THE UNIVERSITY OF CALGARY

Programming with a Quantum Stack

by

Brett Gordon Giles

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

November, 2006

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Programming with a Quantum Stack" submitted by Brett Gordon Giles in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Chairman, Dr. Robin Cockett
Department of Computer Science

Dr. Peter Hoyer
Department of Computer Science

Dr. Barry Sanders
Director, Institute for Quantum Information Science

Date

Abstract

This thesis presents the semantics of *quantum stacks* and a functional quantum programming language, L-QPL. An operational semantics for L-QPL based on quantum stacks in the form of a term logic is developed and used as an interpretation of quantum circuits. The operational semantics is then extended to handle recursion and algebraic datatypes. Recursion and datatypes are not concepts found in quantum circuits, but both are generally required for modern programming languages.

The language L-QPL is introduced in a discussion and example format. Various example programs using both classical and quantum algorithms are used to illustrate features of the language. Details of the language, including handling of **qubit**s, general data types and classical data are covered.

The quantum stack machine is then presented. Supporting data for operation of the machine are introduced and the transitions induced by the machine's instructions are given.

Acknowledgments

Anyone who has written a thesis knows that an incredible number of people support and inspire the writer. I have been incredibly fortunate to have many gifted people in my life during this period.

My greatest appreciation to Dr. Robin Cockett for working with me these past years. His tireless efforts and indefatigable enthusiasm are a constant source of inspiration.

Thanks to Robin Cockett, Peter Hoyer and Barry Sanders for serving on my thesis committee.

Thank you to Peter Selinger for support and encouragement during the initial development and experimentation with quantum programming languages.

Thank you to my many proof-readers from the programming languages lab, Xiuzhan Guo, Ning Tang, Sean Nichols and Pieter Hofstra . Any remaining errors are mine.

The author would like to acknowledge Bryan Eastin and Steven T. Flammia, the developers of the Qcircuit package used to draw the circuits presented in this thesis and M. Tatsuya, who developed the proof package used in the judgements.

Finally, a wonderful thanks to my sweetheart Marie, who supported me, inspired me and proofread the thesis. C'est pour toi que je suis.

Dedication

Dedicated to my wonderful wife Marie Gélinas Giles and to my grandchildren.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis introduces a *quantum stack* and its semantics together with a quantum programming language which has quantum control and classically controlled data types. The work was motivated by the desire to provide a semantically correct programming language for programming quantum algorithms at a higher level than **bit**s and **qubit**s.

The thesis starts with a brief comparison with, and contrast to other Haskell based quantum simulators. This is followed by a short (re-)introduction of the basic concepts of linear algebra and quantum computing.

The two main contributions of this thesis, the quantum stack and L-QPL, may be reviewed independently of each other. Some of the appendices refer to both the quantum stack and L-QPL.

For those interested in reviewing the introduction of the quantum stack machine and its implementation the recommended reading path is the chapter on semantics, chapter 3, followed by chapter 5. In appendix C, further details are given on quantum stack machine instructions and the translation of L-QPL into quantum stack machine code. The details of the Haskell implementation of the machine are also covered in appendix D. Implementation details of the quantum stack are in appendix D.1.2 while details of the staged implementation of the quantum stack machine are in appendix D.2.4.

Chapter 4 is the primary required reading for learning L-QPL. All facets of the language are presented in that chapter interspersed with a number of examples. Further examples of programs in L-QPL can be found in appendix E. A complete BNF description of L-QPL is available in appendix B and a description of how L-QPL is translated into quantum stack machine code in appendix C.2. Details of running the compiler are in appendix F.

1.1 Why a quantum programming language?

Currently, it is not clear that there will ever be a quantum computer having a number of `qubit`s comparable to the number of `bit`s available on classical computers. Thus, it is reasonable to ask what is the point of a quantum programming language. The compiler for L-QPL presented in this thesis targets a virtual machine — the quantum stack machine, which is implemented with a classical language on a classical computer. Given that, it is highly likely that any significant quantum algorithm written in L-QPL will be woefully inefficient.

However, there *are* many reasons to create and use a quantum programming language.

Theory of algorithms. The current understanding of the limits of practical computability has led researchers and practitioners to probabilistic (and quantum) algorithms to solve problems. This has increased the understanding of those algorithms and led back to classical algorithms. An example of this is the recent polynomial time algorithm for primality testing in [7].

Quantum algorithms subsume probabilistic algorithms in that providing language support for quantum computing also gives probabilistic support. A simple example of this can be seen in the program to generate a coin toss given in figure 4.5 on page 79.

Quantum algorithm experimentation. Thinking about quantum algorithms is enhanced by having a high level way of expressing these algorithms. A high level language such as L-QPL allows the researcher or practitioner to design an algorithm at an altogether different level than the standard $\mathtt{bit}$s and $\mathtt{qubit}$s of quantum circuits. A simulator and virtual machine as provided by this thesis allows experimentation with quantum algorithms, allowing a broader exploration of the field.

Quantum computer design. The thesis presents a novel view of quantum computation using a quantum stack machine. This suggests a way of organizing quantum computation with a quantum stack machine as the central element. This may stimulate others to consider how such a machine can be realized efficiently.

1.2 Haskell based quantum computation emulators

1.2.1 Functional quantum programming

Shin-Cheng Mu and Richard Bird wrote [26] in 2001, which explores ways to write quantum algorithms in Haskell. In this paper, n $\mathtt{qubit}$s are represented as a list of 2^n values. The paper refers to these as *quregs*.

The paper provides Haskell code to perform unitary operations and measurements.

Two examples of quantum algorithms; the Deutsch-Jozsa and Grover's search algorithm are provided.

Bird and Mu then make the point that a *qureg* resembles a Haskell monad, in that a *join* function and *return* function may be defined on *quregs*. This resemblance is used to restate the Deutsch-Jozsa algorithm in a monadic format. The importance of this approach is that it encourages programmers to consider the algorithm used, rather than concentrating on the multitude of possible values a *qureg* may assume.

The approach given in [26] is not used by the simulator introduced in this thesis.

1.2.2 Modelling quantum computation in Haskell

In 2003, Amr Sabry [29] described a way to emulate quantum computation in Haskell.

Representation of quantum values

In the paper a **qubit** is represented as a map from *basis* values to values in $\mathbb{C}$. The *basis* is a list of elements that can be used as an orthogonal basis of a vector space. Examples presented in the paper include [**False**,**True**], [*Up*,*Down*] and [*Red*, *Green*, *Blue*], where the latter two are made up of constructors for some of the new types introduced in the paper.

This allows a programmer to represent any **qubit** once the basis is chosen. For example, a "False" **qubit** is represented as $\{\mathsf{False} : 1.0{+}0.0i\}$ and an indeterminate

"False - True" `qubit` may be represented as

$$\{\mathsf{False} : \frac{1}{\sqrt{2}} + 0.0i; \mathsf{True} : \frac{1}{\sqrt{2}} + 0.0i\}$$

The Haskell type used for this is **type** *QV a* = *FiniteMap a* (**Complex Double**).

Entanglement

The concept of *entangled* `qubit`s is handled by extending the basis to pairs of basis elements. Given a *basis* definition for elements of type a, extend this to a definition of a *basis* for elements of type (a,a) by using the product type.

As an example, the standard EPR pair under the (**Bool**, **Bool**) basis would be represented as

$$\{(\mathsf{False}, \mathsf{False}) : \frac{1}{\sqrt{2}} + 0.0i; (\mathsf{True}, \mathsf{True}) : \frac{1}{\sqrt{2}} + 0.0i\}.$$

Unitary operations

Unitary operations are defined as functions of type *QV* **Bool** -> *QV* **Bool**. The definition of the Hadamard gate is given as

```
had v = let a = pr v False
            b = pr v True
        in qv [(False, a+b), (True, a-b)]
```

In the above *pr* retrieves the value of a basis element from a `qubit` and *qv* is a smart constructor. *qv* takes a list of (*basis*, **Complex Double**) pairs and turns the list into an element of type *QV basis*.

It is possible to lift operations on standard classical values to ones on quantum values *provided the operation is reversible*. The paper provides Haskell functions for lifting of the base case where the classical function is of type (*a*->*b*).

Measurement

Measurements of quantum values are done by "collapsing" the measured value. The code as supplied in the paper requires that measurable quantum values be stored in *IORef* variables. *IORef* variables are the Haskell idiom for mutable variables. The measurement will generate a random value to pick one of the basis elements (a) as the result of the measure and then update the variable so that it is now:

$$\{a : 1.0 + 0.0i\}$$

This also ensures all future measures will return the same value.

Multiple qubits

Operations on multiple **qubit**s are discussed next in the paper and the issues with isolating specific sets of **qubit**s to work with are addressed. In Dr. Sabry's simulator, *adaptor*s are required to restructure the form of multiple **qubit**s. A specific adaptor would, for example, take a quantum value of type *QV* $(a,(b,c))$ to one of *QV* $((a,c),b)$. This then allows the application of a unitary transformation to the first and third **qubit**s of the first type. Dr. Sabry assumes that it will be possible to generate any needed adaptors rather than writing code for each individual one. Candidates for doing this would be template Haskell or a pre-compiler that creates adaptors as required.

1.2.3 Comparison with and contrast to the quantum stack machine

Dr. Sabry's paper provided the inspiration for the use of a **class** *Basis* and a Haskell **Map** from basis elements to values when defining the simulator presented in this thesis. It is used in a slightly different way in that the elements of a `qubit`'s density matrix are indexed by pairs of basis elements as opposed to a single basis element.

Classical data and constructed datatypes in the quantum stack machine also use the idea of a Haskell **Map** from their "basis" elements to values. The basis elements for classical data are the integers and Booleans. The basis elements for a constructed data type are the constructors together with bound nodes.

In contrast to Dr. Sabry's approach, multiple `qubit`s are handled by increasing the depth of the tree, classical and quantum data are easily mixed and measurement is approached in a very different way.

One of the largest differences is the approach to handling quantum values. The simulator presented in this thesis (QSM) keeps all the probabilities of values as the program proceeds. This allows us to perform the quantum computation just once, and then pick random values as needed to evaluate different results. In Dr. Sabry's presentation, the actual quantum algorithm needs to be re-run to determine the different possible results.

1.3 QPL by Peter Selinger

In 2002, Peter Selinger presented a description and categorical semantics for a functional quantum programming language in [30].

Much of the work in this thesis was inspired by and often based upon the language described therein. Dr. Selinger first presented a diagrammatic language consisting of picture fragments corresponding to various operations in the language. In later sections of the paper, QPL and Block QPL are introduced. QPL closely mirrors the diagrammatic language with the addition of a few minor restrictions. Block QPL restricts the language further by creating a structured language that would allow allocation of data in a stack based environment rather than on a heap as required for QPL. The syntax for QPL is given in the appendix, figure A.1 on page 128.

1.3.1 Comparison with and contrast to L-QPL

While inspired by QPL, L-QPL has diverged considerably. The syntax of L-QPL has been changed and extended, data construction has been added and a demarcation between classical and quantum data is included in L-QPL.

The two languages are similar in that they both have `qubit`s as a first class datatype and provide standard operations on `qubit`s. The provided operations include unitary transformations and measurement.

Both languages provide procedures, although with different syntax. Each has an assignment statement. Each is a functional language in the sense that a statement of the language is a function from its inputs to its outputs.

The greatest difference between QPL and L-QPL is that QPL is a `bit` and `qubit` oriented language, while L-QPL was designed to work with algebraic data types. L-QPL retains `qubit`s, but `bit`s are not built-in to L-QPL.

The syntax for unitary transformations differs between the languages, with QPL using an operator-assignment type of syntax and L-QPL syntactically treating transformations in the same way as function calls. Quantum control in QPL is done via assuming various built-in controlled transforms, while L-QPL requires specifying the control variables of a transform explicitly.

QPL provides explicit looping based upon a `bit`'s value. In L-QPL, all looping is done via recursion.

The semantics of L-QPL is inspired by QPL. We only provide an operational semantics in this thesis and semantic comparison is future work.

1.4 QML by Altenkirch and Grattage

Thorsten Altenkirch, Jonathan Grattage and others have introduced QML in [22], [23], [8], and [9]. This is a functional quantum programming language intended to compile to a simulator based on [29].

In the papers, the authors give an operational semantics of the language in terms of quantum circuits and a denotational semantics in terms of superoperators, similar to that of [30].

This is somewhat different from the approach of this thesis, which provides a translation from quantum circuits to L-QPL and an operational semantics of L-QPL with respect to quantum stacks. Using the translation of circuits to L-QPL this also gives us an operational semantics of quantum circuits.

Chapter 2

Quantum computation and circuits

2.1 Linear algebra

Quantum computation requires familiarity with the basics of linear algebra. This section will give definitions of the terms used throughout this thesis.

2.1.1 Basic definitions

The first definition needed is that of a *vector space*.

Definition 2.1.1 (Vector Space)**.** *Given a field* F*, whose elements will be referred to as scalars, a* vector space *over* F *is a non-empty set* V *with two operations,* vector addition *and* scalar multiplication*.* Vector addition *is defined as* $+ : \mathsf{V} \times \mathsf{V} \to \mathsf{V}$ *and denoted as* $\boldsymbol{v} + \boldsymbol{w}$ *where* $\boldsymbol{v}, \boldsymbol{w} \in \mathsf{V}$*. The set* V *must be an abelian group under* $+$*.* Scalar multiplication *is defined as* $\ : \mathsf{F} \times \mathsf{V} \to \mathsf{V}$ *and denoted as* $c\boldsymbol{v}$ *where* $c \in \mathsf{F}, \boldsymbol{v} \in \mathsf{V}$*. Scalar multiplication distributes over both vector addition and scalar addition and is associative.* F*'s multiplicative identity is an identity for scalar multiplication.*

The specific algebraic requirements are:

1. $\forall \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathsf{V},\ (\boldsymbol{u} + \boldsymbol{v}) + \boldsymbol{w} = \boldsymbol{u} + (\boldsymbol{v} + \boldsymbol{w})$;

2. $\forall \boldsymbol{u}, \boldsymbol{v} \in \mathsf{V},\ \boldsymbol{u} + \boldsymbol{v} = \boldsymbol{v} + \boldsymbol{u}$;

3. $\exists 0 \in \mathsf{V}$ such that $\forall \boldsymbol{v} \in \mathsf{V}, 0 + \boldsymbol{v} = \boldsymbol{v}$;

4. $\forall \mathbf{u} \in V, \exists \mathbf{v} \in V$ such that $\mathbf{u} + \mathbf{v} = 0$;

5. $\forall \mathbf{u}, \mathbf{v} \in V, c \in F,\ c(\mathbf{u} + \mathbf{v}) = c\mathbf{u} + c\mathbf{v}$;

6. $\forall \mathbf{u} \in V, c, d \in F,\ (c + d)\mathbf{u} = c\mathbf{u} + d\mathbf{u}$;

7. $\forall \mathbf{u} \in V, c, d \in F,\ (cd)\mathbf{u} = c(d\mathbf{u})$;

8. $\forall \mathbf{u} \in V,\ 1\mathbf{u} = \mathbf{u}$.

Examples of vector spaces over F are: $F^{n\times m}$ – the set of $n \times m$ matrices over F; and F^n – the $n-$fold Cartesian product of F. $F^{n\times 1}$, the set of $n \times 1$ matrices over F is also called the space of column vectors, while $F^{1\times n}$, the set of row vectors. Often, F^n is identified with $F^{n\times 1}$.

This thesis shall identify F^n with the column vector space over F.

Definition 2.1.2 (Linearly independent). *A subset of vectors* $\{\mathbf{v}_i\}$ *of the vector space* V *is said to be* linearly independent *when no finite linear combination of them,* $\sum a_j\mathbf{v}_j$ *equals* 0 *unless all the* a_j *are zero.*

Definition 2.1.3 (Basis). *A* basis *of a vector space* V *is a linearly independent subset of* V *that generates* V. *That is, any vector* $u \in V$ *is a linear combination of the basis vectors.*

2.1.2 Matrices

As mentioned above, the set of $n \times m$ matrices over a field is a vector space. Additionally, matrices compose and the tensor product of matrices is defined.

Matrix composition is defined as usual. That is, for $A = [a_{ij}] \in F^{m\times n}, B = [b_{jk}] \in F^{n\times p}$:

$$A\,B = \left[\left(\sum_j a_{ij}b_{jk}\right)_{ik}\right] \in F^{m\times p}.$$

Definition 2.1.4 (Diagonal matrix)**.** *A* diagonal matrix *is a matrix where the only non-zero entries are those where the column index equals the row index.*

The diagonal matrix $n \times n$ with only 1's on the diagonal is the identity for matrix multiplication, and is designated by I_n.

Definition 2.1.5 (Transpose)**.** *The* transpose *of an* $n\times m$ *matrix* $A = [a_{ij}]$ *is an* $m\times n$ *matrix* A^t *with the* i, j *entry being* a_{ji}.

When the base field of a matrix is $\mathbb{C}$, the complex numbers, the *conjugate transpose* (also called the *adjoint*) of an $n \times m$ matrix $A = [a_{ij}]$ is defined as the $m \times n$ matrix A^* with the i, j entry being $\overline{a}_{ji}$, where $\overline{a}$ is the complex conjugate of $a \in \mathbb{C}$.

When working with column vectors over $\mathbb{C}$, note that $\mathbf{u} \in \mathbb{C}^n \implies \mathbf{u}^* \in \mathbb{C}^{1\times n}$ and that $\mathbf{u}^* \times \mathbf{u} \in \mathbb{C}^{1\times 1}$. This thesis will use the usual identification of $\mathbb{C}$ with $\mathbb{C}^{1\times 1}$. A column vector $\mathbf{u}$ is called a *unit vector* when $\mathbf{u}^* \times \mathbf{u} = 1$.

Definition 2.1.6 (Trace)**.** *The* trace, $\mathrm{Tr}(A)$ *of a square matrix* $A = [a_{ij}]$ *is* $\sum a_{ii}$.

Tensor product

The tensor product of two matrices is the usual Kronecker product:

$$U \otimes V = \begin{bmatrix} u_{11}V & u_{12}V & \cdots & u_{1m}V \\ u_{21}V & u_{22}V & \cdots & u_{2m}V \\ \vdots & \vdots & \ddots & \\ u_{n1}V & u_{n2}V & \cdots & u_{nm}V \end{bmatrix} = \begin{bmatrix} u_{11}v_{11} & \cdots & u_{12}v_{11} & \cdots & u_{1m}v_{1q} \\ u_{11}v_{21} & \cdots & u_{12}v_{21} & \cdots & u_{1m}v_{2q} \\ \vdots & \vdots & \vdots & \ddots & \\ u_{n1}v_{p1} & \cdots & u_{n2}v_{p1} & \cdots & u_{nm}v_{pq} \end{bmatrix}$$

Special matrices

When working with quantum values certain types of matrices over the complex numbers are of special interest. These are:

Unitary Matrix : Any $n \times n$ matrix A with $AA^* = I\ (= A^*A)$.

Hermitian Matrix : Any $n \times n$ matrix A with $A = A^*$.

Positive Matrix : Any Hermitian matrix A in $\mathbb{C}^{n\times n}$ where $\mathbf{u}^* A \mathbf{u} \geqslant 0$ for all vectors $\mathbf{u} \in \mathbb{C}^n$. Note that for any Hermitian matrix A and vector u, $\mathbf{u}^* A \mathbf{u}$ is real.

Completely Positive Matrix : Any positive matrix A in $\mathbb{C}^{n\times n}$ where $I_m \otimes A$ is positive.

The matrix

$$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

is an example of a matrix that is *unitary*, *Hermitian*, *positive* and *completely positive*.

Superoperators

A *Superoperator* S is a matrix over $\mathbb{C}$ with the following restrictions:

1. S is *completely positive*. This implies that S is positive as well.

2. For all positive matrices A, $\mathrm{Tr}(S\,A) \leqslant \mathrm{Tr}(A)$.

2.2 Basic quantum computation

2.2.1 Quantum bits

Quantum computation deals with operations on $\mathtt{qubit}$s. A $\mathtt{qubit}$ is typically represented in the literature on quantum computation as a complex linear combination of $|0\rangle$ and $|1\rangle$, respectively identified with (1,0) and (0,1) in $\mathbb{C}^2$. Because of the identification of the basis vectors, any $\mathtt{qubit}$ can be identified with a non-zero vector in $\mathbb{C}^2$. In standard quantum computation, the important piece of information in a $\mathtt{qubit}$ is its direction rather than amplitude. In other words, given $q = \alpha\,|0\rangle + \beta\,|1\rangle$ and $q' = \alpha'\,|0\rangle + \beta'\,|1\rangle$ where $\alpha = \gamma\alpha'$ and $\beta = \gamma\beta'$, then q and q' represent the same quantum state.

A $\mathtt{qubit}$ that has either α or β zero is said to be in a *classical state*. Any other combination of values is said to be a *superposition*.

Section 2.3 on page 19 will introduce quantum circuits which act on $\mathtt{qubit}$s. This section will have some forward references to circuits to illustrate points introduced here.

2.2.2 Quantum entanglement

Consider what happens when working with a pair of $\mathtt{qubit}$s, p and q. This can be considered as the a vector in $\mathbb{C}^4$ and written as

$$\alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle\,. \tag{2.1}$$

In the case where p and q are two independent $\mathtt{qubit}$s, with $p = \alpha\,|0\rangle + \beta\,|1\rangle$ and $q = \gamma\,|0\rangle + \delta\,|1\rangle$,

$$p \otimes q = \alpha\gamma\,|00\rangle + \alpha\delta\,|01\rangle + \beta\gamma\,|10\rangle + \beta\delta\,|11\rangle \tag{2.2}$$

where $p \otimes q$ is the standard tensor product of p and q regarded as vectors. There are states of two $\mathtt{qubit}$s that cannot be written as a tensor product. As an example, the state

$$\frac{1}{\sqrt{2}}\,|00\rangle + \frac{1}{\sqrt{2}}\,|11\rangle \tag{2.3}$$

is not a tensor product of two distinct $\mathtt{qubit}$s. In this case the two $\mathtt{qubit}$s are said to be *entangled*.

2.2.3 Quantum gates

Quantum gates operate on $\mathtt{qubit}$s. These gates are conceptually similar to logic gates in the classical world. In the classical world the only non-trivial single $\mathtt{bit}$ gate is the Not gate which sends 0 to 1 and 1 to 0. However, there are infinitely many non-trivial quantum gates.

An $n-\mathtt{qubit}$ quantum gate is represented by a $2^n \times 2^n$ matrix. A necessary and sufficient condition for such a matrix to be a quantum gate is that it is *unitary*.

The entanglement of two $\mathtt{qubit}$s, p and q, is accomplished by applying a Hadamard transformation to p followed by a Not applied to q controlled by p. The circuit in figure 2.2 on page 19 shows how to entangle two $\mathtt{qubit}$s that start with an initial state of $|00\rangle$. See figure E.1 on page 223 for how this can be done in L-QPL.

A list of some common gates, together with their usual quantum circuit representation is given in the next section in table 2.1 on page 21.

2.2.4 Measurement

The other allowed operation on a **qubit** or group of **qubit**s is measurement. When a **qubit** is measured it assumes only one of two possible values, either $|0\rangle$ or $|1\rangle$. Given

$$q = \alpha |0\rangle + \beta |1\rangle \tag{2.4}$$

where $|\alpha|^2 + |\beta|^2 = 1$, then measuring q will result in $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. Once a **qubit** is measured, re-measuring will always produce the same value.

In multi-**qubit** systems the order of measurement does not matter. If p and q are as in equation (2.1) on page 14, let us suppose measuring p gives $|0\rangle$. The measure will result in that value with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$, after which the system collapses to the state:

$$\alpha_{00} |00\rangle + \alpha_{01} |01\rangle \tag{2.5}$$

Measuring the second **qubit**, q, will give $|0\rangle$ with probability $|\alpha_{00}|^2$ or $|1\rangle$ with probability $|\alpha_{01}|^2$.

Conversely, if q was measured first and gave us $|0\rangle$ (with a probability of $|\alpha_{00}|^2 + |\alpha_{10}|^2$) and then p was measured, p will give us $|0\rangle$ with probability $|\alpha_{00}|^2$ or $|1\rangle$ with probability $|\alpha_{10}|^2$.

Thus, when measuring both p and q, the probability of getting $|0\rangle$ from both measures is $|\alpha_{00}|^2$, regardless of which **qubit** is measured first.

Considering states such as in equation (2.3), measuring either $\mathtt{qubit}$ would actually force the other $\mathtt{qubit}$ to the same value. This type of entanglement is used in many quantum algorithms such as quantum teleportation.

2.2.5 Mixed states

The notion of *mixed states* refers to an outside observer's knowledge of the state of a quantum system. Consider a 1 $\mathtt{qubit}$ system

$$v = \alpha\left|0\right\rangle + \beta\left|1\right\rangle. \tag{2.6}$$

If v is measured but the results of the measurement are not examined, the state of the system is either $|0\rangle$ or $|1\rangle$ and is no longer in a superposition. This type of state is written as:

$$v = |\alpha|^2\{|0\rangle\} + |\beta|^2\{|1\rangle\}. \tag{2.7}$$

An external (to the state) observer knows that the state of v is as expressed in equation (2.7). Since the results of the measurement were not examined, the exact state (0 or 1) is unknown. Instead, a probability is assigned as expressed in the equation. Thus, if the $\mathtt{qubit}$ v is measured and the results are not examined, v can be treated as a probabilistic $\mathtt{bit}$ rather than a $\mathtt{qubit}$.

2.2.6 Density matrix notation

The state of any quantum system of $\mathtt{qubit}$s may be represented via a *density matrix*. In this notation, given a $\mathtt{qubit}$ v, the coefficients of $|0\rangle$ and $|1\rangle$ form a column vector u. Then the density matrix corresponding to v is uu^*. If $v = \alpha\left|0\right\rangle + \beta\left|1\right\rangle$,

$$v = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}\begin{pmatrix}\overline{\alpha} & \overline{\beta}\end{pmatrix} = \begin{pmatrix}\alpha\overline{\alpha} & \alpha\overline{\beta}\\ \beta\overline{\alpha} & \beta\overline{\beta}\end{pmatrix}. \tag{2.8}$$

When working with mixed states the density matrix of each component of the mixed state is added. For example, the mixed state shown in equation (2.7) on the preceding page would be represented by the density matrix

$$|\alpha|^2 \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + |\beta|^2 \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} |\alpha|^2 & 0 \\ 0 & |\beta|^2 \end{pmatrix}. \tag{2.9}$$

Note that since the density matrix of mixed states is a linear combination of other density matrices, it is possible to have two different mixed states represented by the same density matrix.

The advantage of this notation is that it becomes much more compact for mixed state systems. Additionally, scaling issues are handled by insisting the density matrix has a trace = 1. During a general quantum computation, as we shall see, the trace can actually fall below 1 indicating that the computation is not everywhere total.

Gates and density matrices

When considering a **qubit** q as a column vector and a unitary transform T as a matrix, the result of applying the transform T to q is the new vector Tq. The density matrix of the original **qubit** is given by $q\, q^*$, while the density matrix of the transformed **qubit** is $(Tq)(Tq)^*$, which equals $T(qq^*)T^*$. Thus, when a **qubit** q is represented by a density matrix A, the formula for applying the transform T to q is TAT^*.

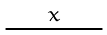
2.3 Quantum circuits

2.3.1 Contents of quantum circuits

Currently a majority of quantum algorithms are defined and documented using *quantum circuits*. These are wire-type diagrams with a series of **qubit**s input on the left of the diagram and output on the right. Various graphical elements are used to describe quantum gates, measurement, control and classical **bit**s.

Gates and qubits

The simplest circuit is a single wire with no action:



The next simplest circuit is one **qubit** and one gate. The **qubit** is represented by a single wire, while the gate is represented by a box with its name, G, inside it. This is shown in the circuit in figure 2.1. In general, the name of the wire which is input to the gate G may be different from the name of G's output wire. Circuit diagrams may also contain constant components as input to gates as in the circuit in figure 2.3 on the following page.

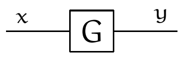


Figure 2.1: Simple single gate circuit

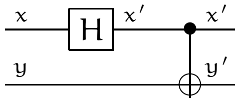


Figure 2.2: Entangling two **qubit**s.

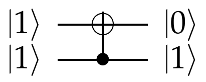


Figure 2.3: Controlled-Not of $|1\rangle$ and $|1\rangle$

Future diagrams will drop the wire labels except when they are important to the concept under discussion.

Controlled gates, where the gate action depends upon another `qubit`, are shown by attaching a wire between the wire of the control `qubit` and the controlled gate. The circuit in figure 2.2 on the previous page shows two `qubit`s, where a Hadamard is applied to the top `qubit`, followed by a controlled-Not applied to the second `qubit`. In circuits, the control `qubit` is on the vertical wire with the solid dot. This is then connected via a horizontal wire to the gate being controlled.

A list of common gates, their circuits and corresponding matrices is given in table 2.1 on the following page.

Measurement

Measurement is used to transform the quantum data to classical data so that it may be then used in classical computing (e.g. for output). The act of measurement is placed at the last part of the quantum algorithm in many circuit diagrams and is sometimes just implicitly considered to be there.

While there are multiple notations used for measurement in quantum circuit diagrams, this thesis will standardize on the *D-box* style of measurement as shown in figure 2.4.

A measurement may have a double line leaving it, signifying a `bit`, or nothing, signifying a destructive measurement.

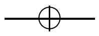
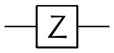
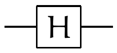
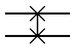
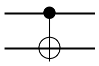
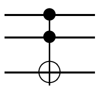
Gate	**Circuit**	**Matrix**
Not (X)		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Z		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Swap		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Controlled-Not		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Toffoli		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Table 2.1: Gates, circuit notation and matrices

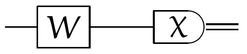


Figure 2.4: Measure notation in quantum circuits

Operations affecting multiple **qubit**s at the same time are shown by extending the gate or measure box to encompass all desired wires. In the circuit in figure 2.5, the gate U applies to all of the first three **qubit**s and the measurement applies to the first two **qubit**s.

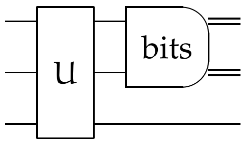


Figure 2.5: Examples of multi-**qubit** gates and measures

0-control and control by bits

The examples above have only shown control based upon a **qubit** being $|1\rangle$. Circuits also allow control on a **qubit** being $|0\rangle$ and upon classical values. The circuit in figure 2.6 on the following page with four **qubit**s $(r_1, r_2, p$ and $q)$ illustrates all these forms of control.

At g_1, a Hadamard is 1—controlled by r_2 and is applied to each of r_1 and p. This is followed in column g_2 with the Not transform applied to r_2 being 0—controlled by r_1 . In the same column, a Z gate is 0—controlled by q and applied to p. p and q are then measured in column g_3 and their corresponding classical values are used for control in g_4. In g_4, the U_R gate is applied to both r_1 and r_2, but only when the measure result of p is 0 and the measure result of q is 1.

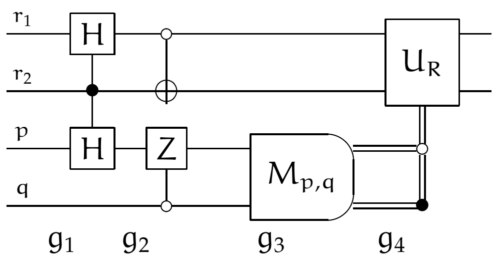


Figure 2.6: Other forms of control for gates

Multi-qubit lines

It is common to represent multiple **qubit**s on one line. A gate applied to a multi-**qubit** line must be a tensor product of gates of the correct dimensions. The circuit in figure 2.7 shows n **qubit**s on one line with the Hadamard gate (tensored with itself n times) applied to all of them. That is followed by a unary gate U_R tensored with $I^{\otimes(n-2)}$ and tensored with itself again. This will have the effect of applying an U_R gate to the first and last **qubit**s on the line.

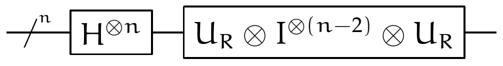


Figure 2.7: n **qubit**s on one line

Other common circuit symbols

Two other symbols that are regularly used are the swap and controlled-Z, shown in the circuit in figure 2.8 on the following page. Note that swap is just shorthand for a series of three controlled-Not gates with the control **qubit** changing. This can also be seen directly by multiplying the matrices for the controlled-Not gates as shown in equation (2.10) on the next page.

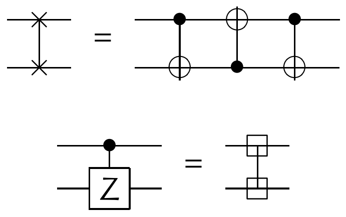


Figure 2.8: Swap and controlled-Z

$$\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix} \begin{bmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{bmatrix} \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}. \tag{2.10}$$

2.3.2 Syntax of quantum circuits

Quantum circuits were originally introduced by David Deutsch in [17]. He extended the idea of standard classical based gate diagrams to encompass the quantum cases. In his paper, he introduced the concepts of quantum gates, sources of `bit`s, sinks and universal gates. One interesting point of the original definition is that it *does* allow loops. Currently, the general practice is not to allow loops of `qubit`s. The commonly used elements of a circuit are summarized in table 2.2 on the following page.

A valid quantum circuit must follow certain restrictions. As physics requires `qubit`s must not be duplicated, circuits must enforce this rule. Therefore, three restrictions in circuits are the *no fan-out*, *no fan-in* and *no loops* rules. These conditions are a way to express the *linearity* of quantum algorithms. Variables (wires) may not be duplicated, may not be destroyed without a specific operation and may not be amalgamated.

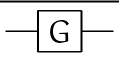
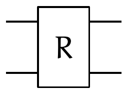
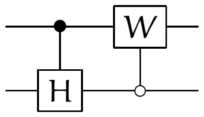
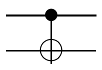
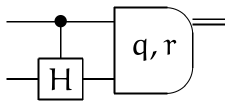
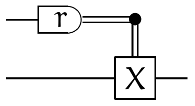
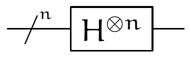
Desired element	**Element in a quantum circuit diagram.**	**Example**
qubit	A single horizontal line.	—
Classical bit	A double horizontal line.	═
Single-qubit gates	A box with the gate name (G) inside it, one wire attached on its left and one wire attached on the right.	
Multi-qubit gates	A box with the gate name (R) inside it, n wires on the left side and the same number of wires on the right.	
Controlled qubit gates	A box with the gate name (H, W) inside, with a solid (1-control) or open (0-control) dot on the control wire with a vertical wire between the dot and the second gate.	
Controlled-Not gates	A *target* $\oplus$, with a solid (1-control) or open (0-control) dot on the control wire with a vertical wire between the dot and the gate.	
Measurement	A *D-box* shaped node with optional names or comments inside. One to n single wires are attached on the left (qubits coming in) and 0 to n classical bit wires on the left. Classical bits may be dropped as desired.	
Classical control	Control bullets attached to horizontal classical wires, with vertical classical wires attached to the controlled gate.	
Multiple qubits	Annotate the line with the number of qubits and use tensors on gates.	

Table 2.2: Syntactic elements of quantum circuit diagrams

2.3.3 Examples of quantum circuits

This section will present three quantum algorithms and the associated circuits. Each of these circuits presented may be found in [27].

First, *quantum teleportation*, an algorithm which sends a quantum bit across a distance via the exchange of two classical bits. This is followed by the *Deutsch-Jozsa algorithm*, which provides information about the global nature of a function with less work than a classical deterministic algorithm can. The third example is circuits for the *quantum Fourier transformation* and its inverse.

Quantum teleportation

The standard presentation of this algorithm involves two participants A and B. (Henceforth known as Alice and Bob). Alice and Bob first initialize two $\mathtt{qubit}$s to $|00\rangle$, then place them into what is known as an *EPR* (for Einstein, Podolsky and Rosen) state. This is accomplished by first applying the Hadamard gate to Alice's $\mathtt{qubit}$, followed by a controlled-Not to the pair of $\mathtt{qubit}$s controlled by Alice's $\mathtt{qubit}$.

Then, Bob travels somewhere distant from Alice, taking his $\mathtt{qubit}$ with him[1].

At this point, Alice receives a $\mathtt{qubit}$, ν, in an unknown state and has to pass ν on to Bob. She then uses ν as the control and applies a controlled-Not transform to this new pair. Alice then applies a Hadamard transform applied to ν.

Alice now measures the two $\mathtt{qubit}$s and sends the resulting two $\mathtt{bit}$s of classical information to Bob.

Bob then examines the two $\mathtt{bit}$s that he receives from Alice. If the $\mathtt{bit}$ resulting

[1]Notice that all other physical constraints are ignored in this algorithm. There is no concern about how one separates the $\mathtt{qubit}$s, transports the $\mathtt{qubit}$ or potential decoherence of the $\mathtt{qubit}$s.

from measuring Alice's original bit is 1, he applies the Not (also referred to as X) gate (= $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$) to his **qubit**. If the measurement result of ν is one, he applies the Z gate (= $\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$). Bob's **qubit** is now in the same state as the **qubit** Alice wanted to send. The circuit for this is shown in figure 2.9.

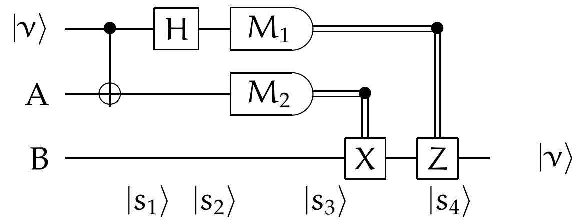


Figure 2.9: Quantum teleportation

For comparison , see figure E.1 on page 223 showing how this would be implemented in L-QPL.

Deutsch-Jozsa algorithm

The Deutsch-Jozsa algorithm describes a way of determining whether a function f[2] is *constant* (i.e. always 0 or 1) or *balanced* (i.e. produces an equal number of 0 or 1 results) based on applying it to one quantum bit. The function takes n **bit**s as input and produces a single **bit**.

f is assumed to be an expensive function, therefore, a desired effect is to evaluate f as few times as possible before determining if f is balanced or constant. The worst case scenario when evaluating f classically is that determining the result requires $2^{n-1}+1$ invocations of the function. The best possible case is 2 invocations, which occurs when f is balanced and the first two inputs chosen produce different

[2]The obvious pre-condition for the Deutsch-Jozsa algorithm is that the function f is *either* balanced or constant and not some general function. The results are not well-defined if f does not fit into one of the two possible categories.

results.

The quantum circuit requires only one application of the function to $n + 1$ **qubit**s which have been suitably prepared to make the decision.

The algorithm relies on being able to construct an $n+1$ order unitary operator based upon f. In general, a unitary operator like this may be constructed by mapping the state $|a, b\rangle$ to $|a, b \oplus f(a)\rangle$ where $\oplus$ is the exclusive-or operator and a is n **bit** values. If we name this operator U_f, the circuit in figure 2.10 will solve the problem with just one application. See the appendix, appendix E.1.3 on page 225 for how this would be done in L-QPL.

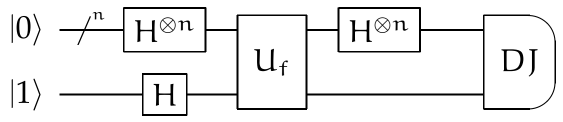


Figure 2.10: Circuit for the Deutsch-Jozsa algorithm

The idea of quantum parallelism is what makes this and many other quantum algorithms work. The initial state of the system is set to $|0^{\otimes n} \otimes 1\rangle$ after which the Hadamard gate is applied to all of the **qubit**s. This places the input **qubit**s into a superposition of all possible input values and the answer **qubit** is a superposition of 0 and 1. At this point, the unitary transformation U_f is applied to the **qubit**s. Then the Hadamard transform is applied again to the input **qubit**s.

To complete the algorithm, measure *all* the **qubit**s. It can be shown that if f is constant, the input **qubit**s will all measure to 0, while if it is balanced, at least one of those **qubit**s will be 1.

Quantum Fourier transform

The circuits for the quantum Fourier transformation and its inverse are in figure 2.11 and figure 2.12 respectively. These transforms are used extensively in many quantum algorithms, including Shor's factoring algorithm.

The quantum Fourier transform is definable on an arbitrary number of $\mathtt{qubit}$s. This is typically presented by eliding the 3rd to the $n-3$rd lines and interior processing. The L-QPL code for the quantum Fourier transform is in the appendix, figure E.2 on page 224. In this circuit, the parametrized transform R_n is the rotation transform, given by:

$$R_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^n}} \end{bmatrix}$$

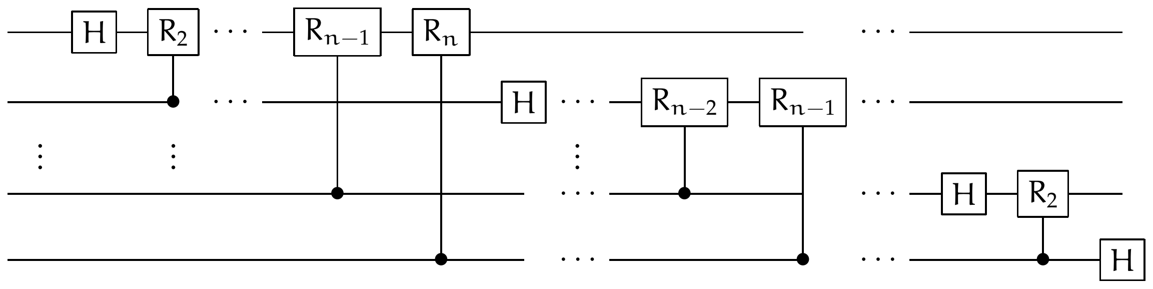


Figure 2.11: Circuit for the quantum Fourier transform

The inverse of a circuit is determined by traversing the circuit from right to left. This process changes the original quantum Fourier circuit to its inverse as shown in figure 2.12 on the following page. The L-QPL code for the inverse quantum Fourier transform is in the appendix, figure E.14 to figure E.15 on pages 233–234.

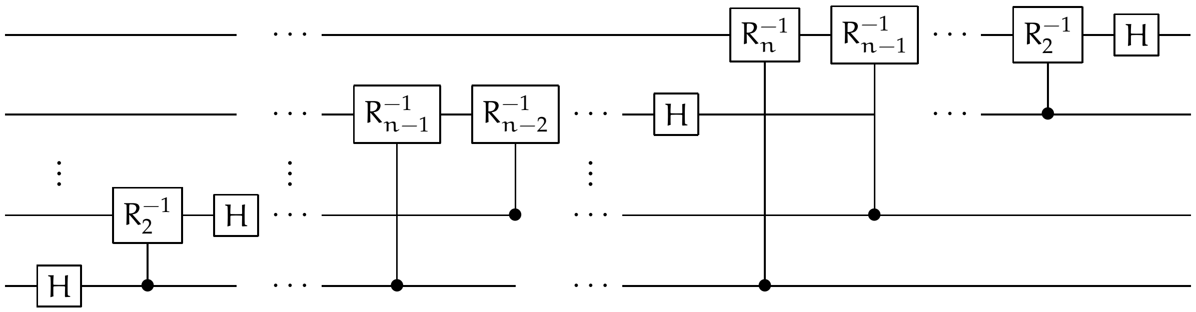


Figure 2.12: Circuit for the inverse quantum Fourier transform

2.4 Extensions to quantum circuits

To facilitate the transition to the programming language L-QPL, this section introduces three extensions to quantum circuits. The extensions are *renaming*, *wire bending and crossing*, and *scoped control*. Each extension adds expressive power to quantum circuits but does not change the semantic power. For each of the extensions, examples of how to re-write the extension in standard quantum circuit terminology will be provided.

2.4.1 Renaming

Quantum circuits currently allow renaming to be an implicit part of any gate. The circuit in figure 2.13 gives an operation to explicitly do this and its rewriting in standard circuit notation.

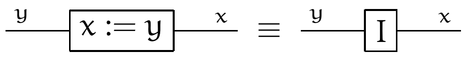


Figure 2.13: Renaming of a `qubit` and its equivalent diagram

2.4.2 Wire crossing

Crossing and bending of wires in a circuit diagram is added to allow a simpler presentation of algorithms. The circuit in figure 2.14 illustrates the concept of re-organizing and bending of wires.

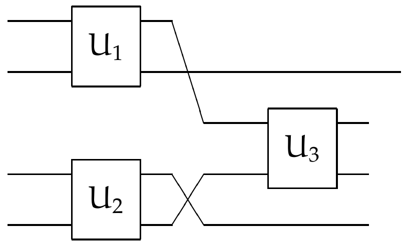


Figure 2.14: Bending

2.4.3 Scoped control

This extension allows us to group different operations in a circuit and show that all of them are controlled by a particular `qubit`. This is the same as attaching separate control wires to each of the gates in the grouped operations. Measurements are not affected by control. Figure 2.15 on the next page shows a scoped control box on the left which includes a measure. The right hand side of the same figure shows the circuit translated back to standard circuit notation, with the measure not being affected by the control.

Scoping boxes correspond to procedures and blocks in L-QPL.

Naturally, both scoping and bending may be combined as in figure 2.16.

However, note that exchanging wires is not the same as swap. Exchanging a pair of wires is not affected by control, but a swap is affected by control, as shown in figure 2.17 on the following page.

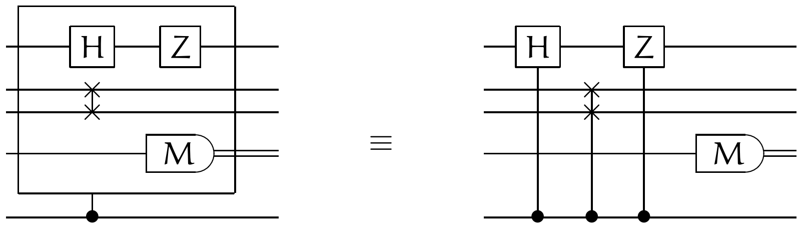


Figure 2.15: Scope of control

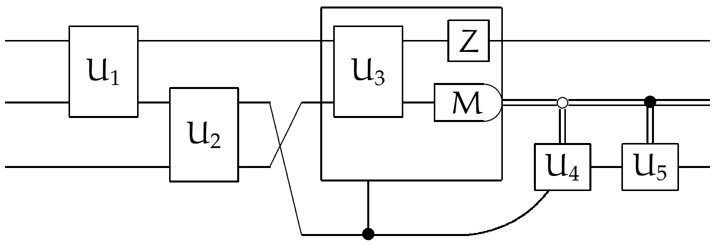


Figure 2.16: Extensions sample

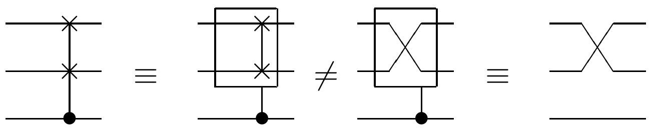


Figure 2.17: Swap in control vs. exchange in control

2.4.4 Circuit identities

Circuits allow the writing of the same algorithm in multiple ways. This subsection will list some of the circuit identities that hold with the extended notation.

First, note that although a measure may appear inside a control box, it is not affected by the control, as in figure 2.18. Conversely, a measurement commutes with control of a circuit as in figure 2.18.

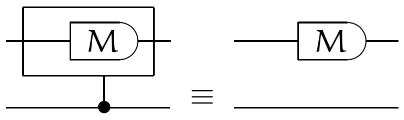


Figure 2.18: Measure is not affected by control

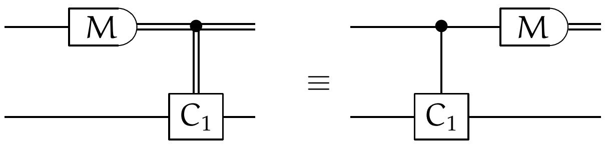


Figure 2.19: Control is not affected by measure

One of the notations introduced earlier was that of *0-control*. This type of control is the same as applying a Not transform before and after a *1-control*, as shown in figure 2.20.

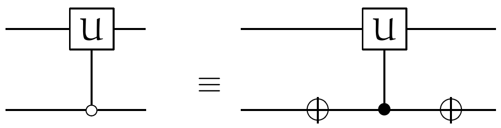


Figure 2.20: Zero control is syntactic sugar

Figure 2.21 on the following page shows that scoped control of multiple transforms is the same as controlling those transforms individually. Figure 2.22 similarly shows that scoped control of multiple transforms of the same qubit is the same as controlling those transforms serially.

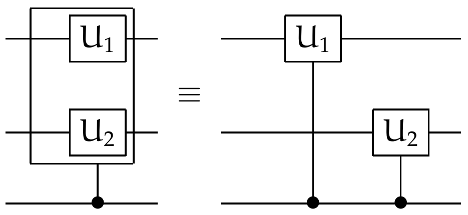


Figure 2.21: Scoped control is parallel control

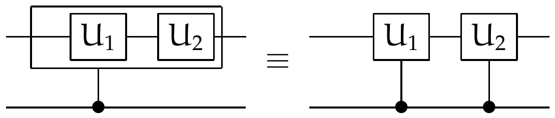


Figure 2.22: Scoped control is serial control

Multiple control commutes with scoping as shown in figure 2.23 to figure 2.24 on pages 34–35.

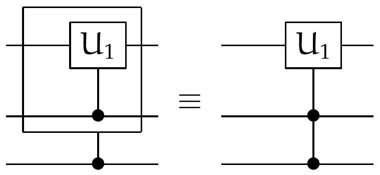


Figure 2.23: Multiple control

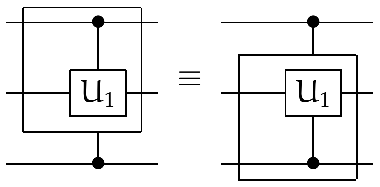


Figure 2.24: Control scopes commute

Chapter 3

Semantics

3.1 Basic L-QPL statements

Quantum circuits can be translated into basic L-QPL statements, which form a core fragment of the L-QPL language. Later sections in this chapter will provide the semantics for these statements and progressively build up to providing a semantics of L-QPL.

The language is introduced in a series of judgements which give the construction of valid L-QPL statements. Judgements have the following form:

$$\Gamma; \Gamma' \Vdash \mathcal{I}$$

where Γ is the context input to the statement $\mathcal{I}$ and Γ' is the context after it is completed.

Basic L-QPL has seven distinct statements: *identity, new, assign, discard, measure, control* and *unitary*. Statements may also be composed, creating a *block* of statements which may be used wherever a statement is required.

The judgements for the creation of basic L-QPL statements are given in figure 3.1 on the next page.

The *identity* statement is a do-nothing statement that does not change the stack or context. The statements *new* and *assign* share a similar syntax and create new variables in the output context. *Assign* will remove an existing variable, while *new* just creates the new variable. *Discard* removes a variable from the input context. The *measure* statement performs a destructive measure of a $\mathtt{qubit}$, while applying different sets of dependent statements contingent on the result of the measure. *Control* modifies the execution of dependent statements, contingent upon the value of the attached control variable. The final statement, *unitary*, applies unitary transformations to a $\mathtt{qubit}$.

$$\frac{}{\Gamma; \Gamma \Vdash \varepsilon}\ \text{identity}$$

$$\frac{i \in \{0,1\}}{\Gamma; (x{::}\mathbf{bit}), \Gamma \Vdash x = i}\ \text{new } \mathbf{bit} \qquad \frac{k \in \{0,1\}}{\Gamma; (q{::}\mathbf{qubit}), \Gamma \Vdash x = |k\rangle}\ \text{new } \mathbf{qubit}$$

$$\frac{}{(x{::}\tau), \Gamma; (y{::}\tau), \Gamma \Vdash x = y}\ \text{assign} \qquad \frac{}{(x{::}\tau), \Gamma; \Gamma \Vdash \text{disc } x}\ \text{discard}$$

$$\frac{\Gamma; \Gamma' \Vdash \mathcal{I}_0 \quad \Gamma; \Gamma' \Vdash \mathcal{I}_1}{(x{::}\mathbf{qubit}), \Gamma; \Gamma' \Vdash \text{meas } x\ |0\rangle \Rightarrow \mathcal{I}_0\ |1\rangle \Rightarrow \mathcal{I}_1}\ \text{measure}$$

$$\frac{\Gamma; \Gamma' \Vdash \mathcal{I}}{(z{::}\tau), \Gamma; (z{::}\tau), \Gamma' \Vdash \mathcal{I} \Leftarrow z}\ \text{control} \qquad \frac{\Gamma; \Gamma' \Vdash \mathcal{I}_1 \quad \Gamma'; \Gamma'' \Vdash \mathcal{I}_2}{\Gamma; \Gamma'' \Vdash (\mathcal{I}_1; \mathcal{I}_2)}\ \text{compose}$$

$$\frac{\Gamma_1; \Gamma_1' \Vdash \mathcal{I}_1 \quad \Gamma_2; \Gamma_2' \Vdash \mathcal{I}_2}{\Gamma_1\Gamma_2; \Gamma_1'\Gamma_2' \Vdash (\mathcal{I}_1;; \mathcal{I}_2)}\ \text{tensor compose}$$

$$\frac{}{(x{::}\mathbf{qubit}), \Gamma; (x{::}\mathbf{qubit}), \Gamma \Vdash U\ x}\ \text{unitary}$$

Figure 3.1: Judgements for creation of basic L-QPL statements.

3.1.1 Examples of basic L-QPL programs

The examples in this sub-section use the statements as introduced in figure 3.1 to give some example programs.

Program to swap two $\mathbf{qubit}$s. This program will swap two $\mathbf{qubit}$s by successive renaming. Both the input and output contexts have two $\mathbf{qubit}$s.

$q, v :: \mathbf{qubit};\ v, q :: \mathbf{qubit} \Vdash$

```
w = q;
q = v;
v = w;
```

Program to do a coinflip. The program has an empty input context and a single $\mathbf{bit}$ in its output context. At the end, the $\mathbf{bit}$ `b` will be 0 with probability .5 and 1 with an equal probability.

$\emptyset;\ b :: \mathbf{bit} \Vdash$

```
q = |0>;
Had q;
meas q
 |0> => {b = 0}
 |1> => {b = 1}
```

Program to entangle two $\mathbf{qubit}$s. This program places two input $\mathbf{qubit}$s into an EPR state.

$q, r :: \mathbf{qubit};\ q, r :: \mathbf{qubit} \Vdash$

```
Had q;
Not r <= q;
```

3.2 Translation of quantum circuits to basic L-QPL

Quantum circuits are translatable to L-QPL statements. The semantic meaning of a circuit is denoted by enclosing the circuit between $[\![$ and $]\!]$.

When a new wire is added to a diagram (at the start of the diagram, this can be done for all wires) the meaning is an assignment of $|0\rangle$ to that variable.

$$\left[\!\!\left[|0\rangle\!\!-\!\!\!-_{q} \right]\!\!\right] = q = |0\rangle \tag{3.1}$$

An existing line with no operations on it translates to an identity statement:

$$[\![\text{———}]\!] = \varepsilon. \tag{3.2}$$

A unitary transform U translates to the obvious corresponding L-QPL statement:

$$\left[\!\!\left[\overset{q}{\text{——}}\boxed{U}\overset{q}{\text{—}} \right]\!\!\right] = U\ q. \tag{3.3}$$

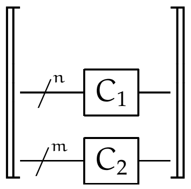
A renaming of a **qubit** wire translates to an assign statement:

$$\left[\!\!\left[\overset{y}{\text{——}}\boxed{x := y}\overset{x}{\text{—}} \right]\!\!\right] = x = y. \tag{3.4}$$

A quantum circuit followed by another circuit is the composition of the meanings:

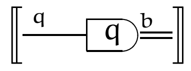
$$\left[\!\!\left[\text{—}\boxed{C_1}\overset{n}{\text{—/—}}\boxed{C_2}\text{—} \right]\!\!\right] = [\![C_1]\!]\,;[\![C_2]\!]\,. \tag{3.5}$$

A circuit that is above another is the tensor composition of the two circuits meanings:



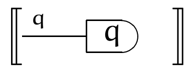
$$= [\![C_1]\!]\ ;\ ;\ [\![C_2]\!]\ . \tag{3.6}$$

Measurement of a **qubit** and outputting a **bit** translates to the measure instruction:



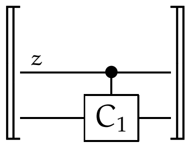
$$= \text{meas q } |0\rangle \text{ => } \{b = 0\}\ |1\rangle \text{ => } \{b = 1\}. \tag{3.7}$$

A destructive measure, where the resulting **bit** is discarded, is translated as a discard:



$$= \text{disc q}\ . \tag{3.8}$$

Control of a circuit translates to L-QPL control constructions. The controlling variable may be a **qubit** or **bit**:



$$= [\![C_1]\!] \Leftarrow z. \tag{3.9}$$

Equation (3.1) through equation (3.9) give a meaning for all the standard quantum circuit elements and their extensions described in section 2.4 on page 30. Certain constructs such as $0-$`control` have not been included as they are describable in terms of other circuit elements that have been assigned a semantics. See section 2.4 on page 30 for those identities.

3.3 Quantum stacks

This section will define a quantum stack and use quantum stacks to provide an operational semantics for L-QPL statements. The previous section has provided a translation of quantum circuits into these statements.

3.3.1 Definition of a quantum stack

In classical computing, a *stack* is an object together with operations for pushing new items onto the stack, retrieving items from the stack and examining whether the stack is empty or not. Refinements are often made in terms of adding linear addressing to the stack, multiple push-pull operations or defining specific data elements that are allowed on the stack.

A *quantum stack* is a somewhat more complex object, due to the properties of entanglement and superpositions. However, the basic idea of an object together with methods of adding and removing elements is retained. The initiating idea of the quantum stack was to find a stack-like way to represent the state of a quantum system and therefore the density matrix describing it.

The construction of a quantum stack is given by four judgements, shown in figure 3.2 on the next page. These judgements show the quantum stack is a pair, $\Gamma \vdash S$. The Γ, called the *context*, is a list of names and their types. The S, called the *stack*, contains the actual stack nodes and traces. The stack will often be written as S^t where the t is an explicitly shown trace.

The judgement rules do not restrict quantum stacks to be Hermitian (i.e., correspond to a tuple of matrices, all of which are Hermitian) or to have a trace less than 1. However, when applied to a quantum stack with these properties, the statements used to interpret quantum circuits do not increase the trace and take Hermitian stacks to Hermitian stacks.

$$\frac{}{\vdash \emptyset^{0}} \qquad \frac{}{\vdash \alpha^{\sqrt{\alpha\overline{\alpha}}}}\ \text{scalar}$$

$$\frac{\Gamma \vdash S_{00}^{t_{00}}, S_{01}^{t_{01}}, S_{10}^{t_{10}}, S_{11}^{t_{11}}}{q{::}\mathbf{qubit}, \Gamma \vdash q\{ij \to S_{ij}\}^{t_{00}+t_{11}}}\ \mathbf{qubit}$$

$$\frac{\Gamma \vdash S_{0}^{t_{0}}, S_{1}^{t_{1}}}{b{::}\mathbf{bit}, \Gamma \vdash b\{i \to S_{i}\}^{t_{0}+t_{1}}}\ \mathbf{bit}$$

Figure 3.2: Judgements for a quantum stack

Description of the quantum stack

A quantum stack is a tree where each node is associated with a particular **bit** or **qubit**. The nodes associated with a **qubit** have four branches, corresponding to the density matrix representation of a **qubit**. The nodes associated with a **bit** have two branches, corresponding to the probabilistic representation of a bit. (i.e., **bit** $b = \alpha 0 + \beta 1$, where $\alpha + \beta \leqslant 1.0$). All nodes have a *trace*. Data values, which are the result of multiplying the probabilities of **bit**s and density matrix entries for **qubit**s, are stored at the leaves of the tree.

The trace of a quantum stack is a non-negative real number. It is defined recursively as shown in figure 3.3 on the following page.

$$\text{trace}(Q) = \begin{cases} \text{trace}(Q_{00}) + \text{trace}(Q_{11}) & \text{when } Q = q\left\{\begin{array}{ll} 00 \to Q_{00}; & 01 \to Q_{01} \\ 10 \to Q_{10}; & 11 \to Q_{11} \end{array}\right\} \\ \text{trace}(B_0) + \text{trace}(B_1) & \text{when } Q = b\{0 \to B_0; 1 \to B_1\} \\ \sqrt{v\overline{v}} & \text{when } Q = v \text{ (a scalar at the leaf)} \end{cases}$$

Figure 3.3: Definition of the trace of a quantum stack

Examples

The quantum stack of a single **bit**, b, which is 0 with a probability of .25 and 1 with a probability of .75, is represented as:

$$b\{0 \to .25; 1 \to .75\}^{1.0}.$$

A **qubit** q that was 0 and then subjected to the Hadamard transform followed by a Phase transformation is represented as:

$$q\{00 \to .5; 01 \to -.5i; 10 \to .5i; 11 \to .5\}^{1.0}.$$

A general **qubit** r having the density matrix $\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ is represented as

$$r\{ij \to a_{ij}\}^{(a_{00}+a_{11})}$$

where the ij of the labelling correspond to the row and column of q's density matrix. The number at the upper right of the node representation is its trace.

Given a single **bit** b as above and a **qubit** q_1 which is 0, the representation is

$$b\{0 \to q_1\{00 \to .25\}^{.25}; 1 \to q_1\{00 \to .75\}^{.75}\}^{1.0}.$$

As shown in this example, values that are zero probabilities or zero entries of the matrix can be elided without loss of meaning.

3.3.2 Quantum stack equivalence

The equality judgements for quantum stacks are given in figure 3.4.

$$\frac{\Gamma \vdash S}{\Gamma \vdash S = \Gamma \vdash S}\ \text{reflexivity} \qquad \frac{\Gamma \vdash S_2 = \Gamma \vdash S_1}{\Gamma \vdash S_1 = \Gamma \vdash S_2}\ \text{symmetry}$$

$$\frac{\Gamma \vdash S_1 = \Gamma \vdash S_2 \quad \Gamma \vdash S_2 = \Gamma \vdash S_3}{\Gamma \vdash S_1 = \Gamma \vdash S_3}\ \text{transitivity}$$

$$\frac{\Gamma, (t{::}\tau), \Gamma' \vdash S}{\Gamma, (t{::}\tau), \Gamma' \vdash S = (\text{rotate } t) \cdot \Gamma, (t{::}\tau), \Gamma' \vdash S}\ \text{rotation}$$

Figure 3.4: Judgements for quantum stack equality

Rotation brings a desired node to the top of the stack and then reorganizes the tree so that the Huffman-like encoding of the leaves is invariant. Given a tree with four nodes (a, b, c, d), each of the leaves can be encoded by the indexes depending on the node names. e.g., $v_{i_a;i_b;i_c;i_d}$. Then, after rotating a tree, following a particular index path in the tree will take you to the same node. That is, if the tree order is now (d, b, c, a) with d, c **qubit**s, a, b **bit**s, then following the path $01 \to 1 \to 10 \to 0$ will take you to the leaf $v_{0;1;10;01}$, where the labelling is from the original a, b, c, d ordering. An example of this in the **bit**, **qubit** case is provided below.

Rotation example

Consider the following single **bit**, single **qubit** quantum stack,

$$c\left\{\begin{array}{l} 0 \to r\left\{\begin{array}{ll} 00 \to v_{0;00}; & 01 \to v_{0;01} \\ 10 \to v_{0;10}; & 11 \to v_{0;11} \end{array}\right\}^{v_{0;00}+v_{0;11}} \\ 1 \to r\left\{\begin{array}{ll} 00 \to v_{1;00}; & 01 \to v_{1;01} \\ 10 \to v_{1;10}; & 11 \to v_{1;11} \end{array}\right\}^{v_{1;00}+v_{1;11}} \end{array}\right\}^{\sum v},$$

where $\sum \nu = \nu_{0;00} + \nu_{0;11} + \nu_{1;00} + \nu_{1;11}$. If the $\mathtt{qubit}$ r is rotated to the top of the quantum stack, the resulting stack will be

$$r \left\{ \begin{array}{ll} 00 \to c\{0 \to \nu_{0;00}; 1 \to \nu_{1;00}\}^{\nu'_{00}}; & 01 \to c\{0 \to \nu_{0;01}; 1 \to \nu_{1;01}\}^{\nu'_{01}} \\ 10 \to c\{0 \to \nu_{0;10}; 1 \to \nu_{1;10}\}^{\nu'_{10}}; & 11 \to c\{0 \to \nu_{0;11}; 1 \to \nu_{1;11}\}^{\nu'_{11}} \end{array} \right\}^{\sum \nu},$$

where $\sum \nu$ is as before and $\nu'_{ij} = \nu_{0;ij} + \nu_{1;ij}$.

Details of rotation on a quantum stack

Rotation is defined recursively on a quantum stack. Given a target name t and a quantum stack $\Gamma \vdash S$, rotation brings the node named t to the top and thus changes $\Gamma \vdash S$ to the quantum stack $(t : \tau), \Gamma' \vdash S'$. The quantum stack $\Gamma \vdash S$ is equivalent to $(t : \tau), \Gamma' \vdash S'$.

If a quantum stack has duplicate names, only the highest node with that name will be rotated up. Rotation's definition can be broken into cases depending on whether the target name is at the top, the second element or somewhere lower in the stack and whether the nodes being affected are $\mathtt{bit}$s or $\mathtt{qubit}$s.

When the target name is the top element of the stack, there is no change to the stack and rotation is just an identity transformation of the quantum stack.

In the case where the target name is the second element of the stack, there are four separate cases to consider: $\mathtt{bit}$ above a $\mathtt{bit}$; $\mathtt{bit}$ above a $\mathtt{qubit}$; $\mathtt{qubit}$ above a $\mathtt{qubit}$ and $\mathtt{qubit}$ above a $\mathtt{bit}$.

$\mathtt{bit}$ above a $\mathtt{bit}$:

$$(\mathrm{rotate}_1\ b_2) \cdot b_1 \left\{ \begin{array}{c} 0 \to b_2\{0 \to S_{0;0}; 1 \to S_{0;1}\} \\ 1 \to b_2\{0 \to S_{1;0}; 1 \to S_{1;1}\} \end{array} \right\}^{t} = b_2 \left\{ \begin{array}{c} 0 \to b_1\{0 \to S_{0;0}; 1 \to S_{1;0}\} \\ 1 \to b_1\{0 \to S_{0;1}; 1 \to S_{1;1}\} \end{array} \right\}^{t}.$$

bit above a **qubit**

$$(\mathrm{rotate}_1\ q_2)\cdot b_1\left\{\begin{array}{l} 0\to q_2\left\{\begin{array}{ll} 00\to S_{0;00}; & 01\to S_{0;01}\\ 10\to S_{0;10}; & 11\to S_{0;11}\end{array}\right\}\\ 1\to q_2\left\{\begin{array}{ll} 00\to S_{1;00}; & 01\to S_{1;01}\\ 10\to S_{1;10}; & 11\to S_{1;11}\end{array}\right\}\end{array}\right\}^t$$

$$= q_2\left\{\begin{array}{ll} 00\to b_1\{0\to S_{0;00};1\to S_{1;00}\}; & 01\to b_1\{0\to S_{0;01};1\to S_{1;01}\}\\ 10\to b_1\{0\to S_{0;10};1\to S_{1;10}\}; & 11\to b_1\{0\to S_{0;11};1\to S_{1;11}\}\end{array}\right\}^t.$$

qubit above a **bit**:

$$(\mathrm{rotate}_1\ b_2)\cdot q_1\left\{\begin{array}{l} 00\to b_2\{0\to S_{00;0};1\to S_{00;1}\}\\ 01\to b_2\{0\to S_{01;0};1\to S_{01;1}\}\\ 10\to b_2\{0\to S_{10;0};1\to S_{10;1}\}\\ 11\to b_2\{0\to S_{11;0};1\to S_{11;1}\}\end{array}\right\}^t$$

$$= b_2\left\{\begin{array}{l} 0\to q_1\{00\to S_{00;0};01\to S_{01;0};10\to S_{10;0};11\to S_{11;0}\}\\ 1\to q_1\{00\to S_{00;1};01\to S_{01;1};10\to S_{10;1};11\to S_{11;1}\}\end{array}\right\}^t.$$

qubit above a **qubit**:

$$(\mathrm{rotate}_1\ q_2)\cdot q_1\left\{\begin{array}{l} 00\to q_2\left\{\begin{array}{ll} 00\to S_{00;00}; & 01\to S_{00;01}\\ 10\to S_{00;10}; & 11\to S_{00;11}\end{array}\right\}\\ 01\to q_2\left\{\begin{array}{ll} 00\to S_{01;00}; & 01\to S_{01;01}\\ 10\to S_{01;10}; & 11\to S_{01;11}\end{array}\right\}\\ 10\to q_2\left\{\begin{array}{ll} 00\to S_{10;00}; & 01\to S_{10;01}\\ 10\to S_{10;10}; & 11\to S_{10;11}\end{array}\right\}\\ 11\to q_2\left\{\begin{array}{ll} 00\to S_{11;00}; & 01\to S_{11;01}\\ 10\to S_{11;10}; & 11\to S_{11;11}\end{array}\right\}\end{array}\right\}^t$$

$$= q_2\left\{\begin{array}{l} 00\to q_1\left\{\begin{array}{ll} 00\to S_{00;00}; & 01\to S_{01;00}\\ 10\to S_{10;00}; & 11\to S_{11;00}\end{array}\right\}\\ 01\to q_1\left\{\begin{array}{ll} 00\to S_{00;01}; & 01\to S_{01;01}\\ 10\to S_{10;01}; & 11\to S_{11;01}\end{array}\right\}\\ 10\to q_1\left\{\begin{array}{ll} 00\to S_{00;10}; & 01\to S_{01;10}\\ 10\to S_{10;10}; & 11\to S_{11;10}\end{array}\right\}\\ 11\to q_1\left\{\begin{array}{ll} 00\to S_{00;11}; & 01\to S_{01;11}\\ 10\to S_{10;11}; & 11\to S_{11;11}\end{array}\right\}\end{array}\right\}^t.$$

When the node to be rotated up is deeper in the quantum stack, we recurse down. Otherwise, we just apply rotate_1. That is:

$$(\mathrm{rotate}\ n_3)\cdot n_1\{\ell_k\to n_2\{\ell'_m\to S\}\}$$

$$= (\mathrm{rotate}_1\ n_3)\cdot n_1\{\ell_k\to(\mathrm{rotate}\ n_3)\cdot n_2\{\ell'_m\to S\}\}$$

and

$$(\text{rotate } n_2) \cdot n_1\{\ell_k \to n_2\{\ell'_m \to S\}\} = (\text{rotate}_1\, n_2) \cdot n_1\{\ell_k \to n_2\{\ell'_m \to S\}\}.$$

In this way, the desired node "bubbles" to the top of the quantum stack.

3.3.3 Basic operations on a quantum stack

Quantum stacks may be added, tensored and multiplied by scalars. Multiplication of a quantum stack $\Gamma \vdash S$ by a scalar α recurses down the stack of S until the leaves are reached. Each leaf value is then multiplied by α. This will also have the effect of multiplying the trace of S by $\sqrt{\alpha\overline{\alpha}}$.

When adding two stacks, $\Gamma \vdash S$ and $\Gamma \vdash T$, the first step is to align the stacks. Aligning is done by recursively rotating nodes in the second stack to correspond with the same named nodes in the first stack. When the alignment is completed, the values at the corresponding leaves are added to produce a result stack. The trace of the resulting quantum stack will be the sum of the traces of S and T.

Tensoring of two quantum stacks $\Gamma_1 \vdash S_1$ and $\Gamma_2 \vdash S_2$ proceeds by first creating a new context Γ, which is the concatenation of Γ_1 and Γ_2. The new stack portion S is created by replacing each leaf value of S_1 with a copy of S_2 multiplied by the previous leaf value. The trace of the tensor will be the product of the traces of S_1 and S_2. The notation for this operation is

$$S = S_1 \otimes S_2.$$

The judgements for these stack operations are shown in figure 3.5 on the following page.

$$\frac{\Gamma \vdash S^{t}}{\Gamma \vdash (\alpha S)^{t\sqrt{\alpha\overline{\alpha}}}}\ \text{scalar multiply}$$

$$\frac{\Gamma \vdash S_1^{t_1}, S_2^{t_2}}{\Gamma \vdash (S_1 + S_2)^{t_1+t_2}}\ \text{add}$$

$$\frac{\Gamma_1 \vdash S_1^{t_1} \quad \Gamma_2 \vdash S_2^{t_2}}{\Gamma_1\Gamma_2 \vdash (S_1 \otimes S_2)^{t_1\times t_2}}\ \text{tensor}$$

Figure 3.5: Judgements for basic stack operations

When recursing down stacks to do an addition or a scalar multiplication, any elided branches are treated as having the required structure with leaf values of 0.

As an example, consider the following two quantum stacks:

$$S_1 = q\{00 \to b\{0 \to .1; 1 \to .1\}^{.2}; 11 \to b{::}\{1 \to .1\}^{.25}\}^{.45}$$
$$S_2 = q\{00 \to b\{0 \to .01; 1 \to .1\}^{.15}; 11 \to b\{0 \to .15; 1 \to .25\}^{.4}\}^{.55}.$$

In the first stage, recursively add the 00 branches of each, i.e.:

$$S_1[00] = b\{0 \to .1; 1 \to .1\}^{.2}$$
$$S_2[00] = b\{0 \to .05; 1 \to .1\}^{.15}.$$

When adding these two branches, each of the labels immediately point to leaves. As leaves are the base case for addition, this gives a result of:

$$S'[00] = b\{0 \to .15; 1 \to .2\}^{.35}.$$

Similarly, apply the addition process in the same manner to the 11 branches of S_1

and S_2:

$$S_1[11] = \mathfrak{b}\{1 \to .25\}^{.25}$$

$$S_2[11] = \mathfrak{b}\{0 \to .15; 1 \to .25\}^{.4}.$$

Note that in this case, the first sub-stack does not have a branch labelled by 0, due to elision of zero leaf values. The result of adding this is

$$S'[11] = \mathfrak{b}\{0 \to .15; 1 \to .5\}^{.65},$$

which gives us the final result of:

$$S = \mathfrak{q}\{00 \to \mathfrak{b}\{0 \to .15; 1 \to .2\}^{.35}; 11 \to \mathfrak{b}\{0 \to .15; 1 \to .5\}^{.65}\}^{1}.$$

The general cases for addition are given in figure 3.6. In the figure, the token $\langle\mathit{empty}\rangle$ stands for any sub-branches that have been elided because all leaves below that node are 0.

$$\begin{aligned}
S_1 + \langle\mathit{empty}\rangle &= S_1 \\
\langle\mathit{empty}\rangle + S_2 &= S_2 \\
v_1 + v_2 &= v_1 + v_2 \\
v_1 + \langle\mathit{empty}\rangle &= v_1 \\
\langle\mathit{empty}\rangle + v_2 &= v_2 \\
nm{::}\{\ell_i \to S_i\} + nm{::}\{\ell_i \to S_i'\} &= nm{::}\{\ell_i \to (S_i + S_i')\} \\
S_1 + S_2 &= S_1 + (\text{rotate } nm_{S_1}) \cdot S_2.
\end{aligned}$$

Figure 3.6: Definition of addition of quantum stacks

Notation	**Meaning**
$\widetilde{z}$	A list of names
$\Gamma;\Gamma' \Vdash \mathcal{I} \cdot \Gamma\Gamma'' \vdash S$	Application of the statements $\mathcal{I}$ to the quantum stack S.
$(\Gamma;\Gamma' \Vdash \mathcal{I} \Leftarrow \widetilde{z}) \cdot \Gamma \vdash S$	Application of the statements $\mathcal{I}$ to the quantum stack S, controlled by the $\mathtt{bit}$s and $\mathtt{qubit}$s in $\widetilde{z}$.
$(\Gamma;\Gamma' \Vdash \mathcal{I}) \cdot \Gamma \vdash S^{t} \rightsquigarrow \Gamma' \vdash S'^{t'}$	Application of the statements $\mathcal{I}$ with inputs Γ and output Γ' to the quantum stack S in context Γ results in the quantum stack S' in context Γ'.
$\Gamma \vdash \mathcal{I} \cdot S^{t} \rightsquigarrow \Gamma' \vdash S'^{t'}$	Alternate way to write the application of the statements $\mathcal{I}$ to the quantum stack S. Γ will contain the input variables of $\mathcal{I}$ and Γ' will contain the output variables.

Table 3.1: Notation used in judgements for operational semantics

3.4 Semantics of basic L-QPL statements

Section 3.1 introduced basic L-QPL statements, giving the judgements for their formation. This section introduces the concept of a statement modifier. Modifiers will be one of *IdOnly*, *Left*, or *Right*. The judgement for formation of statements with a modifier is:

$$\frac{\mathcal{M} \in \{\mathtt{IdOnly,\ Left,\ Right}\} \quad \Gamma;\Gamma' \Vdash \mathcal{I}}{\Gamma;\Gamma' \Vdash \mathcal{M}\mathcal{I}}\ \text{modifier}$$

An operational semantics for basic L-QPL statements is given by the judgement diagrams in figure 3.7 on page 52 and figure 3.8 on page 53. Table 3.1 gives the additional notation used in the judgements.

The modifiers (*IdOnly, Left, Right*) come from examining the effect of a controlled transformation in a general setting. When a controlled transform and a general density matrix are multiplied:

$$\begin{pmatrix} 1 & 0 \\ 0 & U \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & U^* \end{pmatrix} = \begin{pmatrix} A & BU^* \\ UC & UDU^* \end{pmatrix} \tag{3.10}$$

Equating the density matrix to the quantum stack notation, a controlled transformation is controlled by the **qubit** at the top of the quantum stack. It has the four sub-stacks A, B, C and D and the $\mathtt{controlled} - U$ transform is applied as in equation (3.10). The resulting quantum stack, with the same **qubit** at the top now has the four sub-stacks A, BU^*, UC and UDU^*. This leads to the obvious terminology of "`IdOnly`U" for the action on A, "`Right`U" for the action on B and "`Left`U" for the action on C. The action on D is a normal application of the transform U to D. As such, it does not need a modifier to describe the action.

Figure 3.7 on the following page presents the operational semantics of the the L-QPL statements *identity, new, discard, measure, compose* and *control*. Figure 3.8 on page 53 presents the operational semantics of *unitary* and the effects of statement modifiers.

3.4.1 Operational semantics commentary

Identity

This has no effect on the quantum stack or the context.

New bit and new qubit

These two statements transform the quantum stack S by adding a new node at the top of S. Both **bit** and **qubit** nodes are added with a single sub-branch which points to S. In the case of a **bit** the new sub-branch is labelled with 0, for a **qubit**

$$\frac{}{\Gamma \vdash \varepsilon \cdot S \rightsquigarrow \Gamma \vdash S}\ \text{identity} \qquad \frac{}{\begin{array}{l}\Gamma \vdash (b = 0) \cdot S \rightsquigarrow \\ \qquad (b{::}\mathbf{bit}), \Gamma \vdash b\{0 \to S\}^{t}\end{array}}\ \text{new } \mathbf{bit}$$

$$\frac{}{\Gamma \vdash (q = |0\rangle) \cdot S \rightsquigarrow (q{::}\mathbf{qubit}), \Gamma \vdash q\{00 \to S\}^{t}}\ \text{new } \mathbf{qubit}$$

$$\frac{}{(b{::}\mathbf{bit}), \Gamma \vdash (\text{disc } b) \cdot b\{0 \to S_0; 1 \to S_1\}^{t} \rightsquigarrow \Gamma \vdash (S_0 + S_1)^{t}}\ \text{delete } \mathbf{bit}$$

$$\frac{}{(q{::}\mathbf{qubit}), \Gamma \vdash (\text{disc } q) \cdot q\{ij \to S_{ij}\}^{t} \rightsquigarrow \Gamma \vdash (S_{00} + S_{11})^{t}}\ \text{delete } \mathbf{qubit}$$

$$\frac{\Gamma \vdash I_0 \cdot S_{00}^{t_0} \rightsquigarrow \Gamma' \vdash S_0^{t_0'} \quad \Gamma \vdash I_1 \cdot S_{11}^{t_1} \rightsquigarrow \Gamma' \vdash S_1^{t_1'}}{(q{::}\mathbf{qubit})\Gamma \vdash \left\{\begin{array}{l} \text{meas } q : \\ \ |0\rangle => I_0 \\ \ |1\rangle => I_1 \end{array}\right\} \cdot q\{ij \to S_{ij}\}^{t_0 + t_1} \rightsquigarrow \Gamma' \vdash (S_0 + S_1)^{(t_0' + t_1')}}\ \text{measure}$$

$$\frac{\Gamma \vdash Op_1 \cdot S^{t} \rightsquigarrow \Gamma' \vdash S'^{t'} \quad \Gamma' \vdash Op_2 \cdot S'^{t'} \rightsquigarrow \Gamma'' \vdash S''^{t''}}{\Gamma \vdash (Op_1; Op_2) \cdot S^{t} \rightsquigarrow \Gamma'' \vdash S''^{t''}}\ \text{compose}$$

$$\frac{\Gamma_1 \vdash Op_1 \cdot S_1^{t_1} \rightsquigarrow \Gamma_1' \vdash S_1'^{t_1'} \quad \Gamma_2 \vdash Op_2 \cdot S_2{}^{t_2} \rightsquigarrow \Gamma_2' \vdash S_2'^{t_2'}}{\Gamma_1\Gamma_2 \vdash (Op_1;;Op_2) \cdot (S_1 \otimes S_2)^{t_1 \times t_2} \rightsquigarrow \Gamma_1'\Gamma_2' \vdash (S_1' \otimes S_2')^{t_1' \times t_2'}}\ \text{tensor}$$

$$\frac{\Gamma \vdash \mathcal{I} \Leftarrow \widetilde{z} \cdot S_1^{t} \rightsquigarrow \Gamma \vdash S'^{t'}_1}{\begin{array}{l}(z{::}\mathbf{bit}), \Gamma \vdash (\mathcal{I} \Leftarrow \widetilde{z}, z) \cdot z \left\{\begin{array}{l} 0 \to S_0^{t_0} \\ 1 \to S_1^{t} \end{array}\right\} \rightsquigarrow \\ \qquad\qquad (z{::}\mathbf{bit}), \Gamma \vdash z \left\{\begin{array}{l} 0 \to S_0 \\ 1 \to S_1' \end{array}\right\}^{t_0 + t'}\end{array}}\ \text{control } \mathbf{bit}$$

$$\frac{\begin{array}{c}\Gamma \vdash ((\texttt{IdOnly}\ \mathcal{I}) \Leftarrow \widetilde{z}) \cdot S_{00} \rightsquigarrow \Gamma \vdash S_{00}' \\ \Gamma \vdash ((\texttt{Left}\ \mathcal{I}) \Leftarrow \widetilde{z}) \cdot S_{10} \rightsquigarrow \Gamma \vdash S_{10}' \\ \Gamma \vdash ((\texttt{Right}\ \mathcal{I}) \Leftarrow \widetilde{z}) \cdot S_{01} \rightsquigarrow \Gamma \vdash S_{01}' \\ \Gamma \vdash (\mathcal{I} \Leftarrow \widetilde{z}) \cdot S_{11}^{t_{11}} \rightsquigarrow \Gamma \vdash S'^{t_{11}'}_{11}\end{array}}{\begin{array}{l}(z{::}\mathbf{qubit}), \Gamma \vdash (\mathcal{I} \Leftarrow \widetilde{z}, z) \cdot z\{ij \to S_{ij}\} \rightsquigarrow \\ \qquad\qquad (z{::}\mathbf{qubit}), \Gamma \vdash z \left\{\begin{array}{ll} 00 \to S_{00}'; & 01 \to S_{01}' \\ 10 \to S_{10}'; & 11 \to S_{11}' \end{array}\right\}^{t_{00} + t_{11}'}\end{array}}\ \text{control } \mathbf{qubit}$$

Figure 3.7: Operational semantics of basic L-QPL statements

$$\frac{\mathtt{U}\begin{pmatrix} S_{00} & S_{01} \\ S_{10} & S_{11} \end{pmatrix}\mathtt{U}^* = \begin{pmatrix} S'_{00} & S'_{01} \\ S'_{10} & S'_{11} \end{pmatrix}}{\begin{array}{l}(q{::}\mathbf{qubit}), \Gamma \vdash \mathtt{U}\ q \cdot q\{ij \rightarrow S_{ij}\}^t \rightsquigarrow \\ \qquad (q{::}\mathbf{qubit}), \Gamma \vdash q \left\{ \begin{array}{ll} 00 \rightarrow S'_{00}; & 01 \rightarrow S'_{01} \\ 10 \rightarrow S'_{10}; & 11 \rightarrow S'_{11} \end{array} \right\}^t \end{array}} \text{ U a transform}$$

$$\frac{}{\begin{array}{l}(q{::}\mathbf{qubit}), \Gamma \vdash \texttt{IdOnly}\ \mathtt{U}\ q \cdot q\{ij \rightarrow S_{ij}\}^t \rightsquigarrow \\ \qquad (q{::}\mathbf{qubit}), \Gamma \vdash q\{ij \rightarrow S_{ij}\}^t \end{array}} \begin{array}{l}\texttt{IdOnly}\ \mathtt{U}, \\ \text{U a transform}\end{array}$$

$$\frac{\mathtt{U}\begin{pmatrix} S_{00} & S_{01} \\ S_{10} & S_{11} \end{pmatrix} = \begin{pmatrix} S'_{00} & S'_{01} \\ S'_{10} & S'_{11} \end{pmatrix}}{\begin{array}{l}(q{::}\mathbf{qubit}), \Gamma \vdash \texttt{Left}\ \mathtt{U}\ q \cdot q\{ij \rightarrow S_{ij}\}^t \rightsquigarrow \\ \qquad (q{::}\mathbf{qubit}), \Gamma \vdash q \left\{ \begin{array}{ll} 00 \rightarrow S'_{00}; & 01 \rightarrow S'_{01} \\ 10 \rightarrow S'_{10}; & 11 \rightarrow S'_{11} \end{array} \right\}^t \end{array}} \begin{array}{l}\texttt{Left}\ \mathtt{U}, \\ \text{U a transform}\end{array}$$

$$\frac{\begin{pmatrix} S_{00} & S_{01} \\ S_{10} & S_{11} \end{pmatrix}\mathtt{U}^* = \begin{pmatrix} S'_{00} & S'_{01} \\ S'_{10} & S'_{11} \end{pmatrix}}{\begin{array}{l}(q{::}\mathbf{qubit}), \Gamma \vdash \texttt{Right}\ \mathtt{U}\ q \cdot q\{ij \rightarrow S_{ij}\}^t \rightsquigarrow \\ \qquad (q{::}\mathbf{qubit}), \Gamma \vdash q \left\{ \begin{array}{ll} 00 \rightarrow S'_{00}; & 01 \rightarrow S'_{01} \\ 10 \rightarrow S'_{10}; & 11 \rightarrow S'_{11} \end{array} \right\}^t \end{array}} \begin{array}{l}\texttt{Right}\ \mathtt{U}, \\ \text{U a transform}\end{array}$$

$$\frac{\Gamma \vdash \mathcal{I} \cdot S \rightsquigarrow \Gamma' \vdash S'}{\Gamma \vdash (\mathcal{M}\mathcal{I}) \cdot S \rightsquigarrow \Gamma' \vdash S'} \ \mathcal{M}\mathcal{I}, (\mathcal{I} \text{ not a transform})$$

$$\frac{\Gamma \vdash (\mathcal{M}_1 \cdots \mathcal{M}_n)\mathcal{I} \cdot S \rightsquigarrow \Gamma' \vdash S' \quad \mathcal{M}_0 \equiv \mathcal{M}_1}{\Gamma \vdash (\mathcal{M}_0\, \mathcal{M}_1 \cdots \mathcal{M}_n)\mathcal{I} \cdot S \rightsquigarrow \Gamma' \vdash S'} \text{ modifier reduction}$$

$$\frac{\mathcal{M}_0 \neq \mathcal{M}_1 \quad \Gamma \vdash (\text{IdOnly } \mathcal{I}) \cdot S \rightsquigarrow \Gamma' \vdash S'}{\Gamma \vdash ((\mathcal{M}_0\, \mathcal{M}_1 \cdots \mathcal{M}_n)\mathcal{I}) \cdot S \rightsquigarrow \Gamma' \vdash S'} \text{ modifier elimination}$$

Figure 3.8: Operational semantics of basic L-QPL statements, continued

the new sub-branch is labelled with 00. These labels mean the **bit** is 0 with 100% probability and the **qubit** is $1.0\,|0\rangle + 0.0\,|1\rangle$.

Delete bit and delete qubit

On the quantum stack S, deletion removes the top node x, merging the appropriate branches below x. Deletion of a **bit** merges both the 0 and 1 branches. Deletion of a **qubit** merges only the 00 and 11 branches.

Measure

This statement applies separate groups of statements to the 00 and 11 sub-stacks of a **qubit** node, q. After applying those statements, the node q is discarded.

Compose

Composition of the operations on a quantum circuit are done in left to right order of their application.

Control by bit

When the statement $\mathcal{I}_b$ is controlled by the **bit** z, the effect is to apply $\mathcal{I}_b$ only to the 1 sub-branch of z.

Control by qubit

If the statement $\mathcal{I}_q$ is controlled by the **qubit** w, different operations are applied to each of w's sub-branches. The modifiers `IdOnly`, `Left` and `Right` are used: (`IdOnly` $\mathcal{I}_q$) is applied to the the 00 sub-branch of w; (`Left` $\mathcal{I}_q$) to the 10 sub-branch; (`Right` $\mathcal{I}_q$) to the 01 sub-branch and the unmodified statement, $\mathcal{I}_q$, is applied to the 11 sub-branch.

Unitary transforms

Given the unitary transform U and a quantum stack S having the **qubit** q at the top, transforming q by U is defined in terms of matrix multiplication. Note that while figure 3.8 only gives the operational semantics for a single **qubit** transform, the same process is applied for higher degree transforms. A quantum stack with two **qubit**s at its top may be converted to the 4×4 density matrix of those two **qubit**s in the obvious way. A similar process may be used to extend the operation to more **qubit**s.

Modifiers of unitary transforms

With U a unitary transform, the three rules for `IdOnly` U, `Left` U and `Right` U are given in terms of the appropriate matrix multiplications. As above, these may be extended to multiple **qubit** transformation in the obvious way.

Modifiers of statements other than unitary transforms

Modifiers have no effect on any statement of L-QPL except for unitary transformations.

Combinations of modifiers: reduction and elimination

When controlling by multiple **qubit**s, it is possible to have more than one modifier applied to a statement. The modifier reduction rule states that applying the same modifier twice in a row has the same effect as applying it once.

Modifier elimination, however, reduces the list of modifiers to the single modifier `IdOnly` whenever two different modifiers follow one another in the modifier list.

3.5 Example — putting it all together on teleport

Circuit diagrams are interpreted in the algebra of quantum stacks. The preceding sections have given a translation of quantum circuits to L-QPL programs and an operational semantics of L-QPL programs. From this, we have an operational semantics of quantum circuits.

3.5.1 Translation of teleport to basic L-QPL

The quantum circuit for teleportation makes a variety of assumptions, including that the two starting `qubit`s are placed in an EPR state and that the target `qubit` is in some unknown state. Figure 3.9 shows the standard teleport circuit assuming that v is in some unknown state, but that the `qubit`s *alice* and *bob* still need to be placed into an EPR state.

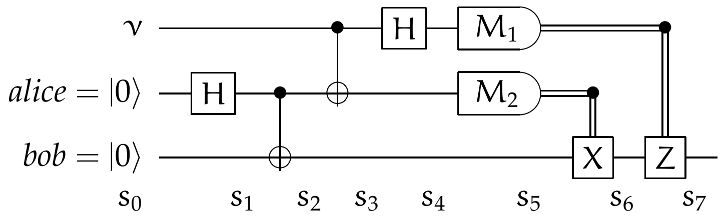


Figure 3.9: Preparation and quantum teleportation

The translation of this circuit to a basic L-QPL program is straightforward and is given in figure 3.10 on the next page.

3.5.2 Unfolding the operational semantics on teleport

This section will step through the program, using the operational semantics to show the changes to the quantum stack at each of the labelled stages. Input to

$$
\begin{array}{lr}
\mathit{alice} = |0\rangle\,; & \\
\mathit{bob} = |0\rangle\,; & (\mathcal{I}_{s_0}) \\
\mathsf{Had}\,\mathit{alice}; & (\mathcal{I}_{s_1}) \\
(\mathsf{Not}\,\mathit{bob}) \Leftarrow \mathit{alice}; & (\mathcal{I}_{s_2}) \\
\mathsf{Not}\,\mathit{alice} \Leftarrow \nu; & (\mathcal{I}_{s_3}) \\
\mathsf{Had}\,\nu; & (\mathcal{I}_{s_4}) \\
\quad \text{meas}\,\nu : & \\
\qquad |0\rangle => \nu_b = 0\ ; & \\
\qquad |1\rangle => \nu_b = 1 & \\
\quad \text{meas}\,\mathit{alice} : & \\
\qquad |0\rangle => \mathit{alice}_b = 0\ ; & (\mathcal{I}_{s_5}) \\
\qquad |1\rangle => \mathit{alice}_b = 1 & \\
\mathsf{Not}\,\mathit{bob} \Leftarrow \mathit{alice}_b; & \\
\text{disc}\,\mathit{alice}_b; & (\mathcal{I}_{s_6}) \\
\mathsf{Z}\,\mathit{bob} \Leftarrow \nu_b; & \\
\text{disc}\,\nu_b & (\mathcal{I}_{s_7})
\end{array}
$$

Figure 3.10: Basic L-QPL program for teleport

the algorithm is assumed to be a single **qubit**, ν, which is in an unknown state, $\alpha\,|0\rangle + \beta\,|1\rangle$. Therefore, the starting stack is

$$\nu \left\{ \begin{array}{ll} 00 \rightarrow \alpha\overline{\alpha}; & 01 \rightarrow \alpha\overline{\beta} \\ 10 \rightarrow \beta\overline{\alpha}; & 11 \rightarrow \beta\overline{\beta} \end{array} \right\}.$$

When the instructions $\mathfrak{I}_{s_0}$ are applied, the stack will change to

$$\mathit{bob}\{00 \rightarrow \mathit{alice}\{00 \rightarrow \nu \left\{ \begin{array}{ll} 00 \rightarrow \alpha\overline{\alpha}; & 01 \rightarrow \alpha\overline{\beta} \\ 10 \rightarrow \beta\overline{\alpha}; & 11 \rightarrow \beta\overline{\beta} \end{array} \right\}\}\}.$$

The next instruction, $\mathfrak{I}_{s_1}$, applies the Hadamard transform to *alice*. *alice* is first rotated to the top and then the transformation is applied. The stack will then be in the state

$$\mathit{alice} \left\{ \begin{array}{l} 00 \rightarrow \mathit{bob}\{00 \rightarrow \nu \left\{ \begin{array}{ll} 00 \rightarrow .5 \times \alpha\overline{\alpha}; & 01 \rightarrow .5 \times \alpha\overline{\beta} \\ 10 \rightarrow .5 \times \beta\overline{\alpha}; & 11 \rightarrow .5 \times \beta\overline{\beta} \end{array} \right\}\} \\ 01 \rightarrow \mathit{bob}\{00 \rightarrow \nu \left\{ \begin{array}{ll} 00 \rightarrow .5 \times \alpha\overline{\alpha}; & 01 \rightarrow .5 \times \alpha\overline{\beta} \\ 10 \rightarrow .5 \times \beta\overline{\alpha}; & 11 \rightarrow .5 \times \beta\overline{\beta} \end{array} \right\}\} \\ 10 \rightarrow \mathit{bob}\{00 \rightarrow \nu \left\{ \begin{array}{ll} 00 \rightarrow .5 \times \alpha\overline{\alpha}; & 01 \rightarrow .5 \times \alpha\overline{\beta} \\ 10 \rightarrow .5 \times \beta\overline{\alpha}; & 11 \rightarrow .5 \times \beta\overline{\beta} \end{array} \right\}\} \\ 11 \rightarrow \mathit{bob}\{00 \rightarrow \nu \left\{ \begin{array}{ll} 00 \rightarrow .5 \times \alpha\overline{\alpha}; & 01 \rightarrow .5 \times \alpha\overline{\beta} \\ 10 \rightarrow .5 \times \beta\overline{\alpha}; & 11 \rightarrow .5 \times \beta\overline{\beta} \end{array} \right\}\} \end{array} \right\}.$$

The preparation of EPR state is completed by doing a controlled-Not between *alice* and *bob*. This results in

$$alice\left\{\begin{array}{l}00\rightarrow bob\{00\rightarrow \nu\left\{\begin{array}{ll}00\rightarrow .5\times\alpha\overline{\alpha}; & 01\rightarrow .5\times\alpha\overline{\beta}\\ 10\rightarrow .5\times\beta\overline{\alpha}; & 11\rightarrow .5\times\beta\overline{\beta}\end{array}\right\}\}\\ 01\rightarrow bob\{10\rightarrow \nu\left\{\begin{array}{ll}00\rightarrow .5\times\alpha\overline{\alpha}; & 01\rightarrow .5\times\alpha\overline{\beta}\\ 10\rightarrow .5\times\beta\overline{\alpha}; & 11\rightarrow .5\times\beta\overline{\beta}\end{array}\right\}\}\\ 10\rightarrow bob\{01\rightarrow \nu\left\{\begin{array}{ll}00\rightarrow .5\times\alpha\overline{\alpha}; & 01\rightarrow .5\times\alpha\overline{\beta}\\ 10\rightarrow .5\times\beta\overline{\alpha}; & 11\rightarrow .5\times\beta\overline{\beta}\end{array}\right\}\}\\ 11\rightarrow bob\{11\rightarrow \nu\left\{\begin{array}{ll}00\rightarrow .5\times\alpha\overline{\alpha}; & 01\rightarrow .5\times\alpha\overline{\beta}\\ 10\rightarrow .5\times\beta\overline{\alpha}; & 11\rightarrow .5\times\beta\overline{\beta}\end{array}\right\}\}\end{array}\right\}.$$

This is followed by the application of the controlled-Not to ν and *alice*, giving

$$\nu\left\{\begin{array}{l}00\rightarrow alice\left\{\begin{array}{ll}00\rightarrow bob\{00\rightarrow .5\times\alpha\overline{\alpha}\}; & 01\rightarrow bob\{10\rightarrow .5\times\alpha\overline{\alpha}\}\\ 10\rightarrow bob\{01\rightarrow .5\times\alpha\overline{\alpha}\}; & 11\rightarrow bob\{11\rightarrow .5\times\alpha\overline{\alpha}\}\end{array}\right\}\\ 01\rightarrow alice\left\{\begin{array}{ll}00\rightarrow bob\{01\rightarrow .5\times\alpha\overline{\beta}\}; & 01\rightarrow bob\{11\rightarrow .5\times\alpha\overline{\beta}\}\\ 10\rightarrow bob\{00\rightarrow .5\times\alpha\overline{\beta}\}; & 11\rightarrow bob\{10\rightarrow .5\times\alpha\overline{\beta}\}\end{array}\right\}\\ 10\rightarrow alice\left\{\begin{array}{ll}00\rightarrow bob\{00\rightarrow .5\times\beta\overline{\alpha}\}; & 01\rightarrow bob\{10\rightarrow .5\times\beta\overline{\alpha}\}\\ 10\rightarrow bob\{01\rightarrow .5\times\beta\overline{\alpha}\}; & 11\rightarrow bob\{11\rightarrow .5\times\beta\overline{\alpha}\}\end{array}\right\}\\ 11\rightarrow alice\left\{\begin{array}{ll}00\rightarrow bob\{11\rightarrow .5\times\beta\overline{\beta}\}; & 01\rightarrow bob\{01\rightarrow .5\times\beta\overline{\beta}\}\\ 10\rightarrow bob\{10\rightarrow .5\times\beta\overline{\beta}\}; & 11\rightarrow bob\{00\rightarrow .5\times\beta\overline{\beta}\}\end{array}\right\}\end{array}\right\}.$$

The Hadamard transform is now applied to ν. Unfortunately, the notation used to this point is somewhat too verbose to show the complete resulting quantum stack. It begins with:

$$\nu\{00 \to alice\{00 \to bob \left\{\begin{array}{ll} 00 \to \frac{\alpha\overline{\alpha}}{4}; & 01 \to \frac{\alpha\overline{\beta}}{4} \\ 10 \to \frac{\beta\overline{\alpha}}{4}; & 11 \to \frac{\beta\overline{\beta}}{4} \end{array}\right\} \ldots\} \ldots\}$$

$$\vdots$$

In the next stage, *alice* is measured and a Not transform is applied to *bob* depending on the result of the measure. The bit $alice_b$ is then discarded, resulting in

$$\nu \left\{\begin{array}{l} 00 \to bob \left\{\begin{array}{ll} 00 \to \alpha\overline{\alpha}/2; & 01 \to \alpha\overline{\beta}/2 \\ 10 \to \beta\overline{\alpha}/2; & 11 \to \beta\overline{\beta}/2 \end{array}\right\} \\ 01 \to bob \left\{\begin{array}{ll} 00 \to \alpha\overline{\alpha}/2; & 01 \to -\alpha\overline{\beta}/2 \\ 10 \to \beta\overline{\alpha}/2; & 11 \to -\beta\overline{\beta}/2 \end{array}\right\} \\ 10 \to bob \left\{\begin{array}{ll} 00 \to \alpha\overline{\alpha}/2; & 01 \to \alpha\overline{\beta}/2 \\ 10 \to -\beta\overline{\alpha}/2; & 11 \to -\beta\overline{\beta}/2 \end{array}\right\} \\ 11 \to bob \left\{\begin{array}{ll} 00 \to \alpha\overline{\alpha}/2; & 01 \to -\alpha\overline{\beta}/2 \\ 10 \to -\beta\overline{\alpha}/2; & 11 \to \beta\overline{\beta}/2 \end{array}\right\} \end{array}\right\}.$$

Then, ν is measured and a RhoZ transform is applied when the resulting **bit** is 1. The **bit** ν_b is then discarded. The resulting quantum stack is

$$bob \left\{\begin{array}{ll} 00 \to \alpha\overline{\alpha}; & 01 \to \alpha\overline{\beta} \\ 10 \to \beta\overline{\alpha}; & 11 \to \beta\overline{\beta} \end{array}\right\},$$

which shows *bob* has been changed to the same state ν was in at the start of the program.

3.6 Semantics of datatypes

To this point, this chapter has only considered quantum stacks with two node types, **bit** and **qubit**. This section will add nodes for constructed datatypes and probabilistic classical data and statements that operate on these nodes.

Constructed datatypes will allow the addition of algebraic datatypes to our quantum stacks. This includes sum, product, and recursive data types. For example, types such as *List* (a recursive type), *Either* (a sum type) and *Pair* (a product type) are now definable.

3.6.1 Statements for constructed datatypes and classical data

Two new statements are required to implement datatypes on the quantum stack: *node construction* and a *case* statement. A third statement *discard data* is also added. *Discard data* is semantic sugar for doing a *case* without any dependent statements.

With the addition of classical data, arithmetic and logical *expressions* are added to L-QPL. The standard *new* and *discard* statements as well as a *use* statement are added to the language. The *if-else* statement is added to allow choices based on classical expressions.

The judgements for expressions are given in figure 3.11 and for the new statements in figure 3.12. Judgements for expressions use the notation

$$\Gamma_c|\Gamma \Vdash e :: \tau_C,$$

which means that given the context Γ_c of classical variables in the porch and the context Γ of quantum variables, e is a valid expression, with the classical type τ_C. Judgements for statements will also carry the classical context in the porch as above. Statements introduced to this point have not been affected by the classical context, nor have their operational semantics affected the classical context. The new format of judgements for statements will be:

$$\Gamma_c|\Gamma; \Gamma_c'|\Gamma' \Vdash \mathcal{J}.$$

For the operational semantics, the classical context must also now be carried in the porch. Additionally, the data in the classical context is not stored in the quantum stack, but in an adjacent standard stack. The full format of the semantics of a statement operating on the stacks will now be:

$$\Gamma_c|\Gamma \vdash \mathcal{J} \cdot (S, C) \rightsquigarrow \Gamma_c'|\Gamma' \vdash (S', C')$$

In many cases, the classical portions are not involved in the semantics and thus may be elided, resulting in our original syntax for the operational semantics.

$$\frac{n \in \mathbb{Z}}{\Gamma_c|\Gamma \Vdash\!\dashv n :: \mathit{Int}}\ \text{integers} \qquad \frac{n \in \{\text{true, false}\}}{\Gamma_c|\Gamma \Vdash\!\dashv b :: \mathit{Boolean}}\ \text{Booleans}$$

$$\frac{\Gamma_c|\Gamma \Vdash\!\dashv e_1 \quad \Gamma_c|\Gamma \Vdash\!\dashv e_2}{\Gamma_c|\Gamma \Vdash\!\dashv (e_1 \text{ op } e_2) :: \tau_{op}}\ \text{operations}$$

$$\frac{}{(n :: \tau_C), \Gamma_c|\Gamma \Vdash\!\dashv n :: \tau_C}\ \text{identifiers}$$

Figure 3.11: Judgements for expressions in L-QPL.

3.6.2 Operational semantics

Semantics for datatype statements

In the quantum stack, a datatype node of type τ has multiple branches, where each branch is labelled with a datatype constructor C of τ and the *bound nodes* for

$$\frac{}{(v_1 :: \tau_0, \ldots v_n :: \tau_n), \Gamma; (x :: \tau(C)), \Gamma \Vdash x = C(v_1, \ldots, v_n)}\ \text{new data}$$

$$\frac{\{\Gamma; \Gamma' \Vdash \mathcal{J}_i\}_i}{(nd :: \tau), \Gamma; \Gamma' \Vdash \left\{ \begin{array}{l} \text{case } nd : \\ \{\ C_i(v_{ij}) \Rightarrow \mathcal{J}_i\}_{C_i \in \tau(C_i)} \end{array} \right\}}\ \text{case}$$

$$\frac{}{(nd :: \tau), \Gamma; \Gamma \Vdash \text{disc } nd}\ \text{discard data}$$

$$\frac{\Gamma_c | \Gamma \Vdash\!\!\dashv e}{\Gamma_c | \Gamma; \Gamma_c | (x :: \tau_C), \Gamma \Vdash x = e}\ \text{new classical}$$

$$\frac{(n :: \tau_C), \Gamma_c | \Gamma; \Gamma_c | \Gamma' \Vdash \mathcal{J}}{\Gamma_c | (n :: \tau_C), \Gamma; \Gamma_c | \Gamma' \Vdash \{\text{use } n :\ \{\mathcal{J}\}\}}\ \text{use}$$

$$\frac{\{\Gamma_c | \Gamma \Vdash\!\!\dashv e_i\text{::Boolean}\}_{i=0,\ldots,n-1} \quad \{\Gamma_c | \Gamma; \Gamma_c^i | \Gamma' \Vdash \mathcal{J}_i\}_{i=0,\ldots,n}}{\Gamma_c | \Gamma; \Gamma_c | \Gamma' \Vdash \left\{ \begin{array}{l} \text{if } e_0 \Rightarrow \mathcal{J}_0 \\ \{\ e_i \Rightarrow \mathcal{J}_i\}_i \\ \text{else } \Rightarrow \mathcal{J}_n \end{array} \right\}}\ \text{if-else}$$

$$\frac{}{\Gamma_c | (n :: \tau_C), \Gamma; \Gamma_c' | \Gamma' \Vdash \text{disc } n}\ \text{discard classical}$$

Figure 3.12: Judgements for datatype and classical data statements.

C. Constructors may require 0 or more bound nodes.

The *case* statement provides dependent statements to be executed for each of the branches of a datatype node d. Upon completion of the case, the node d is no longer available.

In discussing datatypes, the notation θ_z is used for creating a "fresh" variable. The notation $\tau(Cns)$ for the type of the constructor Cns.

The operational semantics for the datatype statements are given in figure 3.13. As *new data* and *discard data* have no effect on the classical context, the original syntax of the semantics statements is retained for these two statements. The *case* statement, however, does affect the classical context in that the classical context is reset at the beginning of execution of each set of dependent statements and at completion of the statement. Therefore, the semantics for *case* will use the full syntax introduced in this section.

$$\frac{(x_i{::}\tau_i)_{i=1,\ldots m}, \Gamma \vdash ([\theta_{x_1}/x_1];\ldots;[\theta_{x_m};x_m]) \cdot S \rightsquigarrow (\theta_{x_i}{::}\tau_i)_{i=1,\ldots m}, \Gamma' \vdash S'}{\begin{array}{l}(x_i{::}\tau_i)_{i=1,\ldots m}, \Gamma \vdash (nd = Cns(x_1,\ldots,x_m)) \cdot S \rightsquigarrow \\ \qquad (nd{::}\tau(Cns)), (\theta_{x_i}{::}\tau_i)_{i=1,\ldots m}, \Gamma' \vdash nd\{Cns(\theta_{x_1},\ldots,\theta_{x_m}) \to S'\}\end{array}} \text{ new data}$$

$$\frac{\{\Gamma \vdash (\{\text{disc } x_{ij}\}_j) \cdot S_i \rightsquigarrow \Gamma' \vdash S_i'\}_i}{(nd :: \tau), \Gamma \vdash \text{disc } nd \cdot nd\{C_i(x_{ij}) \to S_i\} \rightsquigarrow \Gamma' \vdash \sum S_i'} \text{ discard data}$$

$$\frac{\{\Gamma_c|\Gamma \vdash ([v_{ij}/x_{ij}])_j; \mathcal{J}_i \cdot (S_i, C) \rightsquigarrow \Gamma_c'|\Gamma' \vdash (S_i', C_i)\}_i}{\begin{array}{l}\Gamma_c|(nd :: \tau), \Gamma \vdash \left\{\begin{array}{l}\text{case } nd : \\ \{\ Cns_i(v_{ij}) => \mathcal{J}_i\}\end{array}\right\} \cdot (nd\{Cns_i(x_{ij}) \to S_i\}, C) \rightsquigarrow \\ \qquad\qquad \Gamma_c|\Gamma' \vdash (\sum S_i', C)\end{array}} \text{ case}$$

Figure 3.13: Operational semantics for datatype statements

Classical data

Adding probabilistic classical data led to adding a second data structure for the operational semantics. In order to perform standard arithmetic and logical (or other classically defined) operations on this data, a *classical stack* is paired with the quantum stack.

Classical data nodes extend the concept of probabilistic `bit`s to other types. For example, a node c could hold an integer that had value 7 with probability .3, 5 with probability .2 and 37 with probability .5. A probabilistic node is restricted to items of all the same type, e.g., all integers, all floats.

The classical stack is used to operate on non-probabilistic classical data and will interact with the classical nodes via *classical construction* and *use*.

The *use* statement had a set of dependent statements, $\mathcal{I}$, which are executed for each sub-stack S_i of a a classical node c. Prior to executing $\mathcal{I}$, the classical value k_i which labels the sub-branch S_i is moved onto the top of the classical stack. When the statements $\mathcal{I}$ have been executed for each of c's sub-stacks, c is removed and the resulting sub-branches are added together.

The classical stack has standard arithmetic and logic operations defined on it. The *if - else* statement is defined to execute various statements depending upon classical expressions.

The judgements for the classical note construction, discarding, use and interaction with the classical stack are shown in figure 3.14.

An important point to note in the transitions for the `use` statement is that the classical stack is *reset* at the beginning of each execution and in the final result. This enforces a block-like scoping on the classical stack. This also applies to

$$\frac{\Gamma_c|\Gamma \Vdash e}{\Gamma_c|\Gamma \vdash (n = e) \cdot (S, C) \rightsquigarrow \Gamma_c|(n{::}\tau_C), \Gamma \vdash (n\{e \to S\}, C)} \text{ new classical}$$

$$\frac{\{(n{::}\tau_C), \Gamma_c|\Gamma \vdash \mathcal{J} \cdot (S_i, k_i : C) \rightsquigarrow \Gamma_c'|\Gamma' \vdash (S_i', C_i')\}_i}{\Gamma_c|(n{::}\tau_C), \Gamma \vdash \{\text{use } nd : \{\mathcal{J}\}\} \cdot (nd\{k_i \to S_i\}, C) \rightsquigarrow \Gamma_c|\Gamma' \vdash (\sum S_i', C)} \text{ use}$$

$$\frac{}{\Gamma_c|(n{::}\tau_C), \Gamma \vdash \text{disc } n \cdot (nd\{cv_i \to S_i\}, C) \rightsquigarrow \Gamma_c|\Gamma \vdash (\sum S_i, C)} \text{ discard classical}$$

$$\frac{e_k = \texttt{True} \quad \{e_j = \texttt{False}\}_{j<k} \quad \Gamma_c|\Gamma \vdash \mathcal{J}_k \cdot (S, C) \rightsquigarrow \Gamma_c^k|\Gamma' \vdash (S', C_k)}{\Gamma_c|\Gamma \vdash \left\{ \begin{array}{l} \text{if } e_0 => \mathcal{J}_0 \\ \{ e_i => \mathcal{J}_i\}_i \\ \text{else } => \mathcal{J}_n \end{array} \right\} \cdot (S, C) \rightsquigarrow \Gamma_c^k|\Gamma' \vdash (S', C_k)} \text{ if-else, } e_k \text{ true}$$

$$\frac{\{e_j = \texttt{False}\}_{j=0,\dots,n-1} \quad \Gamma_c|\Gamma \vdash \mathcal{J}_n \cdot (S, C) \rightsquigarrow \Gamma_c^n|\Gamma' \vdash (S', C_n)}{\Gamma_c|\Gamma \vdash \left\{ \begin{array}{l} \text{if } e_0 => \mathcal{J}_0 \\ \{ e_i => \mathcal{J}_i\}_i \\ \text{else } => \mathcal{J}_n \end{array} \right\} \cdot (S, C) \rightsquigarrow \Gamma_c^n|\Gamma' \vdash (S', C_n)} \text{ if-else, else}$$

$$\frac{v_i \text{ op}_c\, v_j = v_k}{i :: \tau_1, j :: \tau_2, \Gamma_c|\Gamma \vdash (\text{capp op}_c) \cdot (S, v_i : v_j : C) \rightsquigarrow k :: \tau_3, \Gamma_c|\Gamma \vdash (S, v_k : C)} \text{ binary op}$$

Figure 3.14: Operational semantics for classical data statements

`measure` and `case`. The revised judgement for measure is given in figure 3.15.

$$\frac{\begin{array}{c}\Gamma_c|\Gamma \vdash I_0 \cdot (S_{00}, C) \rightsquigarrow \Gamma_c'|\Gamma' \vdash (S_0, C_0)\\ \Gamma_c|\Gamma \vdash I_1 \cdot (S_{11}, C) \rightsquigarrow \Gamma_c''|\Gamma' \vdash (S_1, C_1)\end{array}}{\Gamma_c|\Gamma \vdash \left\{\begin{array}{c}\text{meas q :}\\ |0\rangle \Rightarrow I_0\\ |1\rangle \Rightarrow I_1\end{array}\right\} \cdot (q\{ij \to S_{ij}\}, C) \rightsquigarrow \Gamma_c|\Gamma' \vdash ((S_0 + S_1), C)}\ \text{measure}'$$

Figure 3.15: Revised semantics for measure with classical stack.

3.7 Semantics of recursion

This section will extend L-QPL with statements for recursion and provide the operational semantics for them.

3.7.1 Statements for recursion

The only statements added for recursion are the *proc* and *call* statements. *Proc* is used to create a subroutine that is available for the *call* statement.

All *proc* statements are global in scope to a program, hence a subroutine may *call* itself or be called by other subroutines defined elsewhere in a program. The judgements for these two new statements are given in figure 3.16 on the next page. The judgements reuse the notation introduced earlier of $\widetilde{z_j}$ signifying a vector of elements.

$$\frac{}{\Gamma_c|\Gamma \Vdash \mathrm{prc}_x :: (\widetilde{c_i}{:}\widetilde{\tau_{Ci}}|\widetilde{qv_j}{:}\widetilde{\tau_j};\widetilde{r_k}{:}\widetilde{\tau_k}) = \mathcal{I}}\ \text{proc}$$

$$\frac{\{\Gamma_c|(\widetilde{qv_j} :: \widetilde{\tau_j})\Gamma \Vdash\!\dashv e_i\}_i}{\Gamma_c|(\widetilde{qv_j} :: \widetilde{\tau_j})\Gamma \Vdash (\widetilde{r_k}) = \mathrm{prc}_c(\widetilde{e_i}|\widetilde{qv_j})}\ \text{call}$$

Figure 3.16: Judgements for formation of proc and call statements

Notation	**Meaning**
$[z'/z]\Gamma \vdash [z'/z]S$	Substituting z' for z in the context and stack.
$\Gamma_c\|\Gamma \vdash \mathcal{I} \cdot (S, C) \overset{d}{\rightsquigarrow} \Gamma_c'\Gamma' \vdash (S', C')$	Application of the statements $\Gamma_c\|\Gamma; \Gamma_c'\Gamma' \Vdash \mathcal{I}$ to to the quantum stack / classical stack pair (S, C) in classical context Γ_c and context Γ results in the quantum stack / classical stack pair (S', C') in classical context Γ_c' and context Γ' *at the* dth *iteration*.

Table 3.2: Notation used in operational semantics once iteration is added

3.7.2 Operational semantics for recursion

In order to provide an operational semantics for recursion, the quantum stacks will now be considered a *stream* (also known as an infinite list) of quantum stacks. The semantics as given previously in this chapter will stay the same, with the provision that all of these apply across the entire stream.

Additional notation is required for calling subroutines, re-namings (formal in and out parameters) and for applying a different transition dependent on the level in the stream. This additional notation is explained in table 3.2.

The operational semantics for recursion is given in figure 3.17 on the next page. Recursion is done in the semantics by first *diverging* at the head of the infinite list and then calling a subroutine once at the second element, calling twice in the third and so forth. In this way, the program gives a closer and closer approximation of the actual results the further one looks down the stream of quantum stacks.

$$\frac{}{(x{::}T),\Gamma \vdash [x'/x]\cdot x\{v_i \to S_i\}^t \rightsquigarrow (x'{::}T),\Gamma \vdash x'\{v_i \to S_i\}^t}\ \text{rename}$$

$$\frac{\mathrm{prc}_a :: (\widetilde{c_i{:}\tau_{C\,i}}|\widetilde{z_j{:}\tau_j};\widetilde{w_k{:}\tau_k}) = \mathcal{I}_a}{\Gamma_c|(\widetilde{z_j} :: \widetilde{\tau_j})\Gamma \vdash (\widetilde{w_k}) = \mathrm{prc}_a(\widetilde{c_i}|\widetilde{z_j})\cdot(S,C) \overset{0}{\rightsquigarrow} \Gamma_c|\emptyset \vdash (\emptyset^0, C)}\ \text{base call}$$

$$\frac{\begin{array}{c}\Gamma_c|\Gamma \vdash \{[z_j'/z_j]\}_j;\mathcal{I}_a;\{[w_k/w_k']\}_k\cdot(S,C) \overset{d}{\rightsquigarrow} \Gamma_c'|(\widetilde{w_k} :: \widetilde{\tau_k})\Gamma' \vdash (S',C') \\ \mathrm{prc}_a :: (\widetilde{c_i'{:}\tau_{C\,i}}|\widetilde{z_j'{:}\tau_j};\widetilde{w_k'{:}\tau_k}) = \mathcal{I}_a \\ \{\Gamma_c|(\widetilde{z_j} :: \widetilde{\tau_j})\Gamma \Vdash e_i{::}\tau_{C\,i}\}_i\end{array}}{\Gamma_c|(\widetilde{z_j} :: \widetilde{\tau_j})\Gamma \vdash (\widetilde{w_k}) = \mathrm{prc}_a(\widetilde{c_i}|\widetilde{z_j})\cdot(S,C) \overset{d+1}{\rightsquigarrow} \Gamma_c|\Gamma' \vdash (S',C)}\ \text{call } d+1$$

Figure 3.17: Operational semantics for recursion

Chapter 4

An Informal Introduction to Linear QPL

4.1 Introduction to L-QPL

This chapter presents an overview of the linear quantum program language. Explanations of L-QPL programs, statements, and expressions are given. The explanations are done using expository presentation with many short examples to illustrate relevant points.

L-QPL is a language for experimenting with quantum algorithms. The language provides an expressive syntax for creating functions and working with different datatypes. L-QPL has `qubit`s as first class citizens of the language, together with quantum control. Classical operations and classical control are also available to work with classical data.

4.1.1 Linearity of L-QPL

The language L-QPL treats all quantum variables as *linear*. This means that any variable *may only be used once*. The primary reason for implementing this is the underlying aspect of linearity of quantum systems, as exemplified by the *no-duplication* rule which must be respected at all times. This allows us to provide compile-time checking that enforces this rule.

The compiler and language do provide ways to "ease the burden" of linear thinking. For example, function calls (sub-section 4.3.6 on page 85) provide a specialized syntax for variables which are both input and output to a function. The classical use statements (sub-section 4.3.5 on page 83) place values on to the classical stack where the values may be used multiple times.

```
1 #Import Prelude.qpl
2
3 len ::(listIn: List (a) ; length:Int) =
4 {   case listIn of
5      Nil => {length = 0}
6      Cons (_, tail) => {
7         tlLen := len(tail);
8         length = 1 + tlLen;  }
9 }
```

Figure 4.1: L-QPL code to return the length of the list

An example illustrating linearity is given in figure 4.1. In line 3, the function `len` is defined as taking one argument of type `List (a)` and returning a variable of type **`Int`**. The input only argument, `listIn`, *must be destroyed in the function*. When the case statement refers to `listIn`, the argument is destroyed, fulfilling the requirement of the function to do so.

4.2 L-QPL programs

L-QPL programs consist of combinations of functions and data definitions, with one special function named **`main`**. The functions and data definitions are *simultaneously declared* and so may be given in any order. The program will start executing at the **`main`** function.

A physical program will typically consist of one or more source files with the suffix `.qpl`. Each source file may contain functions and data definitions. It may also *import* the contents of other source files. The name of a source file is not significant in L-QPL. Common practice is to have one significant function per source file and then to import all these files into the source file containing the **`main`** function.

4.2.1 Data definitions

L-QPL provides the facilities to define datatypes with a syntax reminiscent of Haskell [28].

Natively, the language provides **`Int`**, **`Qbit`** and **`Bool`** types. **`Bool`** is the standard Boolean type with values `true` and `false`. **`Int`** is a standard 32-bit integer. **`Qbit`** is a single qubit.

In L-QPL both native types and other constructed datatypes may be used in the definition of constructed datatypes. These constructed datatypes may involve sums, products, singleton types and parametrization of the constructed type. For example, a type that is the sum of the integers and the Booleans can be declared as follows:

```
qdata Eitherib = {Left (Int) | Right (Bool)}
```

The above example also serves to illustrate the basic syntax of the data declaration.

Syntax of datatype declarations. Each datatype declaration must begin with the keyword **`qdata`**. This is followed by *the type name*, which must be an identifier starting with an uppercase letter. The type name may also be followed by any number of *type variables*. This is then followed by an equals sign and completed by a list of *constructors*. The list of constructors must be surrounded by braces and each constructor must be separated from the others by a vertical bar. Each constructor is followed by an optional parenthesized list of *simple types*. Each simple type is either a built-in type, (one of `Int`, `Bool`, `Qbit`), a type variable that was used in the type declaration, or another declared type, surrounded by parenthesis. L-QPL allows recursive references to the type currently being declared. All constructors must begin with an upper case letter. Constructors and types are in different namespaces, so it is legal to have the same name for both. For example:

```
qdata Record a b = {Record (Int, a, b)}
```

In the above type definition, the first `Record` is the type, while the second is the constructor. The triplet `(Int,a,b)` is the product of the type `Int` and the type variables `a` and `b`. Since constructors may reference their own type and other declared types, recursive data types such as lists and various types of trees may be created:

```
qdata List a   = {Nil       | Cons (a, List (a))}
qdata Tree a   = {Leaf  (a) | Br (Tree (a), Tree (a))}
qdata STree a  = {Tip       | Fork (STree (a), a, STree (a))}
qdata RTree a  = {Rnode (a, (List a))}
qdata Rose a   = {Rose  (List (a,Rose (a)))}
qdata Colour   = {Red       | Black}
qdata RBSet a  = {Empty     |
                  RBTip (Colour, RBSet (a), a, RBSet (a))}
```

4.2.2 Function definitions

Function definitions may appear in any order within a L-QPL source file.

Syntax of function definitions. The first element of a function definition is *the name*, an identifier starting with a lower case letter. This is always followed by a double semi-colon and *a signature*, which details the type and characteristics of input and output arguments. The final component of the function definition is *a body*, which is a block of L-QPL statements. Details of statements are given in section 4.3 on page 76.

Let us examine two examples of functions. The first, in figure 4.2 is a fairly standard function to determine the greatest common divisor of two integers.

```
gcd :: (a : Int, b: Int
        | ;  ans: Int) =
{ if b == 0 => {ans = a}
     a == 0 => {ans = b}
     a >= b =>
        {ans = gcd(b, a mod b)}
  else      =>
        {ans = gcd(a, b mod a)}
}
```

Figure 4.2: L-QPL function to compute the GCD

The first line has the name of the function, `gcd`, followed by the signature; the name and the signature are separated by a double colon. The signature is `(a :Int, b:Int |; ans:Int)`. This signature tells the compiler that `gcd` expects two input arguments, each of type `Int` and that they are *classical*. The compiler deduces this from the fact that they both appear before the '`|`'. In this case there are no quantum input arguments as there are no parameters between the '`|`' and the '`;`'. The last parameter tells us that this function returns one quantum item of type `Int`. Returned items are always quantum data.

The signature specifies variable names for the parameters used in the body. All input parameters are available as variable names in expression and statements. Output parameters are available to be assigned and, indeed, must be assigned by the end of the function.

The next example, in figure 4.3 highlights the linearity of variables in L-QPL. This function is used to create a list of **qubit**s corresponding to the **bit** representation of an input integer.

At line 1, the program uses the import command. `#Import` must have a file name directly after it. This command directs the compiler to stop reading from the current file and to read code in the imported file until the end of that file, after which it continues with the current file. The compiler will not reread the same file in a single compilation, and it will import from any file.

```
1  #Import Prelude.qpl
2  //No classical arguments
3  intToQbitList :: (n :Int ;  //Input a quantum int
4         nq : List (Qbit), n :Int) = // out-> qbit list , original int
5  { use n in
6    { if n == 0 => { nq = Nil; }
7          ((n mod 2) == 0) =>  //Even
8              {  n' = n >> 1;
9                 (nq', n') = intToQbitList(n');
10                nq = Cons(|0>, nq'); }
11     else => { n' = n >> 1;
12                intToQbitList(n'; nq',n');
13                nq = Cons(|1>, nq') };
14     n = n;
15   }
16 }
```

Figure 4.3: L-QPL function to create a **qubit** register

For example, consider a case with three source files, A, B and C. Suppose file A has import commands for both B and C, with B being imported first. Further suppose that file B imports C. The compiler will start reading A, suspend at the first import and start reading B. When it reaches B's `#Import` of C, it will suspend the processing of B and read C. After completing the read of C, the compiler reverts to processing B. After completing the read of B, the compiler does a final reversion and finishes processing A. However, when A's `#Import` of C is reached, the compiler will ignore this import as it keeps track of the fact C has already been read.

In the signature on lines 3-4, the function accepts one quantum parameter of type **`Int`**. It returns a **`Qbit`** list and an **`Int`**. The integer returned, in this program, is computed to have the same value as the one passed in. If this had not been specified in this way, *the integer would have been destroyed by the function*. Generally, any usage of a quantum variable destroys that variable.

In the body of the function, note the **`use`** `n` in the block of statements. This allows repeated use of the variable `n` at lines 6, 7, 8, 11 and 14. In these uses, `n` is a classical variable, no longer on the quantum stack. The last usage on line 14 where `n` is assigned to itself, returns `n` to the quantum world.

4.3 L-QPL statements

The L-QPL language has the following statements:

Assignment: The assign statement, e.g. `x=t;`

Classical control: The **`if`** - **`else`** statement.

Case: The **`case`** for operating on constructed data types.

Measure: The **`measure`** statement which measures a `qubit` and executes dependent statements.

Use: The **use** and classical assign statements which operate on classical data, moving it on to the classical stack for processing.

Function calls: The various ways of calling functions or applying transformations.

Blocks: A group of statements enclosed by '{' and '}'.

Quantum control: Control of statements by the <= qualifier.

Divergence: The **zero** statement.

Other: the **discard** statement.

4.3.1 Assignment statement

Assignments create variables. Typical examples of these are:

```
q1 = |0>;
i = 42;
bt1 = Br(Leaf (q1), Br (Leaf(|0>),Leaf(|1>)));
```

Here the first line creates a **qubit**, `q1`, with initial value $|0\rangle$. The second line creates an integer ,`i`, with the value 42. The last line creates a binary tree, `bt1`, with `q1` as its leftmost node, and the right node being a sub-tree with values $|0\rangle$ and $|1\rangle$ in the left and right nodes respectively. Note that after the execution of the third statement, the variable `q1` is no longer in scope so the name may be reused by reassigning some other value to it.

The variables on the left hand side of an assignment are always quantum variables.

Syntax of assignment statements. An assignment statement always begins with an *identifier*. This must be followed by a single equals sign and then an *expression*.

Identifiers in L-QPL must always start with a lower case letter.

Expressions are discussed below in section 4.4 on page 92

4.3.2 Classical control

Classical control provides a way to choose sets of instructions to execute based upon the values on the classical stack.

```
smallPrime :: (a : Int | ;
                isSmallPrime: Bool) =
{ if a =< 1 => {isSmallPrime = false}
     a == 2 || a == 3 || a == 5  =>
                {isSmallPrime = true}
     a == 4 => {isSmallPrime = false}
  else      => {isSmallPrime = false}
}
```

Figure 4.4: L-QPL program demonstrating **`if-else`**

The expressions in the selectors *must* be classical. This means they can only consist of operations on constants and classical identifiers. It is a semantic error to have an expression that depends on a quantum variable. For an example, see the code to determine if an input number is a small prime in figure 4.4.

Syntax of the `if - else` statement. The statement starts with the word *if*, followed by one or more *selectors*. Each selector is composed of a classical Boolean expression e_b, the symbols => and a dependent block. The statement is completed by a special selector where the Boolean expression is replaced with the word *else*.

In the list of e_i => b_i selectors, b_i is executed only when e_i is the first expression to evaluate to `true`. All others are skipped. The final grouping of **`else =>`** *block* is a default and will be executed when all the selector expressions in a list evaluate to `false`.

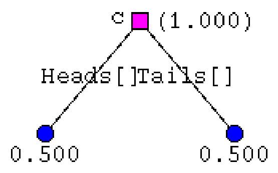
4.3.3 Measure statement

The **`measure`** statement performs a measurement of a **qubit** and executes code depending on the outcome. Currently in L-QPL, all measures are done with respect to the basis $\{|0\rangle, |1\rangle\}$. Referring to figure 4.5, there is a **`measure`** on line 5.

Consider the program in figure 4.5 which emulates a coin flip.

```
1  qdata Coin = {Heads | Tails}
2  cflip ::(; c:Coin) =
3  {  q = |0>;
4     Had q;
5     measure q of
6       |0> => {c = Heads}
7       |1> => {c = Tails}
8  }
9  main :: () =
10 {   c = cflip() ; }
```

(a) Coin flip code



(b) Stack machine state at end

Figure 4.5: L-QPL program to do a coin flip

In the function `cflip`, a **qubit** is prepared by initializing it to $|0\rangle$ and applying the Hadamard transform. This creates a **qubit** whose density matrix is $\begin{pmatrix} .5 & .5 \\ .5 & .5 \end{pmatrix}$. When this **qubit** is measured, it has a 50% chance of being 0 and an equal chance of being 1.

In the branches of the measure, different values are assigned to the return variable c. Each of these assignments happens with a probability of 50%. Once the measure statement is completed, the variable `c` will be `Heads` and `Tails` each with a probability of 50%. In the quantum stack machine this is represented as in sub-figure b of figure 4.5 on the preceding page.

This illustrates the largest difference between quantum and classical processing of choices. In classical programming languages, a choice such as a case type statement *will only execute the code on one of the branches of the case*. In L-QPL, *every* branch may be executed.

```
1 main :: () =
2 {   q = |0>;
3     Had q;
4     measure q of
5       |0> => {i = 0}
6       |1> => {i = 17};
7     q = |0>;
8 }
```

(a) Balanced creation

```
1 main :: () =
2 {   q = |0>;
3     Had q;
4     measure q of
5       |0> => {c = 0}
6       |1> => {d = 17};
7     q = |0>;
8 }
```

(b) Unbalanced creation

Figure 4.6: L-QPL programs contrasting creation

When writing the dependent blocks of **`measure`** (and **`case`** in sub-section 4.3.4 on the next page) variable creation must be the same in each dependent list of statements. The compiler will give you a semantic warning if a variable is created in one branch and not another.

For example, consider figure 4.6. In the left hand program on the **`measure`** starting at line 4, each branch creates a variable named '`i`'. This is legal and from line 7 forward, '`i`' will be available.

On the other hand, the measure in the right hand program starting in line 4 assigns to the variable '`c`' in the $|0\rangle$ branch and '`d`' in the $|1\rangle$ branch. At line 7, neither variable will be available. The compiler will give the warnings:

```
Warning:  Unbalanced creation, discarding c of type INT
Warning:  Unbalanced creation, discarding d of type INT
```

Syntax of the `measure` statement. This statement starts with the word *measure*, followed by a variable name, which must be of type `Qbit`. Next, the keyword *of* signals the start of the two case selections. The case selection starts with either $|0\rangle$ or $|1\rangle$, followed by => and the block of dependent statements.

Note that *both* case selections for a **qubit** must be present. However, it is permissible to not have any statements in a block.

4.3.4 Case statement

The `case` statement is used with any variable of a declared datatype.

```
qdata List a = {Nil | Cons (a, (List a))}

reverse  :: (l:List (a) ; rl: List (a))=
{  rev' (l, Nil ; rl) }

rev'  :: (l:List (a), accumIn:List(a) ; returnList: List (a))=
{  case l of
     Nil => {returnList = accumIn}
     Cons (hd, tail) =>
       {  acc = Cons(hd, accumIn);
          returnList = rev'(tail , acc)}
}
```

Figure 4.7: L-QPL program demonstrating `case`, a function to reverse a list.

In figure 4.7, the program declares the `List` data type, which is parametrized by one type variable and has two constructors: `Nil` which has no arguments and `Cons` which takes two arguments of types `a` and `List a` respectively.

The function `reverse` takes a list as an input argument and returns a single list, which is the original list in reverse order. Because of the linearity of the language the original input list is not in scope at the end of the function. The function `reverse` delegates to the function `rev'` which uses an accumulator to hold the list as it is reversed.

The case statement begins on line 7. For `Nil`, it assigns the accumulator to the return list. For `Cons`, it first adds the current element to the front of the accumulator list, then it uses a recursive call to reverse the tail of the original list with the new accumulator.

```
1  qdata TTree a = {Tip | Br ((TTree a), a, (TTree a)) | Node a}
2
3  treeMaxDepth :: (t:TTree (a); depth : Int) =
4  {   case t of
5        Tip => {depth = 0}
6        Node(_) => {depth = 1}
7        Br(t1, _, t2) => {
8          { j = treeMaxDepth(t1);
9            k = treeMaxDepth(t2);
10           use j,k;
11           if j > k => { depth = 1 + j}
12           else => {depth = 1 + k}
13         }
14 }
```

Figure 4.8: L-QPL program demonstrating **`case`**, a function to compute the max tree depth.

Considering the example in figure 4.8. `TTree` is a parametrized data type which depends on the type variable `a`. It has three constructors: `Tip` which takes no arguments; `Br` which takes three arguments of types `TTree a, a` and `TTree a`; and `Node` which takes one argument of type `a`.

In `treeMaxDepth`, the case statement on line 4 illustrates a "don't care" pattern for both the `Node` and `Br` constructors. This function returns the maximum depth of the `TTree` and actually discards the actual data elements stored at nodes.

Syntax of the `case` statement. This statement starts with the word *case*, followed by a variable of some declared type. Next, the keyword *of* signals the start of the case selections. The number of constructors in a type determine how many case selections the statement has. There is one selection for each constructor. Each case selection consists of a *constructor pattern*, a '=>' and dependent statements.

Constructor patterns are the constructor followed by a parenthesized list of variables and / or *don't care* symbols, '_'. Non-parametrized constructors appear without a list of variable names. The don't care symbol causes data to be discarded.

4.3.5 Use and classical assignment statements

The `use` statement is used with any variable of type `Int` or `Bool`. This statement has a single set of dependent statements. These may either be explicitly attached to the `use` statement or implicit. Implicit dependent statements are all the statements following the `use` until the end of the current block.

Classical assignment is grouped here as it is syntactic sugar for a `use` with implicit statements. This is illustrated in figure 4.9.

The three types of classical use are semantically equivalent, but do have different syntaxes as illustrated in figure 4.10 on the next page.

In sub-figure (a) of figure 4.10, the `use` statement starts on line 4. The next two statements are explicitly in its scope, which ends at line 6. In sub-figure (b) of the same figure, the `use` at line 4 is implicit. Its scope extends to line 7. Finally, in

```
⋮                  ⋮
                   i = exp;
i := exp;     ≡    use i;
s1;                s1;
⋮                  ⋮
```

Figure 4.9: Syntactic sugar for `use` / classical assignment

```
Br(t1, _, t2) =>
  { j = treeMaxDepth(t1);
    k = treeMaxDepth(t2);
    use j,k in
    {  if j > k => { depth = 1 + j}
       else => {depth = 1 + k} }
  }
```

(a) Explicit dependence

```
Br(t1, _, t2) =>
  {  j = treeMaxDepth(t1);
     k = treeMaxDepth(t2);
     use j,k;
     if j > k => { depth = 1 + j}
     else => {depth = 1 + k}
  }
```

(b) Implicit dependence

```
Br(t1, _, t2) =>
  { j := treeMaxDepth(t1);
    k := treeMaxDepth(t2);
    if j > k => { depth = 1 + j}
    else => {depth = 1 + k}
  }
```

(c) Classical assign

Figure 4.10: Fragments of L-QPL programs contrasting `use` syntax

sub-figure(c), the same effect is achieved with two classical assignments at lines 2 and 3. The scope of these assignments extend to line 6.

Unlike data types and `Qbit`s, which have a maximum number of sub-stacks, an `Int` has the potential to have an unbounded number of values and therefore sub-stacks. The dependent statements of the `use` statement are executed for *each* of these values.

To execute different pieces of code depending on the value, L-QPL provides the `if` - `else` statement as discussed in sub-section 4.3.2 on page 78.

Syntax of the `use` statement. This statement starts with the word `use`, followed by a list of variable names, which must be of type `Int` or `Bool`. If there is an explicit dependent block for the statement, it is given by the keyword `in` followed by the dependent block.

When the `use` statement is *not* followed by a dependent block, the rest of the statements in the enclosing block are considered in the scope of the `use`.

Classical assign syntax is a variable name, followed by the symbol ':=' followed by an expression. The expression must have type `Int` or `Bool`.

4.3.6 Function calls

Function calls include calling functions defined in programs and the predefined transforms. The list of predefined transforms valid in a L-QPL program are given in table 4.1 on the next page.

In addition to the predefined transforms, L-QPL allows you to prefix any of the predefined transformations with the string `Inv-` to get the inverse transforma-

L-QPL	A.K.A.	Matrix
`Not`	X, Pauli-X, ρ_X	$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$
`RhoY`	Y, Pauli-Y, ρ_Y	$\begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$
`RhoZ (=Rot(0))`	Z, Pauli-Z, ρ_Z	$\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$
`Had`	Had, H	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$
`Swap`	Swap	$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix}$
`Phase (=Rot(2))`	S, Phase $\sqrt{Z}$	$\begin{bmatrix}1 & 0\\0 & i\end{bmatrix}$
`T(=Rot(3))`	T, $\frac{\pi}{8}$, $\sqrt{S}$	$\begin{bmatrix}1 & 0\\0 & e^{i\pi/4}\end{bmatrix}$
`Rot(n)`	R_n, Rotation	$\begin{bmatrix}1 & 0\\0 & e^{i\pi/2^{n-1}}\end{bmatrix}$

Table 4.1: L-QPL transforms

tion. Controlled versions of all of these are available by using the quantum control construction as defined in sub-section 4.3.8 on page 90. For example, the **`Toffoli3`** gate as shown in table 2.1 on page 21 is simply a controlled-controlled-Not transformation.

The signatures of transforms are dependent upon the size of the associated matrix. A 2×2 matrix gives rise to the signature `(q:Qbit ; q:Qbit)`. In general, a $2^n \times 2^n$ matrix will require n **qubit**s in and out. The parametrized transforms such as **`Rot`** will require one or more integers as input.

Syntax of function calls

There are three different calling syntaxes for functions:

1. *Functional* — $(y_1, \ldots, y_m) = f(n_1, \ldots, n_k \,|\, x_1, \ldots, x_j)$.
2. *Procedural* — $f(n_1, \ldots, n_k \,|\, x_1, \ldots, x_j\,;\, y_1, \ldots, y_m)$.
3. *Transformational* — $f(n_1, \ldots, n_k)\ z_1\ z_2\ \ldots\ z_j$.

Functions with classical and quantum inputs. These functions may be called[1] in each of the three ways.

```
f ::(c1:Int,c2:Int, c3:Int | q1:Qbit, i1:Int ; a:Qbit, b:Int)
= { ... }
...
   (a,b) = f(c1,c2,c3 | q1,i2);
   f(c1,c2,c3 | q1,i2; a,b);
   f(c1,c2,c3) q i;
```

[1]A function call with a single unparenthesized variable appears on the left hand side of the equals are actually assignment statements. The right hand side of the assignment is a function expression. See sub-section 4.4.4 on page 95

When a function is called in the transformational syntax, as on the last line of the above code, the arguments (`q i` in the example) are both passed into the function as arguments and used as return variables. The arguments in the parenthesis (`c1,c2,c3` in our example), *must* be classical and a semantic error will result if a quantum variable is used. If the number of in and out quantum arguments are not the same or their types do not match, this syntax is not available.

Functions which have no quantum input arguments. Functions which have only classical inputs may be called in either the functional or procedural syntax. As there are no input quantum arguments, transformational syntax is not allowed.

```
g :: (c1:Int, c2:Int | ; r:Int, d:Int)
= { ... }
...
  (a,b) = g(c1,c2 |);
  g(c1,c2 | ; a,b);
```

Functions which have no classical input arguments. Functions having only quantum inputs may use all three syntaxes. In this case, where the classical variable list of arguments is empty, the "`|`" may be eliminated in procedural or functional calling, and the parenthesis may be eliminated in transformational calling.

```
h :: (q1:Int, q2:Int ; r:Int, d:Int)
= { ... }
...
  (a,b) = h(|c,d);
  (a,b) = h(c,d);
  h(|a,b ; c,d);
```

```
h(a,b; c,d);
h a b;
```

Linearity of function call arguments. Each input argument is no longer in scope after the function call. If the input is a simple identifier, the same identifier may be used in the output list of the function. The transformational syntax uses this technique to leave the variable names unchanged.

Syntactic forms of function and transform calls. In all three of the forms for function calling, the number and type of input arguments must agree with the definition of the function. Output identifiers must agree in number and their type is set according to the definition of the functions output parameters. Output variables are always quantum.

The *functional* syntax for function calling has three parts. The first part is a parenthesized list of variable names separated by commas. The parenthesized list is then followed by an equals sign. The right hand side consists of the function name followed by the parenthesized input arguments. The input arguments consists of two lists of arguments separated by '`|`'. The first list consists of the classical arguments, the second of the quantum arguments. Each argument must be a valid expression as defined in section 4.4 on page 92. If there are no classical arguments, the '`|`' is optional.

The *procedural* syntax for function calling starts with the function name, followed by a parenthesized grouping of input and output arguments. As in the functional form, the list of classical input arguments are separated from the quantum ones by '`|`', which may be eliminated when there are no classical arguments.

The input arguments are separated from the output arguments by `';'`.

The *transformational* syntax starts with the function name followed by a parenthesized list of classical expressions and then by a series of identifiers, separated by white space. A requirement for using this syntax is that the number and types of the input and output quantum arguments must be the same. The identifiers will be passed as input to the function and will be returned by the function. The parenthesis for the list of classical expressions may be eliminated when there are no classical parameters.

Function calls may also be expressions, which is discussed in sub-section 4.4.4 on page 95.

4.3.7 Blocks

Blocks are created by surrounding a list of statements with braces. A block may appear wherever a statement does. All of the "grouping" types of statements require blocks rather than statements as their group. See, for example, the discussion on **`case`** statements in sub-section 4.3.4 on page 81.

4.3.8 Quantum control

Quantum control provides a general way to create and use controlled unitary transforms in an L-QPL program. An example of quantum control is shown in the prepare and teleport functions in figure 4.11 on the following page.

Syntax of quantum control In L-QPL any statement, including block statements and procedure calls may be quantum controlled. Semantically, this will affect

```
prepare :: (;a:Qbit, b:Qbit)=
{ a = |0>; b=|0>;
  Had a;
  Not b ⇐ a;
}
teleport :: (n:Qbit, a:Qbit, b:Qbit ; b:Qbit) =
{ Not a ⇐ n ;
  Had n;
  measure a of
      |0> => {} |1> => {Not b};
  measure n of
      |0> => {}  |1> => {RhoZ b}
}
```

Figure 4.11: L-QPL program demonstrating quantum control

all transforms that occur within the controlled statement, including any of the transforms in a called function. The syntax is *statement* followed by the symbols <=, followed by a list of identifiers. These identifiers may be of any type, but are typically either **qubit**s or constructed data types with **qubit** elements, such as a `List(Qbit)`.

The identifiers that are controlling a statement can not be used in the statement. Controlling identifiers are exempt from linearity constraints in that they remain in scope after the quantum control construction.

The effect of the control is that all **qubit**s in the control list, or contained in items in that list are used to control all unitary transformations done in the controlled statement.

4.3.9 Divergence

To force the execution of a program to diverge, you can use the **`zero`** statement. This will set the probability of the values of a program to 0, and is interpreted as non-termination.

4.3.10 Discard

In some algorithms, such as in recursive functions when they process the initial cases of constructed data, the algorithm does not specify any action. In these cases, the programmer may need to examine the requirements of linearity with respect to any passed in parameters. Often, these will need to be explicitly discarded in base cases of such algorithms. This is done via the discard statement. An example may be seen in figure E.15 on page 234.

4.4 L-QPL expressions

Expressions in L-QPL are used in many of the statements discussed in section 4.3 on page 76. The four basic types of expressions are identifiers, constants, constructor expressions, and calling expressions. Arithmetic and logical combinations of classical constants and classical identifiers are allowed. As well, constructor and calling expressions often take lists of other expressions as arguments.

4.4.1 Constant expressions

The possible constant expressions in L-QPL are shown in table 4.2 on the next page. The category column in this table contains the word "Classical" when the constant may be used in arithmetic or Boolean expressions.

Expression	**Type**	**Category**
integer	`Int`	Classical
`true`	`Bool`	Classical
`false`	`Bool`	Classical
`|0>`	`Qbit`	Quantum
`|1>`	`Qbit`	Quantum

Table 4.2: Allowed constant expressions in L-QPL

4.4.2 Identifier expressions

These expressions are just the identifier name. While an identifier may be used wherever an expression is expected, the reverse is not true. As an example, in function calls, any of the input arguments may be expressions but the output arguments *must* be identifiers.

Identifier expressions may be either quantum or classical in nature. As discussed in sub-section 4.3.1 on page 77, an identifier is first created by an assignment statement. When initially created, the identifier is always quantum. Using it where a classical expression is required will result in an error. When it is desired to operate on an identifier classically, it must first be the object of a **`use`** statement. In all statements in the scope of that **`use`** statement the identifier will be considered classical. See sub-section 4.3.5 on page 83 for further information and examples.

4.4.3 Constructor expressions

These expressions are used to create new instances of declared data types. Consider this sample fragment of code.

```
qdata TTree a = {Tip | Br ((TTree a), a, (TTree a)) | Node a}
qdata List a = {Nil | Cons (a, (List a))}

qbtree = Br(Tip, |1>, Br (Node(|0>),|1>,Node(|1>)));
intlist = reverse(Cons(5,Cons(4,Cons(3,Cons(2,Cons(1,Nil))))));
```

These statements create a tree as in figure 4.12 and the list $[1,2,3,4,5]$. Compare the logical representation of `qbtree` as in figure 4.12 versus how it is stored in the quantum stack machine, shown in figure 4.13 The assignment statement

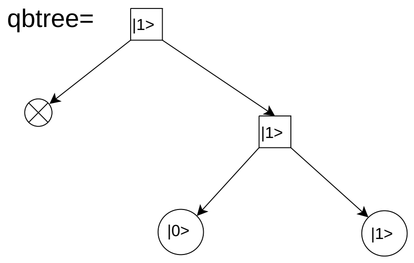


Figure 4.12: Pictorial representation of `qbtree`

which creates `qbtree` uses five constructor expressions. The second assignment statement, which creates `intlist` uses six constructor expressions and one function expression.

Constructor expressions either have no arguments (e.g. `Tip`, `Nil` above), or require a parenthesized list of expressions which agree in both number and type with the template supplied at the declaration of the type. These expressions are unrestricted otherwise. They may be constants, identifiers, other constructor expressions, expression calls or compound expressions. Any expressions that are

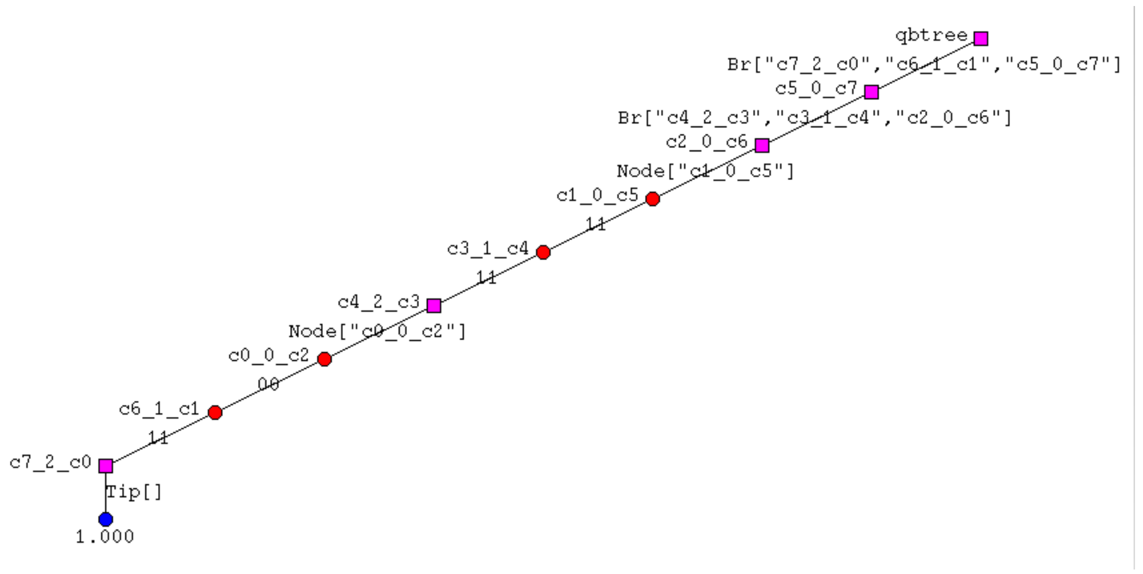


Figure 4.13: Quantum stack contents after creation of `qbtree`

classical in nature, such as constants, are upgraded to quantum automatically.

4.4.4 Function expressions

When a function returns a single value, it may be used in a function expression. The bottom two lines of listing below shows two examples of function expressions.

```
f ::(c1:Int,c2:Int, c3:Int | q1:Qbit, i1:Int ; out:Qbit)
= { ... }
...
   qout = f(c1,c2,c3 | q1,i2);
   qlist = Cons(f(1,2,3 | qout, 5),Nil);
```

In the first function expression, `f` is the right hand side of an assignment statement. The assignment statement creates the variable `qout` with the value returned by the function.

In the second function expression, `f` is the first argument of a constructor expression which will create a one element `List(`**`Qbit`**`)`. The constructor expression is part of an assignment statement which creates the variable `qlist` and sets it to the one element list. Note that due to linearity, the variable `qout` is no longer available after the second function expression.

A function expression is always a quantum expression, so it may only be used in those places where quantum expressions are allowed. Nesting of these calls inside constructor expressions, other function expressions and function calls is allowed.

```
1 qdata List a = {Nil | Cons (a, (List a))}
2
3 append :: (list1:List (a), list2:List (a); appendList: List (a))=
4 {   case list1 of
5       Nil => {appendList = list2}
6       Cons(hd, tail) => {
7           appendList = Cons(hd, append(tail, list2)); }
8 }
```

Figure 4.14: L-QPL code for appending two lists

In figure 4.14, line 7 shows the `append` being used as a function expression inside of a constructor expression.

Chapter 5

The quantum stack machine

5.1 Introduction to the quantum stack machine

The quantum stack machine provides an execution environment where quantum and classical data may be manipulated. The primary component of this machine is the *quantum stack*, which stores both quantum and probabilistic data.

The quantum stack has the same function as a classical stack in that it provides the basic operations and data structures required for quantum computation. Chapter 3 initially showed how quantum circuits can be interpreted as acting on a simple quantum stack consisting of `bit`s and `qubit`s. Later sections of chapter 3 extended the quantum stack with datatype and classical data nodes, together with operations on those nodes. Section 3.7 gave the interpretation of recursive functions acting on a quantum stack.

This chapter describes a machine using this full quantum stack and other data structures to provide an execution environment for L-QPL programs.

5.2 Quantum stack machine in stages

The quantum stack machine is described in terms of four progressively more elaborate stages. The first stage is the *basic QS-machine*, labelled BQSM. This stage provides facilities for the majority of operations of our machine, including classical operations, adding and discarding data and classical control. The second stage, the *labelled QS-machine*, called LBQSM adds the capability of applying unitary transforms with the modifiers `Left, Right` and `IdOnly` as introduced in section 3.4. The third stage, the *controlled QS-machine*, is labelled CQSM and provides the ability to do quantum control. The final stage, the *QS-machine*, is labelled QSM and adds the ability to call subroutines and do recursion.

These stages are ordered in terms of complexity and the operations definable on them. The ordering is:

$$\text{BQSM} < \text{LBQSM} < \text{CQSM} < \text{QSM}$$

When a function is defined on one of the lower stages, it is possible to lift it to a function on any of the higher stages. The details of the Haskell implementation of these stages and the lifting functions are given in appendix D.2.4 on page 185.

5.2.1 Basic quantum stack machine

The quantum stack machine transitions for the quantum instructions are defined at this stage. The state of the basic quantum stack machine has a code stream, $\mathcal{C}$, a classical stack, S, a quantum stack, Q, a dump, D and a name supply, N.

$$(\mathcal{C}, S, Q, D, N) \tag{5.1}$$

The code, $\mathcal{C}$, is a list of machine instructions. Transitions effected by these instructions are detailed in sub-section 5.4.1 on page 106. English descriptions of the instructions and what they do are given in appendix C.1 on page 141.

The classical stack, S, is a standard stack whose items may be pushed or pulled onto the top of the stack and specific locations may be accessed for both reading and updating. Classical arithmetic and Boolean operations are done with the top elements of the classical stack. Thus, an add will pop the top two elements of the classical stack and then push the result on to the top of the stack.

The dump, D, is a holding area for intermediate results and returns. This is used when measuring quantum bits, using probabilistic data, splitting constructed data types and for calling subroutines. Further details are given in sub-section 5.2.6 on page 101.

The name supply, N, is an integer that is incremented each time it is used. The name supply is used when binding nodes to constructed data nodes. As they are bound, they are renamed to a unique name generated from the name supply. For further details on this, see the transitions for `QBind` at sub-section 5.4.2 on page 106.

5.2.2 Labelled quantum stack machine

The labelled QS-machine, LBQSM, extends BQSM by labelling the quantum stack, $L(Q)$. The quantum stack is labelled to control the application of unitary transformations, which allows quantum control to be implemented.

The labelled QS-machines state is a tuple of five elements:

$$(\mathcal{C}, S, L(Q), D, N) \tag{5.2}$$

The quantum stack is labelled by one of four labels: *Full, RightOnly, LeftOnly* or *IdOnly*. These labels describe how unitary transformations will be applied to the quantum stack.

When this labelling was introduced in section 3.4, it was used as an instruction modifier rather than a labelling of the quantum stack. While the implementation of these modifiers is changed, the effect on the quantum stack is the same. The quantum stack machine transitions for unitary transformations are detailed in sub-section 5.4.8 on page 117.

5.2.3 Controlled quantum stack machine

The controlled quantum stack machine, CQSM, adds the capability to add or remove quantum control. This stage adds a control stack, C, and changes the tuple of classical stack, labelled quantum stack, dump and name supply into a list of tuples of these elements. In the machine states, a list will be denoted by enclosing the list items or types in square brackets.

The CQSM state is a tuple of three elements, where the third element is a list of four-tuples:

$$(\mathcal{C}, C, [(S, L(Q), D, N)]) \tag{5.3}$$

The control stack is implemented using a list of functions, each of which is defined on the third element of CQSM. The functions in the control stack transform the list of tuples $(S, L(Q), D, N)$. Control points are added to the control stack by placing an identity function at the top of the stack. Control points are removed by taking the top of the control stack and applying it to the current third element of CQSM, resulting in a new list of tuples. Adding a **qubit** to control will modify the function on top of the control stack and change the list of tuples of $(S, L(Q), D, N)$.

5.2.4 The complete quantum stack machine

The complete machine, QSM, allows the implementation of subroutine calling. Its state consists of an infinite list of CQSM elements.

$$\text{Inflist}(\mathcal{C}, C, [(S, L(Q), D, N)]) \tag{5.4}$$

Subroutine calling is done in an iterative manner. At the head of the infinite list, no subroutines are called, but result in divergence. Divergence is represented in the quantum stack machine by a quantum stack with a trace of 0.

In the next position of the infinite list, a subroutine will be entered once. If the subroutine is recursive or calls other subroutines, those calls will diverge. The next position of the infinite list will call one more level. At the n^{th} position of the infinite list subroutines are executed to a call depth of n.

5.2.5 The classical stack

The machine uses and creates values on the classical stack when performing arithmetic operations. This object is a standard push-down stack with random access. Currently it accommodates both integer and Boolean values.

5.2.6 Representation of the dump

When processing various operations in the machine, such as those labelled as quantum control (measure et. al.), the machine will need to save intermediate stack states and results. To illustrate, when processing a case deconstruction of a datatype, the machine saves all partial trees of the node on the dump together with an empty stack to accumulate the results of processing these partial trees. After processing each case the current quantum stack is merged with the result stack and the next partial stack is processed. The classical stack is also saved in the dump element at the beginning of the process and reset to this saved value when each case is evaluated.

The dump is a list of *dump elements*. There are two distinct types of dump elements, one for quantum control instructions and one used for call statements. The details of these elements may be found in the description of the quantum control transitions in sub-section 5.4.5 on page 111 and function calling in sub-section 5.4.9 on page 119.

5.2.7 Name supply

The name supply is a read-only register of the machine. It provides a unique name when binding nodes to a data node. The implementation uses an integer value which is incremented for each of the variables in a selection pattern in a case statement. It is reset to zero at the start of each program.

5.3 Representation of data in the quantum stack

The quantum stack was introduced and described in chapter 3. This section will give further details of the implementation of the quantum stacks and show example nodes.

5.3.1 Representation of qubits.

A single **qubit** is represented on the quantum stack as a node with four possible branches. This assumes a basis for quantum computation of two elements, which is identified with $(0,1)$ and $(1,0)$ in $\mathbb{C}^2$. The four possible values of the branches represent the elements of the **qubit**'s density matrix. This is illustrated in figure 5.1. From left to right, the branches are labelled with $00, 01, 10$ and 11. The value at each branch is .5. This corresponds to the density matrix $\begin{bmatrix} .5 & .5 \\ .5 & .5 \end{bmatrix}$

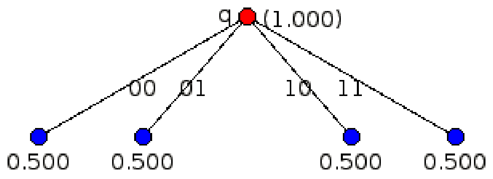


Figure 5.1: A **qubit** after a Hadamard transform

With multiple **qubit**s, the representation becomes hierarchical. For example, two **qubit**s will be represented by a tree with one of the **qubit**s at the top and each of its sub-branches having the second **qubit** below it. Consider applying a Hadamard transform to one **qubit**, followed by a controlled-Not with that **qubit** as the control. This is a standard way to entangle two **qubit**s. As illustrated in figure 5.2 on the next page, this creates a tree in the quantum stack with a total of four non-zero leaves. The quantum stack in the figure corresponds to a sparse representation of the density matrix:

$$\left[\begin{array}{cc|cc} .5 & 0 & 0 & .5 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ .5 & 0 & 0 & .5 \end{array}\right]$$

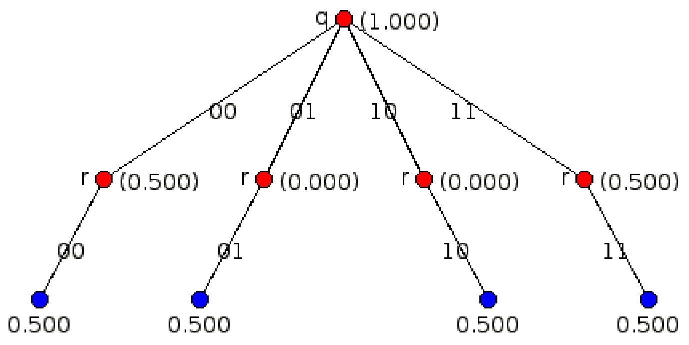


Figure 5.2: Two entangled `qubits`

5.3.2 Representation of integers and Boolean values

Numeric and Boolean data in the quantum stack machine is represented by a node with a sub-branch for each value that occurs with a non-zero probability. These values may be of either integer or Boolean type.

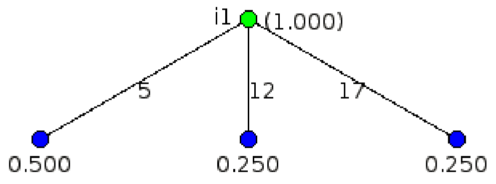


Figure 5.3: An integer with three distinct values

Figure 5.3 depicts an integer `i1` which has a 50% probability of being 5, and 25% each of being 12 or 17.

5.3.3 Representation of general data types

The general datatype is represented as a node with one branch for each of the constructors that occurs with a non-zero probability. Each branch is labelled by the constructor and the names of any nodes that are bound to it[1]. These nodes will be referred to as *bound nodes*.

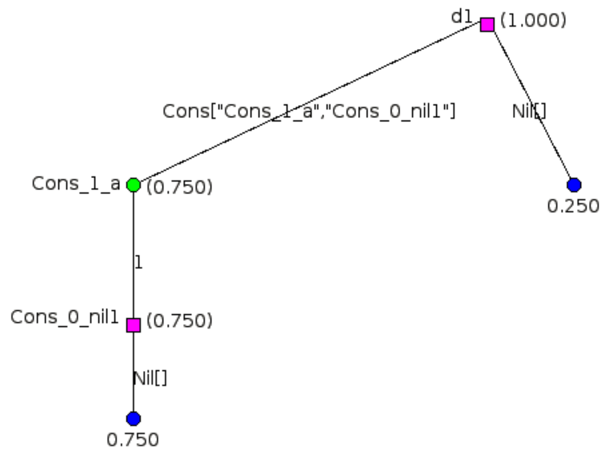


Figure 5.4: A list which is a mix of [ ] and [1].

For example, in the **List** that appears in figure 5.4, the top node is a mix of values. The node d1 has a 25% chance of being `Nil` and 75% of being a `Cons` of two bound nodes. The first bound node is an element of the base type, integer. It is labelled `Cons_1_a` which is an integer node having the single value 1. The second bound element is `Cons_0_nil1`, which is another list having the single value of `Nil`.

5.4 Quantum stack machine operation

This section describes the actual transitions of the stack machine for each of the instructions in the machine.

[1]For example, in **List**s of integers, the `Cons` constructor requires a base integer and another **List**.

5.4.1 Machine transitions

The majority of the transitions presented in this section are defined on a machine of type $\text{B}\textsc{qsm} = (\mathcal{C}, S, Q, D, N)$ as was introduced in sub-section 5.2.1 on page 98. As discussed in that section, labelling only affects the transition of unitary transforms. All other instruction transitions ignore it, giving us:

$$Ins(L(Q)) = L(Ins(Q))$$

where Ins is the transition of some other instruction.

The transition for the application of transformations will use $\text{LB}\textsc{qsm}$, while the transition for the add/remove control instructions uses the machine state of $\text{C}\textsc{qsm}$. The call instruction uses the state of the complete machine, QSM, which allows recursion. All of these stages and their associated states were defined and discussed in section 5.2 on page 97.

5.4.2 Node creation

There are three instructions which allow us to create data on the stack and one which binds sub-nodes into a data type. These are `QLoad`, `QCons`, `QMove` and `QBind`. The transitions are shown in figure 5.5 on the following page.

The instructions do the following tasks:

`QLoad` nm $|i\rangle$ — Load a new **qubit** named nm on top of the quantum stack with the value $|i\rangle$;

`QCons` nm Cns – Load a new datatype node on top of the quantum stack with name nm and value Cns. Sub-nodes are not bound by this instruction.

`QMove` nm — Load a new classical node on top of the quantum stack with name nm and value taken from the top of the classical stack. If the classical stack is empty, the value is defaulted to 0.

`QBind` nm — Binds a sub-node down the branch of the node to the datatype constructor on top of the quantum stack. Furthermore, the act of binding will cause the newly bound sub-node to be renamed so that it is hidden until an unbind is performed. `QBind` uses the name supply, N, to create the new name for the sub-node. The machine will generate an exception if the top of the quantum stack is not a single branched datatype or if a node named nm is not found.

$$
\begin{aligned}
&(\mathrm{QLoad}\ x\ |k\rangle : \mathcal{C}, S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[|k\rangle \to Q], D, N) \\
&(\mathrm{QCons}\ x\ c{:}\mathcal{C}, S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[c\{\} \to Q], D, N) \\
&(\mathrm{QMove}\ x{:}\mathcal{C}, v{:}S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[\bar{v} \to Q], D, N) \\
&(\mathrm{QBind}\ z_0{:}\mathcal{C}, S, x{:}[c\{z_1', \ldots, z_n'\} \to Q], D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[c\{z(N), z_1', \ldots, z_n'\} \to Q[z(N)/z_0]], D, N')
\end{aligned}
$$

Figure 5.5: Transitions for node construction

5.4.3 Node deletion

Three different instructions, `QDelete`, `QUnbind` and `QDiscard` remove data from the quantum stack. These instructions are the converses of `QBind`, `QLoad` and `QMove`. Their transitions are shown in figure 5.6 and figure 5.7. The instructions do the following tasks:

`QDelete` — removes the top node of the stack *and any bound sub-nodes*. This instruction has no restrictions on the number of sub-stacks or bindings in a data node;

`QDiscard` — removes the top node of the stack. In all cases, the top node can only be removed when it has a single sub-stack. For datatype nodes, `QDiscard` also requires there are no bound sub-nodes.

`QUnbind` nm — removes the first bound element from a data type *provided it has a single sub branch*. The datatype node must be on top of the quantum stack. The newly unbound sub-node is renamed to nm.

$$\begin{array}{lcl}
(\text{QDelete:}\mathcal{C}, S, Q{:}[|k_{ij}\rangle \to Q_{ij}], D, N) & \Longrightarrow & (\mathcal{C}, S, (Q_{00} + Q_{11}), D, N) \\
(\text{QDelete:}\mathcal{C}, S, DT{:}[c_i\{b_{ij}\} \to Q_i], D, N) & \Longrightarrow & (\mathcal{C}, S, \sum_i (del(\{b_{ij}\}, Q_i)), D, N) \\
(\text{QDelete:}\mathcal{C}, S, I{:}[\overline{v}_i \to Q_i], D, N) & \Longrightarrow & (\mathcal{C}, S, \sum_i Q_i, D, N)
\end{array}$$

Figure 5.6: Transitions for destruction

For the `QDelete` instruction, the type of node is irrelevant. It will delete the node and, in the case of datatype nodes, any bound nodes. This instruction is required to implement sub-routines that have parametrized datatypes as input arguments. For example, the algorithm for determining the length of a list is to return 0 for the “Nil” constructor and add 1 to the length of the tail list in the “Cons” constructor. When doing this, the elements of the list are deleted due to the linearity of L-QPL. The compiler will have no way of determining the type of the elements in the list and therefore could not generate the appropriate quantum split and discards. The solution is to use a `QDelete` instead.

The subroutine `del` used in the transitions in figure 5.6 will recursively rotate up and then delete the bound nodes of a datatype node.

$$(\text{QDiscard}{:}\mathcal{C}, S, x{:}[|k\rangle \to Q], D, N) \quad\Longrightarrow\quad (\mathcal{C}, S, Q, D, N)$$

$$(\text{QDiscard}{:}\mathcal{C}, S, x{:}[c\{\} \to Q], D, N) \quad\Longrightarrow\quad (\mathcal{C}, S, Q, D, N)$$

$$(\text{QDiscard}{:}\mathcal{C}, S, x{:}[\overline{v} \to Q], D, N) \quad\Longrightarrow\quad (\mathcal{C}, v{:}S, Q, D, N)$$

$$(\text{QUnbind } y{:}\mathcal{C}, S, x{:}[c\{z_1', \dots, z_n'\} \to Q], D, N) \\ \Longrightarrow\ (\mathcal{C}, S, x{:}[c\{z_2', \dots, z_n'\} \to Q[y/z_1']], D, N)$$

Figure 5.7: Transitions for removal and unbinding

The renaming is an integral part of the `QUnbind` instruction, as a compiler will not be able to know the bound names of a particular data type node. The instruction does *not* delete the data type at the top of the stack or the unbound node. If the top node is not a data type or has more than a single branch or does not have any bound nodes, the machine will generate an exception.

The machine ensures that it does not create name capture issues by rotating the bound node to the top of the sub-stack before it does the rename. That is, given the situation as depicted in the transitions, the quantum stack machine performs the following operations:

1. $Q' \leftarrow \mathtt{pull}(z_1', Q)$;
2. $Q'' \leftarrow Q'[y/z_1']$;
3. z_1' is removed from the list of constructors;
4. The new quantum stack is now set to $x{:}[c\{z_2', \dots, z_n'\} \to Q'']$.

5.4.4 Stack manipulation

Most operations on a quantum stack affect only the top of the stack. Therefore, the machine must have ways to move items up the stack. This requirement is met by the instructions `QPullup` and `QName`. The transitions are shown in figure 5.8.

The instructions do the following tasks:

`QPullup` nm — brings the *first* node named nm to the top of the quantum stack. It is not an error to try pulling up a non-existent address. The original stack will not be changed in that case.

`QName` nm_1 nm_2 — renames the first node in the stack having nm_1 to nm_2.

$$\begin{aligned}
(\text{QPullup } x{:}\mathcal{C}, S, Q, D, N) &\quad\Longrightarrow\quad (\mathcal{C}, S, \mathsf{pull}(x, Q), D, N)\\
(\text{QName } x\ y{:}\mathcal{C}, S, Q, D, N) &\quad\Longrightarrow\quad (\mathcal{C}, S, Q[y/x], D, N)
\end{aligned}$$

Figure 5.8: Transitions for quantum stack manipulation

A `QPullup` nm has the potential to be an expensive operation as the node nm may be deep in the quantum stack . In practice, many pullups interact with only the top two or three elements of the quantum stack.

The algorithm for pullup is based on preserving the bag of *path signatures*. A path signature for a node consists of a bag of ordered pairs (consisting of the node name and the branch constructor) where every node from the top to the leaf is represented. Pulling up a node will reorder the sub-branches below nodes to keep this invariant.

Due to the way arguments of recursive subroutines are handled in L-QPL, it is actually possible to get multiple nodes with the same name, however, this does not cause a referencing problem as only the highest such node is actually available in the L-QPL program.

5.4.5 Measurement and choice

The instructions `Split`,`Measure` and `Use` start the task of operating on a node's partial stacks, while the fourth, `EndQC` is used to iterate through the partial stacks. The transitions are shown in figure 5.9 on the following page.

The instructions do the following tasks:

`Use` Lbl — uses the classical node at the top of the quantum stack and executes the code at Lbl for each of its values.

`Split` $(c_1, lbl_1), \dots, (c_n, lbl_n)$ — uses the datatype node at the top of the stack and execute a jump to the code at lbl_i when there is a branch having constructor c_i. Any constructors not mentioned in the instruction are removed from the node first. There is no ordering requirement on the pairs of constructors and labels in `Split`.

`Measure` Lbl_{00} Lbl_{11} — using the **qubit** node on top of the quantum stack, executes the code at its two labels for the 00 and 11 branches. The off-diagonal elements of the **qubit** will be discarded. This implements a non-destructive measure of the **qubit**.

`EndQC` — signals the end of processing of dependent instructions and begins processing the next partial stack. When all values are processed, merges the results and returns to the instruction after the corresponding `Measure`, `Use` or `Split` instruction.

$$(\text{Use } \triangleright\mathcal{C}_U{:}\mathcal{C}, S, x{:}[\bar{v_i} \to Q_i], D, N)$$
$$\implies (\text{EndQC}, S, 0, \mathcal{Q}c(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)], \triangleright\mathcal{C}, 0){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)]_{i=j,\dots,m}, \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}_U, S, x_j, \mathcal{Q}c(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)]_{i=j+1,\dots,m}, \triangleright\mathcal{C}, Q + Q'){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [], \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}, S, Q + Q', D, N)$$

$$(\text{Split } [(c_i, \triangleright\mathcal{C}_i)]{:}\mathcal{C}, S, x{:}[c_i\{V_i\} \to Q_i], D, N)$$
$$\implies (\text{EndQC}, S, 0, \mathcal{Q}c(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)], \triangleright\mathcal{C}, 0){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)]_{i=j,\dots,m}, \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}_j, S, x_j, \mathcal{Q}c(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)]_{i=j+1,\dots,m}, \triangleright\mathcal{C}, Q + Q'){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [], \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}, S, Q + Q', D, N)$$

$$(\text{Meas } \triangleright\mathcal{C}_0\ \triangleright\mathcal{C}_1{:}\mathcal{C}, S, x{:}[|0\rangle \to Q_0, |1\rangle \to Q_1, \dots], D, N)$$
$$\implies (\text{EndQC}, S, 0, \mathcal{Q}c(S, [(x_k{:}\,|k\rangle \to Q_k, \triangleright\mathcal{C}_k)]_{k\in\{0,1\}}, \triangleright\mathcal{C}, 0){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [(x_k{:}\,|k\rangle \to Q_k, \triangleright\mathcal{C}_k)]_{k\in\{0,1\}}, \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}_0, S, x_0, \mathcal{Q}c(S, [(x_1{:}\,|1\rangle \to Q_1, \triangleright\mathcal{C}_1)], \triangleright\mathcal{C}, Q + Q'){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [(x_1{:}\,|1\rangle \to Q_1, \triangleright\mathcal{C}_1)], \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}_1, S, x_1, \mathcal{Q}c(S, [], \triangleright\mathcal{C}, Q + Q'){:}D, N)$$

$$(\text{EndQC}, S', Q, \mathcal{Q}c(S, [], \triangleright\mathcal{C}, Q'){:}D, N)$$
$$\implies (\mathcal{C}, S, Q + Q', D, N)$$

Figure 5.9: Transitions for quantum node choices

Each of the code fragments pointed to by the instruction labels *must* end with the instruction `EndQC`. The `EndQC` instruction will trigger execution of the code associated with the next partial stack.

The `QUnbind` is meant to be used at the start of the dependent code of a `Split` instruction. The sequencing to process a datatype node is to do a `Split`, then in each of the dependent blocks, execute `QUnbind` instructions, possibly interspersed with `QDelete` instructions when the bound node is not further used in the code. This is always concluded with a `QDiscard` that discards the data node which was the target of the `Split`.

In the following discussion there are no significant differences between the `Split` and `Measure`. The action of `Split` is described in detail.

The `Split`, `Measure` and `Use` instructions make use of the dump. The dump element used by these instructions consists of four parts:

- *The return label*. This is used when the control group is complete.
- *The remaining partial stacks*. A list consisting of pairs of quantum stacks and their corresponding label. These partial stacks are the ones waiting to be processed by the control group.
- *The result quantum stack*. This quantum stack accumulates the merge result of processing each of the control groups partial stacks. This is initialized to an empty stack with a zero trace.
- *The saved classical stack*. The classical stack is reset to this value at the start of processing a partial stack and at the end. This occurs each time an `EndQC` instruction is executed.

The instruction `Split` $[(c1, l1), (c2, l2)]$ starts by creating a dump entry with $[(c1 \to Q_1, l1), (c2 \to Q_2, l2)]$ as the list of partial stacks and label pairs. The dump entry will hold 0 quantum stack as the result stack, the current state of the classical stack and the address of the instruction following the `Split`. The final processing of the `Split` instruction sets the current quantum stack to zero and sets the next code to be executed to be `EndQC`.

The `EndQC` will change the top dump element by removing the first pair $(\mathtt{c1} \to Q_1, \mathtt{l1})$ from the execution list. It will set the current quantum stack to the first element of this pair and the code pointer to the second element. Execution then proceeds with the first instruction at $\mathtt{l1}$.

When the next `EndQC` instruction is executed, the dump will again be changed. First the current quantum stack will be merged with the result stack on the dump. Then the next pair of partial quantum stack $\mathtt{P}_q$ $(= \mathtt{c2} \to Q_2)$ and code pointer $\mathtt{l2}$ is removed from the execution list. The current quantum stack is set to $\mathtt{P}_q$ and the code pointer is set to $\mathtt{l2}$. Finally, the classical stack is reset to the one saved in the dump element.

When the partial stack list on the dump element is empty, the `EndQC` instruction will merge the current quantum stack with the result stack and then set the current quantum stack to that result. The classical stack is reset to the one saved on the dump, the code pointer is set to the return location saved in the dump element and the dump element is removed. Program execution then continues from the saved return point.

Normally, the first few instructions pointed to by the $\mathtt{Label}$ in the pairs of constructor and code labels will unbind any bound nodes and delete the node at the top of the stack. QSM does not *require* this, hence, it is possible to implement both destructive and non-destructive measurements and data type deconstruction.

Using classical values. The `Use` instruction introduced above differs from both `Split` and `Measure` in that it works on a node that may an unbounded number of sub-nodes. The `Use` lbl instruction moves all the partial stacks to the quantum stack, one at a time, and then executes the code at lbl for the resulting machine states. Normally, this code will start with a `QDiscard`, which will put the node value for that partial stack onto the classical stack, and finish with an `EndQC` to trigger the processing of the next partial stack.

The dump and EndQC processing for a `Use` lbl is the same as for a `Split` or `Measure`. The execution list pairs will all have the same label, the lbl on the instruction.

5.4.6 Classical control

The machine provides the three instructions `Jump`, `CondJump` and `NoOp` for branch control. Jumps are allowed only in a forward direction. The transitions for these are shown in figure 5.10 on the next page. The instructions do the following tasks:

`Jump` lbl — causes execution to continue with the code at lbl.

`CondJump` lbl — examines the top of the classical stack. When it is `False`, execution will continue with the code at lbl. If it is any other value, execution continues with the instruction following the `CondJump`.

`NoOp` — does nothing in the machine. Execution continues with the instruction following the `NoOp`.

$$\begin{aligned}
(\text{Jump} \triangleright \mathcal{C}_J{:}\mathcal{C}, S, Q, D, N) &\implies (\mathcal{C}_J, S, Q, D, N)\\
(\text{CondJump} \triangleright \mathcal{C}_J{:}\mathcal{C}, \texttt{False}{:}S, Q, D, N) &\implies (\mathcal{C}_J, S, Q, D, N)\\
(\text{CondJump} \triangleright \mathcal{C}_J{:}\mathcal{C}, \texttt{True}{:}S, Q, D, N) &\implies (\mathcal{C}, S, Q, D, N)\\
(\text{NoOp}{:}\mathcal{C}, S, Q, D, N) &\implies (\mathcal{C}, S, Q, D, N)
\end{aligned}$$

Figure 5.10: Transitions for classical control.

No changes are made to the classical stack, the quantum stack or the dump by these instructions. While `NoOp` does nothing, it is allowed as the target of a jump. This is used by the L-QPL compiler in the code generation as the instruction following a `Call`.

5.4.7 Operations on the classical stack

The machine has five instructions that affect the classical stack directly. They are `CGet`, `CPut`, `CPop`, `CApply` and `CLoad`, with transitions shown in figure 5.11 on the following page. The instructions perform the following tasks:

`CPop` — destructively removes the top element of the classical stack.

`CGet` n — copies the n^{th} element of the classical stack to the top of the classical stack.

`CApply` op — applies the operation op to the top elements of the classical stack, replacing them with the result of the operation. Typically, the op is a binary operation such as *add*.

`CLoad` v — places the constant v on top of the classical stack.

$$
\begin{array}{lcl}
(\mathrm{CPop}{:}\mathcal{C}, v{:}S, Q, D, N) & \Longrightarrow & (\mathcal{C}, S, Q, D, N) \\
(\mathrm{CGet}\ n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) & \Longrightarrow & (c, v_n{:}v_1{:}\cdots{:}v_n{:}S, Q, D, N) \\
(\mathrm{CPut}\ n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) & \Longrightarrow & (c, v_1{:}\cdots{:}v_1{:}S, Q, D, N) \\
(\mathrm{CApply}\ \mathsf{op}_n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) & \Longrightarrow & (\mathcal{C}, \mathsf{op}_n(v_1, \ldots, v_n){:}S, Q, D, N) \\
(\mathrm{CLoad}\ n\ {:}\mathcal{C}, S, Q, D, N) & \Longrightarrow & (\mathcal{C}, n{:}S, Q, D, N)
\end{array}
$$

Figure 5.11: Transitions for classical stack operations.

5.4.8 Unitary transformations and quantum control

The QS-Machine has three instructions which add or remove **qubit**s (and other nodes) from quantum control. The instruction transitions in this group are defined directly on CQSM or LBQSM, as they will either affect the control stack (`AddCtrl`, `QCtrl`, `UnCtrl`) or need to take into account the labelling of the quantum stacks (`QApply`).

The first three instructions do not affect the actual state of the quantum stack, classical stack or dump. The `QApply` does affect the state of the quantum stack. The transitions are shown in figure 5.12 on the next page.

The instructions perform the following tasks:

`AddCtrl` — starts a new control point on the control stack.

`QCtrl` — adds the node at the top of the quantum stack, together with any dependent sub-nodes to the control stack.

`UnCtrl` — removes *all* the nodes in the top control point of the control stack.

`QApply` n T — parametrized the transform T with the top n elements of the classical stack and then applies the parametrized transform to quantum stack.

Control is respected because of the labelling of the quantum stack.

$$
\begin{aligned}
&(\mathrm{AddCtrl}{:}\mathcal{C}, C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})]) \\
&\qquad\qquad\qquad\qquad \Longrightarrow\ (\mathcal{C}, \mathsf{id}{:}C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})]) \\
&(\mathrm{QCtrl}{:}\mathcal{C}, f{:}C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})]) \\
&\qquad\qquad\qquad\qquad \Longrightarrow\ (\mathcal{C}, (g \circ f){:}C, [(S_j', L(Q_j)', D_j')]_{j=1,\cdots m})]) \\
&(\mathrm{UnCtrl}{:}\mathcal{C}, f{:}C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})]) \\
&\qquad\qquad\qquad\qquad \Longrightarrow\ (\mathcal{C}, C, [(S_j'', L(Q_j)'', D_j'')]_{j=1,\cdots p})]) \\
&(\mathrm{QApply}\ m\ t{:}\mathcal{C}, (v_1{:}\cdots{:}v_m{:}S), L(Q), D, N) \\
&\qquad\qquad\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, \mathsf{cTrans}([v_1, \ldots, v_m], t, L(Q)), D, N)
\end{aligned}
$$

Figure 5.12: Transitions for unitary transforms

The function $\mathtt{cTrans}$ in the transition for `QApply` must first create the transform. In most cases, this is a fixed transform (e.g., Not, Hadamard), but both `rotate` and the `UM` transforms are parametrized. The transform `rotate` is used in the quantum Fourier transform and `UM` is the $a^x \mod N$ transform used in order finding.

When the top node is a **qubit**, the function expects its required number of **qubit**s to be the top nodes. For example, a Hadamard expects only 1, a *swap* expects 2 and an `UM` will expect as many **qubit**s as N requires **bit**s.

When the top node is a datatype node the machine will attempt to rotate up the required number of **qubit**s to the top, perform the operation and then re-rotate the datatype node back to the top. It will rotate the bound nodes of a datatype node starting at the left and proceeding to the right. Left to right is determined by the ordering in the original constructor expression used to create the datatype node. The machine will throw an exception if there are insufficient bound nodes (e.g., `Nil` for a list) or if the rotation would be indeterminate. Indeterminacy happens whenever a subject datatype node has more than one sub-stack.

Once this is accomplished the function will transform the top parts of the stack into a matrix Q of appropriate size (2×2 for a $1 - \mathbf{qubit}$ transform, 16×16 for a $4 - \mathbf{qubit}$ and so forth) with entries being the sub-stacks.

At this point, the control labelling of the quantum stack is considered and one of the following four transforms will happen. If the actual transform is named T, the result will be:

$$\text{cTrans } \mathsf{T}\ \mathsf{L}(\mathsf{Q}) = \begin{cases} \mathsf{L}(\mathsf{Q}) & \mathsf{L} = \text{IdOnly} \\ \mathsf{L}(\mathsf{TQ}) & \mathsf{L} = \text{LeftOnly} \\ \mathsf{L}(\mathsf{QT}^*) & \mathsf{L} = \text{RightOnly} \\ \mathsf{L}(\mathsf{TQT}^*) & \mathsf{L} = \text{Full} \end{cases} \tag{5.5}$$

Following this the quantum stack is reformed from the resulting matrix.

5.4.9 Function calling and returning

The `Call` and `Return` instructions are used for function calling. The `Call` instruction is the only instruction that needs to directly work on QSM, the infinite list of CQSM items. The transition for this is defined in terms of a subordinate function *enterF* which is defined on BQSM. Its transition is also described below.

Recall the QS-machine stages have the states:

$$\mathrm{B{\small QSM}} = (\mathcal{C}, S, Q, D, N)$$

$$\mathrm{C{\small QSM}} = (\mathcal{C}, C, [(S, L(Q), D, N)])$$

$$\mathrm{QSM} = \mathrm{Inflist}(\mathcal{C}, C, [(S, L(Q), D, N)])$$

For the state QSM, an infinite list will be expressed as

$$H_0 \blacktriangleright T = H_0 \blacktriangleright H_1 \blacktriangleright H_2 \blacktriangleright \cdots$$

where H is an element of the correct type for the infinite list.

The instructions do the following tasks:

`Call` n lbl — Calls the subroutine at lbl, copying the top n elements of the classical stack to a classical stack for the subroutine.

`Return` n — Uses the return label in the head of the dump to return from the subroutine. It also copies the top n elements from the classical stack and places them on top of the saved classical stack from the dump element.

$$\begin{aligned}
&(\mathrm{Call}\; n \triangleright\mathcal{C}_C{:}\mathcal{C}, C, [(S_i, L(Q)_i, D_i, N_i)])_0 \blacktriangleright T \\
&\qquad\qquad \Longrightarrow\; (\mathcal{C}, C, [(S_i, L(\emptyset)_i, D_i, N_i)])_0 \blacktriangleright \mathit{lift}\; (\mathit{enterf}\; n \triangleright\mathcal{C}_C)\; T \\
&\mathit{enterf}\; n \triangleright\mathcal{C}_C(\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\; (\mathcal{C}_C, [v_1, \ldots, v_n], Q, R(S, \triangleright\mathcal{C}){:}D, N) \\
&(\mathrm{Return}\; n, v_1{:}\cdots{:}v_n{:}S', Q, R(S, \triangleright\mathcal{C}){:}D, N) \\
&\qquad\qquad \Longrightarrow\; (\mathcal{C}, [v_1, \ldots, v_n]{:}S, Q, D, N)
\end{aligned}$$

Figure 5.13: Transitions for function calls.

To illustrate how `Call` is being processed, consider the following diagram:

$$\begin{array}{lrrrr}
(0) & M_0 \blacktriangleright & M_1 \blacktriangleright & M_2 \blacktriangleright & M_3 \blacktriangleright \dots \\
(1)\ (\texttt{Call}\ f): & 0 \blacktriangleright & f \cdot M_1 \blacktriangleright & f \cdot M_2 \blacktriangleright & f \cdot M_3 \blacktriangleright \dots \\
(2)\ (\texttt{Call}\ f): & 0 \blacktriangleright & 0 \blacktriangleright & f \cdot f \cdot M_2 \blacktriangleright & f \cdot f \cdot M_3 \blacktriangleright \dots \\
(3)\ (\texttt{Call}\ f): & 0 \blacktriangleright & 0 \blacktriangleright & 0 \blacktriangleright & f \cdot f \cdot f \cdot M_3 \blacktriangleright \dots \\
 & & & \vdots &
\end{array}$$

At the start, in line (0), the machine has state $M_0 \blacktriangleright M_i$. After the first call to f, at line (1), the head of the infinite list state has been zeroed out, indicating divergence. However, at every position further down the infinite list, the subroutine f is entered.

Continuing to line (2) and calling f again, the divergence has moved one position to the right and we now have a state of $0 \blacktriangleright 0 \blacktriangleright f \cdot f \cdot M_i$. Line (3) follows the same pattern. Thus, the further along in the infinite list one goes, the greater the *call depth*.

For details of how this is handled in the Haskell implementation of the quantum stack machine, see appendix D.2.5 on page 191.

The `Call` and `Return` instructions use a dump element as part of subroutine linkage. The `Call` i lbl instruction creates a dump element to store the current classical stack and the address of the instruction following the `Call` instruction. The `Return` n instruction will use the top dump element to reset the code pointer to the saved return location. `Return` also takes the classical stack from the top dump element and the top n values from the current classical stack are added to the top of it. `Return` then removes the top dump element.

Chapter 6

Future Work

As with any piece of research there are many avenues of exploration still left open with respect to L-QPL and its quantum stack machine.

6.1 Language extensions

The programming language L-QPL, while currently interesting and useful, could use additional features to make it a fully functional and powerful tool for developing quantum algorithms.

6.1.1 Refinements to the type system

In addition to the current `qdata` construction, a corresponding construction for classical data creation could be added. This declared classical data would be held in a classical node on the quantum stack, but could be moved back and forth to the classical stack.

A type *aliasing* declaration and potentially a *class* system to allow for closed types may also be useful.

6.1.2 Transform definition

Currently the language has a finite set of built-in transforms, two of which are parametrized by integer values. However, one common feature of quantum algorithms seems to be the application of generic reversible classical computation on $\mathtt{bit}$s as a unitary transformation on $\mathtt{qubit}$s.

This is typically done by using the equivalences:

$$\cfrac{\cfrac{\mathtt{bit}^n \to \mathtt{bit}}{\mathtt{bit}^{n+1} \to \mathtt{bit}^{n+1}}}{\mathtt{qubit}^{n+1} \to \mathtt{qubit}^{n+1}}$$

We would like to add supporting syntax and semantics to the language to accomplish this, without sacrificing the compile time type safety.

6.1.3 Input/Output

The ability to allow a program to read in values and print out results, likely outside of the bounds of any $\mathtt{qubit}$ manipulation, is a necessity for future development.

6.1.4 Miscellaneous enhancements

Additional base types in the language, such as characters and floating point numbers, would be useful in some algorithms and in I/O.

Bibliography

[1] Samson Abramsky. High-level methods for quantum computation and information. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LiCS'04), IEEE Computer Science Press.*, pages 410–414, 2004.

[2] Samson Abramsky. Abstract scalars, loops, and free traced and strongly compact closed categories. In *CALCO*, pages 1–29, 2005.

[3] Samson Abramsky and Bob Coecke. Physical traces: Quantum vs. classical information processing. *Electr. Notes Theor. Comput. Sci*, 69, 2002.

[4] S[amson] Abramsky and B[ob] Coecke. A categorical semantics of quantum protocols. In *LICS*, pages 415–425, 2004.

[5] S[amson] Abramsky and B[ob] Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LiCS'04), IEEE Computer Science Press. (extended version at arXiv:quant-ph/0402130)*, pages 415–425, 2004.

[6] S[amson] Abramsky and B[ob] Coecke. Abstract physical traces. *Theory and Applications of Categories*, 14:114–124, 2005.

[7] Manindra Agrawal, Neeraj Kayal, and Nitin Saxenz. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[8] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *LICS*, pages 249–258, 2005.

[9] Thorsten Altenkirch, Jonathan Grattage, Juliana K. Vizzotto, and Amr Sabry. An algebra of pure quantum programming. In *3rd International Workshop on Quantum Programming Languages*, 2005. to appear in ENTCS.

[10] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, CB2 1RP, Great Britain, 1998. ISBN 0-521-62277-8.

[11] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge Universtiy Press, The Edinburgh Building, Cambridge CB2 2RU, United Kingdom, 1998. ISBN 0-521-58274-1.

[12] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. Divincenzo, Peter Shor, Tycho Sleator, John Smolin, and Harald Weinfurter. Elementary gates for quantum computation, March 31 1995.

[13] R. Cleve, A. Ekert, L. Henderson, C. Macchiavello, and M. Mosca. On quantum algorithms. *Complexity*, 4:33–42, March 1999.

[14] Richard Cleve. An introduction to quantum complexity theory, June 28 1999. Comment: 28 pages, LaTeX, 11 figures within the text, to appear in "Collected Papers on Quantum Computation and Quantum Information Theory", edited by C. Macchiavello, G.M. Palma, and A. Zeilinger (World Scientific).

[15] Richard Cleve and John Watrous. Fast parallel circuits for the quantum fourier transform. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.

[16] D[avid] Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proc. Royal Society of London*, A400:97–117, 1985.

[17] David Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London Ser. A*, A425:73–90, 1989.

[18] David P. DeVincenzo. Quantum gates and circuits. In *Proceedings of the ITP Conference on Quantum Coherence and Decoherence*, December 1996. arXiv:quant-ph/9705009 v1.

[19] Thomas G. Draper. Addition on a quantum computer, 2000.

[20] Bryan Eastin and Steven T. Flammia. Q-circuit Tutorial. Available at `http://info.phys.unm.edu/Qcircuit`.

[21] Phil Gossett. Quantum carry-save arithmetic, 1998.

[22] Jonathan Grattage and Thorsten Altenkirch. A compiler for a functional quantum programming language. submitted for publication, January 2005.

[23] Jonathan Grattage and Thorsten Altenkirch. Qml: Quantum data and control. submitted for publication, February 2005.

[24] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, Berlin, Heidelberg, Germany, second edition, 1997. ISBN 0-387-98403-8. Dewey QA169.M33 1998.

[25] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In R. J. M. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture '91*, New York, NY, 1991. Springer-Verlag. Lecture Notes in Computer Science 523.

[26] Shin-Cheng Mu and Richard Bird. Functional quantum programming. In *Asian Workshop on Programming Languages and Systems*, KAIST, Dajeaon, Korea, December 2001.

[27] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 2000. ISBN 0 521 63235 8.

[28] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[29] Amr Sabry. Modeling quantum computing in Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell (HASKELL-03)*, pages 39–49, New York, August 28 2003. ACM Press.

[30] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[31] Peter Selinger. Dagger compact closed categories and completely positive maps. In *Proceedings of the 3rd International Workshop on Quantum Programming Languages, Chicago*, 2005.

[32] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54:147, 1995.

[33] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1999. ISBN 0 521 64176 4.

Appendix A

QPL syntax and semantics

A.1 The language QPL

QPL is a fairly low level language with a minimum of constructs. It supports only the types **bit** and **qubit**. Because of this design, it is fairly straightforward to present a semantics for the language. The syntax for QPL is given in figure A.1.

As the reader will see, some definitions in figure A.1 on the next page are not required. For example, the distinctions between *qbitid, bitid, procid* can normally be handled when a compiler does the semantic analysis of the language.

As a prelude to the work in this thesis, I implemented a simulator and compiler based directly on *Block QPL*. That compiler determined **bit**s, **qubit**s and procedure identifiers at the semantic analysis stage.

A.2 Semantics of QPL

Each statement in QPL is to be viewed as a function from tuples of matrices to tuples of matrices. For each **bit** in the programs' context there will be two tuple elements. For each **qubit** within a **bit**'s sphere of control the size of the matrices in the corresponding tuple position will grow exponentially. Given n **bit**s and m_i **qubit**s for each **bit**, the context will be $\mathbb{C}^{m_1 \times m_1} \times \cdots \times \mathbb{C}^{m_{2n} \times m_{2n}}$. As explained in Dr. Selinger's paper, **bit** values correspond to *classical control*. For each distinct bit

```
<qplProg>    ::= <statement>

<statement> ::= new bit <bitid> := 0 | new qbit <qbitid> := 0 |
                discard <id> | <bitid> := 0 | <bitid> := 1 |
                <qbitid list> *= <transform> | skip |
                <stmt list>|
                if <bitid> then <statement> else <statement> |
                measure <qbitid> then <statement> else <statement> |
                while <bitid> do <statement> |
                proc <procid>:<signature>{<statement>} in <statement> |
                <idlist> = <procid>(idlist)

<stmt list> ::= <statement> ; <stmt list> | <statement>

<transform> ::= Had | Chad | Not | CNot | Z | X | CZ | CX | V | CV

<id>        ::= <bitid> | <qbitid>

<bitid>     ::= b | <bitid><letterOrDigit>

<qbitid>    ::= q | <qbitid><letterOrDigit>

<procid>    ::= <uppercaseLetter> |
                <procid><letterOrDigit>

<qbitid list>
            ::= <qbitid> | <qbitid> , <qbitid list>

<id list>   ::= <id> | <id> , <id list>

<signature> ::= <typespeclist> -> <typespeclist>

<typespeclist>
            ::= <typespec> | <typespec>,<typespeclist>

<typespec>  ::= <bitid>::bit |
                <qbitid>::qbit
```

Figure A.1: Description of QPL

value, there may be a different number of **qubit**s. Consider the program fragment in figure A.2.

```
1  new bit b:= 0;
2  new qbit q := 0;
3  q *= Had;
4  measure q then
5     b:=0
6   else
7     b:= 1;
8
9  if b then
10   new qbit r = 0;
11   new qbit s = 0;
12   r *= Had;
13   r,s *= Cnot
14  else
15   skip;
```

Figure A.2: Fragment of QPL code showing different contexts

Measurement in QPL is not destructive, therefore the **qubit** `q` remains in scope throughout the program. In the fragment at line 12, the context has three **qubit**s: (`q, r, s`). The **bit** `b` is not available to the program inside an `if`. At line 15 only one **qubit** (`q`) is in context. The context after completion of the if statement is the merge of these two contexts and would therefore be:

$$\mathbb{C}^{8\times 8} \times \mathbb{C}^{2\times 2}.$$

The system can be regarded as being in $\mathbb{C}^{8\times 8}$ when the **bit** `b` is 0, and in $\mathbb{C}^{2\times 2}$ when it is 1.

Each statement defined for QPL has a realization in these contexts.

A.2.1 Formal semantics for QPL

Dr. Selinger presents a categorical semantics for QPL as the category Q of superoperators over signatures. I will give an outline of the major points of this semantics. This section uses concepts from linear algebra that are introduced later in this thesis in section 2.1 on page 10.

Signatures

Signatures are the objects of Q. Each signature is a list of non-zero natural numbers:

$$\ell = [n_1, \ldots, n_m].$$

Each of these signatures is identified with a vector space V_ℓ over the complex numbers

$$V_\ell = \mathbb{C}^{n_1 \times n_1} \times \cdots \times \mathbb{C}^{n_m \times n_m}.$$

This vector space is tuples of matrices over the complex numbers.

Specific objects in Q are given names:

$$\mathbf{bit} = [1, 1] \qquad \mathbf{qubit} = [2]$$

$$I = [1] \qquad 0 = [\,]$$

QPL realizations in Q

Given the details of the category Q it is now straightforward to associate specific maps in Q (superoperators) with specific QPL statements. Dr. Selinger gives a complete list. Table A.1 on the following page repeats a few of these.

QPL	**Map**	**Typing**	**Result**
$b := 0$	$\mathrm{set}_0 :$	$\mathbf{bit} \to \mathbf{bit}$	$\mathrm{set}_0(a,b) = (a+b, 0)$
$b := 1$	$\mathrm{set}_1 :$	$\mathbf{bit} \to \mathbf{bit}$	$\mathrm{set}_1(a,b) = (0, a+b)$
merge	$\mathrm{merge} :$	$\mathbf{bit} \to I$	$\mathrm{merge}(a,b) = a+b$

Table A.1: Instructions for QPL and their meaning

Appendix B

BNF description of the Linear Quantum Programming Language

B.1 Program definition

L-QPL programs consist of a series of definitions at the global level. These are either *data definitions* which give a description of an algebraic data type *procedure definitions* which define executable code.

```
<Linearqplprogram>   :: <global definitions>

<global_definitions> :: <global_definitions> <global_definition>
     | empty

<global_definition> :: <data_definition>
     | <procedure_definition>
```

B.2 Data definition

A data definition consists of declaring a type name, with an optional list of type variables and a list of constructors for that type. It is a semantic error to have different types having the same constructor name, or to redeclare a type name.

Constructor definitions allow either fixed types or uses of the type variables mentioned in the type declaration.

```
<data_definition> :: <type_definition> '='
                     '{' <constructor_list> '}'

<type_definition> :: 'type' <constructorid> <id_list>
```

```
<constructor_list>:: <constructor> <more_constructor_list>

<more_constructor_list> ::
     '|' <constructor> <more_constructor_list>
     | {- empty -}

<constructor> :: <constructorid> '(' <typevar_list> ')'
     | <constructorid>

<typevar_list> :: <typevar> <moretypevar_list>

<moretypevar_list> :: ',' typevar moretypevar_list
     | {- empty -}

<typevar> :: <identifier>
     | <identifier>
     | <constructor>
     | <constructor> '(' <typevar_list> ')'
     | <builtintype>

<builtintype>:: 'Qbit'  | 'Int' | 'Bool'
```

B.3 Procedure definition

Procedures may only be defined at the global level in a L-QPL program. The definition consists of a procedure name, its input and output formal parameters and a body of code. Note that a procedure may have either no input, no outputs or neither.

The classical and quantum inputs are separated by a '|'. Definitions with either no parameters or no classical parameters are specific special cases.

```
<procedure_definition> :: <identifier> '::'
          '(' <parameter_definitions> '|'
              <parameter_definitions> ';'
              <parameter_definitions>  ')'
                  '='  <block>
     | <identifier> '::'
         '(' <parameter_definitions> ';'
             <parameter_definitions>  ')'
```

```
                '='  <block>
     | <identifier> '::' '(' ')' '=' <block>

<parameter_definitions> :: <parameter_definition>
          <more_parameter_definitions>
     | {- empty -}

<more_parameter_definitions> :: ',' <parameter_definition>
          <more_parameter_definitions>
     | {- empty -}

<parameter_definition> :: <identifier> ':' <constructorid>
          '(' <typevar_list> ')'
     | <identifier> ':' <constructorid>
     | <identifier> ':' <builtintype>
```

B.4 Statements

Although L-QPL is a functional language, the language retains the concept of *statements* which provide an execution flow for the program.

The valid collections of statements are *blocks* which are lists of statements.

```
<block> :: '{' <stmtlist> '}'

<stmtlist>:: <stmtlist> ';' <stmt>
     | <stmtlist> ';'
     | <stmt>
     | {- empty -}
```

Statements are broadly grouped into a few classes.

B.4.1 Assignment

Variables are created by assigning to them. There is no ability to separately declare them. Type unification will determine the appropriate type for the variable.

```
<stmt> :: <identifier> '=' <exp>
```

B.4.2 Case statements

These are **`measure`**, **`case`**, **`use`**, **`discard`** and the classical assign, **`:=`**. These statements give the programmer the capability to specify different processing on the sub-stacks of a quantum variable. This is done with dependant statements. For **`measure`** and **`case`**, the dependent statements are in the block specified in the statement. For **`use`**, they may be specified explicitly, or they may be all the statements following the **`use`** to the end of the enclosing block. The classical assign is syntactic sugar for an assignment followed by a **`use`** with no explicit dependent statements.

The **`discard`** statement is grouped here due to the quantum effects. Doing a discard of a `qubit` is equivalent to measuring the `qubit` and ignoring the results. This same pattern is followed for discarding quantum variables of all types.

```
(<stmt> continued)
     | 'case' <exp> 'of' <cases>
     | 'measure' <exp> 'of' <zeroalt> <onealt>
     | <identifier> ':=' <exp>
     | 'use' <identifier_list> <block>
     | 'use' <identifier_list>
```

B.4.3 Functions

This category includes procedures and transforms. A variety of calling syntax is available, however, there are no semantic differences between them.

```
(<stmt> continued)
     | '(' <identifier_list> ') '='
          <callable>  '(' <exp_list> ')'
     | '(' <identifier_list> ') '='
          <callable>  '(' <exp_list> '|' <exp_list> ')'
     | <callable> <ids>
     | <callable> '(' <exp_list> ')' <ids>
     | <callable> '(' <exp_list> '|' <exp_list> ';'
          <ids> ')'
```

B.4.4 Blocks

The block statement allows grouping of a series of statements by enclosing them with `{` and `}`. An empty statement is also valid.

```
(<stmt> continued)
     |  <block>
     | {- empty -}
```

B.4.5 Control

L-QPL provides a statement for classical control and one for quantum control. Note that quantum control affects only the semantics of any transformations applied within the control. Classical control requires the expressions in its guards (see below) to be classical and not quantum.

```
(<stmt> continued)
     | 'if' guards
     | <stmt> '<=' <identifier_list>
```

B.4.6 Divergence

This signifies that this portion of the program does not terminate. Statements after this will have no effect.

```
(<stmt> continued)
     | 'zero'
```

B.5 Parts of statements

The portions of statements are explained below. First is *callable* which can be either a procedure name or a particular unitary transformation.

```
<callable> :: <identifer> | <transform>
```

The alternatives of a measure statement consist of choice indicators for the base of the measure followed by a block of statements.

```
<zeroalt> ::  '|0>' '=>' <block>

<onealt> :: '|1>' '=>' <block>
```

The **`if`** statement requires a list of *guards* following it. Each guard is composed of a classical expression that will evaluate to `true` or `false`, followed by a block of guarded statements. The statements guarded by the expression will be executed only when the expression in the guard is true. The list of guards must end with a default guard called **`else`**. Semantically, this is equivalent to putting a guard of `true`.

```
<guards> :: <freeguards> <owguard>

<freeguards> :: <freeguard> <freeguards>
     | {- empty -}

<freeguard> :: <exp> '=>' <block>

<owguard> :: 'else' '=>' <block>
```

When deconstructing a data type with a **`case`** statement, a pattern match is used to determine which set of dependent statements are executed. The patterns allow the programmer to either throw away the data element (using the '_' special pattern), or assign it to a new identifier.

```
<cases> :: <case> <more_cases>

<more_cases> :: {- empty -}
     | <case> <more_cases>

<case> :: <caseclause> '=>' <block>

<caseclause> ::  <constructorid> '(' <pattern_list> ')'
     | <constructorid>
```

```
<pattern_list>:: <pattern> <more_patterns>

<more_patterns> :: ',' <pattern> <more_patterns>
     | {- empty -}

<pattern> :: <identifier> | '_'
```

B.6 Expressions

L-QPL provides standard expressions, with the restriction that arithmetic expressions may be done only on classical values. That is, they must be on the classical stack or a constant.

The results of comparisons are Boolean values that will be held on the classical stack.

```
<exp>:: <exp0>

<exp0>:: <exp0> <or_op> <exp1> | <exp1>

<exp1>:: <exp1> '&&' <exp2> | <exp2>

<exp2>:: '~' <exp2> | <exp3>  | <exp3> <compare_op> <exp3>

<exp3>:: <exp3> <add_op> <exp4> | <exp4>

<exp4>:: <exp4> <mul_op> <exp5> | <exp5>

<exp5>:: <exp5> <shift_op> <exp6> | <exp6>

<exp6>:: <identifier> | <number> | 'true' | 'false'
     | '(' <exp> ')'
     | <constructorid> '(' <exp_list> ')'
     | <constructorid>
     | <identifier> '('  ')'
     | <identifier> '(' <exp_list> ')'
     | <identifier> '(' <exp_list> ';' ids ')'
     | '|0>' | '|1>'

<exp_list>:: <exp> <more_exp_list>
```

```
<more_exp_list>:: ',' <exp> <more_exp_list>
     | {- empty -}
```

B.7 Miscellaneous and lexical

These are the basic elements of the language as used above. Many of these items are differentiated at the lexing stage of the compiler.

```
<idlist> :: <identifier> more_ids
     | {- empty -}

<more_ids> :: <identifier> more_ids
     | {- empty -}

<identifier_list>:: <identifier> <more_idlist>
     | {- empty -}

<more_idlist>:: ',' <identifier> <more_idlist>
     | {- empty -}


<or_op>:: '||' | '^'

<compare_op>:: '==' | '<' | '>' | '=<' | '>=' | '=/='

<add_op>::'+' | '-'

<mul_op>::'*' | 'div'  | 'rem'

<shift_op>::'>>' | '<<'

<transform>:: <gate>
     | <transform> *o* transform

<gate> :: 'Had' | 'T'  | 'Phase' | 'Not' |  'RhoX'
     | 'Swap' | 'Rot '| 'RhoY ' | '  RhoZ ' | 'Inv-'<gate>

<identifier> :: <lower> | <identifier><letterOrDigit>
<constructorid :: <upper> | <constructorid><letterOrDigit>
<letterOrDigit> :: <upper>|<lower>|<digit>
<number> :: ['+'|'-'] <digit>+
<lower> ::   'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g'
```

```
          | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ | ’n’ | ’o’
          | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’
          | ’x’ | ’y’ | ’z’
<upper> ::    ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’
          | ’H’ | ’I’ | ’J’ | ’K’ | ’L’ | ’M’ | ’N’ | ’O’
          | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’
          | ’X’ | ’Y’ | ’Z’
<digit> ::    ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’
          | ’7’ | ’8’ | ’9’
```

Appendix C

Quantum stack machine additional details

C.1 Instructions

Creation of reasonable set of instructions, balancing brevity and usefulness has been an interesting task. The list of instructions and brief descriptions of them are presented in table C.1. The transitions of these are presented formally in section 5.4 on page 105.

Table C.1: QSM instruction list

Instruction	**Arguments**	**Description**
`QLoad`	*nm:Name*, *k::*$\mathbf{qubit}$	Creates $\mathbf{qubit}$ named *nm* and sets the value to $\lvert k\rangle$.
`QMove`	*nm:Name*	Creates an integer or Boolean named *nm* and sets its value to the top of the classical stack.
`QCons`	*nm:Name*, *c::constructor*	Creates a data type element with the value *c*. Note that if the constructor requires sub-elements, this will need to be followed by `QBind` instructions.
`QBind`	*nm:Name*	Binds the node $[nm]$ to the data element currently on the top of the stack.
`QDelete`	$\emptyset$	Deletes the top node and any bound nodes in the quantum stack.
`QDiscard`	$\emptyset$	Discards the node on top of the quantum stack.

Continued on next page

Instruction	**Arguments**	**Description**
`QUnbind`	*nm:Name*	Unbinds the first bound node from the data element at the top of the stack and assigns it as *nm*.
`QPullup`	*nm::name*	Pulls the node named *nm* to the top of the quantum stack.
`QName`	*nm1::name, nm2::name*	Renames the node named *nm1* to *nm2*.
`AddCtrl`	$\emptyset$	Marks the start of a control point in the control stack. Any following `QCtrl` instructions will add the top node to this control point.
`QCtrl`	$\emptyset$	Moves the top element of the quantum stack to the control stack. Recursively moves any bound nodes to the control stack when the element is a constructed data type.
`UnCtrl`	$\emptyset$	Moves all items in the control stack at the current control point back to the quantum stack.
`QApply`	*i::Int,t:Transform*	Parametrizes the transform *T* with the top *i* elements of the classical stack and applies it to the quantum stack.
`Measure`	*l0::Label, l1::Label*	Measures the **qubit** on top of the quantum stack and sets up the dump for execution of the code at *l0* for the 00 sub-branch and the code at *l1* for the 11 sub-branch.
`Split`	*cls:: [(constructor, Label)]*	Splits the data node at the top of the quantum stack and sets up the dump for execution of the code at the i-th label for the i-th sub-branch.
`Use`	*lbl::Label*	Uses the classical (integer or Boolean) node on top of the quantum stack and sets up the dump to for execution of the code at *lbl* for each of the sub-branches.

Continued on next page

Instruction	**Arguments**	**Description**
`EndQC`	$\emptyset$	Merges the current quantum stack with the results stack of the dump, activates the next partial stack to be processed and jumps to the code at the corresponding label. When there are no more partial stacks, the instruction merges the current stack with the the results stack and sets that as the new quantum stack.
`Call`	*i::Int*, *ep::EntryPoint*	For the first element of the infinite list of states, sets the values at the leaves of the quantum stack to 0. For the remainder of the list, the instruction jumps to the subroutine at *ep*, saving the return location and classical stack on the dump. It copies the top *i* elements of the classical stack for the subroutine.
`Return`	*i::Int*	Restores the location and classical stack from the dump, copies the top *i* items of the current classical stack to the top of the restored classical stack.
`Jump`	*lbl::Label*	Jumps *forward* to the label *lbl*.
`CondJump`	*lbl::Label*	If the top of the classical stack is the value `false`, jumps *forward* to the label *lbl*.
`NoOp`	$\emptyset$	Does nothing.
`CGet`	*i::Int*	Copies the *i*-th element of the classical stack to the top of the classical stack. A negative value for *i* indicates the instruction should copy the $\vert\mathtt{i}\vert^{\text{th}}$ value from the bottom of the classical stack.
`CPut`	*i::Int*	Copies the top of the classical stack to the *i*-th element of the classical stack. A negative value for *i* indicates the instruction should place the value into the $\vert\mathtt{i}\vert^{\text{th}}$ location from the bottom of the classical stack.

Continued on next page

Instruction	Arguments	Description
`CPop`	$\emptyset$	Pops off (and discards) the top element of the classical stack.
`CLoad`	*v::Either Int Bool*	Pushes *v* onto the classical stack.
`CApply`	*op::Classical Op*	Applies *op* to the top elements of the classical stack, replacing them with the result of the operation.

C.2 Translation of L-QPL to stack machine code

This section will discuss the code produced by the various statements and expressions in an L-QPL program. An L-QPL program consists of a collection of data definitions and procedures. Data definitions do not generate any direct code but do affect the code generation of statements and expressions.

Each procedure will generate code. A procedure consists of a collection of statements each of which will generate code. Some statements may have other statements of expressions as dependent pieces, which again will generate code.

The code generation in the compiler, and the description here, follows a standard recursive descent method.

C.2.1 Code generation of procedures

The code generated for each procedure follows a standard pattern of: procedure entry; procedure statements; procedure exit. The procedure statements portion is the code generated for the list of statements of the procedure, each of which is detailed in appendix C.2.2 on the next page.

Procedure entry

Each procedure is identified in QSM by an entry point, using an assembler directive. This directive is a mangled name of the procedure, followed by the keyword `Start`. The only exception to this is the special procedure `main` which is generated without mangling. `main` is always the starting entry point for a QSM program.

Procedure exit

The end of all procedures is denoted by another assembler directive, `EndProc`. For all procedures except `main`, the code generation determines how many classical variables are being returned by the procedure and emits a `Return` n instruction, where n is that count[1].

Procedure body

The code for each statement in the list of statements is generated and used as the body of the procedure. As an example, see the coin flip code and the corresponding generated QSM code in figure C.1 on the following page.

C.2.2 Code generation of statements

Each statement in L-QPL generates code. The details of the code generation for each statement are given in the following pages, together with examples of actual generated code.

[1]This functionality is currently not available in L-QPL, but may be re-introduced at a later date.

```
qdata Coin = {Heads | Tails}
cflip ::(; c:Coin) =
{   q = |0>;
    Had q;
    measure q of
       |0> => {c = Heads}
       |1> => {c = Tails}
}
main :: () =
{    c = cflip() ; }
```

(a) Coin flip code

```
CFlip_fcdlbl0 Start
    QLoad q |0>
    QApply 0 !Had
    Measure l0 l1
    Jump l3
l0  QDiscard
    QCons b #False
    EndQC
l1  QDiscard
    QCons b #True
    EndQC
l3  QPullup b
    Return 0
    EndProc

main Start
    Call 0 CFlip_fcdlbl0
    NoOp
    EndProc
```

(b) Generated code

Figure C.1: L-QPL and QSM coin flip programs

Assignment statements

The sub-section describes code generation for quantum assignment statements and assignments to variables on the classical stack. The classical assignment (:=) statement is described with the `use` statement below, as it is syntactic sugar for that statement.

An assignment of the form $i = \langle expr \rangle$ is actually broken down into 5 special cases. The first is when the left hand side is an in-scope variable that is on the classical stack. The other four all deal with the case of a quantum variable, which is either introduced or overwritten. The four cases depend on the type of expression on the right hand side. Each paragraph below will identify which case is being considered and then describe the code generation for that case.

Left hand side is a classical variable. In this case, generate the code for the expression on the right hand side (which will be classical in nature). This leaves the expression value at the top of the classical stack. Now, emit a `CPut` instruction which will copy that value into the location of the classical variable.

```
i = 5;
```

$\Longrightarrow$

```
CLoad 5
CPut −2
```

Right hand side is a classical expression. First, generate the expression code, which leaves the value on the top of the classical stack. Then emit a `QMove` instruction with the name of the left hand side. This will create a new classical node, which will be set to the value of the top of the classical stack.

```
i = 5;
```

$\Longrightarrow$

```
CLoad 5
QMove i
```

Right hand side is a constant `qubit`. Emit the `QLoad` instruction with the **`qubit`** value and the left hand side variable name.

$\Longrightarrow$

```
q = |1>;
```

```
QLoad q |1>
```

Right hand side is an expression call. First, emit the code for the expression call. This will leave the result quantum value on the top of the quantum stack. If the formal name given by the procedure definition is the same as the left hand side name, do nothing else, as the variable is already created with the proper name. If not, emit a `QName` instruction to rename the last formal parameter name to the left hand side name.

```
random :: (maxval :Int;
        rand :Int) =
  {...}
...
x = random(15);
```

$\Longrightarrow$

```
CLoad 15
QMove c18
QName c18 maxval //In
Call 0 random_fcdlbl0
QName rand x //Out
```

Right hand side is some other expression. Generate the code for the expression. Check the name on the top of the stack. If it is the same as the left hand side name, do nothing else, otherwise emit a `QName` instruction.

```
outqs = Cons(q, inqs');
```

$\Longrightarrow$

```
     QCons c4 #Cons
     QBind inqs'
     QBind q
    QName c4 outqs
```

Measurement code generation

Measurement will always have two subordinate sets of statements, respectively for the $|0\rangle$ and $|1\rangle$ cases. The generation for the actual statement will handle the requisite branching.

The code generation first acquires three new labels, m_0, m_1 and m_f. It will then emit a `Measure` m_0 m_1 statement, followed by a `Jump` m_f. Recall from the transitions in sub-section 5.4.1 on page 106 that when the machine executes the `Measure` instruction, *it then generates and executes a* `EndQC` instruction. The `Jump` will be executed when all branches of the **qubit** have been executed.

Then, for each of the two sub blocks ($i \in \{0, 1\}$), I emit a `Discard` labelled with m_i. This is followed by the code generated from the corresponding block of statements. Finally a `EndQC` is emitted.

The last instruction generated is a `NoOp` which is labelled with m_f.

```
measure q of
  |0> => {n1 = 0}
  |1> => {n1 = 1};
```

$\Longrightarrow$

```
     QPullup q
     Measure l7 l8
     Jump l9
l7   QDiscard
     CLoad 0
     QMove n1
     EndQC
l8   QDiscard
     CLoad 1
     QMove n1
     EndQC
l9   NoOp
```

Case statement code generation

Case statement generation is conceptually similar to that of measurement. The differences are primarily due to the variable number of case clauses and the need to instantiate the variables of the patterns on the case clauses.

As before, the expression will have its code generated first. Then, the compiler will use a function to return a list of triples of a constructor, its generated label and the corresponding code for each of the case clauses. The code generation done by that function is detailed below.

At this point, the code generation resembles measurement generation. The compiler generates a label c_f and emits a `Split` with a list of constructor / code label pairs which have been returned by the case clause generation. This is followed by emitting a `Jump` c_f. After the `Jump` has been emitted, the code generated by the case clause generation is emitted.

The final instruction generated is a `NoOp` which is labelled with c_f.

Case clause code generation. The code generation for a case clause has a rather complex prologue which ensures the assignment of any bound variables to the patterns in the clause.

First, the code generator gets a new label c_l and then calculates the unbinding code as follows: For each *don't care* pattern in the clause, code is generated to delete the corresponding bound node. This is done by first getting a new stack name nm, then adding the instructions `QUnbind` nm, `Pullup` nm and `QDelete` nm. This accomplishes the unbinding of that node and removes it from the quantum stack. Note that `QDelete` is used here to ensure that all subordinate nodes are removed and that no spurious data is added to the classical stack in the case of the don't care node being a classical value.

For each named pattern p, only the instruction `QUnbind` p is added.

The program now has a list of instructions that will accomplish unbinding of the variables. Note this list may be empty, e.g., the `Nil` constructor for `List`.

The clause generation now creates its own return list. When the unbinding list is not empty, these instructions are added first, with the first one of them being labelled with c_l.This is followed by a `Discard` which discards the decomposed data node. When the unbind list is empty, the first instruction is the `Discard` labelled by c_l.

Then, the code generated by the statements in the case clause block are added to the list. Finally, a `EndQC` is added at the end.

The lists from all the case clauses are combined and this is returned to the case code generation.

The example at the end of the use clause will illustrate both the case clause and the use statement code generation.

Use and classical assign code generation

As described earlier in sub-section 4.3.5 on page 83, the classical assignment, `v :=exp` is syntactic sugar for a variable assignment followed by a **`use`** statement. The code generation for each is handled the same way.

There are two different cases to consider when generating this code. The **`use`** statement may or may not have subordinate statements. In the case where it does not have any subordinate statements, (a classical assign or a use with no block), the scope of the classical variables in the use extends to the end of the enclosing block.

The case of a use with a subordinate block is presented first.

Use with subordinate block code generation. While similar to the generation for the measure and case statements, there are differences. The two main differences are that there is only one subordinate body of statements and that multiple variables may be used.

In the case where there is a single use variable nu, the generator gets the body label u_b and end label u_e. It then emits a `Pullup` nu, a `Use` u_b and a `Jump` u_e. This is followed by emitting a `Discard` labelled by u_b and the code generated by the subordinate body of statements and a `EndQC` instruction. This is terminated by a `NoOp` labelled by u_e.

When there are multiple names $n_1, n_2, \ldots, n_j$, the generator first recursively generates code assuming the same body of statements but with a use statement that only has the variables $n_2, \ldots, n_j$. This is then used as the body of code for a use statement with only one variable n_1, which is generated in the same manner as in the above paragraph.

Use with no subordinate block. To properly generate the code for this, including the `EndQC` and end label, the generator uses the concept of delayed code. The prologue (`Use`, `Jump` and `Discard`) and epilogue (`EndQC`, `NoOp`) are created in the same manner as the use with the subordinate block. The prologue is emitted at the time of its generation. The epilogue, however, is added to a push-down stack of delayed code which is emitted at the end of a block. See the description of the block code generation for more details on this.

```
case l of
  Nil => {i = 0}
  Cons (_, l1) => {
    n= len(l1);
    use n;
    i = 1 + n;
}
```

$\Longrightarrow$

```
 Split (#Nil,l0) (#Cons,l1)
   Jump l4
l0 QDiscard
   CLoad 0
   QMove i
   EndQC
l1 QUnbind c0
   QUnbind l1
   QDiscard
   QPullup c0
   QDiscard
   QPullup l1
   QName l1 l
   Call 0 len_fcdl0
   QName i n
   QPullup n
   Use l2
   Jump l3
l2 QDiscard
   CLoad 1
   CGet 0
   CApply +
   QMove i
   EndQC //For Use
l3 NoOp
    EndQC //For Split
l4 NoOp
```

Conditional statements

The **`if ... else`** statement allows the programmer to specify an unlimited number of classical expressions to control blocks of code. Typically, this is done within a **`use`** statement based upon the variables used.

The statement code generation is done by first requesting a new label, g_e. This label is used in the next step, generating the code for all of the guard clauses. The generation is completed by emitting a `NoOp` instruction labelled by g_e.

Guard clauses. Each guard clause consists of an expression and list of statements. The code generator first emits the code to evaluate the expression. Then, a new label g_l is requested and the instruction `CondJump` g_l is emitted. The subordinate statements are generated and emitted, considering them as a single block. The concluding instruction is a `Jump` g_e.

At this point, if there are no more guard clauses, a `NoOp` instruction, labelled by g_l is emitted, otherwise the code generated by the remaining guard clauses is labelled by g_l and emitted.

```
if b == 0 => {
   theGcd = a;
} else =>  {
   (theGcd) =
      gcd(b, a mod b);
}
```

$\Longrightarrow$

```
     CGet -2
     CLoad 0
     CApply ==
     CondJump lbl2
     CGet -1
     QMove theGcd
     Jump lbl1
lbl2 CLoad True //else
     CondJump lbl3
     CGet -1
     CGet -2
     CApply %
     QMove c0
     CGet -2
     QMove c1
     QName c1 a
     QName c0 b
     Call 0 gcd_fcdlbl0
     Jump lbl1
lbl3 NoOp
lbl1 NoOp
```

Function calling and unitary transforms

The code generation of these two statements is practically the same, with the only difference being that built in transformations have a special instruction in QSM, while executing a defined function requires the `Call` instruction.

In each case, the statement allows for input classical and quantum expressions and output quantum identifiers.

The first step is the generation of the code for the input classical expressions. These are generated in reverse order so that the first parameter is on the top of the classical stack, the second is next and so forth. Then, the input quantum expressions are generated, with names of these expressions being saved. Note that it is possible to use expressions which are innately classical (e.g., constants and variables on the classical stack) as a quantum expression. The compiler will generate the code needed to lift it to a quantum expression. See appendix C.2.4 on page 159 for the details.

At this stage, the two types differ slightly. For the unitary transformation case, the code "`QApply` n !t" is emitted, where n is the number of classical arguments and t is the name of the built in transform. The exclamation mark is part of the QSM assembler syntax for transform names. In a defined function, renames of the input quantum expressions to the names of the input formal ids are generated, by emitting a series of `QName` instructions. This is followed by emitting a `Call` n f, where n is again the number of classical arguments and f is the internally generated name of the function.

In both cases, the code checks the formal return parameter names against the list of return value names. For each one that is different, a `QName` $frml$ $retnm$ is emitted.

See the previous code list between the `CondJump lbl3` and `lbl3 NoOp` for an example of this.

C.2.3 Code generation of expressions

In the previous sub-section, a number of examples of code generation were given. These also illustrated most of the different aspects of expression code generation. A few additional examples are given below.

Generation of constants

There are three possible types of constants, a Boolean, an integer or a $\mathtt{qubit}$. Note that constructors are considered a different class of expression.

For both Boolean and integers, the compiler emits a `CLoad` val instruction. For a $\mathtt{qubit}$, it creates a new name q and emits a `QLoad` q qbv instruction.

Examples of these may be seen in the sub-sub-section on assignments.

Generation of classical arithmetic and Boolean expressions

In all cases, these types of expressions are calculated solely on the classical stack. Whenever the generator encounters an expression of the form

$$e_1 \text{ op } e_2$$

it first emits the code to generate e_2, followed by the code to generate e_1. This will leave the result of e_1 on top of the stack with e_2's value right below it. It then emits the instruction `CApply` op, which will apply the operation to the two top elements, replacing them with the result.

The Boolean *not* operation is the only operation of arity 1. Code generation is done in the same manner. The generator emits code for the expression first, followed by `CApply` $\neg$.

See under Guard clauses, appendix C.2.2 on page 153 for examples.

Generation of variables

The semantic analysis of the program will split this into two cases; classical variables and quantum variables. Each are handled differently.

Classical variables. These variables are on the classical stack at a specific offset. The use of them in an expression means that they are to be *copied* to the top of the classical stack. The code emitted is `CGet offset`, where `offset` is the offset of the variable in the classical stack.

Quantum variables. In this case, the variables are at a specific address of the quantum stack. These variables are not allowed to be copied, so the effect of this code is to rotate the quantum stack until the desired variable is at the top. The other consideration is that these variables have a linearity implicitly defined in their usage. In the compiler, this is handled by the semantic analysis phase, but the code generation needs to also consider this. The compiler will add this variable to a *delayed deletion* list. After completion of the statement with this variable expression, the variable will be deleted unless:

- The statement has deleted it. (Measure of a **qubit**, for example, will directly generate the deletion code.)
- It is recreated as the result of an assignment, function call or transformation.

Code generation for expression calls

Each expression call is generated in substantially the same way as the code for a call statement as in appendix C.2.2. The only difference is that the name of the final return variable will not be known and is therefore set to the same name as the name of the last output formal parameter. As an example, consider:

```

gcd :: (a : Int, b: Int; theGcd : Int)
= {
use a,b in {
  if b == 0 => {
          theGcd = a;
      } else =>  {
          (theGcd) = gcd(b, a mod b);
      }
 }
}
```

Suppose this is defined in a program, and at some point, it is called as an expression in the program : `gcd(5,n)`. The code generated for this expression will then leave the integer node named *theGcd* on the top of the quantum stack.

Generation of constructor expressions

These expressions are used to create new data type nodes, such as lists, trees etc. Constructors are similar to functions in that they expect an expression list as input and will return a new quantum variable of a specific type. In L-QPL they are somewhat simpler as the input expressions are all expected to be quantum and there is a single input only. Just as in function calls, any classical expressions input to the constructor will be lifted to a quantum expression.

The first step is for the compiler to emit code that will evaluate and lift any of the input expressions. The names of each of these expressions is saved. Then, it creates a new name d_c and emits the code `QCons` d_c #cid.

The final stage is to emit a `QBind` nm_{e_i} for each of the input expression, *in reverse order* to what was input. The next example illustrates this.

```
lt = Cons(|0>,
        Cons(|0>,
        iTZQList(2×n)));
```

$\Longrightarrow$

```
QLoad c22 |0>
QLoad c23 |0>
CLoad 2  //for the call
CGet −1
CApply ×
QMove c24
QName c24 n
Call 0 iTZQList_fcdlbl5
QCons c25 #Cons
QBind nq
QBind c23 //c23:nq
QCons c26 #Cons
QBind c25
QBind c22 //c22:c23:nq
QName c26 lt
```

C.2.4 Lifting of classical expressions to quantum expressions.

When the compiler requires a quantum expression, but has been given a classical one, it first generates the classical expression. This leaves the expression value on top of the classical stack. The compiler will now generate a new unique name l_c and emit the instruction `QMove` l_c. This now moves the value from the classical stack to the quantum stack.

Appendix D

Quantum stack machine implementation

D.1 Primary data definitions

D.1.1 Description of the classical stack

The classical stack data type is a standard stack of either integers or booleans with random access.

```
type ChoiceStack a b = Stack (Either a b)
type ClassicalStack = ChoiceStack Int Bool
```

The classical operations done on the stack are created by delegating to functions that will work with the *Either* elements on the stack. See appendix D.3.3 on page 209 for the details of these functions.

```
sLtoLfun :: (a → a → a) →
  ChoiceStack a b →
  ChoiceStack a b
sLtoLfun f (Stack (a : b : rest)) = Stack (((eLeftfun f) a b) : rest)
sLtoLfun f (Stack _)
   = error clsstackError
sRtoRfun :: (b → b → b) →
  ChoiceStack a b →
  ChoiceStack a b
sRtoRfun f (Stack (a : b : rest)) = Stack (((eRightfun f) a b) : rest)
sRtoRfun f (Stack _)
   = error clsstackError
sLtoRfun :: (a → a → b) →
  ChoiceStack a b →
  ChoiceStack a b
sLtoRfun f (Stack (a : b : rest)) = Stack (((eLtoRightfun f) a b) : rest)
sLtoRfun f (Stack _)
   = error clsstackError
```

Individual functions for each operation are created here. Below is definition of the add function. The definitions of all the others are similar.

```
stkadd :: ClassicalStack → ClassicalStack
stkadd = sLtoLfun (+)
```

Similarly, a function to convert from the operation to the appropriate stack function is required. Again, only the add function is shown here.

```
getStackOp :: ClassicalOp → ClassicalStack → ClassicalStack
getStackOp CAdd = stkadd
```

Finally, *stkput* is defined, which allows values to be placed into the classical stack at any location. *stkput* takes the top element of the stack and puts it at the designated index.

```
stkput :: Int → Stack a → Stack a
stkput i (Stack l) = Stack l'
  where len = length l
    index = len + i   -- Assume neg or zero
    ival = head l
    l' = (take index l) ++
      (ival : (drop (index + 1) l))
```

D.1.2 Description of the quantum stack

The quantum stack holds both *quantum* and *probabilistic* data in the quantum stack machine.

Quantum data consists of **qubit**s. These are stored in the quantum stack isomorphically to their density matrix notation. Assuming a basis of $|0\rangle$ and $|1\rangle$, four entries are used to store a **qubit**. This is the same as the density matrix for the **qubit** $\alpha\,|0\rangle + \beta\,|1\rangle$ which is $\begin{pmatrix}\alpha\bar{\alpha} & \alpha\bar{\beta}\\ \beta\bar{\alpha} & \beta\bar{\beta}\end{pmatrix}$.

Probabilistic data in the stack may be an `Int`, `Bool` or `datatype`. These are represented as multi-branched trees, with one branch per integer or logical value or data constructor. This is isomorphic to a tuple of branches.

The leaves of the tree are single valued numeric values. These values represent the actual value of the density matrix.

Finally, when all the leaves below a branch are 0, a special representation is used to effect sparseness.

Our code is written assuming a generic *Basis* which may have more than two elements. For example if a *Basis* had three elements, a **qubit** representation would require six values. (A 3×3 matrix with the below diagonal elements not stored).

Representation details

All nodes of the stack follow a common representation pattern of

$$(\text{nodename}, [\texttt{cons} \rightarrow \text{sub} - \text{stack}]).$$

The sub-stack in the case of `datatype`s also contains the list of variables lower in the stack which are *bound* to that constructor.

type *StackAddress* = *String*
type *Constructor* = *String*
type *CV a b* = *Map* (*ClassicalData*) (*QuantumStack a b*)
type *QV a b* = *Map* (*a*, *a*) (*QuantumStack a b*)
type *DV a b* = *Map Constructor* ([*StackAddress*], (*QuantumStack a b*))
class (*Basis a*, *Show a*, *Comp b*) ⇒ *Quantum a b*

For ease of use, a variety of creator and accessor functions are defined. The creator functions are named the same as the type they create, but in lower case. The functions *val* and *valMaybe* are accessor functions that will return a quantum substack given a substack of a *QV* element.

The quantum stack is a tree with a variable number of branches at each level and data at the leaves. Data is always added to the top of the stack. Manipulations of data at the top are performed by applying the proper linear combinations to the sub-stacks. (For more detail see *apply*).

There are five different constructors for *QuantumStack* elements:

StackZero Explicit laziness used when adding **bit**s or **qubit**s to the stack. In those cases, one or more of the sub stacks has all zero entries at the leaves. Rather than computing this, it is simply tagged as such.

StackData a The leaves of the stack where the entries of the series of density matrices are stored. This also serves as a starting point for a stack, with the data being the value 1.

StackBit name (*CV a*) A named **bit** entry in the stack, with the sub-stacks for each basis element.

StackQbit name $(QV\ a)$ A named **qubit** entry, with the sub-stacks corresponding to the product of the the basis elements.

```
data (Quantum a b) ⇒ QuantumStack a b
    = StackZero |
      StackData b |
      StackInt StackAddress (CV a b) |
      StackQbit StackAddress (QV a b) |
      StackCons StackAddress (DV a b)
```

Figure D.1: Haskell definition of the quantum stack

We provide definitions of instances for the predefined Haskell classes *Eq*, *Show*, *Num* and *Comp* for *QuantumStack a b*. These definitions depend upon the *b* type being a member of *Show* and *Comp*. The actual definitions are elided for compactness.

Quantum Stack support functions

This section contains a number of support functions used throughout the machine.

The function *breakQuantum* is used when processing *quantum control* instructions (see appendix D.2.1 on page 179). Each of those types of instructions requires accessing the sub-branches of the tree individually.

```
breakQuantum :: (Quantum a b) ⇒ QuantumStack a b →
    [QuantumStack a b]
breakQuantum (StackInt sa cvals)
    = let citems = Map.toList cvals
      in List.map ((StackInt sa) ∘ cv ∘ (:[])) citems
breakQuantum (StackQbit sa qvals)
```

```
    $= \mathbf{let}\ qitems = Map.toList\ qvals$
      $\mathbf{in}\ List.map\ ((StackQbit\ sa) \circ qv \circ (:[]))\ qitems$
$breakQuantum\ (StackCons\ sa\ dvals)$
    $= \mathbf{let}\ ditems = Map.toList\ dvals$
      $\mathbf{in}\ List.map\ ((StackCons\ sa) \circ dv \circ (:[]))\ ditems$
$breakQuantum\ q = [q]$
```

All the value data is stored at the leaves of the quantum stack, while operations are defined on the actual stacks. The operations, such as setting a value, unitary transformations etc., create linear combinations of the current sub-stacks as new sub-stacks. A left scalar multiple function $\ltimes$ and a corresponding right scalar multiple function $\rtimes$ are defined to support this. Note that scalar multiplication creates a *StackZero* if the scalar is 0.

```
$(\ltimes) :: (Quantum\ a\ b) \Rightarrow b \rightarrow QuantumStack\ a\ b \rightarrow$
  $QuantumStack\ a\ b$
$b \ltimes \_ \mid b \equiv fromInteger\ 0 = StackZero$
$b \ltimes StackZero = StackZero$
$b \ltimes (StackData\ c) = StackData\ (b * c)$
$b \ltimes s = descendMap\ (b\ltimes)\ s$
$(\rtimes) :: (Quantum\ a\ b) \Rightarrow QuantumStack\ a\ b \rightarrow b \rightarrow$
  $QuantumStack\ a\ b$
$x \rtimes y = y \ltimes x$
```

The auxiliary function *trimStack* is used to reduce full stacks of zero values to *StackZero*, which is then used by the *Num*.+ function. The accessor function *qvalues* retrieves the *QV* sub element of a *QuantumStack*. The *descendMap* function works like a functor, applying a *QuantumStack* endomorphism to the sub-stacks. The function *rotateup* works in conjunction with the *pullup* and *descendMap* functions to rotate a **bit** or **qubit** to the top of the stack. The first argument is the name (address) of the **qubit** to bring up. The second is the *QuantumStack* to rotate.

The function works by doing a recursive check and descent. There are three options:

Already at top — Just return the current stack.

Second item on quantum stack — Delegate to *pullup*.

Somewhere else — Use *descendMap* to rotate the desired (q)bit to the top of the sub-stacks, then apply *pullup*.

```
rotateInOrder :: (Quantum a b) ⇒ [String] → QuantumStack a b →
   QuantumStack a b
rotateInOrder [] q = q
rotateInOrder (nm : nms) q = rotateInOrder nms $ rotateup nm q
rotateup :: (Quantum a b) ⇒ String → QuantumStack a b →
   QuantumStack a b
rotateup _ StackZero = StackZero
rotateup _ (StackData b) = StackData b
rotateup s stack
   | s ≡ (theName stack)
      = stack
   | s ≡ (secondName stack)
      = trimStack $
        pullup s stack
   | otherwise
      = trimStack $
        pullup s $
        descendMap (rotateup s) stack
```

The function *pullup* handles the case of the search address being either on top or the second element of the *QuantumStack*. As with *rotateup*, if the target node is already on top, no change is made. In the case where the target node is the next element on the stack, an exchange is done to bring up that stack. This is achieved by building a new stack with the second element at the top and the first element at the first set of sub-stacks. The sub-stacks (up to 16 in the case of two qbits) are re-ordered so that their "tree path" remains the same. The "tree path" can be thought of as the path needed to descend the sub-stacks.

subStack allows us to choose a particular sub stack via the "tree path". It is given a list of basis elements, which are used to descend the *QuantumStack* until the list is exhausted. Two elements of the list are used for each **qubit**.

```
getSecondLevelConsAddresses :: (Quantum a b) ⇒ Constructor →
  QuantumStack a b → [StackAddress]
getSecondLevelConsAddresses cons
   = head ∘ (List.map (getConsAddresses cons)) ∘ getSecondLvlStacks
```

The function *prepare* is used when doing a transform on a quantum stack. It will rotate up the requisite number of **qubit**s for the transform.

```
prepare :: (Quantum a b) ⇒ Int → QuantumStack a b →
  ([StackAddress], QuantumStack a b)
prepare n q@(StackCons _ _) =
  let (oldnms, upnms) = getUpNames n q
     q′ = List.foldl (flip rotateup) q $ reverse upnms
  in (oldnms, q′)
prepare _ qs = ([], qs)
```

The function *getUpNames* is used only by *prepare* to determine the names of both the data nodes that are going to possibly be rotated down and the names of the **qubit**s to rotate up. The function *getUpNames* will only return the required number of **qubit** names.

It uses the subordinate functions *getQbits* which returns a list of the bound variables which are **qubit**s and *getDstacks* which returns a list of pairs of names of other data nodes and **qubit**s at lower levels in the quantum stack. These subordinate functions are explained more fully below.

```
getUpNames :: (Quantum a b) ⇒ Int → QuantumStack a b →
        ([StackAddress], [StackAddress])
getUpNames n qs
   | n ⩽ 0 = ([], [])
getUpNames n (StackCons nm cmap)
   | 1 ≡ Map.size cmap
       = let [(_, (subs, q))] = Map.toList cmap
           qbnms = getQbits subs q
           dstacks = getDStacks subs q
           m = n − (length qbnms)
           morenames = List.foldl (λ(x, y) (a, b) → (x ++ a, y ++ b)) ([], []) $
              (List.map $ getUpNames m) dstacks
           in (nm : fst morenames,
              take n $ qbnms ++ (snd morenames))
   | otherwise =
      error "Indeterminate qbit order for transform"
```

The function *getQbits* filters its first argument (a list of *StackAddress*es) by determining which of them are **qubit**s in the current quantum stack. The filtered list is then returned.

The filtering function *isQbit* returns true if the top node is a **qubit** with the correct name, otherwise it recurses down the quantum stack.

```
getQbits :: Quantum a b ⇒ [StackAddress] → QuantumStack a b →
   [StackAddress]
getQbits nms q = List.filter (flip (isQbit) q) nms
isQbit :: Quantum a b ⇒ StackAddress → QuantumStack a b → Bool
isQbit nm (StackQbit qnm qvls)
   = nm ≡ qnm ∨ (Map.fold (∨) False $ Map.map (isQbit nm) qvls)
isQbit nm (StackInt _ cvls)
   = Map.fold (∨) False $ Map.map (isQbit nm) cvls
isQbit nm (StackCons _ dvls)
   = Map.fold (∨) False $ Map.map ((isQbit nm) ∘ snd) dvls
isQbit _ _ = False
```

The function *getDStacks* first filters its first argument (a list of *StackAddress*es) by determining which of them are data nodes. These are then paired with the current quantum stack and in each case, that node is rotated up to the top of the quantum stack. This is then suitable for further recursion by *getUpNames*

The filtering function *isDtype* returns true if the top node is a data node with the correct name, otherwise it recurses down the quantum stack.

```
getDStacks :: Quantum a b ⇒ [StackAddress] → QuantumStack a b →
  [QuantumStack a b]
getDStacks nms q =
  (List.map (uncurry rotateup)) $ getDtypesNmAndQs nms q
getDtypesNmAndQs :: Quantum a b ⇒ [StackAddress] →
  QuantumStack a b →
  [(StackAddress, QuantumStack a b)]
getDtypesNmAndQs nms q =
  zip (List.filter (flip (isDtype) q) nms) $ repeat q
isDtype :: Quantum a b ⇒ StackAddress → QuantumStack a b → Bool
isDtype nm (StackCons dnm dvls)
   = nm ≡ dnm ∨ (Map.fold (∨) False $
     Map.map ((isDtype nm) ∘ snd) dvls)
isDtype nm (StackInt _ cvls)
   = Map.fold (∨) False $ Map.map (isDtype nm) cvls
isDtype nm (StackQbit _ qvls)
   = Map.fold (∨) False $ Map.map (isDtype nm) qvls
isDtype _ _ = False
```

D.1.3 Description of the control stack

The *ControlStack* is a higher order type, a function from a list of controlled quantum stacks to a list of controlled quantum stacks. The types and data constructs needed are shown in figure D.2.

Representation details

```
type ControlStack a b =
   ([Controlled (QuantumStack a b)], [Controlled (QuantumStack a b)])
      → ([Controlled (QuantumStack a b)], [Controlled (QuantumStack a b)])
instance Show (ControlStack a b) where
   show _ = "Funct"
data Controlled a
   = IdentityOnly a
   | LeftOnly a
   | RightOnly a
   | Full a
     deriving (Show, Eq)
```

Figure D.2: Definition of the control stack

This module also defines a *Functor* for the *Controlled* data modifier as all the functions defined on quantum stacks, with the exception of transformations, commute with *Controlled*. *Transformations* require adjusting according to the constructor of *Controlled* that is used.

```
instance Functor Controlled where
   fmap f (IdentityOnly a) = IdentityOnly (f a)
   fmap f (LeftOnly a) = LeftOnly (f a)
   fmap f (RightOnly a) = RightOnly (f a)
   fmap f (Full a) = Full (f a)
splitcontrolled :: Controlled a → (a, b → Controlled b)
splitcontrolled (IdentityOnly a) = (a, IdentityOnly)
splitcontrolled (LeftOnly a) = (a, LeftOnly)
splitcontrolled (RightOnly a) = (a, RightOnly)
splitcontrolled (Full a) = (a, Full)
unzipControl :: Controlled (a, b) → (a, Controlled b)
unzipControl c = (fst a,f $ snd a)
                  where (a,f) = splitcontrolled c
```

Adding and removing control.

Because of the definition, the addition of new control points is trivial, only requiring concatenating the *id* function to the front of the list.

```
addControl :: [ControlStack a b] → [ControlStack a b]
addControl = (:) ctrlNoOp
ctrlNoOp :: ([a], [a]) → ([a], [a])
ctrlNoOp ([], aas) = (aas, [])
ctrlNoOp (aas, []) = (aas, [])
ctrlNoOp _ = error "Illegal control start"
```

Similarly, removing control becomes simple as well. Note that here, this process affects both the *ControlStack* and the *QuantumStack* of the machine state. If there is no currently active control, this is a "do-nothing" function rather than an error.

When there actually is a control point, it will be removed and applied to the input *QuantumStack*. This new *QuantumStack* and the remaining part of the list of *ControlStack*s are returned by the function.

```
removeAllControl :: (Quantum a b) ⇒ [ControlStack a b] →
   [Controlled (QuantumStack a b)] →
```

```
  Controlled (QuantumStack a b)
removeAllControl [] cqs = head cqs
removeAllControl fs cq = (uncurry removeAllControl) $ removeControl fs cq
removeControl :: (Quantum a b) ⇒ [ControlStack a b] →
                 [Controlled (QuantumStack a b)] →
                 ([ControlStack a b], [Controlled (QuantumStack a b)])
removeControl [] cq = ([], cq)
removeControl (f : fs) cq = (fs, fst $ f ([], cq))
unControl :: (Quantum a b) ⇒ [ControlStack a b] →
             [((Int, ClassicalStack),
               (Controlled (QuantumStack a b), Dump a b))] →
             ([ControlStack a b],
               [((Int, ClassicalStack),
                 (Controlled (QuantumStack a b), Dump a b))])
unControl [] lis = ([], lis)
unControl fs lis = (fs′, newlis)
  where (ncss, cqds) = unzip lis
    (cqs, ds) = unzip cqds
    (fs′, cqs′) = removeControl fs cqs
    cqds′ = zip cqs′ ds
    newlis = zip ncss cqds′
```

Moving items to the *ControlStack*

This is where the actual coding work is required. A function is created that will recreate the quantum stack with the current top element and pass back the new list of *Controlled QuantumStack*s.

This is broken down into simpler pieces. The first delegation is to a function called *withControl* which operates on a list of *ControlStack* items and list of *Controlled QuantumStack* items, returning a pair of those lists. The *withControl* function also delegates to the function *makeControl* which creates the uncontrol function and the new list when given the list of quantum stacks. *withControl* then combines that function with the current one at the head of the control stack and returns the elements.

```
$qControl :: (Quantum\ a\ b) \Rightarrow [ControlStack\ a\ b] \rightarrow$
              $[((Int, ClassicalStack),$
                 $(Controlled\ (QuantumStack\ a\ b), Dump\ a\ b))] \rightarrow$
              $([ControlStack\ a\ b],$
                 $[((Int, ClassicalStack),$
                    $(Controlled\ (QuantumStack\ a\ b), Dump\ a\ b))])$
$qControl\ []\ lis = ([], lis)$
$qControl\ fs\ lis = (fs', newlis)$
   **where** $(ncss, cqds) = unzip\ lis$
      $(cqs, ds) = unzip\ cqds$
      $(fs', cqs') = withControl\ fs\ cqs$
      $cqds' = zip\ cqs'\ (cycle\ ds)$
      $newlis = zip\ (cycle\ ncss)\ cqds'$
$withControl :: (Quantum\ a\ b) \Rightarrow [ControlStack\ a\ b] \rightarrow$
                 $[Controlled\ (QuantumStack\ a\ b)] \rightarrow$
                 $([ControlStack\ a\ b], [Controlled\ (QuantumStack\ a\ b)])$
$withControl\ (f : fs)\ qs =$
   $((f \circ g) : fs, ctrlldqs)$
      **where** $(g, ctrlldqs) = makeControl\ qs$
```

The next function, *makeControl* delegates to the function *makeControl′* with an accumulator to create the control stack function. This function recurses, pasting intermediate results to the accumulator and delegating the work of creating a function and list of an individual quantum stack to *mc*.

```
$makeControl :: (Quantum\ a\ b) \Rightarrow$
      $[Controlled\ (QuantumStack\ a\ b)] \rightarrow$
      $(ControlStack\ a\ b, [Controlled\ (QuantumStack\ a\ b)])$
$makeControl = makeControl'\ (id, [])$
$makeControl' :: (Quantum\ a\ b) \Rightarrow$
      $(ControlStack\ a\ b, [Controlled\ (QuantumStack\ a\ b)]) \rightarrow$
      $[Controlled\ (QuantumStack\ a\ b)] \rightarrow$
```

```
    (ControlStack a b, [Controlled (QuantumStack a b)])
makeControl′ (cs′, qs′) [] = (cs′, qs′)
makeControl′ (cs′, qs′) (q : qs)
  = makeControl′ (f ∘ cs′, qs′ ++ cq) qs
    where (f, cq) = mc q
```

Our base functions, *mc* and *mc′* operate over the various types of the quantum stack. For simple nodes, it moves that node off, creates a list of the sub-nodes and returns a function that would paste that back onto the quantum stack. For data nodes, it also removes all the dependent children named in the constructors.

```
mc :: (Quantum a b) ⇒ Controlled (QuantumStack a b) →
  (ControlStack a b, [Controlled (QuantumStack a b)])
mc (IdentityOnly q@(StackQbit _ _))
  = (f, List.map idonlytr qs)
    where (f, qs) = mc′ IdentityOnly q
mc (LeftOnly q@(StackQbit _ _))
  = (f, List.map leftonlytr qs)
    where (f, qs) = mc′ LeftOnly q
mc (RightOnly q@(StackQbit _ _))
  = (f, List.map rightonlytr qs)
    where (f, qs) = mc′ RightOnly q
mc (Full q@(StackQbit _ _))
  = (f, qs)
    where (f, qs) = mc′ Full q
mc (IdentityOnly q) = mc′ IdentityOnly q
mc (LeftOnly q) = mc′ LeftOnly q
mc (RightOnly q) = mc′ RightOnly q
mc (Full q) = mc′ Full q
```

mc′ returns a function that "pastes" back the controlled data node to the sub-stacks of the node. It memoizes the various pieces needed to do this including the *StackAddress* for all types, the keys of the data values for all types and for constructors, the list of dependent nodes.

For the *StackQbit* nodes, the returned quantum stacks are controlled as per their key placement as a sub-stack. The return function resets the control to its previous value.

```
mc' :: (Quantum a b) ⇒
   (QuantumStack a b → Controlled (QuantumStack a b)) →
   QuantumStack a b →
   (ControlStack a b, [Controlled (QuantumStack a b)])
mc' ctl (StackQbit nm qvs)
    = (f, List.map (uncurry controlIt) ascl)
      where ascl = Map.toAscList qvs
         (keys, vals) = unzip ascl
         f = λ(acc, ctlqvls) →
            (acc ++
               [ctl $ StackQbit nm $ qv $ zip keys $
                  List.map uncontrolIt $ ctlqvls],
               drop (length keys) ctlqvls)
```

For the *StackInt* nodes, the returned quantum stacks are controlled as per the control type passed in. Controlling by an int value does nothing to quantum transformations, therefore it must remain the same as the parent was. Similarly, when uncontrolling, the function sets the control value of the new combined *StackInt* to what it was previously.

```
mc' ctl (StackInt nm cvs)
    = (f, List.map ctl vals)
      where ascl = Map.toAscList cvs
         (keys, vals) = unzip ascl
         f = λ(acc, ctlqvls) →
            (acc ++
               [ctl $ StackInt nm $ cv $ zip keys $
                  List.map uncontrolIt $ ctlqvls],
               drop (length keys) ctlqvls)
```

For the *StackCons* nodes, the direct processing is similar to that of *StackInt*, allowing for the list of dependent nodes.

The main difference is that controlling by a data type controls by *all* of the elements of that data type. For example, controlling by a *List* (*Qbit*) with 3 **qubit**s in it will control by all of those **qubit**s. This is achieved by calling the function *controlAll* defined below.

```
mc' ctl (StackCons nm dvls)
    = controlAll (subnms) (f, List.map (ctl ∘ snd) vals)
      where ascl = Map.toAscList dvls
        (keys, vals) = unzip ascl
        subnameList = List.map fst vals
        subnms = concat $ subnameList
        f = λ(acc, qvls) →
            (acc ++
              [ctl $ StackCons nm $ dv $ zip keys $
                zip subnameList $ List.map uncontrolIt $ qvls],
              drop (length keys) qvls)
```

The *StackZero* nodes are simply propagated as a placeholder.

```
mc' ctl StackZero
    = (f, [ctl StackZero])
      where f = λ(acc, qvls) →
        (acc ++ [ctl $ StackZero],
          tail qvls)
```

The *controlAll* function is used when controlling *StackCons* elements. For each name passed to the function, it rotates that up in the quantum stack and then adds it to the current control.

```
$controlAll :: (Quantum\ a\ b) \Rightarrow [StackAddress] \rightarrow$
   $(ControlStack\ a\ b, [Controlled\ (QuantumStack\ a\ b)]) \rightarrow$
   $(ControlStack\ a\ b, [Controlled\ (QuantumStack\ a\ b)])$
$controlAll\ []\ (f, qs) = (f, qs)$
$controlAll\ (nm : nms)\ (f, qs)$
    $= (g, newqs)$
   where $qs' = List.map\ (fmap\ (rotateup\ nm))\ qs$
     $(f', newqs) = withControl\ [f]\ qs'$
     $g = head\ f'$
```

As an example, consider applying a $\mathtt{controlled} - \mathsf{T}$ transform to the state Q, comprised of two **qubit**s. This gives:

$$\begin{bmatrix} I & 0 \\ 0 & \mathsf{T} \end{bmatrix} \begin{bmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & \mathsf{T}^* \end{bmatrix} = \begin{bmatrix} Q_{00} & Q_{01}\mathsf{T}^* \\ \mathsf{T}Q_{10} & \mathsf{T}Q_{11}\mathsf{T}^* \end{bmatrix} \tag{D.1}$$

From this, a general pattern to implement when controlling by a **qubit** becomes obvious. This is implemented by the function *controlIt*.

```
$controlIt :: (Basis\ a) \Rightarrow (a, a) \rightarrow b \rightarrow$
   $Controlled\ b$
$controlIt\ (a, b)$
    $|\ a \equiv hb \wedge b \equiv hb = IdentityOnly$
    $|\ a \equiv hb \wedge b \not\equiv hb = LeftOnly$
    $|\ a \not\equiv hb \wedge b \equiv hb = RightOnly$
    $|\ otherwise = Full$
   where $hb = head\ basis$
$uncontrolIt :: Controlled\ a \rightarrow a$
$uncontrolIt\ (IdentityOnly\ a) = a$
$uncontrolIt\ (LeftOnly\ a) = a$
$uncontrolIt\ (RightOnly\ a) = a$
$uncontrolIt\ (Full\ a) = a$
$idonlytr :: Controlled\ a \rightarrow Controlled\ a$
$leftonlytr :: Controlled\ a \rightarrow Controlled\ a$
$rightonlytr :: Controlled\ a \rightarrow Controlled\ a$
$idonlytr\ (IdentityOnly\ a) = IdentityOnly\ a$
```

$idonlytr\ (LeftOnly\ a) = IdentityOnly\ a$
$idonlytr\ (RightOnly\ a) = IdentityOnly\ a$
$idonlytr\ (Full\ a) = IdentityOnly\ a$

$leftonlytr\ (IdentityOnly\ a) = IdentityOnly\ a$
$leftonlytr\ (LeftOnly\ a) = LeftOnly\ a$
$leftonlytr\ (RightOnly\ a) = IdentityOnly\ a$
$leftonlytr\ (Full\ a) = LeftOnly\ a$

$rightonlytr\ (IdentityOnly\ a) = IdentityOnly\ a$
$rightonlytr\ (LeftOnly\ a) = IdentityOnly\ a$
$rightonlytr\ (RightOnly\ a) = RightOnly\ a$
$rightonlytr\ (Full\ a) = RightOnly\ a$

D.2 Machine Operation

D.2.1 Description of the machine instructions

The instruction set has been designed to strike a balance between a reasonable instruction (i.e., machine like) and a useful instruction, (i.e., a program does not require hundreds of instructions to do a unit of work).

Instruction Definitions

The *Instruction* data type is a sum type of the different classical and quantum instructions available in the machine.

data *Instruction a* =

Node Creation There are three instructions which allow us to create data on the stack and one which binds sub-nodes into a datatype.

QLoad StackAddress a | *QCons StackAddress Constructor* |

QMove StackAddress | *QBind StackAddress* |

Quantum stack Node Deletion Conversely, three instructions remove data from the quantum stack.

QUnbind StackAddress | *QDiscard* | *QDelete* |

Quantum stack manipulation One of the similarities between a quantum stack and classical stack is the top nodes are the primary targets of instructions. Because of this, instructions to move items up the stack are required.

QPullup StackAddress | *QName StackAddress StackAddress* |

Unitary transformation and control Specific unitary transformations are applied to the top of the stack. These are affected by the control of the transfroms.

AddCtrl | *QCtrl* | *UnCtrl* | *QApply Int UnitaryOp* |

Arbitrary transformations are definable and a compiler may create them. These transformation are unitary matrices, which are applied to the **qubit**s at the top of the stack.

Measurement, Deconstruction and Choice The *Measure* instruction measures a **qubit**, the *Split* does a case deconstruction of a declared data type. The *EndQC* instruction steps through the list of cases set up by a *Split*, *Measure* or *Use*.

EndQC | *Split* [(*Constructor*, *Label*)] |

Measure Label Label |

Using Classical Values

The *Use* instruction will execute the code at *Label* for each possible value a classical element can have.

Use Label |

Classical Control Instructions for standard flow control changes are next.

Jump Label | *CondJump Label* |

Call Int EntryPoint | *Return Int* | *NoOp* |

Classical Operations

Instructions for standard classical stack operations complete our set of instructions.

CGet Int | *CPut Int* | *CApply ClassicalOp* |

CPop | *CLoad* (*Either Int Bool*)

CodeMemory definition

The *Code* type is used to hold the currently active list of instructions. *Memory* holds all of a programs instructions as a map from the functions entry point to its list of instructions.

type *Code a* $=$ [*Instruction a*]
type *EntryPoint* $=$ *String*
type *CodePointer* $=$ (*EntryPoint*, *Label*)
type *Memory a* $=$ *Map EntryPoint* (*Code a*)

D.2.2 Unitary transformations

Unitary transformations are the basic operations for `qubit`s. In the quantum stack machine, a number of basic transformations are defined and capacity for the programmer to define their own is anticipated.

Theoretically, this is not required. All unitary transformations effects can be approximated to within any $\epsilon > 0$ by repeated applications of a few basic transformation.

Practically, however, the determination of what those transforms should be is not straightforward.

The following data definition is a *meta-type* of unitary operations, which defines the operations for the quantum stack machine.

data *UnitaryOp* $=$

The quantum stack machine has the following completely built-in transformations:

Ident Int | *NotGate* | *Tgate* | *Phase* | *Hadamard* | *RhoX* |

RhoY | *RhoZ* | *Swap* | *Toffoli* | *Rotate* | *UM* |

Of these above transforms, note that *Rotate* and *UM* are *parameterized* transforms. *Rotate* is expressible as the matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^n} \end{bmatrix}$$

where the n controls the amount of rotation applied. The transform *UM* requires three parameters, a, n and t. It creates the unitary matrix that will apply the transform

$$a \rightarrow x^{a2^t} \mod n$$

which is required for Shor's factoring algorithm. Note this is expected to be generalized in the future, see chapter 6 on page 122.

Additionally, *Controlled* transformations are easily created from the original transformation.

Controlled UnitaryOp |

Inverse UnitaryOp |

Finally, the type includes *named* transforms. These would be the transforms defined in a quantum assembly or linear qpl program. At this stage, the type simply tracks a name, the number of **qubit**s and the number of integer parameters used by the transform. This is not currently used in the machine or the compiler but is expected to be essential in the future.

DefinedOp String Int

To get the actual matrix used for transformation, the *getTransform* function translates from *UnitaryOp* to *Trans*. Below are a few of the interesting cases.

```
getTransform :: (Comp b) ⇒ [Either Int Bool] → UnitaryOp → Trans b
getTransform ((Left k) : _) Rotate = rotateGate k
getTransform l (Controlled g) = controlled $ getTransform l g
getTransform l (Inverse g) = conjtrans $ getTransform l g
getTransform l (DefinedOp s k) = error "To be defined yet"
getTransform _ NotGate = notGate
```

Transformations are simply matrices. See appendix D.3.6 on page 211 for their definition. Some of the specific matrices are shown below.

The function *controlled* is used to create a controlled version of a transformation, by creating a block matrix with the identity 2×2 matrix in the upper left and the matrix of the subject transformation in the lower right.

```
type Trans a = Matrix a
notGate :: (Comp b) ⇒ Trans b
notGate = [[0,1],[1,0]]
rotateGate :: (Comp b) ⇒ Int → Trans b
rotateGate k
   = [[1,0],
        [0, exp (pi ∗ sqrtMinusOne /
           (2 ∗∗ ((fromInteger $ toInteger k) − 1)))]]
diag :: (Num a) ⇒ Trans a → Trans a → Trans a
diag tl br
   = [a ++ (take (dimx br) (map fromInteger zeros)) |
     a ← tl] ++
        [(take (dimx tl) (map fromInteger zeros)) ++ a |
           a ← br]
controlled :: (Comp b) ⇒ Trans b → Trans b
controlled = diag i2by2
```

D.2.3 Description of the dump

The quantum stack machine has a *Dump*, used by various instructions to hold intermediate states of the machine.

Another essential part of the machine is the *Dump* which is used when working with *Use*, *Split*, *Measure* and *Call*. The *DumpElement* for the first three types,

```
data (Quantum a b) ⇒ DumpElement a b =
  DumpQControl{
    returnLabel :: Label,
    branchesToDo :: [(QuantumStack a b, Label)],
    resultQStack :: QuantumStack a b,
    saveClsStack :: ClassicalStack} |
  DumpCall{returnLabel :: Label,
    returnEp :: EntryPoint,
    saveClsStack :: ClassicalStack}
  deriving Show
type Dump a b = [DumpElement a b]
```

Figure D.3: Haskell definition of the Dump

DumpQControl, provides all that is needed to complete those instructions on the various branches of a *Split*, *Use* or *Measure*. It has the following items:

- *returnLabel*: The *Label* to continue with when the instruction is complete. This is normally the instruction following the quantum control.
- *saveClsStack*: As the *ClassicalStack* is reset before each invocation of a branch and then restored after all are completed, it is saved here.
- *resultQStack*: As the partial branches are processed, the resulting quantum stacks are accumulated, merging after each branch.

- *branchesToDo*: This is a list of pairs, whose first item is derived from the quantum stack. Each one is a single branched *Qstack*. The second item in the pair is the label for the code to execute at this point. Note that in the the case of *Use* for integers, it would have been possible to simply use a single label as all branches execute the same code. However, doing it this way allows for a single data element for Quantum Control.

D.2.4 Description of the quantum stack machine and its operation

The quantum stack machine consists of a variety of stacks, instructions registers and other bookkeeping apparati.

The machine state

As discussed earlier in section 5.4 on page 105, there are a variety of descriptions for the the machine state. In that section, we described BQSM, LBQSM, CQSM and QSM.

These are shown in figure D.4 on the next page for LBQSM, figure D.5 on the following page for BQSM and figure D.6 on page 187 for CQSM and QSM.

Note that in all cases, there are three items to hold what was described as a simple list of code ($\mathcal{C}$) in that earlier section. This is the *runningCode*, *instructionPointer* and *codeMem* in BQSM, with similar names in the other states.

Transforming between the state types

The four stages of machine descriptions were created because various instructions are more naturally implemented on different stages. However, the program must

```
data (Quantum a b) ⇒ BMS a b =
  BMS{quantumStack :: QuantumStack a b,
    classicalStack :: ClassicalStack,
    runningCode :: [Instruction a],
    codeMem :: Memory a,
    instructionPointer :: (EntryPoint, Label),
    dump :: (Dump a b),
    namesupply :: Int}
  deriving Show
```

Figure D.4: Haskell definition of the basic machine state

```
data (Quantum a b) ⇒ LBMS a b =
  LBMS{quantumStackLbld :: Controlled (QuantumStack a b),
    classicalStackLbld :: ClassicalStack,
    runningCodeLbld :: [Instruction a],
    codeMemLbld :: Memory a,
    instructionPointerLbld :: (EntryPoint, Label),
    dumpLbld :: (Dump a b),
    namesupplyLbld :: Int}
  deriving Show
```

Figure D.5: Haskell definition of the labelled machine state

```
data (Quantum a b) ⇒ CMS a b =
  CMS{cmsCodeMem :: Memory a,
    controlStack :: [ControlStack a b],
    cmsInstructionPointer :: (EntryPoint, Label),
    cmsRunningCode :: [Instruction a],
    ctrldMS :: [((Int, ClassicalStack),
      (Controlled (QuantumStack a b), Dump a b))]
    }
  deriving Show
type MachineState a b = Stream (CMS a b)
```

Figure D.6: Haskell definition of the controlled and complete machine states

be able lift all functions defined on any of the states to *MachineState*, the infinite list of CQSM items.

Lift functions are provided to do this such that

$$(\text{BQSM} \to \text{BQSM}) \xrightarrow{\texttt{lift}} (\text{LBQSM} \to \text{LBQSM}) \xrightarrow{\texttt{lift}} (\text{CQSM} \to \text{CQSM}).$$

First, to lift endo-functions defined on BQSM to LBQSM, define helper functions *decontrolLbld* and *recontrolLbld* which pull a *QuantumStack* element from the *Controlled* type and then reapply that control. Lifting the function f is then just

$$\texttt{lift f} = \texttt{recontrolLbld} \circ ([\texttt{id}, \texttt{f}]) \circ \texttt{decontrolLbld}. \tag{D.2}$$

```
decontrolLbld :: (Quantum a b) ⇒ LBMS a b →
  (BMS a b, QuantumStack a b → Controlled (QuantumStack a b))
decontrolLbld (LBMS cqs cs rc cm ip d n) =
  (BMS qs cs rc cm ip d n, ctlr)
  where (qs, ctlr) = splitcontrolled cqs
recontrolLbld :: (Quantum a b) ⇒ (BMS a b, QuantumStack a b →
  Controlled (QuantumStack a b)) → LBMS a b
recontrolLbld (BMS qs cs rc cm ip d n, ctlr) = LBMS (ctlr qs) cs rc cm ip d n
```

```
liftBMStoLBMS :: (Quantum a b) ⇒ (BMS a b → BMS a b) →
   LBMS a b → LBMS a b
liftBMStoLBMS f = recontrolLbld ∘ (app1of2 f) ∘ decontrolLbld
```

In a similar way, to lift endo-functions defined on BQSM up to to CQSM, we define helpers functions *decontrol* and *recontrol*. In this case, these functions are somewhat more complicated, producing and consuming a list of LBQSM items and the CQSM list of *ControlStack*, using other subordinate functions described below. As before,lifting the function f is then just

$$\mathtt{lift}\ \mathsf{f} = \mathtt{recontrol} \circ ([\mathtt{id}, \mathtt{map}\ \mathsf{f}]) \circ \mathtt{decontrol}. \tag{D.3}$$

```
liftLBMStoCMS :: (Quantum a b) ⇒ (LBMS a b → LBMS a b) →
         CMS a b → CMS a b
liftLBMStoCMS f = recontrol ∘ (app1of2 (List.map f)) ∘ decontrol
decontrol :: (Quantum a b) ⇒ CMS a b → ([LBMS a b], [ControlStack a b])
decontrol (CMS cm ctls ip rc cmss)
    = (List.map (buildLBMS cm ip rc) cmss, ctls)
recontrol :: (Quantum a b) ⇒ ([LBMS a b], [ControlStack a b]) → CMS a b
recontrol (cmss, ctls)
    = CMS cm ctls ip rc cmss′
      where (cm, ip, rc) = commonLBMS $ head cmss
         cmss′ = List.map stacksOfLBMS cmss
```

Three helper functions contribute to *decontrol* and *recontrol*. All three are simple structure manipulations.

```
buildLBMS :: (Quantum a b) ⇒ Memory a → (EntryPoint, Label) →
   [Instruction a] →
   ((Int, ClassicalStack),
      (Controlled (QuantumStack a b), Dump a b)) →
```

```
  LBMS a b
buildLBMS cm ip rc ((n, s), (cqs, d)) = LBMS cqs s rc cm ip d n
commonLBMS :: (Quantum a b) ⇒ LBMS a b →
  (Memory a, (EntryPoint, Label), [Instruction a])
commonLBMS msLbld =
    (codeMemLbld msLbld,
      instructionPointerLbld msLbld,
      runningCodeLbld msLbld)
stacksOfLBMS :: (Quantum a b) ⇒ LBMS a b →
  ((Int, ClassicalStack),
    (Controlled (QuantumStack a b), Dump a b))
stacksOfLBMS msLbld =
  ((namesupplyLbld msLbld, classicalStackLbld msLbld),
    (quantumStackLbld msLbld, dumpLbld msLbld))
makeCMS :: (Quantum a b) ⇒ BMS a b → CMS a b
makeCMS ms = cms
    where bms′ = recontrolLbld (ms, Full)
      cms = recontrol ([bms′], [])
```

Lifting an endo-function on BQSM is accomplished by composing the two intermediate lifts.

```
liftBMStoCMS :: (Quantum a b) ⇒ (BMS a b → BMS a b) →
  CMS a b → CMS a b
liftBMStoCMS = liftLBMStoCMS ∘ liftBMStoLBMS
```

Lifting to a *Stream* is accomplished using the *monad* definition of *Stream*:

```
liftS :: (a → a) → Stream a → Stream a
liftS f strm
  = do s ← strm
    return $ f s
```

D.2.5 Machine transitions

Top level functions

The function *go* picks up the current instruction *at a particular depth in the infinite list of machine states* and continues calling *runMachine* until it runs out of instructions.

```
go :: (Quantum a b) ⇒ Int → MachineState a b → MachineState a b
go depth mstate
   = let ci = cmscurrIns $ hd $ dropI depth mstate
     in case ci of
        Just _ → go depth $ runMachine mstate
        Nothing → mstate
```

The function *runMachine* lifts the transition function *runCMS′*, while applying the special logic needed for *Call*, *Jump* and *CondJump*.

```
runMachine′ :: (Quantum a b) ⇒ MachineState a b → MachineState a b
runMachine′ mstate@(Stream mshead mstatetl)
   = newstate
     where ci = cmscurrIns mshead
        newstate =
          case ci of
            (Just (Call n entpt)) →
              do cms ← mstate
                 return $ liftBMStoCMS (enterFunc n entpt) cms
            _ → runMachine mstate
runMachine mstate@(Stream mshead mstatetl)
   = newstate
     where ci = cmscurrIns mshead
        newstate =
          case ci of
            Nothing → Stream mshead $ runMachine mstatetl
            (Just (Call n entpt)) → rcall n entpt mstate
            (Just (Jump lbl)) →
              if $lbl \leqslant (cmscurrIp\ mshead)$
              then $Stream\ (cmsZeroTheQstack\ mshead)$ $
                 $runMachine\ mstatetl$
              else $Stream\ (runCMS'\ ci\ mshead)$ $
                 $runMachine\ mstatetl$
            $(Just\ (CondJump\ lbl)) \rightarrow$
              if $lbl \leqslant (cmscurrIp\ mshead)$
              then $Stream\ (cmsZeroTheQstack\ mshead)$ $
                 $runMachine\ mstatetl$
              else $Stream\ (runCMS'\ ci\ mshead)$ $
                 $runMachine\ mstatetl$
            $(Just\ \_) \rightarrow Stream\ (runCMS'\ ci\ mshead)$ $
              $runMachine\ mstatetl$
```

Recursive Function Transitions

In the Quantum stack machine, all function calls are treated as recursive calls. This means that each function call directly affects the *Stream* of the machine state.

The function *rcall* makes this happen explicitly by creating a new *Stream*. The head of this *Stream* is always the non-terminating representation, i.e. a zeroed stack. The tail creates a new stack, starting with the current stack values, and resetting the instruction pointer to the first instruction of the function, and altering the *Dump* and classical stack appropriately.

The effect of this is that when calling functions, one must look further and further down the stream of stacks to actually see results.

```
$rcall :: (Quantum\ a\ b) \Rightarrow Int \rightarrow EntryPoint \rightarrow MachineState\ a\ b \rightarrow$
   $MachineState\ a\ b$
$rcall\ n\ entpt\ mstate$
    $= Stream\ ms0\ mstaterest$
      where $cms = hd\ mstate$
         $ms0 = liftBMStoCMS\ (incCp \circ zeroTheQstack)\ cms$
```

```
        mstate' = tl mstate
        mstaterest = runMachine' mstate'
enterFunc :: (Quantum a b) ⇒ Int → EntryPoint → BMS a b → BMS a b
enterFunc n entpt ms
  = BMS (quantumStack ms) newcs newcd
    (codeMem ms) newip (d : dump ms) (namesupply ms)
    where newip = (entpt, 0)
      newcd = getCode (codeMem ms) newip
      (newcs, cs') = stackSplitAt n $ classicalStack ms
      d = DumpCall (1 + (currIp ms)) (ep ms) cs'
```

Machine transitions for each instruction

The function *runBMS′* implements the actual machine transitions for each state of the machine.

```
runCMS' :: (Quantum a b) ⇒ Maybe (Instruction a) → CMS a b → CMS a b
runCMS' Nothing c = c
runCMS' (Just ins) c = runCMS ins c
runCMS :: (Quantum a b) ⇒ Instruction a → CMS a b → CMS a b
runCMS AddCtrl cms =
  cmsIncCp cms{controlStack = addControl $ controlStack cms}
runCMS UnCtrl cms =
  cmsIncCp cms{controlStack = tlcs,
    ctrldMS = newCtrldMS}
    where (tlcs, newCtrldMS) = unControl (controlStack cms) (ctrldMS cms)
runCMS QCtrl cms =
  cmsIncCp cms{controlStack = ncs,
    ctrldMS = newCtrldMS}
  where (ncs, newCtrldMS) = qControl (controlStack cms) (ctrldMS cms)
runCMS (QApply n transop) cms
  = cmsIncCp $ cms{ctrldMS = List.map (qapply n transop) $ ctrldMS cms}
runCMS ins cms = liftBMStoCMS (runBMS' ins) cms
qapply :: (Quantum a b) ⇒ Int → UnitaryOp →
    ((Int, ClassicalStack), (Controlled (QuantumStack a b), d)) →
    ((Int, ClassicalStack), (Controlled (QuantumStack a b), d))
```

```
qapply i transop ((n, cs), (cq, d))
    = ((n', cs'), (cq', d))
      where ((n', cs'), cq') = qapply' i transop n cs cq
qapply' :: (Quantum a b) ⇒ Int → UnitaryOp → Int → ClassicalStack →
      Controlled (QuantumStack a b) →
      ((Int, ClassicalStack), Controlled (QuantumStack a b))
qapply' i transop n cs cq =
   ((n, Data.Stack.fromList cs'), cq''')
   where (topn, cs') = splitAt i $ QSM.ClassicalStack.toList cs
      trans = getTransform topn transop
      (names, cq') = unzipControl $ fmap (prepare (qorder trans)) cq
      cq'' = cTransform trans cq'
      cq''' = fmap (rotateInOrder $ reverse names) cq''
```

Node construction is done with the four instructions *QLoad*, *QCons*, *QMove* and *QBind*.

```
runBMS' :: (Quantum a b) ⇒ Instruction a → BMS a b → BMS a b
runBMS' (QLoad nm v) ms
    = let qs = quantumStack ms
      in incCp $ ms{quantumStack = StackQbit nm $ qv [((v, v), qs)]}
runBMS' (QCons nm c) ms
    = let qs = quantumStack ms
      in incCp $ ms{quantumStack = StackCons nm $ dv [(c, ([], qs))]}
runBMS' (QBind nm) ms
    = let qs = quantumStack ms
      ns = namesupply ms
      (ns', qs') = bind nm ns qs
      in incCp $ ms{quantumStack = qs', namesupply = ns'}
runBMS' (QMove nm) ms
    = let qs = quantumStack ms
      (val, cs) = popM $ classicalStack ms
      cval = case val of
         Nothing → Left 0
         (Just a) → a
```

```
        in incCp $ ms{quantumStack = StackInt nm $ cv [(cval, qs)],
          classicalStack = cs}
```

Node destruction is the natural complement of construction and is done by the two instructions *QUnbind* and *QDiscard*. Note that *QDiscard* works on all types of nodes, but requires them to have only a single sub-stack in each case.

```
runBMS′ (QDelete) ms
    = let qs = quantumStack ms
      in incCp $ ms{quantumStack = discard qs}
runBMS′ (QDiscard) ms
    = let qs = quantumStack ms
          bc = branchCount qs
      in if bc ≡ 1
         then incCp $ ms{quantumStack = discard qs,
            classicalStack = pushM (topVal qs) (classicalStack ms)}
         else error badDiscard
runBMS′ (QUnbind nm) ms
    = let qs = quantumStack ms
      in incCp $ ms{quantumStack = unbind nm qs}
```

Quantum stack manipulation consists of two instructions, *QPullup* and *QName*.

```
runBMS′ (QPullup nm) ms
    = let qs = quantumStack ms
      in incCp $ ms{quantumStack = rotateup nm qs}
runBMS′ (QName oldnm newnm) ms
    = let qs = quantumStack ms
      in incCp $ ms{quantumStack = firstRenameInStack oldnm newnm qs}
```

Quantum control instructions allow us to apply instructions to different sub-branches of a node.

The details of the functions used in this code are below.

```
runBMS′ (Use lbl) ms
    = let qs = quantumStack ms
      in doUse qs lbl ms
runBMS′ (Split cns_lbl_list) ms
    = let qs = quantumStack ms
      in doSplit qs (Map.fromList cns_lbl_list) ms
runBMS′ (Measure lbl0 lbl1) ms
    = let qs = quantumStack ms
      in doMeasure qs lbl0 lbl1 ms
runBMS′ EndQC ms
    = let dmp = head $ dump ms
      in doqcEnd dmp ms
```

Classical control comprises the standard jump / call / return types of instructions. Note that the transitions here are based on the assumption one is deep enough in the stream to actually do a *Call*. For example, at the start of the stream, a *Call* instruction actually just returns a zeroed quantum stack. See the sub-section on function calls: appendix D.2.5 on page 191.

```
runBMS′ (Jump lbl) ms
    | lbl > (currIp ms)
       = ms{runningCode = newcode, instructionPointer = newptr}
    | otherwise = error backwardsJump
         where newptr = (ep ms, lbl)
            newcode = getCode (codeMem ms) newptr
runBMS′ (CondJump lbl) ms
    = let (value, cs) = popM $ classicalStack ms
      in case value of
         Just (Right False) → runBMS′ (Jump lbl) ms{classicalStack = cs}
```

```
        _ → incCp ms{classicalStack = cs}
runBMS′ (Call _ _) _ = error wrongCall
runBMS′ (Return n) ms
      = let cs = classicalStack ms
          d = head $ dump ms
          in BMS (quantumStack ms) (addn n cs $ saveClsStack d)
             (getCode (codeMem ms) (returnEp d, returnLabel d))
             (codeMem ms)
             (returnEp d, returnLabel d)
             (tail $ dump ms)
             (namesupply ms)
```

Classical instructions manipulate the classical stack in the machine allowing standard integer and Boolean operations.

```
runBMS′ CPop ms
    = incCp ms{classicalStack = snd $ popM $ classicalStack ms}
runBMS′ (CGet n) ms
    = let cs = classicalStack ms
      in incCp ms{classicalStack = push (stackElem cs n) cs}
runBMS′ (CPut n) ms
    = let cs = classicalStack ms
      in incCp ms{classicalStack = stkput n cs}
runBMS′ (CApply cop) ms
    = let cs = classicalStack ms
      in incCp ms{classicalStack = (getStackOp cop) cs}
runBMS′ (CLoad value) ms
    = let cs = classicalStack ms
      in incCp ms{classicalStack = push value cs}
runBMS′ (NoOp) ms = incCp ms
```

Support for data casing, measure and use

The function *doUse* handles the work of splitting a *StackInt* node down so that a series of instructions may be executed on each of the subbranches. Applying this to a *StackZero* element results in a no-operation, while applying it to anything else will cause a machine exception. The general pattern of this function is repeated in *doSplit* and *doMeasure* below.

```
doUse :: (Quantum a b) ⇒ QuantumStack a b → Label → BMS a b →
   BMS a b
doUse qs@(StackZero) _ ms
    = incCp ms
doUse qs@(StackInt _ _) lbl ms
    = let qss = breakQuantum qs
      cs = classicalStack ms
      dumpU = DumpQControl (1 + (currIp ms)) (zip qss $ repeat lbl)
         (fromInteger 0) cs
      in BMS (fromInteger 0) cs [EndQC] (codeMem ms)
         (instructionPointer ms) (dumpU : dump ms) (namesupply ms)
doUse _ _ _ = error useDataCheck
```

The *doSplit* instruction works only on *StackCons* nodes.

```
doSplit :: (Quantum a b) ⇒ QuantumStack a b → Map Constructor Label →
   BMS a b → BMS a b
doSplit qs@(StackZero) _ ms
    = incCp ms
doSplit qs@(StackCons _ _) jumpMap ms
    = let qss = breakQuantum qs
      cs = classicalStack ms
      dumpU = DumpQControl (1 + (currIp ms)) (associateCons jumpMap qss)
         (fromInteger 0) cs
      in BMS (fromInteger 0) cs [EndQC] (codeMem ms)
         (instructionPointer ms) (dumpU : dump ms) (namesupply ms)
doSplit _ _ _ = error splitDataCheck
```

Finally, *doMeasure* measures a **qubit** and sets up the system for executing code on its $|0\rangle$ and $|1\rangle$ branches. Recall that the density matrix notation, which is implemented by the quantum stack, has four values for a **qubit**. The two off-diagonal values are discarded immediately by this instruction, leaving the diagonal values ($|0\rangle$ and $|1\rangle$).

```
doMeasure :: (Quantum a b) ⇒ QuantumStack a b → Label → Label →
   BMS a b → BMS a b
doMeasure qs@(StackZero) _ _ ms
    = incCp ms
doMeasure qs@(StackQbit _ _) lbl0 lbl1 ms
    = let qss = breakQuantum qs
      qss′ = List.filter (λ(StackQbit _ qvals) →
         (uncurry (≡) ∘ head ∘ keys) qvals) qss
      cs = classicalStack ms
      dumpU = DumpQControl (1 + (currIp ms)) (assocQbs qss′ lbl0 lbl1)
         (fromInteger 0) cs
      in BMS (fromInteger 0) cs [EndQC] (codeMem ms)
         (instructionPointer ms) (dumpU : dump ms) (namesupply ms)
doMeasure _ _ _ _ = error measureDataCheck
```

Once a split, measure or use is started, intermediate results are accumulated on the dump. The structure of this dump element is shared by all three of the instructions. Stepping through the sub-results and finalizing the result is done by the *EndQC* instruction, which uses the function *docqEnd* below. The first part of the definition below handles the case when all sub-stacks have been done. This returns the quantum stack to a merge of all the intermediate results and removes the intermediate result element from the dump.

```
doqcEnd :: (Quantum a b) ⇒ DumpElement a b → BMS a b → BMS a b
doqcEnd (DumpQControl ret [] resultqs savecs) ms
    = BMS (resultqs + (quantumStack ms)) savecs
      (getCode (codeMem ms) ((ep ms), ret))
      (codeMem ms) ((ep ms, ret)) (tail $ dump ms) (namesupply ms)
```

In the second part of the definition, the dump element is changed to add in the current intermediate result, while removing the next leg to be executed and making it the current quantum stack.

```
doqcEnd (DumpQControl ret ((nextqs, nextlbl) : qss) resultqs savecs) ms
    = let dmp = DumpQControl ret qss
      (resultqs + (quantumStack ms)) savecs
      in BMS nextqs savecs
         (getCode (codeMem ms) ((ep ms), nextlbl))
         (codeMem ms) ((ep ms, nextlbl)) (dmp : (tail $ dump ms))
            (namesupply ms)
doqcEnd _ _ = error qcontrolBadEnd
```

Some of the details of associating various sub-stacks with the appropriate labels in the executing code is broken out as separate functions below.

```
assocQbs :: (Quantum a b) ⇒ [QuantumStack a b] → Label → Label →
   [(QuantumStack a b, Label)]
assocQbs [] _ _ = []
assocQbs ((qs@(StackQbit _ qvals)) : qss) lbl0 lbl1
    = let (v, _) = head $ keys qvals
      in (if (v ≡ head basis) then (qs, lbl0)
         else (qs, lbl1)) : assocQbs qss lbl0 lbl1
assocQbs _ _ _ = error measureDataCheck
associateCons :: (Quantum a b) ⇒ Map Constructor Label →
   [QuantumStack a b] → [(QuantumStack a b, Label)]
associateCons jumpMap [] = []
associateCons jumpMap ((qs@(StackCons _ dvals)) : qss)
    = let cons = head $ keys dvals
```

```
        lbl = findWithDefault endQCLabel cons jumpMap
      in (qs, lbl) : associateCons jumpMap qss
associateCons _ _ = error splitDataCheck
```

Support for node construction and deletion

The act of binding nodes to data nodes requires traversing the quantum stack for the desired name and creating a new name for it. That new name is then attached to the data node.

```
bind :: (Quantum a b) ⇒ StackAddress → Int → QuantumStack a b
    → (Int, QuantumStack a b)
bind nm ns theQstack@(StackCons dname dvals)
    | 1 ≡ Map.size dvals
      = let [(c, (bvs, qs))] = Map.toList dvals
            ns' = ns + 1
            numBound = length bvs
            newnm = "c" ++ (show ns) ++ "_" ++
                (show numBound) ++ "_" ++ nm
          in (ns', StackCons dname $ dv [(c, (newnm : bvs,
                firstRenameInStack nm newnm qs))])
    | otherwise = error bindMultiCons
bind _ _ _ = error bindDataCheck
```

Unbinding node from data nodes requires renaming the bound node and removing from the list of those attached to the data node.

```
unbind :: (Quantum a b) ⇒ StackAddress → QuantumStack a b
        → QuantumStack a b
unbind nm (StackCons dname dvals)
    | 1 ≡ Map.size dvals
      = let [(c, (bvs, qs))] = Map.toList dvals
          numBound = length bvs
          oldnm = head bvs
          qs' = rotateup oldnm qs
          in if (numBound ≡ 0)
            then error unbindNothingBound
            else StackCons dname $ dv [(c, (tail bvs,
              setName nm qs'))]
    | otherwise = error unbindBadCons
unbind _ _ = error unbindDataCheck
```

Support for unitary transforms

Applying unitary transforms is done via matrix multiplication. To do the multiplication requires transforming the stack to a matrix, then using the new values of the matrix to reset the stack.

The first defined function, *cTransform*, will transform a *Controlled* quantum stack. This will result in one of four possibilities:

$$\text{cTransform T Q} = \begin{cases} \mathsf{Q} & \text{IdentityOnly control} \\ \mathsf{TQ} & \text{LeftOnly control} \\ \mathsf{QT}^* & \text{RightOnly control} \\ \mathsf{TQT}^* & \text{Full control} \end{cases} \tag{D.4}$$

```
cTransform :: (Quantum a b) ⇒ Trans b → Controlled (QuantumStack a b) →
  Controlled (QuantumStack a b)
cTransform mtrans q@(IdentityOnly q') = q
cTransform mtrans (LeftOnly q)
    = LeftOnly $ trimStack $ ctrans' (qorder mtrans) tr q
      where tr = λqs → matByStack mtrans qs
cTransform mtrans (RightOnly q)
    = RightOnly $ trimStack $ ctrans' (qorder mtrans) tr q
    where tr = λqs → stackByMat qs $ conjtrans mtrans
cTransform mtrans (Full q)
  = Full $ trimStack $ ctrans′ (qorder mtrans) tr q
    where tr = λqs → genmatmul (+) (⋊)
      (matByStack mtrans qs) (conjtrans mtrans)
ctrans′ :: (Quantum a b) ⇒ Int →
  (QuantumStack a b → Matrix (QuantumStack a b)) →
  QuantumStack a b → QuantumStack a b
ctrans′ order f q = setValsFromMat order (f q) q
```

The function *matByStack* performs a matrix multiplication of the subject matrix with a quantum stack. The quantum stack is first transformed to a matrix of the appropriate size. The result is a matrix of quantum stacks.

```
matByStack :: (Quantum a b) ⇒ Matrix b → QuantumStack a b →
  Matrix (QuantumStack a b)
matByStack m StackZero = zeroMat $ qorder m
matByStack _ st@(StackData _)
  = error $ matByStackError "data" (show st)
matByStack _ st@(StackCons _ _)
  = error $ matByStackError "Constructor" (show st)
matByStack m st@(StackInt _ _)
  = error $ matByStackError "Int" $
      (show m) ++ (show st)
matByStack m sqb@(StackQbit _ _)
  = newVals
    where newVals = genmatmul (+) (⋉) m (stackToMat order sqb)
      order = qorder m
```

The function *stackByMat* is complementary to matByStack, multiplying a quantum stack on the right by a matrix and is used in *cTransform* for *RightOnly* controlled transforms.

```
stackByMat :: (Quantum a b) ⇒ QuantumStack a b → Matrix b →
   Matrix (QuantumStack a b)
stackByMat StackZero m = zeroMat $ qorder m
stackByMat st@(StackData _) _
    = error $ stackByMatError "data" (show st)
stackByMat st@(StackCons _ _) _
    = error $ stackByMatError "Constructor" (show st)
stackByMat st@(StackInt _ __) m
    = error $ stackByMatError "Int" $
          (show m) ++ (show st)
stackByMat sqb@(StackQbit _ _) m
    = newVals
      where newVals = genmatmul (+) (⋊) (stackToMat order sqb) m
         order = qorder m
```

The function *setValsFromMat* and dependent function *svfm* extract the values from a matrix of quantum stacks and assigns them as new sub-stacks of the argument quantum stack.

```
setValsFromMat :: (Quantum a b) ⇒ Int →
   Matrix (QuantumStack a b) →
   QuantumStack a b →
   QuantumStack a b
setValsFromMat n m qb@(StackQbit s qvl) = svfm (names n qb) n m qb
setValsFromMat n m q = error $ setValsTypeError n m q
```

```
svfm :: (Quantum a b) ⇒ [String] → Int → Matrix (QuantumStack a b) →
   QuantumStack a b → QuantumStack a b
svfm nms 1 m qb@(StackZero)
    = StackQbit (head nms) (qv [((a, b), indexM (ei a) (ei b) m) |
       a ← basis, b ← basis])
svfm nms n m sq@(StackZero)
    = StackQbit (head nms) (qv [((a, b), svfm (tail nms)
       (n − 1) (grab (ei a) (ei b) m) (StackZero))
       | a ← basis, b ← basis])
```

```
svfm nms n m sq@(StackQbit _ qvl)
   | n ≡ 1 = StackQbit (head nms) (qv [((a,b), indexM (ei a) (ei b) m) |
     a ← basis, b ← basis])
   | n > 1 = StackQbit (head nms) (qv [((a,b), svfm (tail nms)
     (n − 1) (grab (ei a) (ei b) m) (val (a,b) qvl))
      | a ← basis, b ← basis])
   | otherwise = error setValsError
svfm _ _ _ _
   = error setValsDataError
```

Converting a stack to a matrix is done by a recursive descent down the stack. For example, if converting to a 4×4 matrix (two **qubit**s), *stackToMat* converts the four sub matrices of the second level **qubit**. These four 2×2 matrices are then amalgamated via "pasting" to create a 4×4 matrix.

```
stackToMat :: (Quantum a b) ⇒ Int → QuantumStack a b →
              Matrix (QuantumStack a b)
stackToMat 1 qs
   = [[val (a,b) (qvalues qs)
         | b ← basis]
         | a ← basis]
stackToMat n qs
   | n > 1 = reduceM theMat
   | otherwise = error stackToMatError
   where theMat = [[findWithDefault (zeroMat (pow 2 (n − 1))) (a,b)
                        prev | b ← basis] | a ← basis]
     prev = Map.map (stackToMat (n − 1)) (qvalues qs)
pow :: Int → Int → Int
pow n m = foldl (∗) 1 $ take m $ repeat n
zeroMat n = (take n $ repeat $ take n $ repeat StackZero)
```

Miscellaneous transition support

This section has a variety of simple functions used in the operation and setup of the quantum machine.

```
cmsZeroTheQstack :: (Quantum a b) ⇒ CMS a b → CMS a b
cmsZeroTheQstack  = liftBMStoCMS zeroTheQstack

zeroTheQstack :: (Quantum a b) ⇒ BMS a b → BMS a b
zeroTheQstack ms = ms{quantumStack = fromInteger 0}

noCode :: Memory a
noCode = Map.singleton mainproglabel []

startMachine :: (Quantum a b) ⇒ (QuantumStack a b) → Memory a →
   MachineState a b

startMachine qs mem =
   return $ CMS mem [] ("main",0) (getCode mem ("main",0))
      [((0, emptyStack), (Full qs, []))]

hasProc :: (Basis a) ⇒ Memory a → EntryPoint → Bool
hasProc = flip (Map.member)

getCode :: (Basis a) ⇒ Memory a → (EntryPoint, Label) → [Instruction a]
getCode mem (ep, start)
    | start ≡ endQCLabel = [EndQC]
    | otherwise = drop start $
                     findWithDefault emptyCodeBlock ep mem

ep :: (Quantum a b) ⇒ BMS a b → EntryPoint
ep  = fst ∘ instructionPointer

currIp :: (Quantum a b) ⇒ BMS a b → Label
currIp = snd ∘ instructionPointer

cmscurrIp :: (Quantum a b) ⇒ CMS a b → Label
cmscurrIp = snd ∘ cmsInstructionPointer

currIns :: (Quantum a b) ⇒ BMS a b → Maybe (Instruction a)
currIns = ci′ ∘ runningCode

cmscurrIns :: (Quantum a b) ⇒ CMS a b → Maybe (Instruction a)
cmscurrIns = ci′ ∘ cmsRunningCode

ci′ :: [a] → Maybe (a)
ci′ ([]) = Nothing
ci′ (a : _) = Just a

emptyCodeBlock :: (Basis a) ⇒ Code a
emptyCodeBlock = []
```

```
incCp :: (Quantum a b) ⇒ BMS a b → BMS a b
incCp ms = ms{instructionPointer = (ep ms, 1 + (currIp ms)),
      runningCode = tail $ runningCode ms}
cmsIncCp :: (Quantum a b) ⇒ CMS a b → CMS a b
cmsIncCp = liftBMStoCMS incCp
endQCLabel = −1
```

D.3 Basic data elements of the QSM

D.3.1 Permutations

Our permutations are written as a pair between the size of the permutation and a function from *Int* → *Int* which gives the actual permutation.

```
data Perm = Perm{domain :: Int,
   runPerm :: Int → Int}
```

We provide a way to extend the permutation to higher domains. This will be extensively used in the creation of transformations from permutations. The lifted permutation will be the identity on all elements not in the original domain.

```
liftPerm :: Int → Perm → Perm
liftPerm newdomain p =
   if newdomain < (domain p)
      then error "Perms can not be restricted"
      else Perm newdomain f
         where f i | i ⩽ domain p = (runPerm p) i
                   | otherwise = i
permuteList :: Perm → [a] → [a]
permuteList p aas = [aas !! ((runPerm p) j) |
   j ← [0 . . ((length aas) − 1)]]
```

```
orderFindPerm :: Int → Int → Perm
orderFindPerm n k
      = Perm (maxNum n) (λa → aToTheKModN n k a)
numBits :: Int → Int
numBits = numBits′ 0
numBits′ :: Int → Int → Int
numBits′ size 0 = size
numBits′ size n = numBits′ (1 + size) (n ‘shiftR‘ 1)
maxNum :: Int → Int
maxNum n = 2 ↑ (numBits n) − 1
aToTheKModN :: Int → Int → Int → Int
aToTheKModN n k a
   | a ⩾ n = a
   | otherwise = (a ↑ k) ‘mod‘ (maxNum n)
```

D.3.2 Definition of the Stack datatype

The *Stack* is implemented by a *List*. We define instances of *Functor*, *Monad* and *MonadPlus* for the type.

```
newtype Stack a = Stack [a]
  deriving Show
instance Functor Stack where
  fmap f (Stack elts) = Stack (map f elts)
instance Monad Stack where
  return a = Stack [a]
  m ≫= k = stkfoldr (concStack ∘ k) (Stack []) m
  m ≫ k = stkfoldr (concStack ∘ (λ_ → k)) (Stack []) m
  fail _   = Stack []
instance MonadPlus Stack where
  mzero = Stack []
  mplus (Stack a) (Stack b) = Stack (a ++ b)
```

A variety of manipulation functions are provided, giving the standard push and pop functionality of a stack.

```
emptyStack :: Stack a
emptyStack = mzero
push :: a → Stack a → Stack a
push a (Stack as) = Stack (a : as)
pushM :: Maybe a → Stack a → Stack a
pushM Nothing s = s
pushM (Just a) s = push a s
popM :: Stack a → (Maybe a, Stack a)
popM (Stack []) = (Nothing, Stack [])
popM (Stack (a : as)) = (Just a, Stack as)
pop :: Stack a → (a, Stack a)
pop s = let (v, s') = popM s
   in case v of
           Nothing → error "Stack is empty on pop"
           Just v → (v, s')
popNum :: (Num a) ⇒ Stack a → (a, Stack a)
popNum s = let (v, s') = popM s
           in case v of
                Nothing → (fromInteger 0, s)
                Just v → (v, s')
addn :: Int → Stack a → Stack a → Stack a
addn n (Stack e1) (Stack e2)
   = Stack ((take n e1) ++ e2)
multiPop :: (Num a) ⇒ Int → Stack a → (a, Stack a)
multiPop 1 stk = pop stk
multiPop n stk
   | n > 1 = pop $ snd $ multiPop (n − 1) stk
   | otherwise = error "Can not pop 0 or less elements"
getM :: Stack a → Maybe a
getM (Stack []) = Nothing
getM (Stack (a : as)) = Just a
get :: Stack a → a
get s = let v = getM s
   in case v of
                   Nothing → error "Stack is empty on get"
```

```
                 Just v → v
stackElem :: Stack a → Int → a
stackElem (Stack aelems) i
   | i ⩾ 0 = aelemsList. !! i
   | (−i) ⩽ (length aelems) =
                aelemsList. !! ((length aelems) + i)
   | otherwise =
                aelemsList. !!
                   (((length aelems) + i) ‘mod‘ (length aelems))
isEmpty :: Stack a → Bool
isEmpty (Stack [ ]) = True
isEmpty (Stack (a : as)) = False
toList :: Stack a → [a]
toList (Stack a) = a
fromList :: [a] → Stack a
fromList a = Stack a
stkfoldr :: (a → b → b) → b → Stack a → b
stkfoldr f i (Stack a) = foldr f i a
concStack :: Stack a → Stack a → Stack a
concStack = mplus
stackSplitAt :: Int → Stack a → (Stack a, Stack a)
stackSplitAt n (Stack elms) = (Stack l, Stack r)
   where (l, r) = splitAt n elms
```

D.3.3 Definition of extra functions on Either

```
fromLeft          :: Either a b → a
fromLeft (Left a) = a
fromLeft _        = error "fromLeft: Needed left value"
fl2      ::    Either a b → Either a b → (a, a)
fl2 x y  =    (fromLeft x, fromLeft y)
fromRight           :: Either a b → b
fromRight (Right b) = b
fromRight _         = error "fromRight: Needed right value"
fr2      :: Either a b → Either a b → (b, b)
```

$fr2\ x\ y = (fromRight\ x, fromRight\ y)$
$eLeftfun\ \ :: (a \to a \to a) \to Either\ a\ b \to Either\ a\ b \to Either\ a\ b$
$eLeftfun\ f = curry\ \$\ Left \circ (uncurry\ f) \circ (uncurry\ fl2)$
$eRightfun\ \ :: (b \to b \to b) \to Either\ a\ b \to Either\ a\ b \to Either\ a\ b$
$eRightfun\ f = curry\ \$\ Right \circ (uncurry\ f) \circ (uncurry\ fr2)$
$eLtoRightfun\ \ :: (a \to a \to b) \to Either\ a\ b \to Either\ a\ b \to Either\ a\ b$
$eLtoRightfun\ f = curry\ \$\ Right \circ (uncurry\ f) \circ (uncurry\ fl2)$

D.3.4 Classical data and the classical operations

A datatype used as elements of the classical stack in the quantum stack machine.

```
type ClassicalData = Either Int Bool
data ClassicalOp = CAdd | CSub | CTimes | CDiv |
   CMod | CRem | CNeg | COr | CAnd | CXor |
   CNot | CEq | CNeq | CLt | CLte | CGt | CGte |
   CShl | CShr
   deriving (Eq, Read, Show)
showCv :: ClassicalData → String
showCv (Left i) = show i
showCv (Right t) = show t
```

D.3.5 Class for complex numbers

A simple class that extends *Data.Complex* with a predefined value for `i` and a conjugation operation.

```
Num   class (Floating a) ⇒ Comp a where
   conjgt :: a → a
   sqrtMinusOne :: a
instance (RealFloat a) ⇒ Comp (Complex a) where
```

```
conjgt = conjugate
sqrtMinusOne = 0 :+ 1
```

D.3.6 Definition of matrices and matrix functions

A Haskell class for the handling of matrices. A simple representation of lists of lists are used.

```
type Matrix a = [[a]]

mmap :: (a → b) → Matrix a → Matrix b

mmap f = map (map f)
```

Accessor functions *dimx*, *dimy* are defined to provide the matrix dimensions.

```
dimx :: Matrix a → Int
dimx = length
qorder :: Matrix a → Int
qorder = (qorder′ 0) ∘ dimx
qorder′ :: Int → Int → Int
qorder′ acc val =
  if val < 2 then acc
    else qorder′ (acc + 1) (shiftR val 1)
dimy :: Matrix a → Int
dimy [] = 0
dimy (r : _) = length r
```

The idea for the function *gendotprod* was borrowed from APL. This allows for a generalization of the general notion of vector product and therefore matrix multiplication.

```
gendotprod :: (c → c → c) → (a → b → c) → [a] → [b] → c
gendotprod f g xs = (foldl1 f) ∘ zipWith g xs
genmatmul  :: (c → c → c) → (a → b → c) →
                  Matrix a → Matrix b → Matrix c
genmatmul f g m1 m2
    = transpose [[gendotprod f g a b | a ← m1] |
      b ← transpose m2]
```

indexM is a function to retrieve the i, j^{th} element of the matrix.

```
indexM :: Int → Int → Matrix a → a
indexM i j m = (m !! i) !! j
```

In a quantum space, the application of a unitary transform requires taking the conjugate transpose of a matrix.

```
conjtrans :: (Comp a) ⇒ [[a]] → [[a]]
conjtrans = (mmap conjgt) ∘ transpose
```

The module also provides a variation of the standard Haskell *zipWith* that signals an error whenever the lists are of different length. This is used in the definition of matrix addition below.

```
xzipWith :: (a → b → c) → [a] → [b] → [c]
xzipWith _ [] [] = []
xzipWith _ [] ys
   = error "xzipWith: 2nd list longer"
xzipWith _ xs []
   = error "xzipWith: 1st list longer"
xzipWith f (x : xs) (y : ys)
   = (f x y) : xzipWith f xs ys
```

This is a standard vector *dotprod* using addition and multiplication.

```
dotprod :: (Num a) ⇒ [a] → [a] → a
dotprod = gendotprod (+) (∗)
```

Creation of a zero matrix, used in the definition of the numeric instance.

```
zeromat :: (Num a) ⇒ Int → Matrix a
zeromat 0 = error "Invalid dimension"
zeromat n = [[fromInteger 0 |
                 inner ← [1 . . n]] |
                 outer ← [1 . . n]]
idMat :: (Num a) ⇒ Int → Matrix a
idMat n = [[if inner ≡ outer then fromInteger 1 else fromInteger 0 |
                 inner ← [1 . . n]] |
                 outer ← [1 . . n]]
```

When the base elements of a matrix are of the Haskell *Num* class the module defines an instance of the matrix as part of the *Num* class.

```
instance (Num a) ⇒ Num (Matrix a) where
  (+) = xzipWith (xzipWith (+))
  (−) = xzipWith (xzipWith (−))
  (∗) qm1 qm2
     = [[dotprod a b | b ← transpose qm2] |
          a ← qm1]
  negate   = mmap negate
  abs = mmap abs
  signum = mmap signum
  fromInteger n = zeromat $ fromInteger n
mat2ToTheT :: (Num a) ⇒ Matrix a → Int → Matrix a
mat2ToTheT m 0 = m
mat2ToTheT m t
  | t > 0 = m′ ∗ m′
  | otherwise = error "Negative value for t in m^(2^t)"
  where m′ = mat2ToTheT m (t − 1)
```

In certain cases, functions may create matrices whose elements are matrices.

Using our "list of lists" representation, a function to parametrically combine the rows of the *Matrix* creating a column vector is defined. The *paste* function which "pastes" matrices in a side-by-side fashion is defined as

$$\text{paste} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a & b & e & f \\ c & d & g & h \end{bmatrix}.$$

Those functions are then used to take a matrix of matrices of type α to a matrix of type α

```
tocolVector :: (a → a → a) → Matrix a → [a]
tocolVector f m = [foldl1 f a | a ← m]
paste :: Matrix a → Matrix a → Matrix a
paste = zipWith (++)
reduceM :: Matrix (Matrix a) → Matrix a
reduceM = concat ∘ (tocolVector paste)
```

Similarly, the reverse operation is required. That is, to take a matrix to a 2×2 matrix of matrices.

```
tenderize :: Matrix a → Matrix (Matrix a)
tenderize matr =
   [[grab 0 0 matr, grab 0 1 matr],
     [grab 1 0 matr, grab 1 1 matr]]
grab :: Int → Int → Matrix a → Matrix a
grab 0 0 matr =
   (take d2 $ map (take d2) matr)
       where d2 = (dimy matr) `div` 2
grab 0 1 matr
    = (take d2 $ map (take d2 ∘ (drop d2)) matr)
      where d2 = (dimy matr) `div` 2
```

```
grab 1 0 matr
    = (take d2 $ drop d2 $ map (take d2) matr)
      where d2 = (dimy matr) `div` 2
grab 1 1 matr
    = ((take d2 ∘ (drop d2)) $ map (take d2 ∘ (drop d2)) matr)
      where d2 = (dimy matr) `div` 2
```

D.3.7 Definition of the LazyNum type and operations

The semantics of the language QPL specify creation of superoperators over a vector space of Complex numbers. Since computer representation of real numbers is fraught with rounding, this module provides a symbolic type that will allow certain values to be represented exactly, together with a function that will compute the value at the end.

QOperation

```
data QOperation = Plus | Minus | Times | Div
   deriving (Enum, Eq)
```

QFunction

Functions used in Quantum values.

```
data QFunction = SquareRoot | Exp | AbsoluteValue
    | Signum | Negate | Conjugate
    | Log | ExpToThe LazyNum | LogBase LazyNum
    | Sin | Cos | Tan | Asin | Acos | Atan
    | Sinh | Cosh | Tanh | Asinh | Acosh | Atanh
   deriving Eq
```

`LazyNum`

All possible symbolic values for a computation. A recursive type allowing for complex numbers, real numbers, operations and function application and unknowns.

```
data LazyNum = Si
    | Snum Double
    | Sbop QOperation LazyNum LazyNum
    | Sfun QFunction LazyNum
    | Svar String
    | SZero
```

The *approximate* function

When completed processing, the results may be approximated. Note that more processing on the symbolic values could first be done to come to a closer result. For example, squaring a square root value or take the log of an exponential could be cancelled out.

This allows us to compute a numeric value for any *LazyNum* that does not contain an unknown.

```
approximate :: LazyNum → Complex Double
approximate (SZero) = 0 : + 0
approximate (Snum x) = x : + 0
approximate (Si) = 0 : + 1
approximate (Sbop Plus p1 p2) = (approximate p1) + (approximate p2)
approximate (Sfun (ExpToThe b) p) = (approximate p) ** (approximate b)
approximate (Sfun (LogBase b) p) = logBase (approximate p) (approximate b)
approximate (Sfun Log a) = log (approximate a)
approximate (Svar _) = 1.0
approximate (Sfun SquareRoot p) = sqrt (approximate p)
```

qprob

Convert a number into a *LazyNum*.

```
qprob :: Double → LazyNum
qprob x = Snum x
display :: Int → LazyNum → String
display digits ln =
  let c = approximate ln
    re = realPart c
    im = imagPart c
    prec = '.' : (show digits)
  in if (abs im < 0.0001)
    then printf ('%' : prec ++ "g") re
    else printf ('%' : prec ++ "g+" ++ "%" ++ prec ++ "g%s") re im "i"
```

D.3.8 Basis definitions for quantum computing

Quantum values are typically written as:

$$\sum_{i} \alpha_i b_i$$

where the α_i are complex values and the b_i are orthogonal basis vectors in the space of our interest. A typical representation is to consider the complex plane and use two unit vectors $|0\rangle$ and $|1\rangle$. In [29], a method of representing a basis and quantum values over the basis is discussed. This module implements the essence of the idea and the code for the *Basis* class from that paper. There are differences in the actual representation, due to the use of a density matrix representation for **qubit**s and the quantum stack being defined for multiple node types.

The *Basis* class contracts that the type will have a defined list of basis elements, purported to be orthogonal. Due to the representations used for actual elements of the stack, the implementation requires that any type a member of the *Basis* class is also a member of *Eq* and *Ord*.

```
class (Eq a, Ord a) ⇒ Basis a where
  basis :: [a]
  ei :: a → Int
  ei a = case (elemIndex a basis) of
    Nothing → error "Just shouldn't happen"
    Just i → i
  ind :: a → a → Int
  ind a c = (length (basis :: [a])) ∗ (ei a) + (ei c)
```

Figure D.7: Haskell definition of a Basis

Once a type is defined as being an instance of *Basis*, the instance of *Basis* for the pair type is straightforward.

```
instance (Basis a) ⇒ Basis (a, a) where
  basis = [(a, b) | a ← basis, b ← basis]
```

The implementation detailed in this thesis and in L-QPL uses one basis, $|0\rangle$ and $|1\rangle$.

```
data OurBasis = Zero | One
  deriving (Eq, Show, Read, Ord)
instance Basis OurBasis where
  basis = [Zero, One]
offdiag :: (Basis a, Show a, Num b) ⇒ [a] → [((a, a), b)]
offdiag basis = [((a, b), fromInteger 0) |
  a ← basis, b ← basis, a ≢ b]
showQv :: (OurBasis, OurBasis) → String
showQv (Zero, Zero) = "00"
showQv (Zero, One) = "01"
```

$showQv\ (One, Zero) = $ `"10"`
$showQv\ (One, One) = $ `"11"`

D.3.9 A class definition for infinite lists

Infinite lists are given by the co-inductive type defined with destructors hd that return the base type and tl that returns a new infinite list.

$$\mathbb{IL}(A) = \nu x.\{\mathrm{hd} : A, \mathrm{tl} : x\}$$

In this module, a **class** to encapsulate infinite list functionality is defined. Typically, this is all that should need to be defined or used in other modules.

IL

This class handles the interface for types of infinite lists. A class was chosen rather than directly defining on our choice of type, to insulate the rest of the code from the actual type definition.

class *IL t* **where**

This is the standard first destructor of an infinite list, giving the first element of the list.

$hd :: t\ a \rightarrow a$

This is the standard second destructor of an infinite list, giving the remaining elements of the list.

$tl :: t\ a \rightarrow t\ a$

Other "list-like" functions

Many of the functions normally applicable to lists are redefined for infinite lists.

```
takeI :: Int → t a → [a]
takeI  0 il = [ ]
takeI  n il
   | n < 0
      = error "Can not take negative number of list"
   | n > 0
      = hd il : (takeI (n − 1) (tl il))
takeI _ _ = error "Error in takeI patterning"
dropI :: Int → t a → t a
dropI 0 il = il
dropI n il
   | n < 0
      = error "Can not drop negative number of list"
   | n > 0
      = dropI (n − 1) (tl il)
dropI _ _
   = error "Error in dropI patterning"
makeInfinite :: [a] → t a
cycleI :: [a] → t a
cycleI = makeInfinite ∘ cycle
iterateI :: (a → a) → a → t a
iterateI f = makeInfinite ∘ (iterate f)
iterI :: (a → t a) → a → t a
pushI :: a → t a → t a
sumList :: (Num a) ⇒ t a → t a
unzipI :: t (a, b) → (t a, t b)
zipI :: t a → t b → t (a, b)
zipWithI :: (a → b → c) → t a → t b → t c
(!!!) :: t a → Int → a
(!!!) il i = hd $ dropI i il
```

sumL2

Takes an infinite list of pairs, where the second element of the pair is numeric to another list of pairs, but where the second element of the pair is the sum of the elements of index less than it in the original list.

```
sumL2 :: (Num b) ⇒ t (a, b) → t (a, b)
sumL2 il = zipI il1 sumil2
  where (il1, il2) = unzipI il
    sumil2 = sumList il2
```

D.3.10 Definition of the Stream data type.

Although Haskell does not directly implement co-inductive types (Types defined by destructors rather than constructors), an equivalent structure is definable due to Haskell's laziness. As discussed in appendix D.3.9 on page 219, two destructors are required, *hd* and *tl*. To get this effect, the type defines two constructors *strhd* and *strtl*. An instance of a *Stream* can then be defined as the tail is lazily defined and accessed.

```
data Stream a = Stream{strhd :: a,
  strtl :: Stream a}
```

Figure D.8: Haskell definitions of a Stream

```
instance Functor Stream where
  fmap f (Stream a t) = Stream (f a) (fmap f t)
instance Monad Stream where
  return a = Stream a (return a)
```

```
    m >>= k = Stream (strhd ∘ k $ strhd m)
      ((strtl m) >>= (strtl ∘ k))
```

The instance of *IL* for *Stream* is defined.

```
  instance IL Stream where
    hd = strhd
    tl = strtl
    takeI 0 str = []
    takeI n str
       = strhd str : takeI (n − 1) (strtl str)
    makeInfinite []
       = error "May only convert unending lists"
    makeInfinite (a : as)
       = Stream a (makeInfinite as)
    iterI f a
       = Stream a (iterI f ((strhd ∘ f) a))
    pushI = Stream
    sumList il
       = Stream (strhd il) (fmap (+(strhd il))
         (sumList (strtl il)))
    unzipI (Stream (a, b) tail)
       = (Stream a (fstI tail), Stream b (sndI tail))
         where fstI = fst ∘ unzipI
               sndI = snd ∘ unzipI
    zipI (Stream a taila) (Stream b tailb)
       = Stream (a, b) (zipI taila tailb)
    zipWithI f (Stream a taila) (Stream b tailb)
       = Stream (f a b) (zipWithI f taila tailb)
```

Appendix E

Example L-QPL programs

E.1 Miscellaneous examples

E.1.1 Quantum teleportation function

The L-QPL program shown in figure E.1 is an implementation of a function that will accomplish quantum teleportation as per the circuit shown previously in figure 2.9 on page 27. It also provides a separate function to place two **qubit**s into the EPR state.

Note that the teleport function, similarly to the circuit, does not check the precondition that **qubit**s a and b are in the EPR state, which is required to actually have teleportation work.

```
prepare :: (;a:Qbit, b:Qbit)=
{ a = |0>; b=|0>;
  Had a;
  Not b  <=  a;
}
teleport :: (n:Qbit, a:Qbit, b:Qbit ; b:Qbit) =
{ Not a  <=  n ;
  Had n;
  measure a of
       |0>  => {} |1>  => {Not b};
  measure n of
       |0> => {}  |1> => {RhoZ b}
}
```

Figure E.1: L-QPL code for a teleport routine

E.1.2 Quantum Fourier transform

The L-QPL program to implement the quantum Fourier transform in figure E.2 uses two recursive routines, `qft` and `rotate`. These functions assume the **qubit**s to transform are in a `List`.

The routine `qft` first applies the Hadamard transform to the **qubit** at the head of the list, then uses the `rotate` routine to recursively apply the correct Rot transforms controlled by the other **qubit**s in the list. `qft` then recursively calls itself on the remaining **qubit**s in the list.

```
#Import Prelude.qpl

rotate ::(n:Int | h:Qbit, qbsIn :List (Qbit);
                  h:Qbit, qbsOut:List (Qbit))=
{ case qbsIn of
     Nil => {qbsOut = Nil }
     Cons (hd, tl) =>
       { Rot(n) h <= hd;
         m := n+1 ;
         rotate(m) h tl;
         qbsOut = Cons(hd,tl) }
}

qft :: (qsIn:List (Qbit); qsOut:List (Qbit)) =
{ case qsIn of
    Nil => {l = Nil}
    Cons(hd,tl) =>
      {  Had hd;
         rotate (2) hd tl;
         qft tl;
         qsOut = Cons(hd,tl) }
}
```

Figure E.2: L-QPL code for a quantum Fourier transform

E.1.3 Deutsch-Jozsa algorithm

The L-QPL program to implement the Deutsch-Jozsa algorithm is in figure E.3 with supporting routines in figure E.5, figure E.4, figure E.6. The `hadList` function is defined in figure E.12 on page 231.

The algorithm decides if a function is balanced or constant on n $\mathbf{bit}$s. This implementation requires supplying the number of $\mathbf{bit}$s / $\mathbf{qubit}$s used by the function, so that the input can be prepared. Additionally, it currently requires an extension to the language where a $n+1-\mathbf{qubit}$ function can be defined from a $n-\mathbf{bit}$ function. Assuming the existance of this planned extension, the rest of the algorithm is straight-forward.

The function `dj` creates an input list for the function, applies the Hadamard transform to all the elements of that list and applies the candidate function. When that is completed, the initial segment of the list is transformed again by Hadamard and then measured.

```
#Import initList.qpl
#Import addnzeros.qpl
#Import hadList.qpl
#Import measureInps.qpl

dj ::(size:Int | uFunc: (List(Qbit) -> List(Qbit)); resultType :Ftype)=
{  inlist = addNZeroqbs(size | |1>);
   hadList inlist;
   uFunc inlist;
   inputs = initList(inlist);
   hadList inputs;
   resultType = measureInputs(inputs);
}
```

Figure E.3: L-QPL code for the Deutsch-Jozsa algorithm

The function `addNZeroqbs` creates a list of $\mathbf{qubit}$s when given a length and the last value. Assuming the parameters passed to the function were 3 and $|1\rangle$, this would return the list:

$$[|0\rangle, |0\rangle, |0\rangle, |1\rangle]$$

```
#Import Prelude.qpl

addNZeroqbs ::(size:Int | last :Qbit; resultList :List(Qbit))=
{     if (size == 0) => { resultList = Cons(last,Nil)}
      else { resultList = Cons(|0>,addNZeroqbs(size - 1 | last))}
}
```

Figure E.4: L-QPL code to prepend n $|0\rangle$'s to a **qubit**

The `initList` function removes the last element of a list.

```
#Import Prelude.qpl
initList(inlist: List(a) ; outlist: List(a)) =
{   case inlist of
       Nil => {outlist = Nil}
       Cons(hd, tail) =>
         { case tail of
              Nil => {outlist = Nil}
              Cons(hd', tl') =>
                { tlist = Cons(hd', tl');
                   outlist = Cons(hd, initList(tlist)) }
         }
}
```

Figure E.5: L-QPL code accessing initial part of list

The `measureInputs` function recursively measures the **qubit**s in a list. If any of them measure to 1, it returns the value `Balanced`. If all of them measure to 0, it returns the value `Constant`.

```
#Import Prelude.qpl
qdata Ftype = Balanced | Constant

measureInputs::(inputs:List(Qbit); result :Ftype) =
{   case inputs of
       Nil => {result = Constant} //All were zero
       Cons(hd, tail) =>
         {   measure hd of
               |0> => { result = measureInputs(tail)}
               |1> => { result = Balanced} }
}
```

Figure E.6: L-QPL code to measure a list of **qubit**s

E.1.4 Quantum adder

This section provides subroutines that perform *carry-save* arithmetic on **qubit**s. The algorithm is from [32]. The `carry` and `sum` routines in figure E.7 function as gates on four **qubit**s and three **qubit**s respectively.

```

carry::(c0:Qbit, a:Qbit, b:Qbit,c1:Qbit;
        c0:Qbit, a:Qbit, b:Qbit,c1:Qbit) =
{
   Not c1  <=  b  <=  a;
   Not b  <=  a;
   Not c1  <=  b  <=  c0
}

sum::(c:Qbit, a:Qbit, b:Qbit;
      c:Qbit, a:Qbit, b:Qbit) =
{
  Not b  <=  a;
  Not b  <=  c;
}
```

Figure E.7: L-QPL code to implement carry and sum gates

The addition algorithm adds two lists of **qubit**s and an input carried **qubit**. The first list is unchanged by the algorithm and the second list is changed to hold the sum of the lists, as shown in figure E.8.

```
qdata List a = {Nil | Cons (a, List(a))}
#Import carrysumgates.qpl

adder::(c0:Qbit, asin:List(Qbit), bsin:List(Qbit);
        c0:Qbit, asout:List(Qbit), aplusbout:List(Qbit)) =
{
   case asin of
      Nil => { asout = Nil; aplusbout = Nil}
      Cons(a, taila) =>
        { case bsin of
             Nil => { //Error
                      aout = Nil; aplusbout = Nil;}
             Cons(b, tailb) =>
              { c1 = |0>;
                carry c0 a b c1;
                case tailb of
                  Nil =>
                   {Not b  <=  a;
                    sum c0 a b;
                    tailb = Cons(c1,Nil)}
                  Cons( t, tlb') =>
                   { tailb = Cons(t,tlb');
                     adder c1 taila tailb;
                     carry c0 a b c1;
                     sum c0 a b;
                     discard c1};
                asout = Cons(a,taila);
                aplusbout = Cons(b,tailb);
              }
        }
}
```

Figure E.8: L-QPL code to add two lists of **qubit**s

The program proceeds down the lists A and B of input **qubit**s, first applying the `carry` to the input carried **qubit**, the heads of A and B and a new zeroed **qubit**, c_1. When the ends of the lists are reached, a controlled not and the `sum` are applied. The output $A + B$ list is then started with c_1. Otherwise, the program recurses, calling itself with c_1 and the tails of the input lists. When that returns, `carry` and `sum` are applied, the results are "Consed" to the existing tails of the lists, c_1 is discarded and the program returns.

E.2 Shor's factoring algorithm

Shor's algorithm is split into a classical computational part and a quantum computational. The classical part can be considered as three steps:

- Setup;
- Call order finding (a quantum algorithm);
- Check to see if an answer has been found, repeat if not.

The code provided in this section is currently work in progress. It represents my current understanding of the algorithm and how that fits with the capabilities of L-QPL. Future L-QPL enhancements and study of the algorithm will undoubtedly lead to changes and improvements.

E.2.1 Setup for factoring

The algorithm begins by assuming input of a number N for factoring. It then chooses a random number a less than N (see figure E.9 on the next page) and compute the $\gcd(a, N)$ (see figure E.10 on the following page). If this is one, a is a factor of N and the algorithm is complete.

```
random :: (maxval :Int, seed:Int; seed:Int, rand :Int)=
{ use seed in {
      seed = seed ×238457 ≪ 3
   };
  use maxval,seed;
  rand = ((((maxval ≪ 3) − 13987) × (maxval −1) × seed) ×
           ((maxval ≪2) −1)) mod maxval;
  seed = seed;
}
```

Figure E.9: L-QPL function to get a random number

```
gcd :: (a : Int, b: Int; theGcd : Int) =
{ use a,b in
  { if b == 0 => { theGcd = a }
    else => {theGcd = gcd(b, a mod b)} }
}
```

Figure E.10: L-QPL function to compute the GCD

E.2.2 Order finding

The first part involve creating two **qubit** "registers", which in L-QPL are items of type `List(Qbit)`. The lists need to be $\log_2 N$ **qubit**s each,the first initialized to $|0\rangle$, the second to $|1\rangle$, and individual Hadamard transforms must be applied to the first register. This is accomplished by calling the functions `intToZeroQbitList` and `hadList` as shown in figure E.11 and figure E.12.

The next step is to apply the unitary transform corresponding to the function f, where $f(x) = a^x \mod N$. This is done via the specialized transform UM, which takes x, N and t as parametrization inputs and then applies the transform to *a list of* **qubit***s*, controlled by the **qubit**s in the first list. This is shown in figure E.13.

In the example program, the matrix of the transformation UM will be a 16×16 matrix as it operates on 4 **qubit**s. Let us determine what the matrix would be in

```
#Import ../qplprogs/Prelude.qpl

intToZeroQbitList :: (n :Int ; nq : List (Qbit))=
{ use n;
  if  n == 0 => { nq = Nil }
  else       =>
     {  n' = n ≫ 1;
        nq' = intToZeroQbitList(n');
        nq = Cons(|0>, nq') }
}
```

Figure E.11: Function to take an int and return a list of **qubit**s

```
#Import Prelude.qpl

hadList :: (inqs :List(Qbit) ; outqs : List (Qbit))=
{ case inqs of
    Nil => { outqs = Nil }
    Cons(q, inqs') =>
       {  Had q;
          hadList inqs';
          outqs = Cons(q, inqs')}
}
```

Figure E.12: Function to apply Hadamard to all of a list of **qubit**s

```
#Import Prelude.qpl

orderFind :: (x:Int , n:Int, t:Int | l1:List (Qbit), l2:List(Qbit);
                   l1:List(Qbit), l2:List(Qbit))=
{ case l1 of
    Nil =>
      { l2 = l2;
        l1 = Nil }
    Cons(q,l1') =>
      { UM(x,n,t) l2  ⇐  q;
        (l1,l2)=orderFind(x ,n,t+1 | l1',l2)   }
}
```

Figure E.13: Order finding algorithm

the case $N = 15, x = 7$.

First, compute $x^7 \mod 15$ for $1, \ldots 14$.

$$\begin{array}{ccc}
1^7 \equiv 1(=' 0001') & 6^7 \equiv 6(=' 0110') & 11^7 \equiv 11(=' 1011') \\
2^7 \equiv 8(=' 1000') & 7^7 \equiv 13(=' 1110') & 12^7 \equiv 3(=' 0011') \\
3^7 \equiv 12(=' 1100') & 8^7 \equiv 2(=' 0010') & 13^7 \equiv 7(=' 0111') \\
4^7 \equiv 4(=' 0100') & 9^7 \equiv 9(=' 1001') & 14^7 \equiv 14(=' 1110') \\
5^7 \equiv 5(=' 0101') & 10^7 \equiv 10(=' 1010') &
\end{array}$$

Note also that the bit strings $'0000', '1111'$ are mapped to themselves.

Identifying these numbers (i.e., using their corresponding bit strings) with the basis vectors of $\mathbb{C}^4$, the function determines a permutation of the basis vectors:

$$\begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
0 & 1 & 8 & 12 & 4 & 5 & 6 & 13 & 2 & 9 & 10 & 11 & 3 & 7 & 14 & 15
\end{pmatrix}$$

This gives us the transformation matrix:

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

The algorithm then applies the inverse quantum Fourier transform to the first **qubit** list. This is given by the functions as shown in figure E.14 and figure E.15.

```
#Import inverseRotate.qpl
#Import reverse.qpl

inverseQft :: (inqs:List (Qbit); outqs:List (Qbit)) =
{   reverse inqs;
    inverseQft' inqs;
    outqs = reverse(inqs);
}

inverseQft' :: (inqs:List (Qbit); outqs:List (Qbit)) =
{ case inqs of
     Nil => {outqs = Nil}
     Cons(h,inqs') =>
        {   inverseQft' inqs';
            inverseRotate (2) h inqs';
            Had h;
            outqs = Cons(h,inqs') }
}
```

Figure E.14: Function to apply the inverse quantum Fourier transform

The algorithm then measures the first list, creating an integer from its value, as in figure E.16 on the next page. Call the result of this y.

The algorithm now takes y and computes the denominator r' of y/N when reduced via a continued fraction algorithm. The code for the continued fraction algorithm is still under active development and not included at this time.

The final program is in figure E.17 on page 235.

```
#Import Prelude.qpl

inverseRotate ::(n:Int| h:Qbit, inqs:List (Qbit);
                    h:Qbit, outqs:List (Qbit))=
{ case inqs of
     Nil => {outqs = Nil }
     Cons (q, inqs') =>
       { use n;
         m := n+1 ;
          inverseRotate(m) q inqs';
          Inv-Rot(n) h <= q;
          outqs = Cons(q,inqs') }
}
```

Figure E.15: Function to apply inverse rotations as part of the inverse QFT

```
#Import Prelude.qpl

qbitListToInt :: (nq :List(Qbit) ; n :Int)=
{ case nq of
     Nil => { n = 0}
     Cons(q, nq') =>
         { n' = qbitListToInt(nq');
            measure q of
                |0> => {n1 = 0}
                |1> => {n1 = 1};
            use n1, n';
            n = n1 + (n' >> 1)
         }
}
```

Figure E.16: Function to measure a list and create a probabilistic integer

```
#Import random.qpl
#Import gcd.qpl
#Import intToOneQbitList.qpl
#Import intToZeroQbitList.qpl
#Import hadList.qpl
#Import inverseQft.qpl
#Import qbitListToInt.qpl
#Import orderFind.qpl
#Import contFrac.qpl

factor ::(seed:Int, n:Int ; fact:Int) =
{   use n;
    (seed,x) = random(n,seed); //Trial number
    use x;
    g = gcd(n,x);
    use g;
    if g == 1 =>  //Need to apply algorithm
    {  ln = intToOneQbitList(n); //ln is a qbit list of length 4
       lt = Cons(|0>, Cons(|0>,
               intToZeroQbitList(2×n))); //lt length 2×4 + 1 + 1;
       hadList lt; //Create superpositions
      (lt,ln) = orderFind(x,n,0 | lt, ln) ; // Order finding execution
       inverseQft lt;
       sr = qbitListToInt(lt);
            // sr is phase estimate s/r, regard as
            // float with the decimal point at the 10 binary
            // position.
      cfrac(sr,10; ipart,npart, r);
      grn = gcd(r,n);
      use grn in
      {   if grn == 1 => {fact = factor(seed,n)} // Try again
          else => { fact = grn;} }
    }
    else => {fact = g }
}

main :: () =
{
    g = factor(38272,15);
}
```

Figure E.17: Program to factor 15 via Shor's algorithm

Appendix F

Using the system

F.1 Running the L-QPL compiler

The compiler is run from the command line as:

```
  lqplc <options> <infiles>
```

This will run the compiler on each of the input files *infiles* which are expected to have a suffix '`.qpl`'. The compiled files will be written with the suffix '`.qpo`'.

The options allowed for the compiler are:

`-e, --echo_code` Echo the input files to `stderr`.

`-s, --syntactic` This option will cause the compiler to print a syntax parse tree on `stderr`.

`-r, --ir_print` This compiler option will force the printing of the intermediate representation generated during the semantic analysis phase.

`-h, --help` This prints a help message describing these options on `stderr`.

`-V, -?, --version` This option prints the version information of the compiler on `stderr`.

`-o[FILE], --output[=FILE]` This will cause the compiler to write the compiled QSM code to `FILE`.

`-i[DIRLIST], --includes=DIRLIST` For this option, `DIRLIST` is expected to be a list of semi-colon separated directories. The compiler will use the directory list when searching for any import files.