t, u) \mid \mathbf{let}\ p = t\ \mathbf{in}\ u \mid \\
& & \mathbf{if}\ t\ \mathbf{then}\ u\ \mathbf{else}\ u' \mid \mathbf{if}^\circ\ t\ \mathbf{then}\ u\ \mathbf{else}\ u' \mid \\
& & \mathbf{qfalse}^{\vec{y}} \mid \mathbf{qtrue}^{\vec{y}} \mid k \times t \mid t + u.
\end{array}$$

The superscript vectors $\vec{y}$ represent explicit weakening in the given variables, which is treated as a measurement. Most of the terms should be self explanatory: the language has tuples, let bindings which can be used to unpack them, and the quantum basis states in the form **qtrue** and **qfalse**. $k \times t$ indicates a term t with probability amplitude k, and $t + u$ indicates a superposition of t and u. There are two control constructs, one (**if**) being a classical if statement that is treated as a measurement, and the other (**if**$^\circ$) a quantum conditional that is treated as a superposition.

This is where the interesting quantumness of the language is revealed. Classical if statements behave as expected: **if** t **then** u **else** u' measures t as a classical bit and then performs either u or u' depending on the outcome. Quantum if statements, however, do not choose a single path for control to follow. **if**$^\circ$ t **then** u **else** u' evaluates *both* u and u', and gives the outputs probability amplitudes according to t. So if t evaluates to **qtrue**, then the if statement would give u with amplitude one; if t evaluates to an equal superposition, then the if statement would give an equal superposition of u and u', and so on.

We should make a few critical observations here before we proceed. The fact that a quantum conditional proceeds down both branches simultaneously seems to imply that the control flow graph of a program can explode exponentially over time. This is certainly true for a classical simulation of a quantum program, but hardly surprising, as in general simulating quantum computation with a classical computer requires exponential overhead to store all of the possible probability amplitudes. Given an actual quantum computer, a QML program could be compiled down to a single unitary operator describing the computation which could then be carried out with some combination of gates or operators, allowing the state space explosion to be handled by quantum representation.

Note that there is no mention of a duplicability judgment; there does not need to be, as QML is designed to obey the dual of the no-cloning theorem, the equivalent no-deleting theorem. Instead

of restricting duplication of terms, QML restricts weakening. As mentioned, a QML program can be compiled directly to a unitary operator, or a purely quantum computation, unlike QCL or the quantum lambda calculus, which describe essentially classical programs with operations on quantum data thrown in. Any weakening is treated as measurement, and moved to the end of the computation (which can be shown to be a valid rearrangement). For a full discussion of how the type system works, see [3, pp. 104].

There are two interesting conditions that must be enforced by the type system in order for terms to make sense. The first is quite straightforward: superpositions must always be normalized, as would be expected. The second is much more interesting. The branches of a quantum conditional must be orthogonal to each other, as the conditional will create a superposition of them, and superpositions only make sense when the terms are orthogonal. We have avoided this topic so far by considering only superpositions of products of the basis states, which are by definition all orthogonal. But we could imagine trying to form a superposition of $|0\rangle$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. This doesn't make sense in a quantum mechanical sense, so it should be disallowed by the type system.

To handle this, Grattage defines an inner product over terms, with orthogonality treated in the typical way [3, pp. 114]. We will not discuss the details here, but a simple example might be instructive. Consider the term $\mathbf{if}^\circ\ x\ \mathbf{then\ qtrue\ else\ qtrue}$. Regardless of the value of x, the term will simply evaluate to **qtrue**, discarding any information about x in the process. This is equivalent to an inexplicit measurement, which should be disallowed by the type system. The only way to prevent this kind of information loss is to ensure that the branches are orthogonal; they can't "overlap" each other, or it would be unclear which was followed by which component of the superposition.

QML provides a very interesting description of quantum computation, which unlike QCL and the quantum lambda calculus is fundamentally different from the gate and operator models. QML programs are operators, but they are described in a syntax that resembles a functional programming language more than it resembles logic circuits or matrix algebra. The key component is the quantum conditional, and its unintuitive behavior and unusual typing restrictions seem to indicate both the power and the limitations of what we can describe with quantum algorithms.

However, QML doesn't exactly make it easy to write quantum programs. Speaking from experience, the author can attest that most known algorithms are actually much harder to code in QML than in QCL, and no new algorithms have been developed with it. There are many limitations that make it unsuitable as a general quantum programming language. Most importantly, all of the data types are finite, so it is impossible to describe operations on systems of general or parameterized sizes. There is also no notion of recursion, which is unfortunate but probably necessary, as it isn't clear what purely quantum recursion would mean, or if it simply doesn't make sense, like a non-orthogonal superposition.

6 Two Views of Quantum Teleportation

Although the no-cloning theorem states that quantum information cannot be duplicated, it can still be moved from one place to another without a direct quantum channel. The procedure is called quantum teleportation, and it can be described as a quantum circuit, as show in Figure 1.

The purpose of the circuit is to transfer the state of qubit 1 to qubit 3. If the qubits were represented by particles, this would correspond to putting particle 3 into exactly the same state that particle 1 was in initially, but changing the state of particle 1 in the process. The reason for the

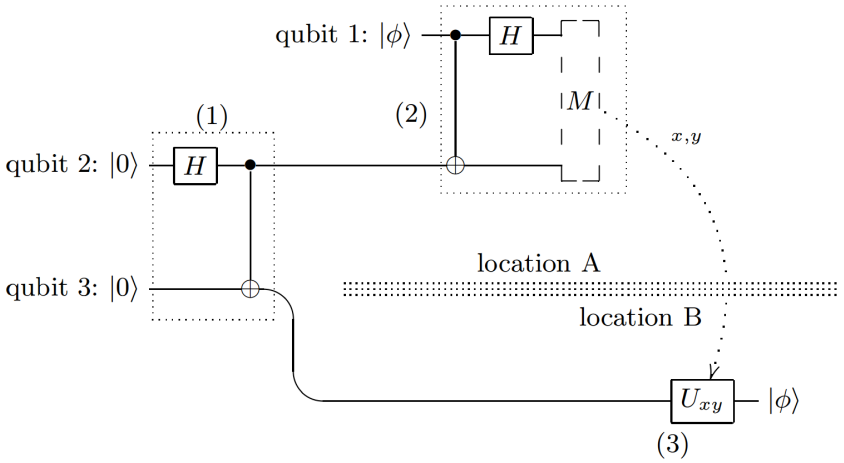

Figure 1: Quantum Teleportation circuit, see [6, pp. 4].

name "teleportation" is that neither of the particles needs to be relocated physically: the quantum information is "teleported" by two classical bits.

The teleportation algorithm works in three phases. First, qubits 2 and 3 are entangled in a Bell state (as it turns out, the same state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ we have already discussed) by a Hadamard gate and a controlled not gate. Then a Bell state measurement is made of qubits 1 and 2 (*not* qubits 2 and 3: indeed, we are measuring the qubit we want to teleport with one of the two entangled qubits, which may seem a bit counterintuitive). This measurement is similar to a standard basis measurement, but its result is one of the four entangled Bell states—the full details are not important here, but one may want to refer to the literature. What is important is that based on the outcome of the measurement, a corrective operation can be performed on qubit 3 to put it in the same state qubit 1 was in initially.

Note that this does not violate the no cloning theorem at all, as the Bell state measurement has disrupted the initial state of qubit 1. Also note that the reason the algorithm works is because of the entanglement between qubits 2 and 3. In theory, they could be separated after the initialization of the Bell state and used for teleportation at a later time, as long as the states were not accidentally measured. In practice, keeping qubits in entangled states is incredibly difficult.

The teleportation algorithm can be expressed in both the quantum lambda calculus and QML. We won't examine a lambda calculus implementation explicitly, but we can learn some interesting things by looking at the types.

The initialization, step (1) of the algorithm, should have type $\top \multimap qbit \otimes qbit$. This is duplicable under the subtyping relation, as there is no embedded quantum data, even though the output type is clearly not duplicable. Step (2) has type $qbit \otimes qbit \multimap bit \otimes bit$ and step (3) has type $qbit \otimes bit \otimes bit \multimap qbit$, again with both being subtypes of a duplicable type as functions.

We can curry the types of steps (2) and (3) to produce $qbit \multimap (qbit \multimap bit \otimes bit)$ and $qbit \multimap (bit \otimes bit \multimap qbit)$, respectively. Both of these types take a $qbit$ as input, so we can combine them into a tuple and then compose with (1), to see that the whole teleportation procedure effectively has type $\top \multimap qbit \otimes qbit \multimap (qbit \multimap bit \otimes bit) \otimes (bit \otimes bit \multimap qbit)$.

In other words, teleportation takes in nothing, creates an entangled pair of qubits, and then outputs two functions we can call f and g, such that $f : qbit \multimap bit \otimes bit$ and $g : bit \otimes bit \multimap qbit$. These functions form a "single-use isomorphism" between the types $bit \otimes bit$ and $qbit$, which are in general not isomorphic, as they satisfy $g(f(|\psi\rangle)) = |\psi\rangle$ for all qubits $|\psi\rangle$, and $f(g(x, y)) = (x, y)$ for all bits x and y [6, pp. 5]. They are actually entangled: thought the whole operation is predictable duplicable (you can use the same method to teleport as many different qubits as you want), the output of a given instance is not.

This should make perfect sense. In creating the functions f and g, we have had to handle quantum data, upon which the behavior of f and g depends. It is not really the functions that are entangled but the qubits which they interact with. What is happening is that the quantum lambda calculus gives us and abstraction that lets us directly embed quantum information into computations and reason about them together. It perfectly satisfies the traditional lambda calculus motto: "Values are functions."

We can also encode teleportation with QML, which will give us a very different understanding. Though the types are relatively uninteresting, a QML implementation will provide an explicit description of the necessary operators in terms of only the QML control primitives. For reference, we provide an alternative circuit diagram in Figure 2.

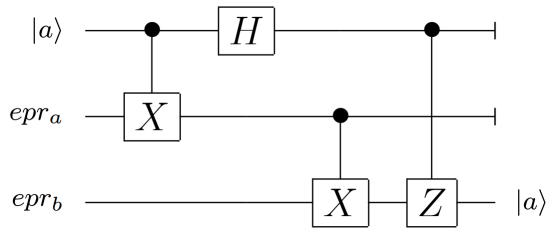

Figure 2: Quantum Teleportation circuit with delayed measurement, see [3, pp. 60].

In this diagram, it is assumed the initialization of the Bell state has already happened, between the qubits labeled erp_a and epr_b. "EPR" refers to an Einstein Podolsky Rosen pair, another name for the Bell state. The first CNOT gate and Hadamard gate prepare qubits 1 and 2 for the Bell measurement; the following CNOT and Pauli Z gates perform the correction. Finally, the lines indicating the termination of qubits 1 and 2 specify measurement. According to the principle of deferred measurement, this circuit is identical to the circuit that measures and sends classical bits to control the final NOT and Z gates, rather than using quantum controlled gates as depicted. The standard basis measurements indicated are equivalent to a Bell measurement due to the preparation done by the first two gates.

We can implement the circuit with QML code. First, we define all of the gates:

$$\begin{aligned}
\text{had } b &= \textbf{if}^\circ\ b\ \textbf{then}\ (-1) \times \textbf{qtrue} + \textbf{qfalse} \\
&\quad\ \textbf{else qtrue} + \textbf{qfalse} \\
\text{qnot } b &= \textbf{if}^\circ\ b\ \textbf{then qfalse else qtrue} \\
\text{cnot } s\ t &= \textbf{if}^\circ\ s\ \textbf{then}\ (\textbf{qtrue}, \text{qnot } t)\ \textbf{else}\ (\textbf{qfalse}, t) \\
\text{z } x &= \textbf{if } x\ \textbf{then}\ (-1) \times \textbf{qtrue else qfalse}.
\end{aligned}$$

Reading the descriptions, and comparing them to the known behavior of the operators, is somewhat interesting. It is sometimes convenient to think of the quantum conditional as an isometry, specifying what each possible value of the argument is mapped to. Notice how the CNOT function preserves the value of s; the definition of the NOT gate is required to show that the states are properly orthogonal. Note that correctly normalized amplitudes are sometimes omitted: when this is the case, it is assumed that they are simply equal.

It is straightforward to combine these gates into an implementation of teleportation.

$$\begin{aligned}
\text{tel } x \quad = \quad & \textbf{let } (a, b) = (\textbf{qfalse}, \textbf{qfalse}) + (\textbf{qtrue}, \textbf{qtrue}) \\
& (a', x') = \text{cnot } a\ x \\
& b' = \textbf{if } a \textbf{ then } \text{qnot } b \textbf{ else } b \\
& b'' = \textbf{if } \text{had } x' \textbf{ then } \text{z } b' \textbf{ else } b' \\
& \textbf{in } b''.
\end{aligned}$$

Looking back at Figure 2, we can see that all of the gates are implemented. The measurements are explicit in the classical if statements, but in constructing the full operator the compiler would move them to the end as shown in the circuit.

Unlike the quantum lambda calculus, QML doesn't let us say anything about the interaction between classical and quantum types. QML is a purely quantum language, so in some ways using it to implement teleportation is silly. The point of teleportation is to transfer a qubit using a classical channel, but for quantum computation the existence of a working quantum channel is assumed; we could have just used the qubit without first teleporting it. However, QML does provide an interesting abstraction of the gates themselves, which cannot really be said about the quantum lambda calculus or QCL. How useful that abstraction is to a programmer is difficult to ascertain.

7 Conclusion

We have discussed two models of quantum computation: the quantum lambda calculus, which extends classical control structures to handle quantum data, and QML, which provides quantum control structures to describe purely quantum operations directly. Both models are interesting in their own right: the quantum lambda calculus allows us to create entangled functions, while QML implements a set of universal quantum operators with functional programming. However, both models have their limitations as well. The quantum lambda calculus does not provide any insightful description of quantum operations, only the relationship between quantum data and the classical control, and QML is limited by finite types.

It would be interesting to combine the two approaches in a language that provided a classical control, quantum data framework as well as a functional description of the basic quantum operations. Such a language might be able to provide a more usable interface for writing programs that take advantage of quantum effects, or establish the meaning of mysteries like quantum recursion. However, in quantum physics one must always be careful to check that things make sense. It is easy to accidentally express a system that is nonsensical, such as a superposition of two non-orthogonal states. A type motivated description that goes beyond the gate level might befall the same fate.

References

[1] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 249–258, 2005. http://arxiv.org/abs/quant-ph/0409065.

[2] Thorsten Altenkirch and Jonathan Grattage. An Algebra of Pure Quantum Programming. *Electronic Notes in Theoretical Computer Science*, 170:23–47, 2007. http://arxiv.org/abs/quant-ph/0506012.

[3] Jonathan James Grattage. *A functional quantum programming language*. PhD thesis, University of Nottingham, September 2006.

[4] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. *ACM SIGPLAN Notices*, 48(6):333–342, 2013. http://arxiv.org/abs/1304.3390.

[5] Bernhard Ömer. *Structured Quantum Programming*. PhD thesis, Vienna University of Technology, September 2009.

[6] Peter Selinger and Benoît Valiron. Quantum Lambda Calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 2009.