Contents

Chapter 1

An Introduction to Linear QPL

1.1 Introduction to L-QPL

This chapter presents an overview of the linear quantum programming language. Explanations of L-QPL programs, statements, and expressions are given.

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

The language design started from QPL in [Sel04] and rapidly evolved to a point where a direct comparison is somewhat difficult. Major differences are the type system, the syntax and structure of functions and the choices of individual statements.

1.1.1 Functional versus imperative

L-QPL is a functional language that uses single assignment. It resembles QPL in this aspect rather than QML of [AG05]. Single assignment means that a variable always has a unique value until its use. (See sub-section 1.1.2 below.)

Like most functional languages, side effects are not allowed in functions, *other than those that occur due to quantum entanglement*. Side effects in imperative languages are those where a global variable is updated or values are read or written. An example of this in an imperative quantum language is a procedure in QCL from [Öme00]. L-QPL does not have the concept of a global variable. Currently, I/O is undefined. Side effects from quantum entanglement can occur when a qubit is passed as a parameter to a function and it is operated on in the function.

1.1.2 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 (see sub-section 1.3.6 on page 15) provide a specialized syntax for variables which are both input and output to a function. The ability to use a classical value (integer or boolean) multiple times is handled by **use** statements (see sub-section 1.3.5 on page 14), which 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 1.1: L-QPL code to return the length of the list

An example illustrating linearity is given in figure 1.1. 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.

1.2 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.

The above structure was chosen thinking of Haskell [Pey03] and C [KR88]. Global definitions of functions and types is borrowed from Haskell, while the **main** start point and import feature is a combination of Haskell and C.

1.2.1 Data definitions

L-QPL provides the facilities to define datatypes with a syntax reminiscent of Haskell [Pey03].

Natively, the language provides **Int**, **Qubit** and **Bool** types. **Bool** is the standard Boolean type with values true and false. **Int** is a standard 32-bit integer. **Qubit** 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 illustrates 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*. Type variables must be and identifier starting with a lower case letter. Typically, a single letter is used. 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**, **Qubit**), 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 Colour   = {Red      | Black}
qdata RBSet a  = {Empty    |
                  RBTip(Colour, RBSet(a), a, RBSet(a))}
qdata RTree a  = {Rnode(a, List(a))}
qdata Rose a   = {Rose(List(a, Rose(a)))}
```

1.2.2 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 1.3 on page 9.

The structure of function definitions is unique to L-QPL, although broadly based on one of the acceptable syntaxes for C function definitions as in [KR88]. The major difference occurs in the signature which, as will be seen below, allows for both classical and quantum input

arguments and specifies the quantum outputs of a function. Another difference is that L-QPL defines all functions globally, hence there is no requirement for declarations separate from the definitions.

Let us examine two examples of functions. The first, in figure 1.2 is a fairly standard function to determine the greatest common divisor of two integers.

```
1  gcd::(a:Int, b:Int | ; ans: Int) =
2  { if b == 0 => {ans = a}
3       a == 0 => {ans = b}
4       a >= b => {ans = gcd(b, a mod b)}
5    else      => {ans = gcd(a, b mod a)}
6  }
```

Figure 1.2: L-QPL function to compute the GCD

The first line has the name of the function, gcd, a separating double colon, and the signature. The signature of the function 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 1.3 on the next page highlights the linearity of variables in L-QPL. This function is used to create a list of qubits 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.

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 **Qubit** 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.

```
1  #Import Prelude.qpl
2
3  toQubitList::(n:Int ;                    //Input: a probabilistic int
4                nq:List(Qubit), n:Int)= //Output: qubit list, original int
5  { use n in
6    { if n == 0       =>
7                { nq         = Nil}
8      (n mod 2) == 0 =>
9                { n'         = n >> 1;
10               (nq', n') = toQubitList(n');
11               nq         = Cons(|0>, nq') }
12     else => { n'         = n >> 1;
13               toQubitList(n'; nq',n');
14               nq         = Cons(|1>, nq') };
15     n = n //Recreate as probabilistic int
16   }
17 }
```

Figure 1.3: L-QPL function to create a qubit register

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, 8, 9, 12 and 15. In these uses, n is a classical variable, no longer on the quantum stack. The last usage on line 15 where n is assigned to itself, returns n to the quantum world.

1.3 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 $\Leftarrow$ qualifier.

Divergence: The **zero** statement.

Other: the **discard** statement.

Most of these correspond to the conceptual statements for quantum pseudo-code as given in [Kni96].

1.3.1 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*. Expressions are introduced and defined in section 1.4 on page 19

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

1.3.2 Classical control

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

```
1 smallPrime::(a:Int | ;
2               isSmallPrime:Bool) =
3 { if a =< 1 => {isSmallPrime = false}
4     a == 2 || a == 3 || a == 5  =>
5                 {isSmallPrime = true}
6     a == 4 => {isSmallPrime = false}
7   else     => {isSmallPrime = false}
8 }
```

Figure 1.4: 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 1.4.

Some form of classical control is encountered in all current quantum programming languages. It is central to the semantics of [Sel04]. Note that many languages also include a classically controlled looping statement (such as while or do). L-QPL does not, relying on recursive functions to achieve the same end.

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.

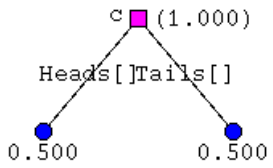
When writing the dependent blocks of the selectors, quantum variable creation must be the same in each block. The compiler will give you a semantic warning if a quantum variable is created in one branch and not another.

1.3.3 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 1.5, there is a **measure** on line 5.

```
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 1.5: L-QPL program to do a coin flip

Consider the program in figure 1.5 which emulates 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 1.5.

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.

When writing the dependent blocks of **measure** (and **case** in sub-section 1.3.4 on the following page) variable creation must be the same in each dependent list of statements. The

```
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 1.6: L-QPL programs contrasting creation

compiler will give you a semantic warning if a variable is created in one branch and not another.

For example, consider figure 1.6. 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 **Qubit**. 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.

1.3.4 Case statement

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

In figure 1.7 on the facing page, 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.

Consider the example in figure 1.8 on the next page. TTree is a parametrized data type which depends on the type variable a. It has three constructors: Tip which takes no argu-

```
1  qdata List a = {Nil | Cons(a, List(a))}
2
3  reverse::(lis:List(a) ; revlis:List(a))=
4  {  rev' (lis, Nil ; revlis) }
5
6  rev'::(lis:List(a), accumIn:List(a) ; returnList:List(a))=
7  {  case lis of
8       Nil            => { returnList = accumIn}
9       Cons(hd, tail) => { acc        = Cons(hd, accumIn);
10                          returnList = rev'(tail, acc)}
11 }
```

Figure 1.7: L-QPL program demonstrating **case**, a function to reverse a list.

```
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                   if j > k => { depth = 1 + j}
11                   else     => { depth = 1 + k}}
12 }
```

Figure 1.8: L-QPL program demonstrating **case**, a function to compute the max tree depth.

ments; 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.

1.3.5 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 1.9.

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

Figure 1.9: Syntactic sugar for **use** / classical assignment

The three types of classical use are semantically equivalent, but do have different syntaxes as illustrated in figure 1.10 on the facing page.

In sub-figure (a) of figure 1.10, 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 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 **Qubits**, 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 statements depending on the value of the **Int**, L-QPL provides the **if - else** statement as discussed in sub-section 1.3.2 on page 10. Typical use would be immediately

```
1 Br(t1,_,t2) =>
2   { j = treeMaxDepth(t1);
3     k = treeMaxDepth(t2);
4     use j,k in
5       { if j > k => { depth = 1 + j}
6         else     => { depth = 1 + k} }
7   }
```

(a) Explicit dependence

```
1 Br(t1,_,t2) =>
2   {   j = treeMaxDepth(t1);
3       k = treeMaxDepth(t2);
4       use j,k;
5       if j > k => { depth = 1 + j}
6       else     => { depth = 1 + k}
7   }
```

(b) Implicit dependence

```
1 Br(t1,_,t2) =>
2   { j := treeMaxDepth(t1);
3     k := treeMaxDepth(t2);
4     if j > k => { depth = 1 + j}
5     else     => { depth = 1 + k}
6   }
```

(c) Classical assign

Figure 1.10: Fragments of L-QPL programs contrasting **use** syntax

after the **use** statement, or as the first statement of the dependent block of an explicit **use** statement.

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**.

1.3.6 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 1.1 on the following 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 transformation. Controlled versions of transforms are accomplished by the built-in control mechanism using <=.

The signatures of transforms are dependent upon the size of the associated matrix. A 2×2 matrix gives rise to the signature (q:**Qubit** ; q:**Qubit**). In general, a $2^n \times 2^n$ matrix will require n qubits 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:

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

Table 1.1: L-QPL transforms

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

When a function is called in the transformational syntax, as on the last line of the above code, the arguments separated by spaces (`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` 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;
```

[1] A function call where a single unparenthesized variable appears on the left hand side of the equals is actually an assignment statement. The right hand side of the assignment is a function expression. See sub-section 1.3.1 and sub-section 1.4.4 for further details.

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 1.4 on the next page. 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 1.4.4.

1.3.7 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 1.3.4.

1.3.8 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 1.11 on the facing page.

Syntax of quantum control In L-QPL any statement, including block statements and procedure calls may be quantum controlled. Semantically, this will affect all transforms that occur within the controlled statement, including any of the transforms in a called function. The

```
1  prepare::( ; a:Qubit, b:Qubit)=
2  { a = |0>;  b = |0>;
3    Had a;
4    Not b  <=  a;
5  }
6  teleport::(n:Qubit, a:Qubit, b:Qubit ; b:Qubit) =
7  { Not a  <=  n ;
8    Had n;
9    measure a of
10       |0> => {}   |1> => {Not b};
11   measure n of
12       |0> => {}   |1> => {RhoZ b}
13 }
```

Figure 1.11: L-QPL program demonstrating quantum control

syntax is *statement* followed by the symbols <=, followed by a list of identifiers. These identifiers may be of any type, but are typically either qubits or constructed data types with qubit elements, such as a List(**Qubit**). Additionally, any identifier may be preceeded by a tilde (~) to indicate 0−control. The default is 1−control.

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 statement is that all qubits in the control list, or contained in items in that list are used to control all unitary transformations done in the controlled statement.

1.3.9 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.

1.3.10 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 4.30 on page 66.

1.4 L-QPL expressions

Expressions in L-QPL are used in many of the statements discussed in section 1.3 on page 9. 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.

1.4.1 Constant expressions

The possible constant expressions in L-QPL are shown in table 1.2. 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
$\|0\rangle$	**Qubit**	Quantum
$\|1\rangle$	**Qubit**	Quantum

Table 1.2: Allowed constant expressions in L-QPL

1.4.2 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 1.3.1 on page 10, 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 1.3.5 on page 14 for further information and examples.

1.4.3 Constructor expressions

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

```
1        qdata TTree a = {Tip | Br(TTree(a), a, TTree(a)) | Node(a)}
2        qdata List a  = {Nil | Cons(a, List(a))}
3
4        qbtree  = Br(Tip, |1>, Br(Node(|0>), |1>, Node(|1>)));
5        intlist = reverse(Cons(5,Cons(4,Cons(3,Cons(2,Cons(1,Nil))))));
```

These statements create a tree as in figure 1.12 and the list $[1, 2, 3, 4, 5]$. Compare the logical representation of qbtree as in figure 1.12 versus how it is stored in the quantum stack machine, shown in figure 1.13. The assignment statement which creates qbtree uses five constructor

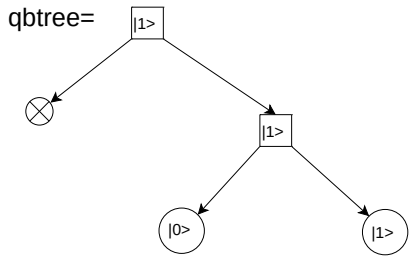


Figure 1.12: Pictorial representation of qbtree

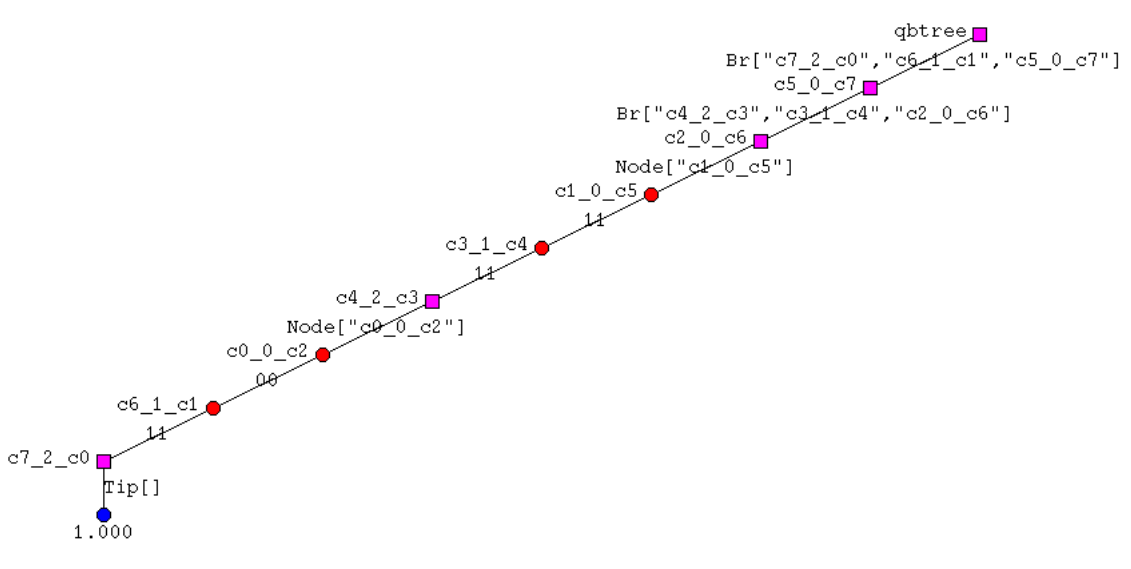


Figure 1.13: Quantum stack contents after creation of qbtree

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

1.4.4 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.

```
1       f::(c1:Int, c2:Int, c3:Int | q1:Qubit, i1:Int ; out:Qubit)
2       = { ... }
3       ...
4           qout  = f(c1,c2,c3 | q1,i2);
5           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(**Qubit**). 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 1.14: L-QPL code for appending two lists

In figure 1.14, line 7 shows the append being used as a function expression inside of a constructor expression.

Chapter 2

BNF description of the Linear Quantum Programming Language

2.1 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>
```

2.2 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>:: 'Qubit'  | 'Int' | 'Bool'
```

2.3 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>
```

2.4 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.

2.4.1 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>
```

2.4.2 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>
```

2.4.3 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> ')'
```

2.4.4 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 -}
```

2.4.5 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> '<=' <control_list>
```

2.4.6 Divergence

This signifies that this portion of the program does not terminate. Statements after this will have no effect.

```
(<stmt> continued)
     | 'zero'
```

2.5 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> | '_'
```

2.6 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 -}
```

2.7 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 -}

<control_list>:: <control> <more_control_list>

<more_control_list>:: ',' <control> <more_control_list>
     | {- empty -}

<control>:: <identifer>   | '~' <identifer>

<opt_identifier_list>:: <identifier_list>
     | {- empty -}

<identifier_list>:: <identifier> <more_idlist>

<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'
```

Chapter 3

Quantum stack machine additional details

3.1 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 3.1. The transitions of these are presented formally in appendix A.4 on page 82.

Table 3.1: QSM instruction list

Instruction	Arguments	Description
`QLoad`	*nm:Name*, *k::qubit*	Creates qubit named *nm* and sets the value to $\|k\rangle$.
`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.
`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.

Continued on next page

Instruction	Arguments	Description
`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.
`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.

Continued on next page

Instruction	Arguments	Description
`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 $\|i\|^{th}$ 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 $\|i\|^{th}$ location from the bottom of the classical stack.
`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.

3.2 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.

3.2.1 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 sub-section 3.2.2.

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 3.1 on the next page.

3.2.2 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.

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

[1] This functionality is currently not available in L-QPL, but may be re-introduced at a later date.

```
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

```
1  CFlip_fcdlbl0 Start
2      QLoad q |0>
3      QApply 0 !Had
4      Measure l0 l1
5      Jump l3
6  l0  QDiscard
7      QCons b #False
8      EndQC
9  l1  QDiscard
10     QCons b #True
11     EndQC
12 l3  QPullup b
13     Return 0
14     EndProc
15
16 main Start
17     Call 0 CFlip_fcdlbl0
18     NoOp
19     EndProc
```

(b) Generated code

Figure 3.1: L-QPL and QSM coin flip programs

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.

```
1 i = 5;
```

$\Longrightarrow$

```
1      CLoad 5
2      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.

```
1 i = 5;
```

$\Longrightarrow$

```
1      CLoad 5
2      QMove i
```

Right hand side is a constant qubit. Emit the `QLoad` instruction with the qubit value and the left hand side variable name.

```
1 q = |1>;        ⟹  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.

```
1 random :: (maxval :Int;          ⟹  1 CLoad 15
2            rand :Int) =              2 QMove c18
  {...}                                3 QName c18 maxval //In
3 ...                                  4 Call 0 random_fcdlbl0
4 x = random(15);                      5 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.

```
1 outqs = Cons(q, inqs');    ⟹  1   QCons c4 #Cons
                                 2   QBind inqs'
                                 3   QBind q
                                 4   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 appendix A.4.1 on page 82 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.

```
1 measure q of
2   |0> => {n1 = 0}
3   |1> => {n1 = 1};
```

$\Longrightarrow$

```
1     QPullup q
2     Measure l7 l8
3     Jump l9
4  l7 QDiscard
5     CLoad 0
6     QMove n1
7     EndQC
8  l8 QDiscard
9     CLoad 1
10    QMove n1
11    EndQC
12 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 1.3.5 on page 14, 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. These items are illustrated in figure 3.2 on the facing page.

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.

```
1 case l of
2   Nil => {i = 0}
3   Cons (_, l1) => {
4     n= len(l1);
5     use n;
6     i = 1 + n;
7 }
```

$\Longrightarrow$

```
1  Split (#Nil,l0) (#Cons,l1)
2    Jump l4
3 l0 QDiscard
4    CLoad 0
5    QMove i
6    EndQC
7 l1 QUnbind c0
8    QUnbind l1
9    QDiscard
10   QPullup c0
11   QDiscard
12   QPullup l1
13   QName l1 l
14   Call 0 len_fcdl0
15   QName i n
16   QPullup n
17   Use l2
18   Jump l3
19 l2 QDiscard
20   CLoad 1
21   CGet 0
22   CApply +
23   QMove i
24   EndQC //For Use
25 l3 NoOp
26    EndQC //For Split
27 l4 NoOp
```

Figure 3.2: QSM code generated for a case and use statement.

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.

```
1 if b == 0 => {
2     theGcd = a;
3 } else =>  {
4     (theGcd) =
5       gcd(b, a mod b);
6 }
```

$\Longrightarrow$

```
1       CGet -2
2       CLoad 0
3       CApply ==
4       CondJump lb12
5       CGet -1
6       QMove theGcd
7       Jump lb11
8  lb12 CLoad True //else
9       CondJump lb13
10      CGet -1
11      CGet -2
12      CApply %
13      QMove c0
14      CGet -2
15      QMove c1
16      QName c1 a
17      QName c0 b
18      Call 0 gcd_fcdlb10
19      Jump lb11
20 lb13 NoOp
21 lb11 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 sub-section 3.2.4 on page 43 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.

3.2.3 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 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 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` ¬.

See under Guard clauses, sub-section 3.2.2 on the facing page 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 sub-section 3.2.2. 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:

```
1 gcd::(a:Int, b:Int; theGcd:Int)=
2 { use a,b in
3   { if b == 0 => { theGcd    = a}
4     else      => { (theGcd) = gcd(b, a mod b)}
5   }
6 }
```

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.

```
1 lt = Cons(|0>,                    ⟹  1   QLoad c22 |0>
2         Cons(|0>,                     2   QLoad c23 |0>
3         iTZQList(2×n)));              3   CLoad 2  //for the call
                                        4   CGet -1
                                        5   CApply ×
                                        6   QMove c24
                                        7   QName c24 n
                                        8   Call 0 iTZQList_fcdlbl5
                                        9   QCons c25 #Cons
                                        10  QBind nq
                                        11  QBind c23 //c23:nq
                                        12  QCons c26 #Cons
                                        13  QBind c25
                                        14  QBind c22 //c22:c23:nq
                                        15  QName c26 lt
```

3.2.4 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.

Chapter 4

Example L-QPL programs

4.1 Basic examples

This section covers a number of the standard examples of quantum algorithms covered in most introductions to the subject.

4.1.1 Quantum teleportation function

The L-QPL program shown in figure 4.2 on the next page is an implementation of a function that will accomplish quantum teleportation as per the circuit shown in figure 4.1. (See also [Wat] or [NC00]). It also provides a separate function to place two qubits into the EPR state.

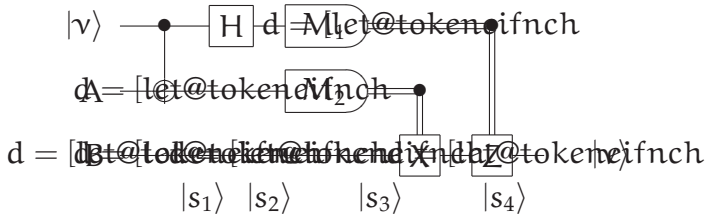


Figure 4.1: Quantum teleportation

Note that the teleport function, similarly to the circuit, does not check the precondition that qubits a and b are in the EPR state, which is required to actually have teleportation work.

4.1.2 Quantum Fourier transform

The L-QPL program to implement the quantum Fourier transform in figure 4.3 on the next page uses two recursive routines, qft and rotate. These functions assume the qubits to transform are in a List. (See also [Sel04] and either [Wat] or [NC00]).

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 qubits in the list. qft then recursively calls itself on the remaining qubits in the list.

```
1  prepare::( ; a:Qubit, b:Qubit)=
2  { a = |0>;  b = |0>;
3    Had a;
4    Not b  <=  a;
5  }
6  teleport::(n:Qubit, a:Qubit, b:Qubit ; b:Qubit) =
7  { Not a  <=  n ;
8    Had n;
9    measure a of
10       |0> => {}   |1> => {Not b};
11   measure n of
12       |0> => {}   |1> => {RhoZ b}
13 }
```

Figure 4.2: L-QPL code for a teleport routine

```
1  #Import Prelude.qpl
2  rotate::(n:Int | h:Qubit, qbsIn :List(Qubit);
3                   h:Qubit, qbsOut:List(Qubit))=
4  { case qbsIn of
5      Nil           => {qbsOut = Nil }
6      Cons(hd, tl) =>
7        { Rot(n) h     <=  hd;
8          rotate(n+1) h tl;
9          qbsOut = Cons(hd,tl) }
10 }
11 qft::(qsIn:List(Qubit); qsOut:List(Qubit)) =
12 { case qsIn of
13     Nil          => {l = Nil}
14     Cons(hd,tl) =>
15       {  Had hd;
16          rotate(2) hd tl;
17          qft tl;
18          qsOut = Cons(hd,tl) }
19 }
```

Figure 4.3: L-QPL code for a quantum Fourier transform

4.1.3 Deutsch-Jozsa algorithm

The L-QPL program to implement the Deutsch-Jozsa algorithm is in figure 4.4 with supporting routines in figure 4.6, figure 4.5, figure 4.7. The hadList function is defined in figure 4.32 on page 67.

The algorithm decides if a function is balanced or constant on n bits. This implementation requires supplying the number of bits / qubits used by the function, so that the input can be prepared. Additionally, it currently requires hand-writing the *oracle* for this function, shown in figure 4.8. The oracle in this example is for a balanced function. (See also [Wat] or [NC00]).

The function dj creates an input list for the function, applies the Hadamard transform to all the elements of that list and then applies the oracle. When that is completed, the initial segment of the list is transformed again by Hadamard and then measured.

```
1  #Import initList.qpl
2  #Import prependnzeros.qpl
3  #Import hadList.qpl
4  #Import measureInps.qpl
5  #Import djoracle.qpl
6
7  dj::(size:Int |; resultType:Ftype)=
8  {   inlist = prependNZeroqbs(size | |1>);
9      hadList inlist;
10     djoracle inlist;
11     inputs = initialList(inlist);
12     hadList inputs;
13     resultType = measureInputs(inputs);
14 }
```

Figure 4.4: L-QPL code for the Deutsch-Jozsa algorithm

The function prependNZeroqbs creates a list of qubits 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:

$$[|0\rangle, |0\rangle, |0\rangle, |1\rangle]$$

```
1  #Import Prelude.qpl
2  prependNZeroqbs::(size:Int | last:Qubit;
3                                resultList:List(Qubit))=
4  {     if (size == 0) => { resultList = Cons(last,Nil)}
5        else           => { last'      = addNZeroqbs(size - 1 | last);
6                            resultList = Cons(|0>,last') }
7  }
```

Figure 4.5: 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
initialList::(inlist:List(a) ; outlist:List(a)) =
{   case inlist of
      Nil           =>
            {outlist = Nil}
      Cons(hd,tail) =>
            {outlist = initial(hd, tail)}
}
initial::(head:a, inlist:List(a) ;
                  outlist:List(a))=
{  init    = initialList(inlist);
   outlist = Cons(head,init)
}
```

Figure 4.6: L-QPL code accessing initial part of list

```
#Import Prelude.qpl
qdata Ftype = {Balanced | Constant}
measureInputs::(inputs:List(Qubit) ; 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 4.7: L-QPL code to measure a list of qubits

The measureInputs function recursively measures the qubits 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.

The oracle for the Deutsch-Jozsa algorithm is normally assumed to be a given. An actual implementation such as this requires an actual function to be provided. The function djoracle effects a transformation on the qubits in the input list by delegating to the function bal.

```
1  balanced :: (ctlqb:Qubit, inlist :List(Qubit) ;
2               ctlqb:Qubit, outlist:List(Qubit))=
3  { case inlist of
4      Nil          => {outlist = Nil}
5      Cons(hd,tl) =>
6        { case tl of
7              Nil => { Not hd    <= ctlqb;
8                       outlist  = Cons(hd,Nil)}
9              Cons(h,t) =>
10               { balanced(ctlqb, t ; ctlqb,outt);
11                 outlist = Cons(hd,Cons(h,outt)) } }
12 }
13 djoracle::(inl:List(Qubit); outl:List(Qubit))=
14 {  case inl of
15     Nil         => { outl = Nil}
16     Cons(hd,tl)=> { balanced hd tl;
17                     outl = Cons(hd,tl)}
18 }
```

Figure 4.8: L-QPL code for Deutsch-Jozsa oracle

4.2 Hidden subgroup algorithms

The two algorithms presented in this section, the Grover search and Simons, are examples of the hidden subgroup problem.

4.2.1 Grover search algorithm

The L-QPL program to implement the Grover search algorithm is in figure 4.9 with supporting routines in figure 4.11, figure 4.10, figure 4.12. The hadList function is defined in figure 4.32 on page 67 and the oracle for this implementation in figure 4.13 on page 52.

For a specific function $f : bit^n \to bit$, the algorithm probabalistically determines the solution to $f(x) = 1$. Classically, this would require 2^n applications of f. The quantum algorithm requires $\mathcal{O}(\sqrt{2^n})$ applications. For the algorithm, first define

$$U_f |x\rangle = (-1)^{f(x)} |x\rangle \text{ and } U_0 |x\rangle = \begin{cases} |x\rangle & \text{if any } x \neq 0 \\ -|x\rangle & \text{if } x = 0^n \end{cases}$$

then:

- Start with n zeroed qubits and apply Hadamard to them.
- Apply $G = -H^{\otimes n} U_0 H^{\otimes n} U_f$ approximately $\sqrt{2^n}$ times.
- Measure the qubits, forming an integer and check the result.

In the implementation that follows, U_0 is given by the function phase, U_f is the function oracle and G is given by gtrans.

For a complete description and analysis of the algorithm, see [Wat] or [NC00].

The function **main** creates a zeroed qubit list for the function, applies the Hadamard transform to all the elements of that list and then applies the G transformation 4 times as this example is for a 4-bit function.

```
1 #Import intListConversion.qpl
2 #Import gtrans.qpl
3 main::()=
4 { dataqbs = intToZeroQubitList(15 | );
5   hadList dataqbs;
6   doNGrovers(4) dataqbs;
7   i = qubitListToInt(dataqbs);
8 }
```

Figure 4.9: L-QPL code to call the grover search algorithm

The function intToZeroQubitList creates a list of zeroed qubits as long as the standard binary representation of the input number. For example, if it were passed the value 3, it would return a list of length 2. If it were passed the value 21 it would return a list of length 5.

```
#Import Prelude.qpl
intToZeroQubitList::(n:Int | ; nq:List(Qubit))=
{ if  n == 0 => { nq  = Nil }
  else       => { nq' = intToZeroQubitList(n >> 1 |);
                  nq  = Cons(|0>, nq') }
}
qubitListToInt::(nq:List(Qubit) ; n:Int)=
{ case nq of
    Nil => { n = 0}
    Cons(q, nq') =>
           { n' = qubitListToInt(nq');
             measure q of
               |0> => {n1 = 0}
               |1> => {n1 = 1};
             use n1, n' in { n = n1 + (n' << 1) }   }
}
```

Figure 4.10: L-QPL code to convert integers from or to lists of qubits

The function qubitListToInt creates a probabilistic integer based on the values of the qubits in the list. Note that the list is assumed to be least significant digit first.

The phase uses phase kickback to implement the U_0 transformation. Note this is an excellent example of the utility of both 0-based quantum control and quantum control by a data structure.

```
#Import Prelude.qpl

phase::(inqs:List(Qubit) ; outqs:List(Qubit))=
{  a=|1>;   Had a;
   Not a <= ~inqs; //Not a when all elts of inqs are |0>
   Had a;   Not a;   discard a;
   outqs=inqs
}
```

Figure 4.11: L-QPL code using phase kickback to transform the list

The function doNGrovers calls the function gtrans repeatedly, based on the input n.

The gtrans function implements the G transform above. Note that we may ignore the minus sign in G's definition as we are using density matrices.

The oracle for the grover algorithm is normally assumed to be a given. We provide a specific implementation where $f(12) = 1$.

4.2.2 Simon's algorithm

Simon's algorithm determines a global property of a given function, f, *provided it is guaranteed to follow some rules*. The function f, from n bits to n bits, must either be one-to-one, or when for any vectors of bits x, y with $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ then $(x_1 \oplus y_1, \ldots, x_n \oplus y_n) = (s_1, \ldots, s_n)$ where $\oplus$ is exclusive-or and the vector s is the same for any choices of x or y.

```
#Import oracle.qpl
#Import hadList.qpl
#Import phase.qpl

gtrans::(dataqbs:List(Qubit);
         dataqbs:List(Qubit))=
{  hadList dataqbs;
   phase dataqbs;
   hadList dataqbs;
   oracle dataqbs;
}


doNGrovers::(n:Int | dataqbs:List(Qubit);
                     dataqbs:List(Qubit))=
{  if (n==0) => {} //Done
   else      =>
      { gtrans dataqbs;
        doNGrovers(n-1) dataqbs}
}
```

Figure 4.12: L-QPL code to do the grover transformation

```
#Import Prelude.qpl

oracle::(dataqbs:List(Qubit);
         dataqbs:List(Qubit))=
{  a = |1>;  Had a;
   case dataqbs of
     Nil =>
         { dataqbs = Nil;
           Had a; Not a; discard a}
     Cons(hd,tl) =>
         { case tl of
             Nil =>
                 { dataqbs = Nil;
                   Had a; Not a; discard a; discard hd;}
             Cons(hd',tl') =>
                 { Not a <= ~hd, ~hd', tl';  // Search for 0 0 1 1 (12)
                   dataqbs = Cons(hd,(Cons(hd',tl')));
                   Had a; Not a; discard a }
         }
}
```

Figure 4.13: L-QPL code using phase kickback when $f(12) = 1$

The L-QPL program to implement Simon's algorithm is in figure 4.14 with supporting routines in figure 4.15. The hadList function is defined in figure 4.32 on page 67 and the oracle and "blackboxes" for this implementation in figure 4.16 on the next page and figure 4.17.

For a specific function $f : bit^n \to bit^n$, the quantum portion of the algorithm returns a value r such that $r \cdot s = 0$. The classical remainder of the algorithm usually runs the quantum portion a number of times to obtain different values of r and then performs Guassian elimintion on the series of linear equations to determine s. We present only the quantum portion in this example.

The quantum portion first creates a list of length n of qubits initialized to $|0\rangle$. The Hadamard transform is applied to the list, followed by the oracle for f followed by another Hadamard. The qubits are then measured and the result is r.

For a complete description and analysis of the algorithm, see [Wat] or [NC00].

In figure 4.14, the type ZorO represents bits. The function qubToBitList converts a list of qubits to elements of type ZorO by repeated measures.

```
1  #Import oracle.qpl
2  #Import hadList.qpl
3
4  main::()=
5  {  dqs=Cons(|0>,Cons(|0>,Cons(|0>,Nil)));
6     hadList dqs;
7     oracle dqs;
8     hadList dqs;
9     qubToBitList(dqs ; bits)
10 }
11
12 qdata ZorO = { Z | O }
13
14 qubToBitList::(inqubits:List(Qubit);
15                    out:List(ZorO))=
16 { case inqubits of
17      Nil => { out = Nil}
18      Cons(hd,tl) =>
19             { outtl = qubToBitList(tl);
20               measure hd of
21                 |0> => {out = Cons(Z,outtl)}
22                 |1> => {out = Cons(O,outtl)}}
23 }
```

Figure 4.14: L-QPL code to do Simon's algorithm

The function ndestLength is a non-destructive measure of the length of a list. The function makeZeroQubitList creates a list of zeroed qubits when given a length.

The oracle for Simon's algorithm is one of the more complex in the examples given in this thesis. It makes use of the decomposition of $f : bit^3 \to bit^3$ into three functions, $f_i : bit^3 \to bit$ where f_i gives the i-th component of f. These are represented by the functions bbox1, bbox2 and bbox3.

The oracle function creates a list of ancilla qubits which are then used in each of the bbox$n$ functions, with the ancilla's being exclusive-or'ed with the results of the functions.

```
#Import Prelude.qpl

ndestLength::(inLis:List(a);
              len:Int, outLis:List(a))=
{  case inLis of
     Nil => { len = 0; outLis = Nil}
     Cons(hd,tl) =>
            { ndestLength(tl; tlen, tail);
              use tlen in { len = 1 + tlen};
              outLis = Cons(hd, tail)}
}

makeZeroQubitList::(len:Int | ; outLis:List(Qubit))=
{  if (len =< 0) => { outLis = Nil}
   else          => { outListTail = makeZeroQubitList( len-1 |);
                      outLis = Cons(|0>,outListTail)}
}
```

Figure 4.15: L-QPL list functions

```
#Import blackboxes.qpl

oracle::(dataqbs:List(Qubit);
         dataqbs:List(Qubit))=
{  ndestLength(dataqbs; len, dataqbs);
   use len;
   makeZeroQubitList(len | ; ancillas);
   bboxfRecurse(1,len) dataqbs ancillas;
   discard ancillas;
}

bboxfRecurse::(start:Int, len:Int | dqs:List(Qubit), anc:List(Qubit);
                                    dqs : List(Qubit), anc :List(Qubit))=
{  if (len < start) => { }
   else => { bbox(start) dqs anc;
             bboxfRecurse((start +1), len) dqs anc }
}

//Assumption of three qubits - customize for each function
bbox::(index:Int | dqs:List(Qubit), anc:List(Qubit);
                   dqs:List(Qubit), anc:List(Qubit))=
{  if (index == 1) => { bbox1 dqs anc}
      (index == 2) => { bbox2 dqs anc}
   else            => { bbox3 dqs anc}
}
```

Figure 4.16: L-QPL code implementing oracle for Simon's algorithm

The code for the black boxes simply applies various combinations of controlled Nots to create the desired boolean functions. The code in figure 4.17 only shows the first component. The code for the second and third components is similar.

```
#Import ListUtils.qpl
// bb1 results in  1,0,0,1,0,1,1,0 when applied to combos of x,y,z
bbox1::(dqs:List(Qubit), anc:List(Qubit);
        dqs:List(Qubit), anc:List(Qubit))=
{  case anc of
     Nil => { anc = Nil}
     Cons(hAnc,atail) =>
       { case dqs of
           Nil => { anc = Cons(hAnc,atail);
                    dqs = Nil}
           Cons(x,tl1)=>
             { case tl1 of
                 Nil  =>{anc = Cons(hAnc,atail);
                         dqs = Cons(x,Nil)}
                 Cons(y,tl2)=>
                   { case tl2 of
                       Nil => {anc = Cons(hAnc,atail);
                               dqs = Cons(x,(Cons(y,Nil)))}
                       Cons(z,tl3)=>
                         { // Now have x, y and z to work with.
                           cq = |0>; Not cq <= y,z;
                           Not cq <= ~y,~z;
                           Not cq <= x;
                           Not hAnc <= cq;  discard cq;
                           anc = Cons(hAnc,atail);
                           dqs = Cons(x,Cons(y,Cons(z,tl3))) }
                   }
             }
       }
}
```

Figure 4.17: L-QPL code implementing first black box for Simon's algorithm

4.3 Quantum arithmetic

This section provides examples of functions that will do arithmetic on quantum values. The primary source of these algorithms is [VBE95], although the algorithms and types used in modular arithmetic are not ones I have encountered yet in the literature.

4.3.1 Quantum adder

This section provides subroutines that perform *carry-save* arithmetic on qubits. The carry and sum routines in figure 4.18 function as gates on four qubits and three qubits respectively.

```
1  carry::(c0:Qubit, a:Qubit, b:Qubit, c1:Qubit;
2          c0:Qubit, a:Qubit, b:Qubit, c1:Qubit) =
3  {  Not c1  <= b, a;
4     Not b   <= a;
5     Not c1  <= b, c0
6  }
7  sum::(c:Qubit, a:Qubit, b:Qubit;
8        c:Qubit, a:Qubit, b:Qubit) =
9  { Not b  <= a;
10   Not b  <= c;
11 }
12 carryRev::(c0:Qubit, a:Qubit, b:Qubit, c1:Qubit;
13              c0:Qubit, a:Qubit, b:Qubit, c1:Qubit) =
14 {  Not c1  <= b, c0;
15    Not b   <= a;
16    Not c1  <= b,a
17 }
```

Figure 4.18: L-QPL code to implement carry and sum gates

Additionally, the reverse of the carry is also defined in the same figure. In order to define a subtraction, which can be defined as the reverse of the add function, we would also need to define the reverse of the sum function.

The addition algorithm adds two lists of qubits 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 4.19 on the next page.

The program proceeds down the lists A and B of input qubits, 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.

4.3.2 Modular arithmetic

Most treatments of quantum modular arithmetic assume the program / algorithm will work with a quantum register with a sufficiently large number of qubits, as in [VBE95]. In L-QPL,

```
#Import carrysave.qpl
adder::(c0:Qubit, asin:List(Qubit), bsin:List(Qubit);
        c0:Qubit, asout:List(Qubit), aplusbout:List(Qubit)) =
{   case asin of
      Nil => { asout = Nil; aplusbout = Nil}
      Cons(a, taila) =>
        { case bsin of
            Nil => { asout = 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;
                          carryRev c0 a b c1;
                          sum c0 a b;
                          discard c1};
               asout = Cons(a,taila);
               aplusbout = Cons(b,tailb)}}
}
```

Figure 4.19: L-QPL code to add two lists of qubits

we define a new datatype for quantum modular integers, called QuintMod. A QuintMod consists of a triple of two integers and a list of qubits. The first integer is the maximum size of the list and the second integer is the modulus.

```
#Import ModFunctions.qpl
#Import ListFunctions.qpl
#Import quints.qpl
qdata QuintMod = {QuintMod(Int,Int,List(Qubit))}

//Convert a probabalistic int to a QuintMod
intToQuintMod::(radix:Int, n:Int ;  nq:QuintMod)=
{  use n, radix;
   determineIntSize(radix;  size);
   cintToQuintMod(n, radix | size ; nq)
}

//Convert a classical int to a QuintMod
cintToQuintMod::(n:Int, radix:Int | size:Int ; nq:QuintMod)=
{ if n == 0 => { nq = QuintMod(size,radix,Nil) }
  else      => { nmr  := n mod radix;
                 qlist = intToQubitList(nmr | );
                 nq    = QuintMod(size,radix,qlist) }
}

//Convert a QuintMod to a probabalistic integer
quintModToInt::(nq:QuintMod ; n:Int)=
{  case nq of
     QuintMod(_,radix,digits) =>
       { n' := qubitListToInt(digits);
         use radix in {n  = n' mod radix}}
}
```

Figure 4.20: L-QPL definitions of QuintMod

The code for the type definition and conversion from and to integers is given in figure 4.20.

Given these definitions and code for adding and subtracting lists of qubits(as in sub-section 4.3.1), it is now possible to define a *smart constructor* for a QuintMod.

A QuintMod triplet has only one invariant that must be maintained. That is the list of qubits must have length less than or equal to the first number of the triplet. Note this implies that the number represented by a qubit list may be outside the range of the modulus. For example, assume a modulus of 5. This implies a length of 3 qubits. The (qu)bit sequences of 101, 011, 111, representing $5, 6$ and 7 are allowed. When converting back to a viewable integer, these would be converted to $0, 1, 2$ respectively.

The function makeQuint defined in figure 4.21 maintains the length invariant, *assuming* that the length is no more than 1 greater than allowed. That is, if working with QuintMod numbers of length n, the List(**Qubit**) argument has length at most $n + 1$.

This will imply the represented number passed in is at most $2^{n+1} - 1$. As the modulus is at least 2^{n-1}, at most two subtractions of the modulus will ensure the passed in list represents a number in the range $0 - 2^n - 1$. The subtractions are controlled by the $n + 1$st qubit in the

```
#Import Prelude.qpl
#Import carrysave.qpl
#Import QuintMods.qpl

// Needs to subtract the modulus 0,1 or 2 times to
// get the correct range for the numbers.
makeQuint::(size:Int, radix:Int, digits:List(Qubit) ;
                                          res:QuintMod)=
{  getLength(digits; digits,length);
   use size,length in
     { if length > size =>
         { // Two controlled subs
           use radix;
           csubModulus(radix | digits ; digits);
           takeOnly(size+1 | digits ; digits); //Last is |0>
           csubModulus(radix | digits ; digits);
           takeOnly(size | digits ; digits); //Last is |0>
           res = QuintMod(size,radix,digits) }
       else =>
         { res = QuintMod(size,radix,digits)}
     }
}

csubModulus::(modulus:Int | digits:List(Qubit) ;
                              digs:List(Qubit))=
{  splitLast(digits, |0> ; digits, last);
   ctldintToQubitList(modulus | last ; last, subRad);
   digs = append(digits,Cons(last,Nil));
   c0   = |0>;
   subLists c0 subRad digs;
   discard c0, subRad;
}
```

Figure 4.21: Semi-Smart constructor for QuintMod

list.

The reason the numbers may take on the full range is due to the representation. For example, if we are working in mod 4, which requires 3 qubits for representation we may attempt to add 7 to itself. This will result in 14, requiring 4 qubits. The final qubit, recalling we store numbers with least significant qubit first, will be $|1\rangle$. Subtracting 4 once leaves us with 10, still requiring 4 qubits. Subtracting 4 once more gives us 6 which brings us back to the correct range.

```
1  #Import smartConstructor.qpl
2
3  addM::(asin:QuintMod, bsin:QuintMod ;
4         asout:QuintMod, aplusbout:QuintMod) =
5  { case asin of QuintMod(sizeA,mA, aDigits) => {
6    case bsin of QuintMod(sizeB,mB, bDigits) =>
7      { use sizeB;
8        normalize( sizeB| bDigits;bDigits);
9        c0 = |0>;
10       addLists c0 aDigits  bDigits;
11       discard c0;
12       asout     = QuintMod(sizeA,mA,aDigits);
13       aplusbout = makeQuint(sizeB,mB,bDigits)}}
14 }
15
16 addListToQuint::(inc:List(Qubit), dest:QuintMod;
17                  dest:QuintMod)=
18 { case dest of QuintMod(sizeD,mdD,dDigits) =>
19     { use sizeD,mdD;
20       a    = QuintMod(sizeD,mdD,inc);
21       dest = QuintMod(sizeD,mdD,dDigits);
22       addM a dest;
23       discard a}
24 }
```

Figure 4.22: Quantum modular addition

figure 4.22 shows the L-QPL code for implementing modular addition. The function addM takes in two QuintMod elements and adds the first to the second. Note the use of the smart constructor makeQuint to produce the result.

At this stage modular multiplication is straightforward to write. First, the support functions ctlCopy and ctlDouble are defined in figure 4.23. The ctlCopy function creates a "copy" of a QuintMod. Note that the qubitsare all created by controlled-Nots of the original list, so the "copy" is entangled with the original. If the control qubit is $|0\rangle$, the resulting QuintMod is zero. The ctlDouble function uses the controlled copying and introduces a new $|0\rangle$ at the beginning of the list of qubits in the QuintMod. This effectively doubles the number by shifting it one position to the right. The resulting QuintMod is created using the smart constructor defined previously.

The function multiplyM is then defined as a recursive function in figure 4.24. The algorithm used is the standard grade school multiplication method.

```
#Import basicArith.qpl

ctlCopy::( ctl:Qubit, src:QuintMod ;
           ctl:Qubit, src:QuintMod, dest:QuintMod)=
{  case src of QuintMod(size,modulus,srcDig)=>
     { use size, modulus;
       ctlCopyList(ctl, srcDig; ctl, srcDig, destDig);
       dest = QuintMod(size,modulus,destDig);
       src  = QuintMod(size,modulus,srcDig)}
}

ctlDouble::( ctl:Qubit, src:QuintMod ;
             ctl:Qubit, src:QuintMod, dest:QuintMod)=
{  case src of QuintMod(size,modulus,srcDig)=>
     { use size, modulus;
       ctlCopyList(ctl, srcDig; ctl, srcDig, destDig');
       destDig = Cons(|0>, destDig');
       dest    = makeQuint(size,modulus,destDig);
       src     = QuintMod(size,modulus,srcDig)}
}

ctlCopyList::(ctl:Qubit, srcList:List(Qubit);
              ctl:Qubit, srcList:List(Qubit), destList:List(Qubit))=
{ case srcList of
    Nil => { srcList = Nil; destList = Nil}
    Cons(hd,tl) =>
           { ctlCopyList(ctl, tl; ctl, tl, desttl);
             desthd   = |0>;
             Not desthd <= ctl, hd;
             srcList  = Cons(hd,tl);
             destList = Cons(desthd,desttl)}
}
```

Figure 4.23: Support functions for Quantum modular multiplication

```
#Import multSupport.qpl

multiplyM::(cor:QuintMod, cand:QuintMod ;
            cor:QuintMod,  res:QuintMod)=
{  case cand of QuintMod(sizeA,modA,aDig) =>
     { use sizeA,modA;
       res = cintToQuintMod(0,modA | sizeA);
       case aDig of Nil => {}
         Cons(hdA,tlA)=>
           { ctlCopy(hdA,cor;hdA,cor,corcopy);
             discard hdA;
             addM corcopy res;
             discard corcopy;
             ctl = |1>;
             ctlDouble(ctl,cor; ctl, cor, cordouble);
             discard ctl;
             res1  = QuintMod(sizeA, modA, tlA);
             multiplyM  cordouble res1; discard cordouble;
             addM res1 res; discard res1}
     }
}
```

Figure 4.24: Quantum modular multiplication

```
#Import multiply.qpl

ctlCopyOne::(ctl:Qubit, src:QuintMod ;
             ctl:Qubit, src:QuintMod, dest:QuintMod)=
{ ctlCopy(ctl,src; ctl,src,dest); //dest == 0 if ctl == 0
  ctlone = |0>;
  Not ctlone <= ~ctl;
  addListToQuint(Cons(ctlone,Nil), dest; dest);
}

square::( src:QuintMod; dest:QuintMod)=
{ ctl = |1>;
  ctlCopy(ctl, src; ctl,dest,cpy);  discard ctl;
  multiplyM cpy dest;  discard cpy
}
```

Figure 4.25: Quantum modular exponentiation support

Modular exponentiation is written in a similar manner. First the support functions ctlCopyOne and square are defined in figure 4.25. ctlCopyOne uses ctlCopy defined above to create a copy of a QuintMod, but adds 1 to the resulting QuintMod when the control qubit is $|0\rangle$, resulting in the identity for multiplication rather than the identity for addition. The function square corresponds to ctlDouble in the multiplication case.

```
1  #Import powerSupport.qpl
2
3  powerM::(base:QuintMod, pow:QuintMod ;
4                          res:QuintMod)=
5  {  case pow of QuintMod(sizeA,modA,aDig) =>
6       { use sizeA,modA;
7         res = cintToQuintMod(1,modA|sizeA);
8         case aDig of Nil => {discard base} //  base^0 is 1
9         Cons(hdA,tlA)=>
10          { ctlCopyOne(hdA,base;hdA,base,basecopy);
11            discard hdA;
12            multiplyM basecopy  res;  discard basecopy;
13            square(base; basesqd);
14            pow' = QuintMod(sizeA, modA, tlA);
15            powerM(basesqd,pow'; res1);
16            multiplyM res1 res;  discard res1}}
17 }
```

Figure 4.26: Quantum modular exponentiation

The code for the actual powerM function is in figure 4.26.

4.4 Order finding

Shor's algorithm for factoring depends on the quantum algorithm for order finding. Shor's algorithm may be summarized by the following three steps:

- Setup;
- Call order finding (a quantum algorithm);
- Check to see if an answer has been found, repeat if not.

In this section, we concentrate on the quantum portion, order finding. Theoretical details may be found in [NC00].

The code for the main order finding function, OrderFind is shown in figure 4.27

```
1  #Import powerFind.qpl
2  #Import inverseQft.qpl
3
4  orderFind::(n:Int, x:Int | ; order:Int)=
5  {
6      size    := determineIntSize(n);
7      sizeT   := 2 × size + 1 + 2;
8      makeZeroQubitList(sizeT | ; tList );
9      hadList tList;
10     xQm     = intToQuintMod(x);
11     powList xQm tList;
12     discard xQm;
13     inverseQft tList;
14     result  = qubitListToInt(tList);
15     gcdrs   := gcd(result,(2 ≪ sizeT));
16     order   = sizeT mod gcdrs;
17 }
```

Figure 4.27: L-QPL function for order finding.

Beginning at lines 1 and 2, the program imports files which provide various other functions. At line 4, the function is declared to take two classical integers as input and return one probabalistic integer. The first two lines of code, 6 and 7 compute the number of qubits required for the algorithm to work with a maximum error of $\frac{1}{4}$. Increasing sizeT will reduce the error.

Lines 8 through 10 complete the initial setup for the algorithm. These include: Creating tList, a list of sizeT qubits initialized to zero; applying the Hadamard transform to each qubit; creating xQm, a QuintMod with the value of the input parameter x. See sub-section 4.3.2 on page 56 for the details of modular arithmetic.

Line 11 performs the modular exponentiation and is shown in figure 4.28. This is similar to the exponentiation function in figure 4.26 on the previous page and has just been modified to work with an exponent in list form and to retain the exponent.

On completion of the exponentiation, the actual result is discarded as it has no further effect on the algorithm. Then, the inverse quantum Fourier transform (shown in figure 4.29

```
1  #Import power.qpl
2
3  powList::(base:QuintMod, pow:List(Qubit) ;
4            res:QuintMod,  pow:List(Qubit))=
5  {  case base of QuintMod(sizeA,modA,baseDig) =>
6       { use sizeA,modA;
7         res = cintToQuintMod(1,modA|sizeA);
8         base = QuintMod(sizeA,modA,baseDig);
9         case pow of
10           Nil           => {pow=Nil; discard base} //  base^0 is 1
11           Cons(hdA,tlA) =>
12             { ctlCopyOne(hdA,base;hdA,base,basecopy);
13               multiplyM basecopy  res;
14               discard basecopy;
15               square(base; basesqd);
16               powList basesqd tlA;
17               multiplyM basesqd res;  discard basesqd;
18               pow = Cons(hdA,tlA)}}
19 }
```

Figure 4.28: L-QPL code for modular power by a list

and figure 4.30) is applied to tList and it is measured to produce a probabalistic integer in line 14.

The final part of the algorithm is normally described as using a continued fraction algorithm to compute the potential order. This is the same as dividing 2^{sizeT} by the greatest common division of itself and the result.

At this point, since this is a probabilistic algorithm, the calling function would need to check whether the result is correct. If so, it may then be used in the completion of the factoring algorithm.

Various other support functions for this algorithm are shown in figure 4.31 and figure 4.32.

```
#Import inverseRotate.qpl
#Import Utils.qpl

inverseQft :: (inqs:List (Qubit); outqs:List (Qubit)) =
{  reverse inqs;
   inverseQft' inqs;
   outqs = reverse(inqs);
}

inverseQft' :: (inqs:List (Qubit); outqs:List (Qubit)) =
{ case inqs of
    Nil => {outqs = Nil}
    Cons(h,inqs') =>
      {  inverseQft' inqs';
         inverseRotate (2) h inqs';
         Had h;
         outqs = Cons(h,inqs') }
}
```

Figure 4.29: Function to apply the inverse quantum Fourier transform

```
#Import Prelude.qpl

inverseRotate ::(n:Int| h:Qubit, inqs:List (Qubit);
                 h:Qubit, outqs:List (Qubit))=
{ 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 4.30: Function to apply inverse rotations as part of the inverse QFT

```
#Import inverseQft.qpl
#Import ListUtils.qpl

gcd::(a:Int, b:Int ; ans: Int) =
{  use a, b in {
     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 4.31: GCD and import of other ulitlties

```
#Import Prelude.qpl

ndestLength::(inLis:List(a);
              len:Int, outLis:List(a))=
{  case inLis of
     Nil => { len = 0; outLis = Nil}
     Cons(hd,tl) =>
            { ndestLength(tl; tlen, tail);
              use tlen in { len = 1 + tlen};
              outLis = Cons(hd, tail)}
}

makeZeroQubitList::(len:Int | ; outLis:List(Qubit))=
{  if (len =< 0) => { outLis = Nil}
   else          => { outListTail = makeZeroQubitList( len-1 |);
                      outLis = Cons(|0>,outListTail)}
}


hadList::(inhqs:List(Qubit) ; outhqs:List (Qubit))=
{ case inhqs of
    Nil             => { outhqs = Nil }
    Cons(q,hadtail) =>
       {  Had q;
          hadList hadtail;
          outhqs = Cons(q, hadtail)}
}

append::(list1:List(a), list2:List(a) ; app:List(a))=
{ case list1 of
     Nil           => {app = list2}
     Cons(a,subl1) =>
       { app = Cons(a, append(subl1, list2)) }
}

reverse::(inlis:List(a) ; rvlis:List(a))=
{  rvlis = rev'(inlis, Nil) }

rev'::(inlist:List(a), accin:List(a) ; revlist:List(a)) =
{ case inlist of
    Nil             => {revlist = accin}
    Cons(a,sublist) =>
      { acc     = Cons(a, accin);
        revlist = rev'(sublist, acc) }
}
```

Figure 4.32: Various list ulitlties

Chapter 5

Using the system

5.1 Running the L-QPL compiler

The compiler is run from the command line as:

```
lqpl <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.

5.2 Running the QSM Emulator

Note: Figures in this section are from an earlier version of the emulator and will be updated as some point in time. The general instructions on how to run the emulator are still applicable.

5.2.1 Window layout

When starting the simulator with the command `emlqpl`, The first thing seen is figure 5.1

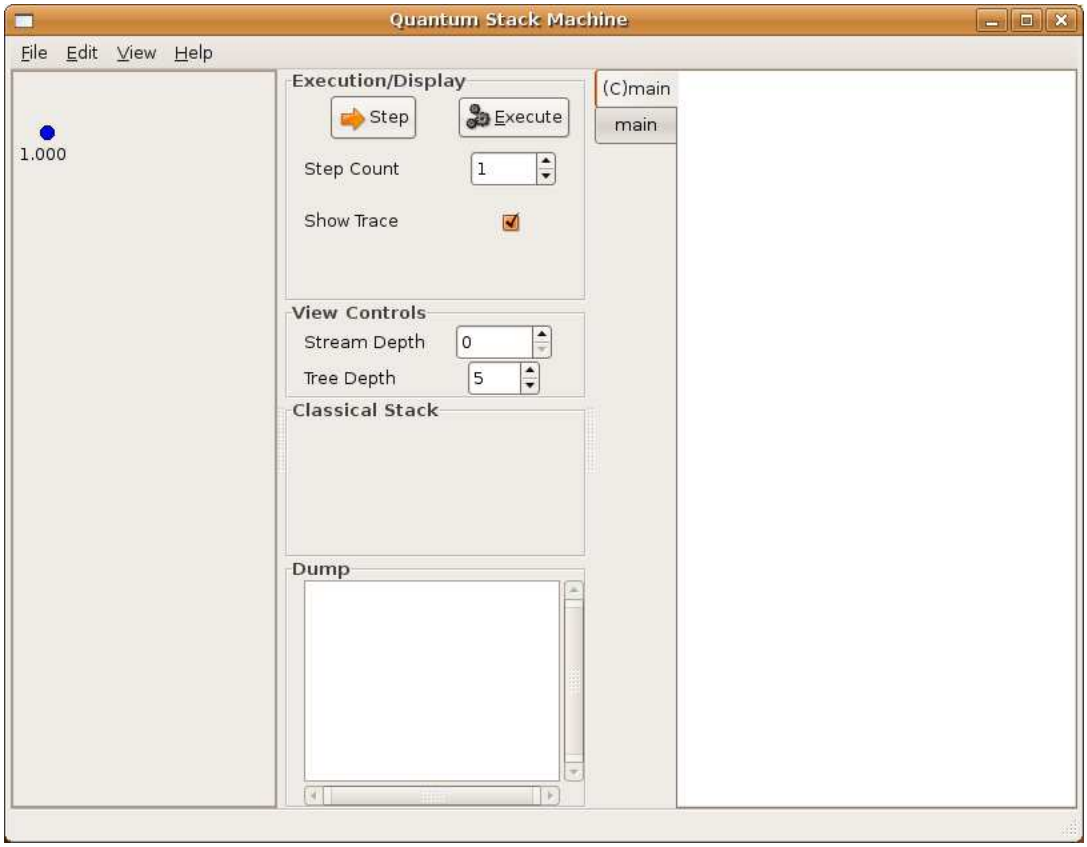


Figure 5.1: The emulator window

The emulator window is composed of three main sections. On the left is the quantum stack display area. On the right is a tabbed display of the emulator assembly/machine code.

The middle section is divided into four parts. At the top is an execution and display control area followed by a view control area. Beneath these is the classical stack display area with the dump display area at the bottom.

5.2.2 Loading a file

The first required step is to actually load a program into the emulator, using the `File --> Open` dialog in figure 5.2.

Opening a QSM assembly file will check the compiler version used to create it and translate it to machine code. If the compiler and emulator versions do not match, a warning dialog, as in figure 5.3 will appear. It is suggested that one ensures the version of the compiler and emulator are the same.

After a successful assemble, the assembled code will appear in the right hand side. In a program, each function will appear as a separate tab in the tabbed code window. The cur-

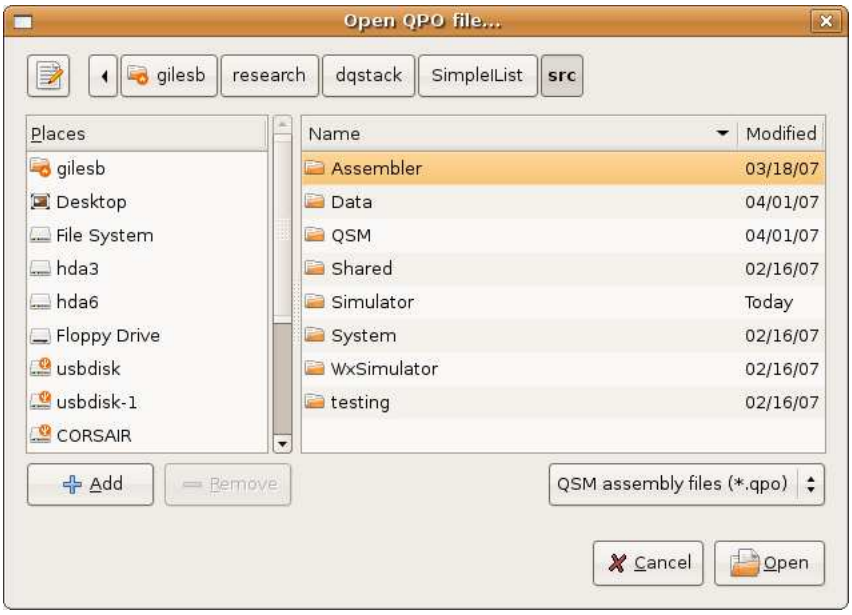


Figure 5.2: Emulator file open dialog

Figure 5.3: Version mis-match warning

rently executing code will be in the top tab, which is labelled with a "(C)" and the name of the currently executing function. Clicking on a tab will show the code associated with that tab.

5.2.3 Setting preferences

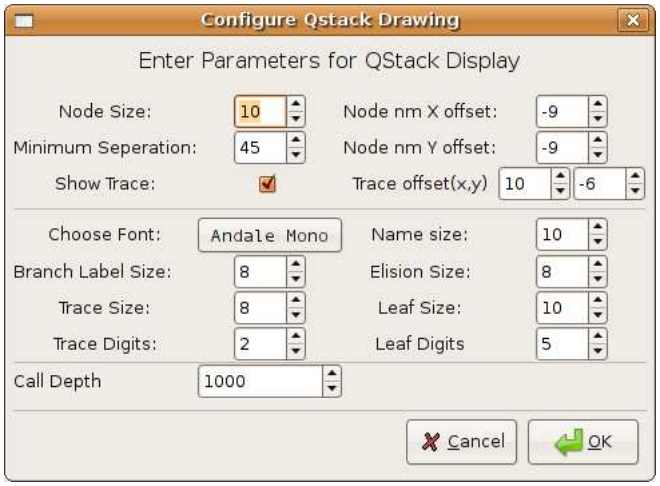


Figure 5.4: Preferences dialog

Prior to executing the program, various display and execution options may be set by using the `Edit --> Preferences` dialog, shown in figure 5.4. The top third of the dialog controls the spacing and size of nodes and their labels. The middle controls the font used and the font size choices for various labels. Typically, the defaults for these are sufficient for general execution.

The final section, with the single item `Call Depth` is used to control the actual depth of recursion *multiplied by the depth in the stream.* In figure 5.4 the call depth is 1000, meaning that at stream depth 0, we will perform 1000 calls before signalling non-termination, at stream depth 5, 6000 calls and so on.

5.2.4 Running the program

The program execution is controlled by the elements of the `Execution/Display` section of the main window. As shown in figure 5.5, there are two buttons (`Step` and `Execute`), a spin control (`Step Count`) and a checkbox (`Show Trace`).

For each click of the `Step` button, the emulator will execute *`Step Count`* instructions and then redisplay the components of the quantum stack machine. The spin control may be set to any positive number.

The `Execute` button will run the program until it completes. Completion may either be due to non-termination (e.g., exceeding the call depth number of calls at the current stream depth), or actual completion. See the description of `Stream Depth` below. As well, while executing, the program will display a progress bar below the `Show Trace` area. After completion, the machine components will be re-displayed.

Figure 5.5: Execution control section of the main window

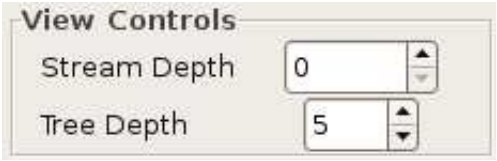


Figure 5.6: View controls section of the main window

The other component that affects execution in both step mode and execute mode is the `Stream Depth`, shown in figure 5.6. Essentially, if one encounters a non-termination (a zero quantum stack) after executing, just increase the `Stream Depth` and continue.

5.2.5 Result interpretation

Obviously, the first step is to visually examine the quantum stack. In case where a simple final result is produced, this is often enough.

However, in cases where there are a significant number of nodes with multiple branches, it can be difficult to determine what the results actually are.

For example, consider the end result of running Simon's algorithm, as shown in figure 5.7. The important result is "What are the resulting bit-strings?"[1]. Using the menu item `File --> Simulate`, we bring up a simulation dialog, which does the "roll the dice" and shows us what our end result would be when transferred to a classical computer. For example, for three invocation of simulation, we get sub-figures (a), (b), and (c) as shown in figure 5.8.

Note that the bit string can simply be read from the top three entries in the simulation results. It should also be noted that the additional simulations do not require re-execution by the emulator. Instead, a new random value is generated and used to determine a single path down the quantum stack for the values.

[1]Obviously, it is possible to determine the results solely by examination of the quantum stack as displayed. The method shown here becomes more relevant the more complicated the result stack.

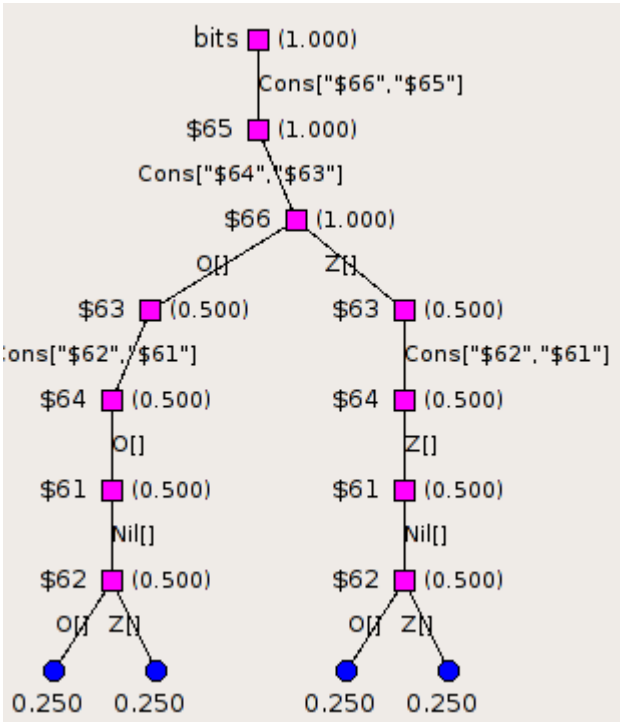
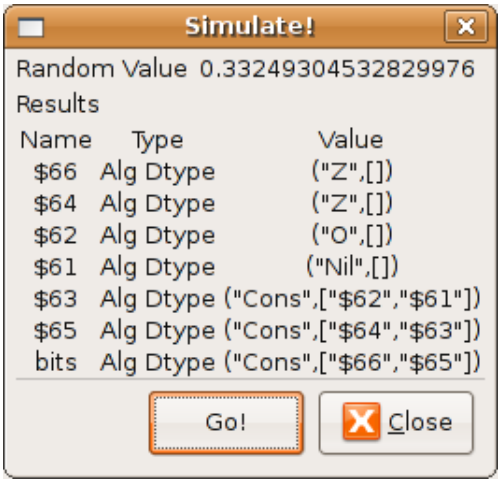
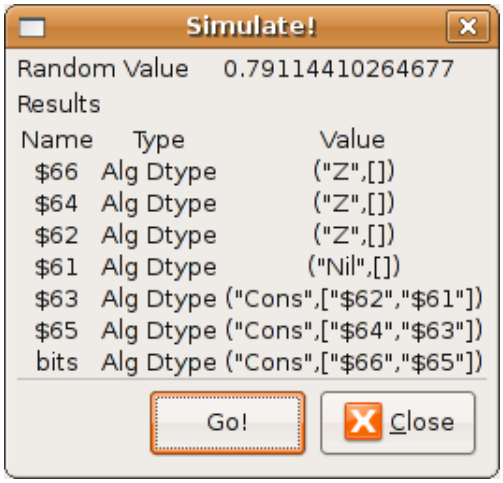


Figure 5.7: Quantum stack at end of Simon's algorithm



(a) First simulation



(b) Second simulation

(c) Third simulation

Figure 5.8: Simulation of Simon's algorithm

Bibliography

[AG05] Thorsten Altenkirch and Jonathan Grattage, *A functional quantum programming language*, LICS, 2005, pp. 249–258.

[Gil07] Brett G. Giles, *Programming with a Quantum Stack*, Master's thesis, University of Calgary, Calgary, Alberta, Canada, April 2007.

[Kni96] E. Knill, *Conventions for quantum pseudocode*, 1996.

[KR88] Brian W. Kernighan and Dennis M. Ritchie, *The C programming language, second edition*, Prentice Hall, Inc,, Cambridge, CB2 1RP, Great Britain, 1988, ISBN 0-13-110362-8.

[NC00] 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.

[Öme00] Bernhard Ömer, *Quantum programming in QCL*, Master's thesis, Department of Computer Science, Technical University of Vienna, January 2000, http://tph.tuwien.ac.at/ oemer/qcl.html.

[Pey03] Simon Peyton Jones (ed.), *Haskell 98 language and libraries – the revised report*, Cambridge University Press, Cambridge, England, 2003.

[Sel04] Peter Selinger, *Towards a quantum programming language*, Mathematical Structures in Computer Science **14** (2004), no. 4, 527–586.

[VBE95] Vlatko Vedral, Adriano Barenco, and Artur Ekert, *Quantum networks for elementary arithmetic operations*, Physical Review A **54** (1995), 147.

[Wat] John Watrous, *Lecture notes for quantum computation*, University of Calgary.

Appendix A

The quantum stack machine

A.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.

For the semantics of this, please refer to [Gil07].

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

A.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 [Gil07]. 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.

A.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.

$$(\mathcal{C}, S, Q, D, N) \tag{A.1}$$

The code, $\mathcal{C}$, is a list of machine instructions. Transitions effected by these instructions are detailed in appendix A.4.1 on page 82. English descriptions of the instructions and what they do are given in section 3.1 on page 31.

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 appendix A.2.6 on the facing page.

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 appendix A.4.2 on page 82.

A.2.2 Labelled quantum stack machine

The labelled QS-machine, designated as 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:

$$(\mathcal{C}, S, L(Q), D, N) \tag{A.2}$$

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

When this labelling was introduced in [Gil07], 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 appendix A.4.8 on page 89.

A.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:

$$(\mathcal{C}, C, [(S, L(Q), D, N)]) \tag{A.3}$$

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 $(S, L(Q), D, N)$.

A.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.

$$\text{Inflist}(\mathcal{C}, C, [(S, L(Q), D, N)]) \tag{A.4}$$

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.

A.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.

A.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 appendix A.4.5 on page 85 and function calling in appendix A.4.9 on page 90.

A.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.

A.3 Representation of data in the quantum stack

The quantum stack was introduced and described in [Gil07] in the chapeter on semantics. This section will give further details of the implementation of the quantum stacks and show example nodes.

A.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 A.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 $\begin{bmatrix} .5 & .5 \\ .5 & .5 \end{bmatrix}$

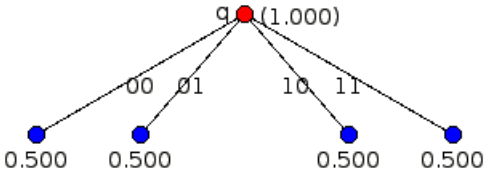


Figure A.1: A qubit after a Hadamard transform

With multiple qubits, the representation becomes hierarchical. For example, two qubits will be represented by a tree with one of the qubits 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 qubits. As illustrated in figure A.2 on the facing 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:

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

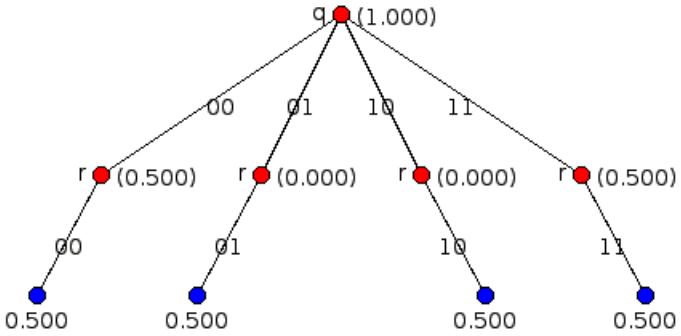


Figure A.2: Two entangled qubits

A.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.

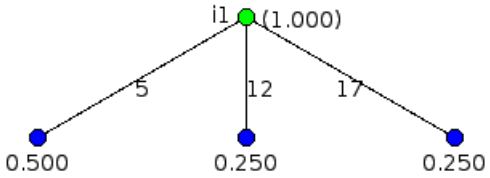


Figure A.3: An integer with three distinct values

figure A.3 depicts an integer i1 which has a 50% probability of being 5, and 25% each of being 12 or 17.

A.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*.

For example, in the **List** that appears in figure A.4 on the following page, 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`.

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

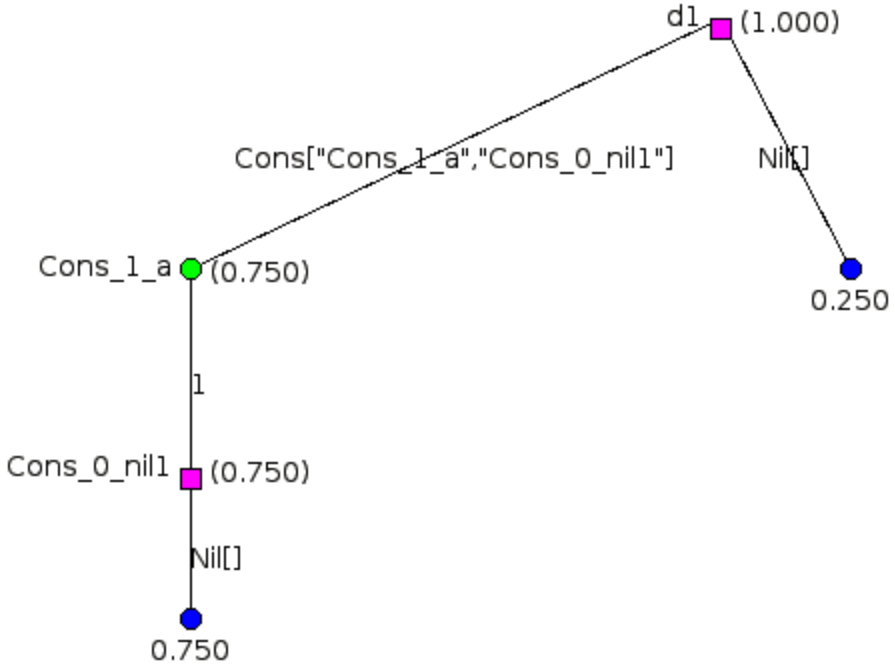


Figure A.4: A list which is a mix of [ ] and [1].

A.4 Quantum stack machine operation

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

A.4.1 Machine transitions

The majority of the transitions presented in this section are defined on a machine of type $\text{BQSM} = (\mathcal{C}, S, Q, D, N)$ as was introduced in appendix A.2.1 on page 78. As discussed in that section, labelling only affects the transition of unitary transforms. All other instruction transitions ignore it, giving us:

$$\text{Ins}(L(Q)) = L(\text{Ins}(Q))$$

where Ins is the transition of some other instruction.

The transition for the application of transformations will use LBQSM, while the transition for the add/remove control instructions uses the machine state of CQSM. 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 appendix A.2 on page 77.

A.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 A.5.

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.

$$
\begin{aligned}
&(\text{QLoad } x\ |k\rangle : \mathcal{C}, S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[|k\rangle \to Q], D, N) \\
&(\text{QCons } x\ c{:}\mathcal{C}, S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[c\{\} \to Q], D, N) \\
&(\text{QMove } x{:}\mathcal{C}, v{:}S, Q, D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[\bar{v} \to Q], D, N) \\
&(\text{QBind } z_0{:}\mathcal{C}, S, x{:}[c\{z'_1, \ldots, z'_n\} \to Q], D, N) \\
&\qquad\qquad \Longrightarrow\ (\mathcal{C}, S, x{:}[c\{z(N), z'_1, \ldots, z'_n\} \to Q[z(N)/z_0]], D, N')
\end{aligned}
$$

Figure A.5: Transitions for node construction

A.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 A.6 and figure A.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.

$$(\text{QDelete:}\mathcal{C}, S, Q\text{:}[|k_{ij}\rangle \to Q_{ij}], D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, (Q_{00} + Q_{11}), D, N)$$

$$(\text{QDelete:}\mathcal{C}, S, DT\text{:}[c_i\{b_{ij}\} \to Q_i], D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, \textstyle\sum_i(\mathtt{del}(\{b_{ij}\}, Q_i)), D, N)$$

$$(\text{QDelete:}\mathcal{C}, S, I\text{:}[\overline{v}_i \to Q_i], D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, \textstyle\sum_i Q_i, D, N)$$

Figure A.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 subroutines 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 A.6 will recursively rotate up and then delete the bound nodes of a datatype node.

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

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

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

$$(\text{QUnbind } y\text{:}\mathcal{C}, S, x\text{:}[c\{z'_1, \ldots, z'_n\} \to Q], D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, x\text{:}[c\{z'_2, \ldots, z'_n\} \to Q[y/z'_1]], D, N)$$

Figure A.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 $x\text{:}[c\{z'_2, \ldots, z'_n\} \to Q'']$.

A.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 A.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.

$$(\text{QPullup } x{:}\mathcal{C}, S, Q, D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, \mathsf{pull}(x, Q), D, N)$$

$$(\text{QName } x\ y{:}\mathcal{C}, S, Q, D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, Q[y/x], D, N)$$

Figure A.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.

A.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 A.9 on the next 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` $(c_1, \mathtt{lbl}_1), \ldots, (c_n, \mathtt{lbl}_n)$ — uses the datatype node at the top of the stack and execute a jump to the code at $\mathtt{lbl}_i$ 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` $\mathtt{Lbl}_{00}$ $\mathtt{Lbl}_{11}$ — 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.

$$(\text{Use } \triangleright\mathcal{C}_U{:}\mathcal{C}, S, x{:}[\bar{v_i} \to Q_i], D, N) \\ \quad \Longrightarrow\ (\text{EndQC}, S, 0, \texttt{Qc}(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)], \triangleright\mathcal{C}, 0){:}D, N)$$

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

$$(\text{EndQC}, S', Q, \texttt{Qc}(S, [], \triangleright\mathcal{C}, Q'){:}D, N) \\ \quad \Longrightarrow\ (\mathcal{C}, S, Q + Q', D, N)$$

$$(\text{Split } [(c_i, \triangleright\mathcal{C}_i)]{:}\mathcal{C}, S, x{:}[c_i\{V_i\} \to Q_i], D, N) \\ \quad \Longrightarrow\ (\text{EndQC}, S, 0, \texttt{Qc}(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)], \triangleright\mathcal{C}, 0){:}D, N)$$

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

$$(\text{EndQC}, S', Q, \texttt{Qc}(S, [], \triangleright\mathcal{C}, Q'){:}D, N) \\ \quad \Longrightarrow\ (\mathcal{C}, S, Q + Q', D, N)$$

$$(\text{Meas } \triangleright\mathcal{C}_0\ \triangleright\mathcal{C}_1{:}\mathcal{C}, S, x{:}[|0\rangle \to Q_0, |1\rangle \to Q_1, \ldots], D, N) \\ \quad \Longrightarrow\ (\text{EndQC}, S, 0, \texttt{Qc}(S, [(x_k{:}|k\rangle \to Q_k, \triangleright\mathcal{C}_k)]_{k\in\{0,1\}}, \triangleright\mathcal{C}, 0){:}D, N)$$

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

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

$$(\text{EndQC}, S', Q, \texttt{Qc}(S, [], \triangleright\mathcal{C}, Q'){:}D, N) \\ \quad \Longrightarrow\ (\mathcal{C}, S, Q + Q', D, N)$$

Figure A.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` $[(c1, l1), (c2, l2)]$ begins with the creation of a dump entry holding $[(c1 \rightarrow Q_1, l1), (c2 \rightarrow Q_2, l2)]$ 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 $(c1 \rightarrow Q_1, l1)$ 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 $l1$.

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 P_q $(= c2 \rightarrow Q_2)$ and code pointer $l2$ is removed from the execution list. The current quantum stack is set to P_q and the code pointer is set to $l2$. 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 `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 the instructions `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.

A.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 A.10. 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`.

$$(\text{Jump} \triangleright \mathcal{C}_J{:}\mathcal{C}, S, Q, D, N) \Longrightarrow (\mathcal{C}_J, S, Q, D, N)$$

$$(\text{CondJump} \triangleright \mathcal{C}_J{:}\mathcal{C}, \texttt{False}{:}S, Q, D, N) \Longrightarrow (\mathcal{C}_J, S, Q, D, N)$$

$$(\text{CondJump} \triangleright \mathcal{C}_J{:}\mathcal{C}, \texttt{True}{:}S, Q, D, N) \Longrightarrow (\mathcal{C}, S, Q, D, N)$$

$$(\text{NoOp}{:}\mathcal{C}, S, Q, D, N) \Longrightarrow (\mathcal{C}, S, Q, D, N)$$

Figure A.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`.

A.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 A.11. 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.

$$(\text{CPop}:\mathcal{C}, v{:}S, Q, D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, Q, D, N)$$

$$(\text{CGet } n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) \quad \Longrightarrow \quad (c, v_n{:}v_1{:}\cdots{:}v_n{:}S, Q, D, N)$$

$$(\text{CPut } n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) \quad \Longrightarrow \quad (c, v_1{:}\cdots{:}v_1{:}S, Q, D, N)$$

$$(\text{CApply } \mathsf{op}_n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D, N) \quad \Longrightarrow \quad (\mathcal{C}, \mathsf{op}_n(v_1, \ldots, v_n){:}S, Q, D, N)$$

$$(\text{CLoad } n :\mathcal{C}, S, Q, D, N) \quad \Longrightarrow \quad (\mathcal{C}, n{:}S, Q, D, N)$$

Figure A.11: Transitions for classical stack operations.

A.4.8 Unitary transformations and quantum control

The QS-Machine has three instructions which add or remove qubits (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 A.12.

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.

$$(\text{AddCtrl}:\mathcal{C}, C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})] \quad \Longrightarrow \quad (\mathcal{C}, \mathsf{id}{:}C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})])$$

$$(\text{QCtrl}:\mathcal{C}, f{:}C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})] \quad \Longrightarrow \quad (\mathcal{C}, (g \circ f){:}C, [(S'_j, L(Q_j)', D'_j)]_{j=1,\cdots m})])$$

$$(\text{UnCtrl}:\mathcal{C}, f{:}C, [(S_i, L(Q_i), D_i, N_i)]_{i=1,\cdots n})] \quad \Longrightarrow \quad (\mathcal{C}, C, [(S''_j, L(Q_j)'', D''_j)]_{j=1,\cdots p})])$$

$$(\text{QApply } m\ t{:}\mathcal{C}, (v_1{:}\cdots{:}v_m{:}S), L(Q), D, N) \quad \Longrightarrow \quad (\mathcal{C}, S, \mathsf{cTrans}([v_1, \ldots, v_m], t, L(Q)), D, N)$$

Figure A.12: Transitions for unitary transforms

The function 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 qubits to be the top nodes. For example, a Hadamard expects only 1, a *swap* expects 2 and an UM will expect as many qubits as N requires bits.

When a transform is applied to a datatype node the machine will attempt to rotate up the required number of qubits to the top, perform the operation and then re-rotate the datatype node back to the top.

The first step is to rotate the bound nodes of the 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. The machine considers a rotation for a transform to be indeterminate whenever the subject datatype node has more than one sub-stack. For example, this means a transform can not be applied to an Either that is a mix of Left(|0>) and Right(|1>).

When the rotation of the qubits succeeds the function will transform the top parts of the stack into a matrix Q of appropriate size (2×2 for a $1-$qubit transform, 16×16 for a $4-$qubit and so forth) with entries in the matrix being the sub-stacks of the qubits used in the transform.

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:

$$\text{cTrans } T\ L(Q) = \begin{cases} L(Q) & L = \texttt{IdOnly} \\ L(TQ) & L = \texttt{Left} \\ L(QT^*) & L = \texttt{Right} \\ L(TQT^*) & L = \texttt{Full} \end{cases}$$

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

A.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:

$$\begin{aligned} B_{QSM} &= (\mathcal{C}, S, Q, D, N) \\ C_{QSM} &= (\mathcal{C}, C, [(S, L(Q), D, N)]) \\ QSM &= \text{Inflist}(\mathcal{C}, C, [(S, L(Q), D, N)]) \end{aligned}$$

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.

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

Figure A.13: Transitions for function calls.

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

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

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*.

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.