
Vadim von Brzeski
Department of Computer Science
University of California, Santa Cruz

December 1, 2004

Abstract
This is a survey of research in the area of programming languages for quantum computing.

1 Introduction

In 1994, Peter Shor [13] produced a quantum algorithm that could solve the factorization problem in polynomial time. Since then there has been a flurry of research into algorithms that could run on future quantum computers. Shor presented his algorithm in terms of rigorous and detailed mathematics that governs the transformations of quantum states in a hypothetical quantum machine, all according to the physical laws of quantum mechanics. However, it was not readily evident how one could translate such a formal mathematical description into a quantum program that could be run on a quantum computer.

Therefore, if and when practical quantum computers become available, there will need to be a practical method of describing quantum algorithms that can abstract the complex mathematics of quantum mechanics to a quantum computer programmer, i.e. a quantum programming language. Once we have quantum computer hardware, we will need quantum computer software, and ideally, software that is hardware independent. Also, we will need compilers for quantum programming languages that produce machine code suitable for quantum computers. Whatever form these quantum programming languages take, they will need to guarantee that the algorithms they describe are well-formed quantum programs, i.e. that they obey the physical laws that govern quantum mechanical systems. Ideally, quantum language compilers will need to determine if a quantum program is well formed and well typed at compile time, rather than letting an illegal (leading to a crash) or nonsensical (leading to incorrect results) operation to be performed at run-time. The primary rules that a quantum programming language will need to enforce are based on the following concepts in quantum mechanics: reversibility, non-determinism, entanglement, measurement, and the no-cloning property.

This paper is a survey of research dealing specifically with issues particular to the development of quantum programming languages, such as:

- How can we succinctly code a quantum algorithm using a quantum programming language?
- How do we determine if a program in a quantum programming language is well formed and well typed?
• How do we formally reason and prove the correctness of such programs?
• What are possible semantics of a quantum programming language?
• Can we simulate quantum algorithms in the absence of quantum computer hardware?

Research into these areas is split into two camps: the formalists, whose goal is to define a formal syntax and semantics for possible programming languages, and the pragmatists, whose goal is to develop quantum simulators using classical hardware. This survey paper is thus organized according to this split. After presenting a brief overview of the basics of quantum computing and associated implications in Sections 2 and 3, Section 4 outlines a number of formal approaches to quantum programming languages. Section 5 describes the practical approaches taken to simulate quantum computations on classical computers. The relevant mathematical notation of quantum computing and quantum mechanics is explained in detail in the Appendix; readers unfamiliar with vector spaces and tensor products, and their relationship to quantum computing, may wish to read the Appendix first.

2 Brief Overview of Quantum Computing

The lure of quantum computing is the tremendous parallelism offers compared to classical computing. The computational ability of a classical computer increases linearly with the size of the system, whereas it increases exponentially in a quantum system (see Appendix 7.1.3). This can be thought of as an exponential ($O(2^n)$) increase in the number of parallel processors resulting from a linear ($O(n)$) increase in the size of the system. Such an exponential increase in computing power could be used to develop efficient (polynomial time) solutions to problems currently lacking efficient algorithms. Factoring a large number into a product of primes, which is the foundation of many commercial encryption algorithms (e.g. RSA), is an example of such a problem.

This exponential increase is due to the fact that although a classical bit can exist in one of two states, either "0" or "1", a quantum bit, or qubit, can exist in any linear combination, called a superposition, of the states "0" and "1". In other words, a qubit can be in states "0", "1", or in a state that can be best described as both "0" and "1". Similarly, a 2-bit classical system can be in one of four states: "00", "01", "10", or "11". A 2-qubit quantum system can exist in each of the above four classical states, plus any state which is a linear superposition of the above four states, e.g. $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|11\rangle$, a state which has no classical counterpart.

An n-qubit quantum computing system can be likened to a classical computer running $2^n$ parallel threads simultaneously looking for a solution among all $2^n$ possible values of the input. The problem is that although you have $2^n$ threads, you don't know which one is computing the correct answer. Each of the $2^n$ possible threads has some probability of yielding the correct answer, and so the approach taken by many quantum algorithms is to perform a sequence of quantum operations on the input data such that over time the probability of the correct answer is much higher than the probabilities of the incorrect answers. At this point, an actual measurement is performed, and an answer is presented based on its probability. However, it may not be the correct answer, since the probability of the correct answer is not usually guaranteed to be 1. However, if the probability is much greater than the probability of any incorrect answer (e.g. 0.6 vs 0.1), then repeating the entire process a few more times should yield the desired answer.

The generally accepted hardware model for a possible quantum computer is the Quantum Random Access Machine (QRAM) model due to Knill [1]. This model is based on a classical control system driving
a quantum subsystem. The QRAM subsystem is composed of a set of quantum registers (registers containing qubits), and can respond to a set of quantum operations. The classical control system translates a quantum program into a sequence of quantum operations, initializes the n-qubit quantum subsystem, performs the sequence of quantum operations on the subsystem, and finally measures the results of the computation. The QRAM model has been used to describe most quantum algorithms today. Another possible model of quantum computation is the quantum Turing machine, in which both the control system and the data are quantum.

3 Quantum Mechanics Rules and Implications for Quantum Programming Languages

3.1 Reversibility of Operations

Any operation performed on a quantum state must be reversible. The reason for this comes from the mathematics of quantum mechanics, and the fact that each matrix $U$ that defines a quantum operation (see Appendix 7.1.2) must be invertible, i.e. it must have an inverse matrix $U^{-1}$ such that $UU^{-1} = I$. Moreover, the matrix $U$ also needs to be unitary, meaning that its conjugate transpose is equal to its inverse, $U^* = U^{-1}$. Not all matrices are unitary, and thus not all matrices define legal operations on qubits. The implication of this, for example, is that the very familiar assignment operation in classical programming languages like $x := 5$ is not legal if $x$ denotes a quantum variable, since after the assignment the previous value of $x$ is lost.

3.2 No-Cloning Rule

The no cloning rule of quantum mechanics states that you cannot clone a quantum system that is an unknown state. This can be seen as an example of the Heisenberg Uncertainty Principle which states that one cannot both measure the momentum and position of a particle to any arbitrary degree of accuracy; the more accurate the measurement of one, the more error in the measurement of the other. Suppose you could clone a particle in an unknown quantum state. Then you could exactly measure the position of the original particle, and exactly measure the momentum of its identical clone, thus violating the uncertainty principle. In terms of mathematics, given that all quantum operations are defined in terms of unitary matrices, there is no unitary matrix $U$ such that $U(|a0\rangle) = |aa\rangle$ for any quantum state $a$ [12]. Note that known quantum states, which are in fact nothing more than classical states, can be cloned.

The implication of the no-cloning rule on programming languages is that the familiar copy-by-value parameter passing convention in procedure calls is not allowed. Also, if a procedure takes in multiple parameters, they must be distinct, i.e. the following is not allowed: $procedure(a,a)$.

3.3 Measurement

The measurement of a quantum value permanently collapses the quantum state of that value to a classical state. Also, the measurement of one value may have non-local side effects on other values, e.g. if the measurement is of a value in a pair of entangled values. For example, if a single qubit is in quantum state $u = 0.8|0\rangle + 0.6|1\rangle$, when measured it will collapse to the classical state $u = 1|0\rangle + 0|1\rangle$ with a probability of $(0.8)^2 = 64\%$ or collapse to $u = 0|0\rangle + 1|1\rangle$ with a probability of $(0.6)^2 = 36\%$. All quantum information
is lost, and the single qubit will from then on act as a classical value (unless again transformed by a Hadamard).

The situation is a bit more interesting for a 2-qubit system. For example, let the quantum state $u = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$. Measuring the first qubit only collapses the quantum state of that qubit, returning $|0\rangle$ or $|1\rangle$ with equal probability of 50%. After measurement, the new state will be either $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle$ or $\frac{1}{\sqrt{2}}|10\rangle + \frac{1}{\sqrt{2}}|11\rangle$.

The consequence of this for quantum programs is that once a value of quantum variable is read and known, this variable loses some or all of its quantum properties (i.e. superposition), and cannot be further manipulated as a quantum variable. Thus care must be taken when and how measurements are performed in quantum programs. Furthermore, as shown below, measurement can have non-local side effects due to entanglement.

### 3.4 Entanglement

Entanglement is an entirely quantum concept with no analog in classical computing. It means that the state of quantum system cannot always be decomposed into a collection of states of its underlying components. Quantum bits are said to be entangled if the measurement of one qubit collapses the quantum state of the other. In the measurement example above, the two qubits described by state $u = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$ are not entangled since reading the first still leaves the second in a quantum superposition of $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. However, the two qubits in the state $w = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ are entangled since they can either both be 0 or both be 1. Measuring one of them as 0 forces the other to be 0, and measuring one of them as 1 forces the other to be 1, i.e. the entire quantum state collapses into one of two classical states: $|00\rangle$ or $|11\rangle$.

Entangled pairs can be produced using the controlled not gate (see Appendix 7.1.2). Suppose the first (control) qubit is in a superposition of true (0) and false (1) defined by state $u = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and the second (target) qubit is false (0) defined by state $v = |0\rangle + 0|1\rangle$. Applying $C_{not}$ (see Appendix 7.1.2) to the combined state $(u \otimes v) = \frac{1}{\sqrt{2}}|00\rangle + 0|01\rangle + \frac{1}{\sqrt{2}}|10\rangle + 0|11\rangle$:

$$
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
\frac{1}{\sqrt{2}} \\
0 \\
\frac{1}{\sqrt{2}} \\
0
\end{bmatrix}
= 
\begin{bmatrix}
\frac{1}{\sqrt{2}} \\
0 \\
0 \\
\frac{1}{\sqrt{2}}
\end{bmatrix}
= \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle.
$$

Another way of looking at entanglement is to note that an entangled state cannot be written as a tensor product (see Appendix 7.1.3) of its independent component states, e.g. there are no such $u, v$ such that $(u \otimes v) = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. Equivalently an operation (matrix) creating an entangled pair of qubits cannot be written as a tensor product of two single qubit operations (matrices), e.g. there are no such unitary matrices $A, B$ such that $A \otimes B = C_{not}$.

The consequence of this for quantum programs is that entangled pairs can produce non-local side effects between quantum variables. This has implications for measurement and debugging. For example, one cannot simply read the value of a qubit to see if the program is working without potentially disrupting the entire computation. Furthermore, a quantum computation that produces entangled n-tuples of qubits cannot be analyzed by what it does to individual qubits. This is one of the central problems in modeling quantum computations using classical computers.
4 Formal Approaches

4.1 Quantum Pseudocode

One of the earliest attempts at a formal specification for quantum programming languages was quantum pseudocode by Knill [1]. Knill defined a set of rules and a syntax for writing down quantum algorithms in pseudocode that is similar to pseudocode syntax for classical algorithms, but which adhere to the rules of quantum computation. Knill defined notation for quantum bit variables versus classical bit variables, as well as notation for quantum operations versus classical operations. For example, quantum operations can be applied only to quantum variables, and include a finite set of unitary operations. Classical operations like "+", ",", are excluded from the set of quantum operations, ensuring that all quantum operations are reversible.

Knill also defined rules for measurement. For example, no quantum variable can appear on both the left and right hand side of an assignment statement, since if a quantum variable is on the right hand side, it means it is being read (measured). If it is being measured, it may cease being a quantum variable after the measurement, so it cannot be assigned to a quantum variable on the left hand side. Also, a variable appearing only on the left hand side of assignment can only be a classical value. Otherwise, if it were quantum and only on the left hand side, its value would be erased by the assignment, violating the principle that all quantum operations must be reversible. A variable appearing on the left hand side as classical and on the right hand side as quantum means that a measurement of that quantum variable has taken place. A procedure that performs quantum operations can be reversed, but only if it does not perform any measurements (which are not reversible).

4.2 Probabilistic Programs and Non-Determinism : pGCL

Quantum computing inherently involves probability and non-determinism. Therefore it is not surprising that much work on formal frameworks for quantum programming languages is based on frameworks for reasoning about probabilistic and non-deterministic computation, some of which were developed independently of quantum computing. One such framework was put forward by Morgan et al [3], based on Dijkstra's Guarded Command Language (GCL) [14]. The following is not meant to be a comprehensive account of pGCL, but is intended to give the reader a flavor of a probabilistic version of an axiomatic semantics for reasoning about non-deterministic programs.

Morgan develops an extension to GCL, adding probabilism and non-determinism, and calls it pGCL. Non-deterministic choice was first presented by Dijkstra in GCL (in 1975) as a set of overlapping guards. If more than one guard evaluated to true in a given expression, the program would choose which guarded list to execute in non-deterministic fashion. Since then non-determinism in programs has acquired a certain formal notation : \textit{option\_A \cap option\_B}, which means the program can choose unpredictably between option\_A and option\_B, either option producing the desired results in the end. The \cap signifies non-deterministic choice. An important note is that non-determinism is not the same concept as probabilism. We follow some of the examples in [3] to introduce these two concepts as they relate to pGCL.

Consider the following sample program :

\[
x := -y \quad 0.25 \oplus x := y
\]

The notation \(0.25 \oplus\) means choose the left branch with probability 0.25 and choose the right branch with probability \(1 - 0.25 = 0.75\).

As Dijkstra introduced the logic of "weakest preconditions" in deterministic programs, Morgan defines the logic of "greatest pre-expectations". Recall that in deterministic programs, a weakest pre-condition
$S = \wp(C, P)$ is the weakest boolean predicate that must hold in the initial state of a program $C$ such that a boolean predicate $P$ holds in the final state of program $C$. So if $S = \wp(C, P)$, then the program fragment \{\{S\}C\{P\}$ means that if $C$ starts out in a state where $S$ holds, then $C$ will end up in a state where $P$ holds. In other words, we can ask what is the weakest such $S$ such that $C$ is guaranteed to end up in a state where $P$ holds?

When dealing with probabilistic programs such as \{\{x := -y \qquad 0.25 \oplus \qquad x := y\} \}, we ask given that $S = \wp(C, P)$, what is the probability that $C$ will reach a state where $P$ holds? To answer this question we need to calculate $\wp.C.P$. In the following example calculation, Morgan uses the "." notation to refer to a function applied to an argument, i.e. $f.x = f(x)$, so we follow that notation here. The calculation of $\wp.C.P$ is done using the following two rules:

$$\wp.(x := E).P \quad := \quad \text{"replace } x \text{ by } E \text{ in } P \quad (1)$$

$$\wp.(A \quad p \oplus \quad B).P \quad := \quad p \times \wp.A.P \quad + \quad (1-p) \times \wp.B.P \quad (2)$$

In our example program above, we can now calculate the probability of the weakest precondition $S = \wp.C.[x \geq 0]$ in the initial state of the program.

$$\wp.(x := -y \qquad 0.25 \oplus \qquad x := y).[x \geq 0]$$

$$= 0.25 \times \wp.(x := -y).[x \geq 0] + 0.75 \times \wp.(x := y).[x \geq 0] \quad \text{by (2)}$$

$$= 0.25 \times \wp.(-y \geq 0) + 0.75 \times \wp.[y \geq 0] \quad \text{by (1)}$$

$$= 0.25[y < 0] + 1.0[y = 0] + 0.75[y > 0] \quad \text{arithmetic}$$

The probability values are read from the result for various values of $y$. Reading the above, we can say that $x \geq 0$ will hold with probability of 25% when $y < 0$, with probability of 75% when $y > 0$, and with probability of 100% when $y = 0$. As we see, the calculation of $\wp.C.P$ results in a probability distribution, or a list of expectations of the pre-condition, i.e. a list of "pre-expectations".

The above example does not contain any (demonic) non-determinism $\sqcap$ operations; it is simply a probabilistic program. The next example extends the above program to include non-deterministic choice. Suppose we say that the left branch $x = -y$ is to be performed with probability between $\frac{1}{4}$ and $\frac{3}{4}$ (we don’t care what the exact probability is as long as it is within that range - the outcome of the program will still be correct), and the $x = y$ right branch otherwise. In this case, our program statement looks like:

$$x := -y \qquad 0.25 \oplus \qquad x := y \quad \sqcap \quad x := -y \qquad 0.30 \oplus \qquad x := y$$

To deal with such programs, Morgan defines as additional rule:

$$\wp.(A \sqcap B).P \quad := \quad \min(\wp.A.P, \wp.B.P) \quad (3)$$

using the minimum function since the "demon" in "demonic non-determinism" will always make a choice such that $P$ is least likely to hold. Repeating our calculation with $P : x \geq 0$:

$$\wp.(x := -y \qquad 0.25 \oplus \qquad x := y \quad \sqcap \quad x := -y \qquad 0.80 \oplus \qquad x := y).[x \geq 0]$$

$$= \min(\wp.(x := -y \qquad 0.25 \oplus \qquad x := y).[x \geq 0], \wp.(x := -y \qquad 0.80 \oplus \qquad x := y).[x \geq 0]))$$

$$= \min(0.25[y < 0] + 1.0[y = 0] + 0.75[y > 0], 0.80[y < 0] + 1.0[y = 0] + 0.20[y > 0])$$
= 0.25[y < 0] + 1.0[y = 0] + 0.20[y > 0]

Looking at our original program, if \( y < 0 \), the "demon" will choose the path that makes the desired result \( x \geq 0 \) least likely (or the undesired result \( x < 0 \) most likely). In this case, this is the left branch of \( \Pi \) since \( x := y \) (leading to the undesired result of \( x < 0 \)) is more likely to happen in the left branch (75\%) that in the right branch (20\%). Reading the results of our calculation above, the best that we can say is that \( x \geq 0 \) will hold with probability of at least 25\% when \( y < 0 \). If \( y > 0 \), then the demon will choose the branch that is most likely to execute \( x := -y \) (leading again to the undesired result of \( x < 0 \)), in this case the right branch (80\% vs. 25\%). We can say that \( x \geq 0 \) will hold with probability of at least 20\% when \( y > 0 \). It is interesting to compare this result with the analysis of the probabilistic program with no demonic non-determinism, where we saw that \( x \geq 0 \) will hold with probability 75\% when \( y > 0 \). This is the difference between simple probabilism and demonic non-determinism. If \( y = 0 \), no matter what the demon does, \( x \geq 0 \) holds with probability 1. The probabilities we just calculated are the "greatest pre-expectations" we can have for \( x \geq 0 \) holding in the final state.

Following such techniques, Morgan builds a semantic framework, including a refinement calculus (in pGCL, a program \( Q \) is a refinement or transformation of program \( P \), written as \( P \subseteq Q \), meaning that \( Q \) is at least as deterministic as \( P \), or \( Q \) is no more non-deterministic than \( P \)), around pGCL for reasoning about random algorithms - see [3] for a complete description of pGCL and proofs of its correctness.

### 4.3 From pGCL to qGCL

Sanders and Zuliani [5] extend pGCL to a "quantum pGCL", and call it qGCL. A program written in qGCL is nothing more than a program written in pGCL that calls quantum procedures. Specifically, Sanders et al define the following three quantum procedures: initialization (via the Hadamard operation), evolution (via the application of unitary operations), and finalization (measurement). For example, using pGCL, a measurement procedure in qGCL of a single qubit in state \( \alpha \left| 0 \right> + \beta \left| 1 \right> \) that returns the classical value \( x \) is written as follows:

```plaintext
measure_single_qubit(qubit: y \equiv \alpha \left| 0 \right> + \beta \left| 1 \right>)
x := \left| 0 \right>
\alpha \left| 0 \right> + \left| 1 \right>
return x
```

The semantics of qGCL are entirely based on that of pGCL, so at first it did not bring much to the table. Besides measurement, it did not address in detail the issues concerning quantum operations, such as reversibility, entanglement, or cloning. However, later on Zuliani specifically dealt with the issue of reversibility of quantum operations in qGCL [6].

Zuliani developed a method and a set of rules for translating a program in pGCL into an equivalent reversible program. Reversibility of classical computation had been first studied by Bennett [15] in 1973, in which he presented a reversible Turing machine that was able to compute any computable function. Zuliani merged the work of Bennett with pGCL to create a reversible pGCL.

Using the semantics of pGCL, Zuliani first defined what it means for two statements to be equivalent. Two statements \( R \) and \( S \) are equivalent if \( wp.R.Q \equiv wp.S.Q \), in other words the probability that \( R \) will reach \( Q \) is equal to the probability that \( S \) will reach \( Q \). A statement \( R \) is reversible if there exists a statement \( S \) such that \( (R; S) \equiv skip \), and a program is reversible if each of its statements is reversible. A reversible program terminates for all inputs, and a terminating computation can be performed in reverse.
Thus for any terminating program $P$, there is an equivalent reversible program $P_r$ and its inverse $P_i$ such that $(P_r ; P_i) \equiv \text{skip}$.

An example of a transformation according to Zuliani’s rules from a non-reversible to a reversible program is as follows. Suppose you have the non-reversible program

$$x := e;$$

The reversible program $P_r$ is :

```plaintext
push x;
```

$$x := e;$$

And its inverse $P_i$ is :

```plaintext
pop x;
```

It is clear that $(P_r ; P_i) \equiv \text{skip}$. The above example outlines the general pattern for the rest of Zuliani’s rules of creating equivalent reversible statement $P_r$ from statements $P$ of the form $(R; S)$: while $c$ do $S$, if $c$ then $R$, else $R \cap S$, and $R \cup S$. Specifically, $P_r$ is an augmented version of $P$ that pushes the initial state of $P$ onto a stack, and also recursively saves the execution history of $P_r$ on the stack (keeping track of which way the probabilities fell out); once $P_r$ is done, the stack is cleaned by $P_i$. For the specifics of each rule, see [6].

But how do you obtain useful results from a reversible computation? The trick is that before you run $P_i$, you copy the results of $P_r$ into a separate location, e.g. a separate (initially empty) stack. This midway copy operation is trivially reversible as well - just pop off all the elements from this utility stack.

Zuliani’s pGCL to reversible-pGCL compiler works as follows. Given a quantum program, it checks that all operations on quantum variables are reversible. The ones that are not (e.g. classical operations applied to quantum variables) are first converted into reversible operations using the defined rules. The entire set of the reversible operations can now safely be passed onto the quantum subsystem.

### 4.4 Quantum Flowcharts

Selinger [8] presents a functional quantum programming language (QPL) that is statically typed. He defines the semantics of QPL using "quantum flowcharts" and defines the semantics of quantum flowcharts entirely in terms of operations on density matrices (see Appendix 7.1.4) of quantum systems. Selinger assumes the QRAM model of quantum hardware.

To give the reader a flavor of quantum flowcharts and their associated semantics, we first present an example of a classical flowchart and its semantics. Suppose you have a 2-bit boolean function : 00 → 00, 01 → 01, 10 → 00, 11 → 10. Suppose that the probability of each possible input (00,01,10,11) is (A,B,C,D) respectively, i.e. A = probability of seeing 00 on the input, B = probability of seeing 01 on the input, etc. In this case, the probability of each possible output (00,01,10,11) is equal to (A+C, B, D, 0), e.g. the output will be 00 if the input is 00 with probability A or if the input is 10 with probability C. Similarly, the output 11 can never happen, i.e. the probability of 11 on the output is zero. Selinger formalizes this reasoning by defining rules for classical flowcharts in terms mappings from input probabilities to output probabilities. He defines seven atomic flowchart fragments and defines the rules for each fragment. As an example, the boolean function above is described by a classical flowchart following Selinger’s rules in Figure 1. It incorporates three of the seven atomic flowchart fragments, namely:
Figure 1: Classical flowchart implementing boolean function \((00,01,10,11) \rightarrow (00,01,00,10)\). Maps input probability distribution \((A,B,C,D)\) to output probability distribution \((A+C,B,D,0)\). Each edge is annotated with a probability tuple denoting the probability of reaching that edge.

branch (once), assignment (twice), and merge (once). Note that each edge in the flowchart is annotated with a tuple of probabilities which describe the probability of reaching this edge.

Selinger now extends this to quantum flowcharts, again defining rules for seven atomic flowchart fragments, including the quantum operations of unitary transformation and measurement. An example of such a quantum flowchart (one measurement, two unitary transformations, and a merge) is shown in Figure 2. In this case, each edge is again annotated with a probability of reaching this edge; however in the quantum case, a probability distribution of a quantum state can be directly specified by the density matrix (see Appendix 7.1.4) associated with this state. Therefore, Selinger uses density matrices to annotate the edges in his quantum flowcharts. The semantics of the program in Figure 2 (and all other quantum programs) is given by a function on density matrices, i.e. a mapping from input probabilities to output probabilities.

By representing the syntax and semantics of quantum programs by quantum flowcharts, Selinger implicitly addresses one of the four issues we mentioned earlier regarding quantum programming languages, namely the no-cloning property. Note that from section 3.2, a cloning operation \(U(|a0\rangle) = |aa\rangle\) is a non-linear operation on \(a\). But Selinger’s flowchart language is inherently linear, i.e. a larger flowchart is composed of a linear combination of smaller flowcharts, and each edge in a flowchart computes a linear function of its inputs. Thus no valid flowchart can have an edge that is annotated by a non-linear function of its inputs. So Selinger’s QPL (based on the syntax of quantum flowcharts) can check for violations of the no-cloning rule statically at compile time, unlike qGCL by Sanders et al [5].

Using this basic notion of quantum flowcharts, Selinger goes on to develop a complete semantics for his language QPL, including an operational semantics for loops, recursion, and procedures. He also introduces a type system and typing rules for QPL, and claims that a well-typed program in QPL can never produce a run-time error.
Figure 2: Quantum flowchart implementing a function of two qubits a and b. The qubit a is measured. If a is 0, the unitary operation X (inverter) is applied to qubit b. If a is 1, X is applied to qubit a (which has now the classical value of 1 since it was measured.). The final density matrix T is shown after the merge operation.

4.5 Quantum Lambda Calculus

Andre van Tonder takes a different approach and puts forward a quantum version of the classical untyped lambda calculus [10] that does not assume QRAM model computation, i.e. classical control and quantum data. Van Tonder develops a quantum lambda calculus to efficiently reason about quantum Turing machines, where both the control and data are quantum.

First van Tonder adds constant terms, c, to the set of valid expressions in the classical lambda calculus:

\[
e ::= x \mid (\lambda x.e) \mid (e e) \mid c
\]

\[
c ::= 0 \mid 1 \mid H \mid X \mid I \mid \text{cnot} \mid \text{etc}....
\]

where c can be a classical bit value or one of the unitary quantum transformations (Hadamard H, inverter X, etc.).

The second issue that van Tonder addresses is issue of reversibility. In the classical lambda calculus a beta reduction consumes the expression and discards information, making it non-reversible. Van Tonder extends the classical lambda calculus to contain reversible beta reductions by adding the concepts of a "history track" h, which holds the sequence of beta reductions performed, and a "computational register" t which holds the current value of the computation. He defines new beta reduction rules that employ a placeholder symbol "_" to indicate which terms do not need to be recorded in the course of a reduction; however, the term on which a reduction is performed is recorded in the history track. Instead of listing all of the rules, it will be helpful to give the reader their basic idea by way of sample reduction which illustrates the concepts of a history track h and a computational register t:

\[
(((\lambda f. (\lambda x. (f \ x))) (\lambda z. z)) w)
\]
→ (((λf. (_· ( f _ ))) _) _); ((λx. (λz.z) x)) w)
We reduced f, and the result is in computational register t = ((λx. (λz.z) x)) w. History track h = (((λf. (_· ( f _ ))) _) _).

→ (((λf. (_· ( f _ ))) _) _); ((λx. ( _ x )) _) ; ((λz.z) w)
We reduced x. Now h = (((λf. (_· ( f _ ))) _) _); ((λx. ( _ x )) _), and t = ((λz.z) w).

→ (((λf. (_· ( f _ ))) _) _); ((λx. ( _ x )) _); ((λz.z) _); w
We reduced z. Now h = (((λf. (_· ( f _ ))) _) _); ((λx. ( _ x )) _); ((λz.z) _), and t = w.

Thus it is clear to see how the reduction can be reconstructed in reverse from the history track.

Van Tonder then develops reduction rules for quantum operations using the ket notation and tensor products (see Appendix 7.1.3). The general beta reduction rule for any unitary quantum operation U (specified by constant cU), applied to a quantum primitive q (0 or 1 in the single bit case), is given by:

\[ | h; (cU \ q) \rangle \rightarrow | h; (cU \ _) \rangle \otimes U|q\]

For example, for the Hadamard operation H has the following reduction rules:

\[ | h; (cH \ 0) \rangle \rightarrow | h; (cH \ _) \rangle \otimes H|0\rangle = | h; (cH \ _) \rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \]
\[ | h; (cH \ 1) \rangle \rightarrow | h; (cH \ _) \rangle \otimes H|1\rangle = | h; (cH \ _) \rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \]

But then van Tonder runs into a slight problem with his reduction rules. The problem can be seen by looking the following (correct) sample reduction:

\[ | (λx.0) \ (H \ 0) \rangle \rightarrow \left| _· (H \ _) \right\rangle \otimes \frac{1}{\sqrt{2}}[ | (λx.0) \ 0 \rangle + | (λx.0) \ 1 \rangle ] \]
\[ \rightarrow \left| _· (H \ _) \right\rangle \otimes \frac{1}{\sqrt{2}}[ | (λx._) \ 0 \rangle + | (λx._) \ 1 \rangle ] \otimes |0\rangle \]

The final value of the computation register is the classical value |0\rangle as expected from the definition of (λx.0). Note that the value of the computation register after the first reduction is \( \frac{1}{\sqrt{2}}[ | (λx.0) \ 0 \rangle + | (λx.0) \ 1 \rangle ] \), which is a quantum superposition. Suppose instead we were to simply reduce the computation register after the first reduction step to just \( \frac{1}{\sqrt{2}}(2|0\rangle) = \sqrt{2}|0\rangle \), in effect ignoring the quantum superposition aspect. This would be invalid since \( \sqrt{2}|0\rangle \) is not a valid quantum state.

Van Tonder rectifies this problem in two steps. First, he adjusts the quantum lambda calculus to not allow functions to discard arguments that represent a quantum superposition. He presents a calculus that keeps track of which values are purely classical (can be discarded) and which ones are in a quantum superposition (cannot be discarded). It turns out that calculi that keep track of different types of resources already exist and are called linear lambda calculi, so van Tonder appeals to an untyped linear lambda calculus as a solution to this problem. He extends the terms of his calculus to include linear and non-linear terms:
\[ e ::= x \mid (\lambda x.e) \mid (e \, e) \mid e \mid !e \mid (\lambda!x.e) \]

where the last two terms are a non-linear term and a non-linear abstraction. The reason for the name linear is that a linear term can only appear in the function body exactly once. Non-linear terms may appear any number of times in a function body, and may be discarded and duplicated at will. Therefore in the quantum linear lambda calculus, linear terms represent quantum variables and non-linear terms represent classical variables. Van Tonder lists the rules for well-formedness of expressions in the calculus, i.e. what constitutes a legal term in the linear lambda calculus, and claims that this property is something that can be checked statically. For example, the expression \((\lambda x. \, (x \, x))\) is not well formed since the linear argument \(x\) appears twice in the function body.

In the second step, van Tonder modifies the reduction rules of the calculus to prevent a reduction inside a non-linear term \(e\). He does this by adding \(!e\) to the list of "values" in the calculus, and defines an evaluation context:

\[ C, C_i ::= \sigma \mid (e \, C) \mid (C \, e) \mid \lambda x.C \mid \lambda!x.C \mid \sum c_i C_i \]

where the last term signifies a superposition of contexts. Note that there is no evaluation context of the form \(!(! \ldots \sigma \ldots)\), thus disallowing evaluation of a non-linear term. The reduction can only now be the correct one:

\[ | (\lambda!x.0) \, !((H \, 0)) \rightarrow |0\rangle, \] since we do not evaluate under the \(!\) and can discard the non-linear expression \(!((H \, 0))\).

5 Quantum Simulators

In contrast to the above formal methods, we now review some of the well known attempts at simulation quantum algorithms on classical computers, using both functional and imperative languages.

5.1 Q-gol

One of the earliest attempts at modeling quantum computation was done by Baker with his language Q-gol [2]. Baker mainly wanted to create a prototype of a quantum programming language as a proof of concept, with the goal of discovering some of the potential issues implicit in quantum programming.

Baker began with an implementation using an imperative, object-oriented language. He had a set of Operation (unitary operation) objects and a set of Universe objects (quantum state), and the Operation objects would act on the Universe objects. However, he ran into a few problems with this approach. First, he could not guarantee that all operations defined by Operation objects would be unitary. Second, he did not see an practical and easy way of composing independent operations. Last, the object-oriented paradigm of message passing (remote side effects) between objects did not have any physical meaning in quantum mechanics, and could possibly lead to programs that violated the laws of quantum mechanics. Furthermore, in a typical OO implementation, a given system is composed of sub-systems (objects) interacting in well defined way; however, as we mentioned previously, this is not always possible in a quantum system due to entanglement.

Baker scrapped his initial OO approach in favor of an implementation using a functional language, Concurrent Clean. Being a "true" functional language, Concurrent Clean disallows assignment statements, thus solving Baker's problem with remote side effects. Furthermore, Concurrent Clean has
referential transparency, meaning that a function does not depend on some global state which can change with each new function call (thus changing the function). Imperative languages do not have referential transparency. Referential transparency solved Baker’s problem regarding unitary operations remaining unitary. Furthermore, Concurrent Clean has a concept of "type attributes". An object can have a type, and a type can have one of two attributes: strict or unique. The strict attribute forces the compiler to evaluate the object as soon as possible (the opposite of lazy evaluation). The unique attribute guarantees that an object can appear exactly once in the right hand side of an expression. A unique type attribute is very similar to a linear term in the linear lambda calculus discussed in section 4.5. The unique type attribute enforces the no-cloning property of quantum mechanics at compile time.

Due to the above features in Concurrent Clean, and after implementing some basic types (abstract type Quantum) and functions (prepare, observe, entangle), Baker concludes that the functional programming model is much better suited to quantum programming than the OO imperative model.

5.2 Quantum Programming Using Haskell

5.2.1 Monads to the Rescue

Mu and Bird propose a monadic style for writing quantum programs in Haskell, a functional programming language [7]. Haskell provides two concise constructs for modeling the non-deterministic nature of quantum computing: list comprehension and the list monad. (There are also other monads, e.g. the state monad which is useful for implementing the concept of mutable state into a functional language like Haskell. For more information on monads, the reader is encouraged to refer to the Haskell tutorial on www.haskell.org).

List comprehension is basically a concise method of writing nested for-loops. For example, the program (pseudocode):

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n \\
\text{for } j &= 1 \text{ to } m \\
\text{for } k &= 1 \text{ to } p \\
& \quad f \ [i,j,k] = a[i] \ast b[j] \ast c[k];
\end{align*}
\]

can be written in Haskell as one line: \[ [i \ast j \ast k \mid i \leftarrow a, j \leftarrow b, k \leftarrow c] \]. The function \( f \) (in this case multiplication) is computed for all possible combinations of the inputs in the three lists \( a, b, \) and \( c, \) of lengths \( n, m, \) and \( p, \) respectively, resulting in a list of length \( n \ast m \ast p. \)

The list monad in Haskell is a construct that allows one to define an operator \( G \) that takes in a list of values \([v]\) and another function \( f : v \rightarrow [f(v)] \) that given a single value \( v \) returns a list \([f(v)]\). The operator \( G \) then returns a list that is the concatenation of all of the possible lists produced by function \( f \) when applied to the list of values \([v]\). Specifically, \( G :: [v] \rightarrow (v \rightarrow [f(v)]) \rightarrow [c], \) where \([c]\) can be thought of as \([f(v_1)], [f(v_2)], \ldots \). The notation for such an operator \( G \) is \( \gg\gg \). In Haskell, the 'do' syntax is a convenient way to specify list monads. It gives one the appearance that one is working with a single value when in reality the function is computed for all possible combinations of the inputs \([7]\), thus providing a nice way to deal with non-determinism. Following an example in the Haskell tutorial available on www.haskell.org:

\[
\begin{align*}
\text{myfun} & \quad :: \ (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c] \\
\text{myfun } f \ x \ y & \quad = \ \text{do } \ i \leftarrow x
\end{align*}
\]
\[
j <- y \\
\text{return } (f \ i \ j)
\]

The function myfun converts a regular function \( f \) of two variables into a function of lists, returning a list of \( f \) applied to all possible combinations of inputs. The do syntax can be thought of a two nested for-loops, i.e. for each value \( i \) of \( x \), for each value \( j \) of \( y \), place \((f \ i \ j)\) into a new list. For example:

\[
\text{myfun (+) [1, 2, 4] [10, 20, 30] = [11, 21, 31, 12, 22, 32, 14, 24, 34]}
\]

Bird and Mu immediately apply the above two concepts to quantum computing. List comprehension is used to define the tensor product as follows, where \( \text{Qureg} \) is a quantum register type, the function \( f \) is \( \times \) (the standard scalar multiplication), and quantum variables \( u \) and \( v \) are represented as lists of the form \([a, b]\) where \( a \) and \( b \) are the associated probability amplitudes in \( a|0\rangle + b|1\rangle \):

\[
(\otimes) \quad :: \quad \text{Qureg} \rightarrow \text{Qureg} \rightarrow \text{Qureg} \\
(\otimes) \ u \ v \quad = \quad [i \times j \mid i <- u, j <- v]
\]

Now we can write the mathematical equation \( u \otimes v = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \) in the language as a list comprehension: \((\otimes) \ [a, b] \ [c, d] = [ab, ac, bc, bd]\).

To show how list monads fit in, Mu and Bird define an "apply" function for any quantum operator \( f \) and a quantum register variable \( q \), such that "apply \( f \) \( q \)" returns a quantum register variable whose contents are the result of applying \( f \) to \( q \) \([7]\). Therefore, the type of the apply operator is:

\[
\text{apply :: } (\text{Qureg} \rightarrow \text{Qureg}) \rightarrow \text{Qureg} \rightarrow \text{Qureg}
\]

However, since \( f \) is a linear operator (see Appendix 7.1.3), \( f(q) \) can be written as the superposition of applying \( f \) to the basis vector components of \( q \), which represent classical values (Integers), i.e. \( f(a|0\rangle + b|1\rangle) = af(|0\rangle) + bf(|1\rangle). \) Thus, the type of "apply" can be written as:

\[
\text{apply :: } (\text{Int} \rightarrow \text{Qureg}) \rightarrow \text{Qureg} \rightarrow \text{Qureg}
\]

or equivalently,

\[
\text{apply :: } \text{Qureg} \rightarrow (\text{Int} \rightarrow \text{Qureg}) \rightarrow \text{Qureg}
\]

The type definition of "apply" then exactly mimics the type definition of our operator \( G \), \( >>= \), for list monads. Therefore, Mu and Bird claim that other quantum algorithms can be easily written using this monadic style of programming.

As an example, Mu and Bird present the Deutsch-Josza algorithm in monadic form. The Deutsch-Josza algorithm \([16]\) determines whether an unknown function \( f \) is constant (\( f(a) = c \) for all \( a \)), or if \( f \) is balanced (\( f(a) = 0 \) for half of the input \( a \), and \( f(a) = 1 \) for the other half of the input \( a \)). Note that with a classical computer, you may need as many as \( 2^n - 1 + 1 \) runs of the function to determine which kind of function \( f \) is, where \( n \) is the size in bits of each input \( a \). Given a quantum computer, this can be done in one quantum computation as follows:

\[
dj f n \ = \ \text{finalize } \left( \text{do } x <- \text{hadamard } n \\
y <- \text{trans } n \ f \ x \\
\text{hadamard } n \ y \right)
\]

An \( n \)-qubit quantum register is initialized by a Hadamard transformation of the classical value 0, putting the \( n \)-qubit register into a superposition of all possible states with equal probability amplitudes (see Appendix 7.1.2), e.g. hadamard 2 0 = hadamard([1,0,0,0]) => x = [1/2, 1/2, 1/2, 1/2]. Then for every \( x \)
in the resulting list, the value y is computed by the function \(trans\ n\ f\). \(Trans\ n\ f\) is a quantum operator that flips the sign of the probability amplitude at location \(k\) of an \(n\)-qubit register if \(f(k)\) returns an odd value. Then every such value y is transformed again by the \(n\)-bit hadamard operation. The \textit{finalize} operation is a measurement that produces a classical value.

Suppose \(f\) is constant, either always 0 or always 1. Then \(y\) will either be \([\frac{1}{2}, \frac{1}{2}, \frac{1}{2}]\) or \([-\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}]\), respectively. Using the 2-qubit Hadamard operation shown in Appendix 7.1.3, the reader can verify that 
\[
H([\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]) = [1, 0, 0, 0] \quad \text{and} \quad H([-\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}]) = [-1, 0, 0, 0].
\]
In either case, when \(y\) is measured, it will read 0 with probability 1 (always). If \(f\) is balanced, then \(y\) will contain an equal number of \(+\frac{1}{2}\) and \(-\frac{1}{2}\). In this case, the reader can again verify that \(H(y)\) will always be of the form \([0, a, b, c]\). In other words, if \(f\) is balanced, the probability of reading a classical value of 0 when \(y\) measured is 0 (never). Thus, if \(dj\) returns 0, \(f\) is constant; otherwise, it is balanced.

Note that although the above computation can be done in one step on a quantum computer, it still requires \(O(2^n)\) steps on a classical computer since in reality the 'do' syntax expands to all possible values of \(x\) and \(y\).

### 5.2.2 Dealing with Measurement and Entanglement in Haskell

Sabry [9] presents a complete implementation of a quantum simulator written in Haskell. One interesting facet of his approach is that Sabry focuses primarily on issues concerning measurement and entangled values.

As noted above, measuring a single qubit collapses the quantum superposition of the qubit into a classical value. From that point on, all future measurements of this qubit will yield the same classical value. Furthermore, if this qubit is entangled with other qubits, measuring it collapses the quantum superpositions of its entangled partners, and all future measurements of those partner qubits will also be fixed. Thus measurement is a non-reversible operation with a potential side-effect of changing the values of other non-local quantum variables.

To model the side-effects inherent in the measurement of entangled quantum variables, Sabry uses a simple scheme of a global side-effect to a shared reference which references a shared data structure [9]. An assignment to the shared reference immediately affects and updates all of the entangled quantum variables modeled by this shared data structure. Therefore, Sabry makes sure that all quantum variables can only be accessed via such global references cells. An observation of a quantum value then proceeds as follows: the contents of the reference cell are read, the value is observed, and the reference cell is updated with the results of the observation. The act of observation involves probability, and Sabry implements a way of making a random choice proportional to the probability amplitudes of the qubit being measured.

A quantum algorithm is composed of a set of quantum operations acting on a data structure that describes the entire quantum state of all qubits in the system. However, when actually writing down (coding) an algorithm as a sequence of operations, one would like to have access to different groups of qubits at different stages of the algorithm. But due to entanglement, this becomes very unwieldy. For example, imagine you are coding a 6-qubit algorithm, and somewhere in the middle of the sequence of operations you need to access to qubits \(b_2\) and \(b_4\). The problem is that at this stage of the computation the six qubits are entangled as \(((b_1, b_2, b_3), (b_4, b_5), (b_2, b_4))\), where parentheses indicate entanglement. How do you isolate \(b_2\) and \(b_4\)? Sabry refers to this as the "wave-particle duality" [9] in quantum programming languages: each qubit acts as an independent particle in the computation, but yet is entangled in the entire computational wave of the algorithm.

Sabry introduces the concept of \textit{virtual values} as a means of accessing individual values that are part of...
an entangled data structure. One can think of virtual values as providing a set of pointers that reference internal values deep in a complex data structure, which itself has reference pointer and is typically used to model a state. Virtual values are declared by specifying the reference to the entire data structure and by a mapping, an adaptor, which maps from the larger structure to the subset of values (decomposition), and from the subset of values to the larger structure (composition). In a way, virtual values act as higher level interfaces (as in object oriented programming) to the larger data structure, and thus the notion of virtual values is closely related to the notion of sub-typing (interface inheritance) in OO languages like Java.

For example, one type of virtual value for \((b_2, b_4)\) can be declared as follows:

\[
\text{Virtual}_1v :: (\text{REF}[\langle b_1, b_2, b_3 \rangle, \langle b_4, b_5 \rangle, \langle b_2, b_4 \rangle], \text{adaptorA})
\]

\[
\text{adaptorA} =
\]

\[
\text{decomposition} ::= \langle \langle b_1, b_2, b_3 \rangle, \langle b_4, b_5 \rangle, \langle b_2, b_4 \rangle \rangle \rightarrow \langle \langle b_2 \text{ and } b_4 \rangle, \langle b_1, b_3, b_5, b_6 \rangle \rangle
\]

\[
\text{composition} ::= \langle \langle b_2 \text{ and } b_4 \rangle, \langle b_1, b_3, b_5, b_6 \rangle \rangle \rightarrow \langle \langle b_1, b_2, b_3 \rangle, \langle b_4, b_5 \rangle, \langle b_2, b_6 \rangle \rangle
\]

Note that there could be different types of virtual values defined for the pair \((b_2, b_4)\) since there are different ways of grouping (entangling) \(b_2\) and \(b_4\) with the rest of the values in the entire data structure. Each virtual value type is operationally defined by its adaptor, and the mappings defined in the adaptor specify how all the values are entangled.

Sabry claims that the notion of virtual values allows one to pretend that there is value of type \((b_2, b_4)\) in the program, even though a variable of the type \((b_2, b_4)\) does not explicitly exist, and \(b_2\) and \(b_4\) are embedded inside a larger data structure. This fact can be appreciated if one thinks of virtual values as being analogous to interfaces in an object oriented programming like Java. An interface of type \(T\) does not represent a concrete class, but nevertheless references of type \(T\) can be manipulated just like objects of concrete types, e.g. formal parameters in methods, in method bodies, etc., the only limitation being that one cannot instantiate an object of an interface type. Similar to how a Java method can be written to act on polymorphic variables of an interface type, Sabry’s quantum algorithms can be written to act on virtual values, thus simplifying the code. The binding of the polymorphic variables to their actual types happens at run-time, and so does the binding of the virtual values. Moreover, just like one cannot instantiate an object of an interface type in Java, one cannot instantiate (isolate) a virtual value from its entangled state. Nevertheless, one can define operations using interfaces and virtual values.

This central property of virtual values allows Sabry to efficiently deal with entanglement and succinctly define quantum algorithms. If an operation needs a value of a particular virtual type from the larger type, it extracts the subtype at run-time using the decomposition mapping in the adaptor; if the operation then needs to recover then "larger" type, it uses the composition mapping in the adaptor.

5.3 Quantum Programming as Extensions of Imperative Languages

The last two attempts surveyed were the works of Omer [4] and Bettelli et al [11]. Both of them take the approach that quantum programming should be an extension of popular (classical) structured and imperative languages. Omer defines a new structured language called QCL, and develops an interpreter for it in C++. Bettelli et al take a similar approach and develop a quantum library for C++.

16
5.3.1 QCL

Omer’s QCL language is an attempt to map the elements found in structured and procedural programming (e.g. hierarchical structure, flow of control, global and local variables, subroutines, dynamic memory, conditionals, loops) to quantum computing. Although QCL does not come with a formal calculus nor an operational semantics, it is still quite powerful for simulating current and future quantum algorithms. A QCL program is compiled on a classical computer into a set of instructions for the quantum subsystem (QRAM model [1]). QCL is a complete language in that it is able to simulate all of the known quantum algorithms we have today. It contains the usual data types, qubit and qureg, but also contains an additional quantum data types, one of them being quscratch. Quscratch is a type of temporary "scratch register" specifically designed to hold the intermediate results during a reversible computation. Qureg type variables are constrained by some of the rules introduced by [1], e.g. cannot do an arbitrary assignment to a variable of type qureg.

QCL also defines four different types of subroutines in a calling hierarchy:

- **procedure** - non-reversible classical operation with side effects
- **operator** - reversible unitary operation with no-side effects and independent of program state
- **qufunct** - reversible quantum function (e.g. permutation of qubits)
- **function** - non-reversible mathematical functions but with no-side effects

The additional constraint is that a subroutine type can only call subroutine types of the same or lower level than itself in the hierarchy, thus preventing unitary operations from calling non-reversible classical procedures. Each defined operator also has a predefined inverse operator, prefixed by a "!". All rules for quantum variables and quantum operations are checked dynamically at run-time.

Omer also implements quantum conditionals as operations controlled by an enable qubit (e.g. controlled-not or cnot, see Appendix 7.1.2). For example, the statement `if e { not(x) }` can be written as `cnot(x,e)`. QCL can automatically derive the conditional form of an operator simply be prefacings its declaration with **cond**. If a conditional statement changes the state of the program (e.g. updates variables), then the QCL simulator forks and evaluates all possible paths through the computation in a serial sequence.

5.3.2 Quantum Operators as Objects

Bettelli et al [11] follow a similar path in implementing a C++ library for simulating quantum algorithms. However, as opposed to Omer, Bettelli treats quantum operators as actual objects instead of functions, objects which can be composed, manipulated, and potentially simplified during run-time, before being applied to quantum subsystem. A composition of two operator objects `Cop = Aop & Bop` is an object that performs the operations Aop and Bop in the order specified in the composition. Moreover, inside the definition of the concatenation primitive (&) for each operator object, Bettelli defines how this operator simplifies when concatenated with another operator. For example, the Hadamard operation applied twice to a qubit leaves the qubit in the same state (see Appendix 7.1.2). Therefore, the concatenation operator defined as `Hcomp = H & H` is really just a no-op. Thus by treating operators as objects and defining simplification routines inside them, Bettelli’s Q language can perform these types of optimizations, thus saving quantum resources at run-time. This is the main advantage of Bettelli’s approach.

There are also a plethora of other quantum simulators. We refer the reader to
http://www.cs.kent.ac.uk/people/staff/jw71/qc/simtable.htm for a complete tabular listing, including links.

6 Conclusion

Practical quantum computer hardware is not here yet. In spite of more than twenty years of research since Richard Feynman declared that a true quantum mechanical system could not be simulated on classical computer, the largest quantum "computers" in research labs today are seven quantum bit systems that compute highly specialized functions. Even though quantum hardware is a bit (or qubit, if you prefer) behind the ball, it is comforting to know that when it arrives, many of the issues relating to programming this hardware will already have been dealt with. However, despite the amount of effort already put into defining quantum programming, there is no standard and accepted model of quantum programming languages (unlike the widely accepted QRAM model for quantum hardware). Some argue for a model based on functional languages with static type systems, since this allows for checking of run-time errors at compile time (e.g. violations of the no-cloning rule). Others argue that we should base quantum programming languages on current popular imperative languages, thus allowing quantum programming to take immediate advantages of enhancements and updates to these languages. It will take the production of an actual and practical quantum computer to settle this debate and define the best model. Furthermore, many other challenges remain, specifically in the areas of concurrent quantum programs, quantum cryptography, and quantum error correction, which is necessary since presumably future quantum hardware will not be ideal.

7 Appendix

7.1 Mathematical Foundations

7.1.1 Vector Spaces

Mathematically, quantum states in a quantum system are specified using vectors in a complex vector space, in which the basis vectors correspond to the classical states of the system. For example, as noted above, the two classical states for a single qubit system are "0" and "1"; therefore "0" and "1" correspond to the basis vectors \( |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \) and \( |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \) in a 2-dimensional complex vector space. A quantum state of the 1-qubit system is then any linear combination of the above basis vectors: \( a|0\rangle + b|1\rangle \), where \( a \) and \( b \) are complex numbers such that \( |a|^2 + |b|^2 = 1 \). The \( |x\rangle \) is called the 'ket', and its complex conjugate \( \langle x| \) is called the 'bra', in 'bra/ket' notation due to Paul Dirac. Thus, the classical state "false" or "0" is written as \( 1|0\rangle + 0|1\rangle \), and the classical state "true" or "1" is written as \( 0|0\rangle + 1|1\rangle \). The complex coefficients \( a \) and \( b \) signify the probability amplitude associated with each basis state, and they represent the probability of measuring the qubit as being in that basis state. Measurement means the act of reading the value of a qubit. Prior to being measured, a qubit can exist in a superposition \( a|0\rangle + b|1\rangle \), but once it is measured (and thereafter), its quantum state "collapses" into one of the two possible classical states: \( |0\rangle \) with probability \( |a|^2 \) or \( |1\rangle \) with probability \( |b|^2 \).

A 2-qubit system has four basis vectors, corresponding to the four possible classical states of 2-bit classical system:
\[
|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},
\]
and each quantum state is a linear combination of the four basis vectors: \[a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle,\]
where again \(a, b, c, d\) are complex numbers such that \(|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1.\)

### 7.1.2 Matrix Operations

Operations on quantum states (vectors) are specified by matrices acting on these vectors. An operation can be thought of as a quantum logic gate, much like a classical digital logic gate. For example, for a 1-qubit system, the not gate inverts the state of the qubit:

\[
|0\rangle \rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{and} \quad |1\rangle \rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}.
\]

The not gate (inverter) is described by the matrix operation \(X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\), whose columns are the transformations of the basis vectors (right hand sides of above). Similarly, the identity operation (i.e. just a wire) is defined by the matrix \(I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\).

A very important operation is called the Hadamard transformation, \(H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\). The effect of a Hadamard operation is to transform a \(n\)-qubit system from an initial state of \(|00...0\rangle\) to a state which is a linear superposition of all \(2^n\) possible states, where each state has an equal probability amplitude. Therefore the Hadamard operation plays a crucial role in quantum computing since its purpose is to initialize the quantum subsystem by transforming the n-qubit register from a classical state of all 0's to a quantum superposition of all possible states. Applying the Hadamard operation again returns the register to its original state, i.e. \(HH = I\). For example, a 1-qubit system initially in the state of \(|0\rangle = 1|0\rangle + 0|1\rangle\) is transformed into a superposition of \(|0\rangle\) and \(|1\rangle\):

\[
H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).
\]

For a 2-qubit system with four basis vectors, 2-qubit operations (gates) are described using \(4 \times 4\) matrices. An important 2-qubit operation is the controlled-not gate, or cnot. The cnot gate inverts the second qubit of a pair if the first qubit is 1, otherwise it does nothing. Specifically: \(|00\rangle \rightarrow |00\rangle, \quad |01\rangle \rightarrow |01\rangle, \quad |10\rangle \rightarrow |11\rangle, \quad |11\rangle \rightarrow |10\rangle\). The cnot operation is defined by the following matrix:

\[
C_{\text{not}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = [I \\ 0 \\ 0 \\ X],
\]

where \(I\) and \(X\) are the \(2 \times 2\) identity and not gate matrices for 1-qubit systems as defined above. Again, the columns of the matrix of the \(C_{\text{not}}\) operation are defined
by the what the operation does on the four basis vectors. Using the full vector value of $|10\rangle$ as shown above, the reader can check that $C_{\text{hot}}(|10\rangle) = |11\rangle$ using straight matrix multiplication. The matrices for other operations are constructed in a similar fashion. The most advanced quantum computers today are in reality quantum circuits composed of simple quantum gates, implementing very specialized functions.

### 7.1.3 Tensor Product

The state space of a n-qubit system is defined by the tensor product of the state spaces of the n 1-qubit systems. For example, the quantum state $w$ of a 2-qubit register is specified by the tensor product of the two single qubits $u$ and $v$:

$$w = (u \otimes v) = (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) = (ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle).$$

As a specific example, consider $w = (|0\rangle \otimes |0\rangle) = (1|0\rangle + 0|1\rangle) \otimes (|0\rangle + 0|1\rangle) = (|00\rangle + 0|01\rangle + 0|10\rangle + 0|11\rangle) = |00\rangle$.

Note that while $u$ and $v$ were both 2-dimensional vectors in a 2-dimensional vector space, $w$ is a 4-dimensional vector in a 4-dimensional vector space. In general, if $u$ is a n-dimensional vector and $v$ is a m-dimensional vector, $(u \otimes v)$ will be an nm-dimensional vector in an nm-dimensional vector space (a vector space with $n \times m$ basis vectors). Thus, the dimension (the number of basis vectors) of $W = \text{dimension of } U \times \text{dimension of } V$.

In contrast, in a classical system, the state space of a collection of particles is defined by the Cartesian product of the individual vector spaces. For example, given two vector spaces $U$ and $V$ with basis vectors $(u_1, u_2)$ and $(v_1, v_2)$, the basis of the combined vector space $W$ will be the union ("sum") of the sets of basis vectors: $(u_1, u_2, v_1, v_2)$. Thus, the dimension (the number of basis vectors) of $W = \text{dimension of } U + \text{dimension of } V$.

It is this exponential growth of the state space (product versus sum) in a quantum system that results in the exponential growth of its computational capacity as compared to classical system [12].

Some useful properties of the tensor product relevant to quantum computing are as follows, where $A, B, C, D, U$ are matrices representing quantum operations, and $u$ and $v$ are vectors representing quantum states:

1. $(A \otimes B)(u \otimes v) = (Au) \otimes (Bv)$, which means that the operation defined by the matrix $(A \otimes B)$ on the combined state $(u \otimes v)$ of qubits is equal to the combined state (tensor product) of $A$ acting on $u$ and $B$ acting on $v$. Therefore, if an operation is needed than operates only on the first qubit $u$, its matrix will be $(A \otimes I)$, since $(A \otimes I)(u \otimes v) = (Au) \otimes (Iv) = (Au) \otimes v$; similarly, an operation acting only the second qubit will have matrix of $(I \otimes B)$ since $(I \otimes B)(u \otimes v) = (Iu) \otimes (Bv) = u \otimes (Bv)$.

As another example, to apply a binary quantum gate $A \otimes B$ to the second and fourth bits (only) of a 5-qubit quantum system, we transform the system by the matrix $I \otimes A \otimes I \otimes B \otimes I$.

Furthermore, we can now see how to construct a Hadamard transformation to initialize a n-qubit subsystem, namely using n tensor products: $(H \otimes H \otimes H, ..., \otimes H)$. For example, for a 2-qubit system:

$$H \otimes H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} H & H \\ H & -H \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$
Note that each $2 \times 2$ rectangular matrix in $H \otimes H$ is the result of multiplying the first 1-qubit matrix $H$ by the corresponding entry in the second 1-qubit matrix $H$. This is an example of the general rule of the tensor product of two matrices.

2. $A \otimes (u + v) = A \otimes u + A \otimes v$, meaning that tensor product is linear.

The above two tensor product rules allow us to deal with entangled pairs of qubits as follows. Suppose we have the entangled state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ and we wish to apply the 2-qubit operation $(A \otimes B)$ to it. First we distribute the operation into a sum using Rule 2, then we write each 2-qubit state as the tensor product of its 1-qubit elements, $|00\rangle = |0\rangle \otimes |0\rangle$ and $|11\rangle = |1\rangle \otimes |1\rangle$, and finally we apply Rule 1:

$$(A \otimes B)(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle) = (A \otimes B)(\frac{1}{\sqrt{2}}|00\rangle) + (A \otimes B)(\frac{1}{\sqrt{2}}|11\rangle)$$

$$= \frac{1}{\sqrt{2}}(A \otimes B)(|0\rangle \otimes |0\rangle) + \frac{1}{\sqrt{2}}(A \otimes B)(|1\rangle \otimes |1\rangle)$$

$$= \frac{1}{\sqrt{2}}(A \otimes |0\rangle) \otimes (B \otimes |0\rangle) + \frac{1}{\sqrt{2}}(A \otimes |1\rangle) \otimes (B \otimes |1\rangle)$$

Thus, we see how we can apply operations $A$ and $B$ to the individual qubits in the entangled pair.

### 7.1.4 Density Matrices

The last important concepts are that of **mixed states** and **density matrices**. A pure quantum state is the underlying unknown quantum state expressed by $a|0\rangle + b|1\rangle$, where $a$ and $b$ are complex numbers. Suppose we have performed a physical measurement of the system but have not yet observed the measured value. Thus from our perspective, the value is either $|0\rangle$ with probability $|a|^2$, or $|1\rangle$ with probability $|b|^2$. This situation is known as a **mixed state**, and can be written as $|a|^2(|0\rangle + b|^1\rangle)$, or in density matrix notation $D = |a|^2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + |b|^2 \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} |a|^2 & 0 \\ 0 & |b|^2 \end{bmatrix}$. Note that the probabilities associated with the measurement can be read directly off the diagonal. Mixed states and density matrices do not have a physical significance - they are merely a convenient way of describing probabilities associated with a measurement. Density matrices completely define the **probability distribution** of measuring a particular value in a collection of qubits. In general, performing a measurement (without observation) on a mixed state described by some density matrix $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$, where $A, B, C, D$ are matrices, yields the mixed state with density matrix $\begin{bmatrix} A & 0 \\ 0 & D \end{bmatrix}$. Now actually observing the measurement yields two density matrices

$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 0 \\ 0 & D \end{bmatrix}$, each with probability equal to the sum of the elements on its diagonal.

Furthermore, in addition to performing unitary matrix operations on vectors representing pure states as we did in section 7.1.2, we can also perform the same unitary operations on the corresponding density matrices. This immediately yields a probability distribution for measurements, and measured values (results) are ultimately what we care about. Suppose a quantum system with an initial density matrix $D$ undergoes a operation described by the unitary matrix $U$. The final density matrix of the system will be given by $D' = UDU^*$, where $U^*$ is the conjugate transpose of $U$. Thus we can see that operations on density matrices take a probability distribution as an input (input = density matrix $D$), and produce another probability distribution as an output (output = density matrix $D'$).
References


