Term Rewriting and Simulation of the Measurement Calculus for One-way Quantum Computing

James Lawson

First Marker:
Herbert Wiklicky

Second Marker:
Alessio R. Lomuscio

IMPERIAL COLLEGE LONDON
Department of Computing

Abstract

Among the models of quantum computation, the One-way Quantum Computer [10, 11] is one of the most promising proposals of physical realization [12]. The Measurement Calculus was developed to formalize one-way quantum computing by introducing the notion of a pattern - a strict sequence of instructions - and gives their operational semantics. We have developed several tools aimed at researchers to explore and discover patterns in the Measurement Calculus. These include tools to perform term rewriting and simulate the execution of patterns.

Acknowledgements

I would like to thank my supervisor Herbert Wiklicky for helping me choose this project and for giving me guidance and inspiration. I would like to thank my parents, whose care and support has got me this far, and finally my friends who have made the past four years at university enjoyable.

Contents

1 Introduction

This project focuses on an area of Quantum Computing [1] known as the *Measurement Calculus* [26]. The Measurement Calculus gives a formal specification for executing instructions on a quantum computer [2]. The Measurement Calculus describes what are called *patterns*. Figure 1.1 has an example of a (simplified) pattern.

$$\mathrm{X}_2 M_1^0 E_{12} N_2$$

Figure 1.1: An example of a *pattern*. Patterns give a formal symbolic representation of quantum computation.

A pattern can be thought of a sequence of instructions. Each letter represents an instruction. In general, instructions are executed one-by-one, from right-to-left. In Figure 1.1, $XMEN$, tells us to first do N (a preparatio*n*), then M (a *m*easurement), then E (an *e*ntanglement), followed by X (a X-correction). We can compare the idea of these patterns on a quantum computer to the assembly code of a classical computer. Instead of add, move, store,... instructions acting on *bits*, we look at so-called *prepare, measure, correct,...* instructions acting on *qubits* [3].

The Measurement Calculus describes the syntax and semantics of patterns and investigates patterns that perform *equivalent computations.* In assembly, we could exchange the order of certain instructions and still have the same result. Likewise, in the Measurement Calculus, exchanging the order of the instructions in the sequence can give the same result. For example the two patterns shown in Figure 1.2 can be shown to give equivalent computations.

What is important here is that a *proof* can be given that $\mathcal{P}_1$ is equivalent to $\mathcal{P}_2$ via equivalence rules. Figure 1.2 (b) shows this proof. In general, we are interested in showing that some pattern $\mathcal{P}_1$ is equivalent to another pattern $\mathcal{P}_2$ by applying these rules. Section 3.2.1 details exactly how these rules work, but the here point is, these proofs, and patterns in general, are *difficult to work with.*

Unfortunately, the simplest of quantum algorithms have patterns with instructions running into the *hundreds.* Figure 1.3 illustrates how this becomes a problem. Patterns can easily grow and their complexity explodes. At the same time, equivalence proofs also become complex and generally require more steps. It is currently too difficult for a researcher of the subject to explore new patterns and reason about them because of how easily their size explode.

[1]The fundamentals of quantum computing are explained in Section 2.2.

[2]Specifically, we are looking at executing instructions using the *measurement-based* model of quantum computation (one-way quantum computing) as described in Section 3.1.1.

[3]A *qubit* is a quantum bit, see Section 2.2.2.

$$\begin{aligned}
\mathcal{P}_1 &: X_4^{s_3} M_3^0 E_{34} E_{13} X_3^{s_2} M_2^0 E_{23} \\
\mathcal{P}_2 &: X_4^{s_3} Z_4^{s_2} Z_1^{s_2} M_3^0 M_2^0 E_{12} E_{23} E_{34}
\end{aligned}$$

(a)

$$\begin{aligned}
\mathcal{P}_1 &= X_4^{s_3} M_3^0 E_{34} E_{13} X_3^{s_2} M_2^0 E_{23} \\
&\equiv_{EX} Z_4^{s_3} Z_1^{s_2} M_3^0 E_{34} X_3^{s_2} M_2^0 E_{13} E_{23} \\
&\equiv_{EX} Z_4^{s_3} Z_4^{s_2} Z_1^{s_2} M_3^0 E_{34} X_3^{s_2} M_2^0 E_{13} E_{23} E_{34} \\
&\equiv_{MX} X_4^{s_3} Z_4^{s_2} Z_1^{s_2} M_3^0 M_2^0 E_{13} E_{23} E_{34} = \mathcal{P}_2
\end{aligned}$$

(b)

Figure 1.2: (a) Two patterns may give *equivalent computations.* We claim $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent.* (b) A *proof* that applies equivalence rules ($\equiv_{EX}, \equiv_{MX}$) can show that $\mathcal{P}_1$ is equivalent to $\mathcal{P}_2$.

$$\begin{aligned}
& X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} [M_3^0]^{s_2} M_2^0 E_{34} E_{23} X_2^{s_1} M_1^0 E_{12} \\
\equiv_{EX}\; & X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} [M_3^0]^{s_2} M_2^0 E_{34} X_2^{s_1} Z_3^{s_1} E_{23} M_1^0 E_{12} \\
\equiv_{FC}\; & X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} [M_3^0]^{s_2} M_2^0 X_2^{s_1} E_{34} Z_3^{s_1} M_1^0 E_{23} E_{12} \\
\equiv_{EZ}\; & X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} [M_3^0]^{s_2} M_2^0 X_2^{s_1} Z_3^{s_1} E_{34} M_1^0 E_{23} E_{12} \\
\equiv_{FC}\; & X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} [M_3^0]^{s_2} Z_3^{s_1} M_2^0 X_2^{s_1} M_1^0 E_{34} E_{23} E_{12} \\
\equiv_{MX}\; & X_5^{s_4} M_4^0 E_{45} X_4^{s_3} Z_4^{s_2} [M_3^0]^{s_2} Z_3^{s_1} [M_2^0]^{s_1} M_1^0 E_{34} E_{23} E_{12} \\
& \vdots
\end{aligned}$$

Figure 1.3: Patterns that are meaningful tend to be *long.* Reasoning about them becomes complex and equivalence proofs, like the one above, become tedious to hand-write.

This project creates a set of tools in Javascript to overcome the complexity in dealing with patterns. These tools are aimed at researcher in the field of Quantum Computer. They aim to make it easier for researchers to investigate and experiment current patterns and make it easier to explore new patterns.

The main tool is a proof assistant to aid in the term rewriting of patterns. [4]. This proof-assistant aids in writing equivalence proofs and would:

- let the user input formulas $\mathcal{P}_1$ and $\mathcal{P}_2$, then the user can choose from preset equivalence rules (such as $\equiv_{EX}, \equiv_{MX}$) and attempt to derive a proof starting at $\mathcal{P}_1$ and ending with $\mathcal{P}_2$.
- improve the clarity and presentation of a proof. The tool highlights (provide colour-highlighting, clean math typesetting, collapsing lines ...)
- significantly reduce human error caused by writing out long lines. The user has confidence in the proof's correctness. Once a proof has been entered, it is never wrong.

[4] An example of a proof-assistant is *Pandora* [1], a proof-assistant for natural deduction.

In addition to the proof-assistant, the project makes further contributions to Measurement Calculus by providing a variety of other useful tools/features. These include:

- an *interactive editor* for patterns. We develop the first tool that can both render patterns on screen and allow the user to create and edit patterns via a user interface. The editor supports editing sequences of commands and the editing of pattern compositions.
- checking the *validity* of a pattern. Not all patterns are valid. An arbitrary sequence of instructions may not describe a valid computation. In Section 3.1.5 we give code that checks that patterns are *valid*. This is the first tool that can perform this check.
- *simulating* the execution of a general pattern and provide the user with the resulting quantum state. We give the operational semantics of pattern execution and our implementation in Section 3.1.6. Although simulation has been attempted before [36], no tool has provided a graphical user interface with novel features to help visualize computation (such as showing an entanglement graph or showing the computation tree along with the branch that was taken).
- finding the *standard form* of a pattern. It can be shown that every valid patten has a special form called the standard form. Our tool gives the first implementation of an algorithm for finding the standard form of a pattern (Section 3.2.3).
- drawing visual aids called *one-way computing diagrams* (also called *open graphs* or *entanglement graphs*). (c.f. 24). Furthermore we give the first implementation of algorithm for finding the causal flow and general flow of an open graph in Sections 3.4.1 and 3.4.2.

2 Background

2.1 Complex Vector Spaces

We first introduce the mathematics required to understand Quantum Computing. The mathematics is that of linear algebra. In particular, vectors and matrices that are over the field of complex numbers. We then introduce the definition of a qubit and see how we perform quantum computation via the use of unitary matrices and measurement operators.

2.1.1 Hilbert Spaces

We first introduce two basic operations on vectors, with the assumption that our vectors have complex numbers in their entries. These two operations are the *complex conjugate* and the *adjoint*.

Definition 1 (Conjugate of a Vector)**:** *The complex conjugate of a column vector* $\mathbf{x} = (x_1, ..., x_n)$ *is defined by* $\mathbf{x}^* = (\mathbf{x}_1^*, ..., \mathbf{x}_n^*)$*, that is, the vector whose components have all been complex conjugated.*

Definition 2 (Adjoint of a Vector)**:** *The adjoint of a column vector* $\mathbf{x} = (x_1, ..., x_n)$*, written* $\mathbf{x}^\dagger$*, is complex conjugate of the corresponding row vector of* $\mathbf{x}$*,* $[x_1^* \, ... \, x_n^*]$*.*

Note that we can express the adjoint of $\mathbf{x}$ as $\mathbf{x}^\dagger = (\mathbf{x}^*)^\top$ or equivalently as $\mathbf{x}^\dagger = (\mathbf{x}^\top)^*$, but it is easier to write a dagger (†) as an abbreviation for taking conjugating and transposing a vector. This combined operation is sometimes called the *conjugate-transpose operation.*

Example:
Let $\mathbf{x} = (1+i, 2-3i, -2)$ and $\mathbf{y} = (4, -i, 1+\frac{i}{2})$. Then: the conjugates are: $\mathbf{x}^* = (1-i, 2+3i, -2)$ and $\mathbf{y}^* = (4, i, 1-\frac{i}{2})$. the adjoints are: $\mathbf{x}^\dagger = (1-i, 2+3i, -2)^T$ and $\mathbf{x}^\dagger = (4, i, 1-\frac{i}{2})^T$.

Definition 3 (Hermitian Product)**:** *The Hermitian Product (complex product) is a function* $\langle ., . \rangle : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}$ *defined by* $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\dagger \mathbf{y}$*.*

That is, the sum of the products of the corresponding components between the adjoint of $\mathbf{x}$ and $\mathbf{y}$. The Hermitian product can be expressed as $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i x_i^* y_i$.

Example:
Let $\mathbf{x} = (1-i,\ 2+3i,\ -i)$ and $\mathbf{y} = (1,\ 2,\ -i)$. Then $\mathbf{x}^\dagger = [1+i\ \ 2-3i\ \ i]$, and so the Hermitian Inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ is given by: $\mathbf{x}^\dagger \mathbf{y} = (1+i)\cdot 1 + (2-3i)\cdot 2 + (-i)\cdot i \cdot (-1) = 3 + -5i$

Definition 4 (Inner Product Space)**:** *An inner product space,* $((K, \cdot \cdot ., . + ., \mathcal{V}), \langle ., . \rangle)$ *is a vector space together with a function* $\langle ., . \rangle : \mathcal{V} \times \mathcal{V} \to \mathbb{C}$.

A finite complex vector space, together with the Hermitian product give an *Inner Product Space*. We shall therefore call the Hermitian Product, the Hermitian *Inner Product* (or complex *inner product*). A certain property [1] about $\mathbb{C}$ means that $(\mathbb{C}, \langle ., . \rangle)$ is a particular inner product space called a *Hilbert Space*.

1. $\langle \mathbf{x}, \mathbf{x} \rangle \geqslant 0$
2. $\langle \mathbf{x}, \mathbf{x} \rangle = 0$ iff $\mathbf{x} = \mathbf{0}$
3. $\langle \mathbf{x}, \alpha\mathbf{y} \rangle = \alpha \langle \mathbf{x}, \mathbf{y} \rangle$
4. $\langle \mathbf{z}, \mathbf{x} + \mathbf{y} \rangle = \langle \mathbf{z}, \mathbf{x} \rangle + \langle \mathbf{z}, \mathbf{y} \rangle$
5. $\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{y}, \mathbf{x} \rangle^*$

$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\dagger \mathbf{y}$

Figure 2.1: Properties of the Hermitian Inner Product

Note that this is similar to the Euclidean Inner Product for real vector spaces except that we take the complex conjugate of the left vector (giving a lack of symmetry).

	$\mathbb{R}^n$	$\mathbb{C}^n$
Inner Product:	*Euclidean*	*Hermitian*
	$\mathbf{x}^T\mathbf{x}$	$\mathbf{x}^\dagger\mathbf{x}$
Unit Length:	$\mathbf{x}^T\mathbf{x} = 1$	$\mathbf{x}^\dagger\mathbf{x} = 1$

Table 2.1: Correspondences between real vectors and complex vectors.

2.1.2 Complex Matrices

The operations defined on vectors can be extended to *matrices*.

[1] $\mathbb{C}$ is a *complete metric space*

Definition 5 (Complex Conjugate of a Matrix)**:** *The complex conjugate of matrix* $\mathbf{M}$*, denoted* $\mathbf{M}^*$*, is defined by* $\mathbf{M}^* = m_{ij}^*$*, that is, the matrix whose components have all been complex conjugated.*

Definition 6 (Adjoint of a Matrix)**:** *The adjoint of* $\mathbf{M}$*, denoted* $\mathbf{M}^\dagger$*, is complex conjugate of the transpose of* $\mathbf{M}$ *given by:* $\mathbf{M}^\dagger = m_{ji}^*$.

Like the adjoint of a vector, the adjoint of a matrix can be written as $(\mathbf{x}^*)^T$ or equivalently as $(\mathbf{x}^T)^*$, but it is easier to write a dagger (†) to denote but it is easier to write a dagger to denote this conjugate transpose operation.

Example:

$$\mathbf{A} = \begin{bmatrix} 1 & 1+2i \\ 2-4i & i \end{bmatrix} \quad \mathbf{A}^\dagger = \begin{bmatrix} 1 & 2+4i \\ 1-2i & i \end{bmatrix}$$

Definition 7 (Hermitian Matrix)**:** *A Hermitian Matrix is any matrix* $\mathbf{M} \in \mathbb{C}^{m\times n}$ *that satisfies* $\mathbf{M} = \mathbf{M}^\dagger$*.*

Example:
The matrices $\mathbf{A}$ and $\mathbf{B}$ are Hermitian. To see why $\mathbf{A}$ is Hermitian, if we take the complex conjugate of $\mathbf{A}$, and then take the transpose, we see that $\mathbf{A}^\dagger = \mathbf{A}$. The same reasoning can be applied to $\mathbf{B}$.

$$\mathbf{A} = \begin{bmatrix} 1 & 1+i \\ 1-i & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 2 & -i & 2i \\ i & 3 & -1-i \\ -2i & -1+i & i \end{bmatrix}$$

Let $\mathbf{M}$ be a square matrix. Then the product $\mathbf{M}^\dagger\mathbf{M}$ computes n^2 different Hermitian inner products involving the n columns of $\mathbf{M}$. That is to say, the (i, j)th element of $\mathbf{M}^\dagger\mathbf{M}$ is $\langle \mathbf{u}_i, \mathbf{u}_j \rangle$, where $\mathbf{u}_i$ denotes the ith column of $\mathbf{M}$. One particular matrix of interest is the *Unitary Matrix.*

Definition 8 (Unitary Matrix)**:** *A matrix* $\mathbf{M} \in \mathbb{C}^{n\times n}$ *is unitary if satisfies* $\mathbf{M}^\dagger\mathbf{M} = \mathbf{I}$.

The equation $\mathbf{U}^\dagger\mathbf{U} = \mathbf{I}$ expresses that out of all the n^2 inner products involving the columns of $\mathbf{U}$, $\langle \mathbf{u}_i, \mathbf{u}_j \rangle$, $i, j \in \{1, ..., n\}$, only distinct columns have a zero inner product, and the inner product of the same column with itself is 1. In other words,

1. the columns are pairwise *orthogonal* ($\langle \mathbf{u}_i, \mathbf{u}_j \rangle = 0$ when $i \neq j$)
2. each column has *unit length* ($\langle \mathbf{u}_i, \mathbf{u}_j \rangle = 1$ when $i = j$)

This condition can be expressed with the *Kronecker Delta*:

$$\langle \mathbf{u}_i, \mathbf{u}_j \rangle = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Example:
The matrices **U**, **V**, and **W** are unitary. Hermitian product of any column with another column is zero. The Hermitian product of a given column with itself is 1.

$$\mathbf{U} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \quad \mathbf{V} = \begin{bmatrix} i & 0 \\ 0 & i \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} i & 0 & 0 \\ 0 & i & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hermitian matrices can be through of as the complex extension of real symmetric matrices. That is, every real symmetric matrix in $\mathbb{R}^{m \times m}$ is a Hermitian matrix in $\mathbb{C}^{m \times m}$. And similarly, unitary matrices can be through of as a complex extension of real orthogonal matrices. Every real orthogonal matrix in $\mathbb{R}^{m \times m}$ is a unitary matrix in $\mathbb{C}^{m \times m}$. Table 2.2 gives a comparison that shows the correspondence between real vector spaces and complex vector spaces. The next table (Table 2.3) gives properties about eigenvalues/eigenvectors:

	$\mathbb{R}^n$	$\mathbb{C}^n$
Symmetry:	*Symmetric* $\mathbf{A}^T = \mathbf{A}$	*Hermitian* $\mathbf{A}^\dagger = \mathbf{A}$
Preserve Inner Product:	*Orthogonal* $\mathbf{Q}^T = \mathbf{Q}^{-1}$	*Unitary* $\mathbf{U}^\dagger = \mathbf{U}^{-1}$

Table 2.2: Correspondences between real matrices and complex matrices.

	$\mathbb{R}^n$	$\mathbb{C}^n$
Symmetry:	Real λ	Real λ
Preserve Inner Product:	$\lvert\lambda\rvert = 1$	$\lvert\lambda\rvert = 1$

Table 2.3: Properties of Eigenvalues.

To implement complex vectors and matrices, we define a Javascript object called `Complex` (see Listings 2.1). This will allow us to store complex numbers and more importantly, vectors and matrices of complex numbers. Although there are existing Javascript libraries that support matrices [9, 10] we implemented our own library as we found the existing libraries to be inflexible and too large for our needs.

Our `Complex` constructor function has functions that implement the basic operations we can perform on complex numbers such as finding: $k\mathbf{x}$, $\mathbf{x}^*$, $|\mathbf{x}|^2$, as well as addition and multiplication. Here, `this.real` and `this.imag` are Javascript `Numbers` which are double-precision 64-bit binary format IEEE 754 values [29].

```
function Complex(re, im) {
  this.real = re;
  this.imag = im;
  // alpha := newc
  this.assignTo = function(newc) {...};
  // alpha := newre + i*newim
  this.assign = function(newre, newim) {...};
  // alpha := k.alpha
  this.scale = function(k) {...};
  // alpha := alpha*
  this.conjugate = function() {...};
  // return = alpha + beta
  this.addTo = function(beta) {...};
  // alpha := alpha + beta
  this.addWith = function(beta) {...};
  // return = alpha * beta
  this.mult = function(beta) {...};
  // alpha = alpha * beta
  this.multWith = function(beta) {...};
  // return = |alpha|^2
  this.lengthSquared = function() {...};
  // return = a clone of alpha
  this.clone = function() {...}
}
```

Listing 2.1: implementation of complex numbers

Then a vector $\in \mathbb{C}^2$ is represented by a Javascript array of `Complex` objects and a matrix is represented by an array of arrays, where each array has equal length and each entry contains a `Complex` object. We define a helper `f` that returns commonly used complex numbers.

Example:
The identity matrix is represented by `[[f.one(),f.zero()],[f.zero(),f.one()]]`

Our implementation includes functions to perform the basic operations on vectors and matrices, namely *scaling*: $k\mathbf{v}$, $k\mathbf{M}$, *addition*: $\mathbf{v}+\mathbf{u}$, $\mathbf{M}+\mathbf{N}$ and *multiplication*: $\mathbf{Mv}$, $\mathbf{MN}$. These are shown in Listings 2.2. Our implementation also uses `matrixClone` that performs a deep clone of a matrix; cloning all the array objects and complex objects for a matrix.

```
// return = ||v||^2 = <v,v>
function vectorLengthSquared(v) {...}
// return = a deep clone of m
function matrixClone(m) {...}
// return = m1 * m2
function matrixMultiply(m1, m2) {...}
// return = m * v
function matrixVectorMultiply(m, v) {...}
// m1 := m1 + m2
function matrixAddWith(m1, m2) {...}
// m := k.m
function scaleMatrix(k, m) {...}
// v := k.v
function scaleVector(k, m) {...}
```

Listing 2.2: functions that implement operations involving vectors and matrices

2.1.3 Tensor Product

We now introduce the tensor product. We will see later in Section 2.2 that the tensor product is one of the most important operations in Quantum Computing. Given two vectors, $\mathbf{x} \in \mathbb{C}^k$ and $\mathbf{y} \in \mathbb{C}^l$, we define the *tensor product of two vectors* by the column vector:

$$\mathbf{x} \otimes \mathbf{y} = \begin{bmatrix} x_1\mathbf{y} \\ x_2\mathbf{y} \\ x_3\mathbf{y} \\ \vdots \\ x_n\mathbf{y} \end{bmatrix}$$

Here, $x_j\mathbf{y}$ denotes the column vector given by the scalar multiplication of complex number x_j with vector $\mathbf{y}$. The result of the tensor product is a vector $\mathbf{x} \otimes \mathbf{y} \in \mathbb{C}^{k \cdot l}$.

Example:

$$\begin{bmatrix} 1+4i \\ 2 \end{bmatrix} \otimes \begin{bmatrix} 3i \\ 5-i \end{bmatrix} = \begin{bmatrix} 1+4i \begin{bmatrix} 3i \\ 5-i \end{bmatrix} \\ 2 \begin{bmatrix} 3i \\ 5-i \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 3i-12 \\ 9+19i \\ 6i \\ 10-2i \end{bmatrix}$$

Example:

$(1,0,i,-1) \otimes (1,2,3,4) = (1,2,3,4,0,0,0,0,i,2i,3i,4i,-1,-2,-3,-4)$
$(x,y,z) \otimes (a,b,c,d) = (xa,xb,xc,xd,ya,yb,yc,yd,za,zb,zc,zd)$

The tensor product of vectors can be extended to the *tensor product of matrices*. Given two square matrices, $\mathbf{M} \in \mathbb{C}^{k \times k}$ and $\mathbf{N} \in \mathbb{C}^{l \times l}$, we define the *matrix tensor product* $\mathbf{M} \otimes \mathbf{N}$ by:

$$\mathbf{M} \otimes \mathbf{N} = \begin{bmatrix} m_{11}\mathbf{N} & m_{12}\mathbf{N} & \dots & m_{1k}\mathbf{N} \\ m_{12}\mathbf{N} & m_{22}\mathbf{N} & \dots & m_{2k}\mathbf{N} \\ \vdots & \vdots & \dots & \vdots \\ m_{k1}\mathbf{N} & m_{k2}\mathbf{N} & \dots & m_{kk}\mathbf{N} \end{bmatrix}$$

The tensor product of $\mathbf{M} \otimes \mathbf{N}$ gives the matrix $\mathbf{M}$, except that now each element, m_{ij} becomes block matrix $m_{ij}\mathbf{N}$ of a larger matrix.

Example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & -1 \\ -i & i \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 1 & -1 \\ -i & i \end{bmatrix} & 2 \begin{bmatrix} 1 & -1 \\ -i & i \end{bmatrix} \\ 3 \begin{bmatrix} 1 & -1 \\ -i & i \end{bmatrix} & 4 \begin{bmatrix} 1 & -1 \\ -i & i \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 & -1 & 2 & -2 \\ -i & i & -2i & i \\ 3 & -3 & 4 & -4 \\ -3i & 3i & -4i & 4i \end{bmatrix}$$

Example:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix} = \begin{bmatrix} a\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix} & b\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix} \\ c\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix} & d\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix} \end{bmatrix} = \begin{bmatrix} au & av & aw & bu & bv & bw \\ ax & ay & az & bx & by & bz \\ cu & cv & cw & du & dv & dw \\ cx & cy & cz & dx & dy & dz \end{bmatrix}$$

Figure 2.4 gives a list of identities involving the tensor product of vectors and matrices. The first ten identities can be proven directly using the definition. The eleventh can be proven using the definition of the Hermitian product along with identity 10.

1.	$(a\mathbf{a} + b\mathbf{b}) \otimes (c\mathbf{c} + d\mathbf{d})$	$= ac(\mathbf{a} \otimes \mathbf{c}) + ad(\mathbf{a} \otimes \mathbf{d})$ $+bc(\mathbf{b} \otimes \mathbf{c}) + bd(\mathbf{b} \otimes \mathbf{d})$
2.	$(\mathbf{A} \otimes \mathbf{B})(\mathbf{a} \otimes \mathbf{b})$	$= \mathbf{Aa} \otimes \mathbf{Bb}$
3.	$(\mathbf{a} \otimes \mathbf{b})^*$	$= \mathbf{a}^* \otimes \mathbf{b}^*$
4.	$(\mathbf{a} \otimes \mathbf{b})^T$	$= \mathbf{a}^T \otimes \mathbf{b}^T$
5.	$(\mathbf{a} \otimes \mathbf{b})^\dagger$	$= \mathbf{a}^\dagger \otimes \mathbf{b}^\dagger$
6.	$(a\mathbf{A} + b\mathbf{B}) \otimes (c\mathbf{C} + d\mathbf{D})$	$= ac(\mathbf{A} \otimes \mathbf{C}) + ad(\mathbf{A} \otimes \mathbf{D})$ $+bc(\mathbf{B} \otimes \mathbf{C}) + bd(\mathbf{B} \otimes \mathbf{D})$
7.	$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D})$	$= \mathbf{AC} \otimes \mathbf{BD}$
8.	$(\mathbf{A} \otimes \mathbf{B})^*$	$= \mathbf{A}^* \otimes \mathbf{B}^*$
9.	$(\mathbf{A} \otimes \mathbf{B})^T$	$= \mathbf{A}^T \otimes \mathbf{B}^T$
10.	$(\mathbf{A} \otimes \mathbf{B})^\dagger$	$= \mathbf{A}^\dagger \otimes \mathbf{B}^\dagger$
11.	$\langle \mathbf{a} \otimes \mathbf{b}, \mathbf{c} \otimes \mathbf{d} \rangle = \langle \mathbf{a}, \mathbf{c} \rangle \cdot \langle \mathbf{b}, \mathbf{d} \rangle$	

Table 2.4: Properties of the Tensor Product

Listings 2.3 shows the implementation of the vector tensor product. To use this function, we simply pass in two arrays of complex objects and the function will return a new array of new complex objects representing the vector tensor product.

```
// return = v1 tensor v2
function vectorTensor(v1, v2) {
    var r = [];
    for (var i = 0, l1 = v1.length; i < l1; i++) {
        for (var j = 0, l2 = v2.length; j < l2; j++) {
            r[i*l2 + j] = v1[i].mult(v2[j]);
        }
    }
    return r;
}
```

Listing 2.3: implementation of the vector tensor product $\mathbf{v}_1 \otimes \mathbf{v}_2$

Listings 2.4 shows the implementation of the matrix tensor product. Similar to `vectorTensor`, we input two two-dimensional arrays and the function returns a newly allocated two-dimensional array representing the matrix tensor product. Here we use `~~(a/b)` that computes the *integer division* of a and b (as apposed to *floating-point* division).

```
// return = m1 tensor m2
function matrixTensor(m1, m2) {
    var r = [];
    var r1 = m1.length, c1 = m1[0].length;
    var r2 = m2.length, c2 = m2[0].length;

    for (var i = 0, m = r1*r2; i < m; i++) {
        r[i] = [];
        for (var j = 0, n = c1*c2; j < n; j++) {
            r[i][j] = m1[~~(i/r2)][~~(j/c2)].mult(m2[i%r2][j%c2]);
        }
    }
    return r;
}
```

Listing 2.4: implementation of the matrix tensor product $\mathbf{M}_1 \otimes \mathbf{M}_2$

2.2 Quantum Computing

Now that we have the notation of a Hilbert Space, we can introduce the definitions of a qubit (Section 2.2.2), and look computation is perform in a very classical, general sense in Sections 2.2.3 and 2.2.4. But first, we set the scene for quantum computing in Section 2.2.1.

2.2.1 Introduction to Quantum Computing

The first two paragraphs of this introduction to quantum computing borrows excepts from [3]

In 1982, the Nobel prize-winning physicist Richard Feynman thought up the idea of a *quantum computer*, a computer that uses the effects of quantum mechanics to its advantage.

For some time, the notion of a quantum computer was primarily of theoretical interest only, but developments of certain quantum algorithms bought the idea to everybody's attention. One such quantum algorithm is *Shor's Algorithm*, an algorithm that gives the ability to factor large numbers on a quantum computer quickly. In fact it is fast enough to break current cryptography techniques in a matter of seconds. With the motivation provided by this algorithm, the topic of quantum computing has gathered momentum over, what is now, several decades, and researchers around the world are racing to be the first to create a practical quantum computer.

Yet currently there is no widely accepted physical implementation of a Quantum Computing. It is purely a theoretical machine (like the Turing Machine). Recently the invention of by company D-wave [11] claims to have a physical realization of a quantum computer. Verifying that the D-wave computer is indeed a quantum computer is non-trivial. The main issues are that the D-wave machine is a black box and that it may mix classical computing with quantum computing. It is unclear how much computation is actually quantum. As of April 2014, there has been an ongoing debate in literature concerning how quantum the D-wave computer is [12]. Some researchers such as are opposed to D-wave [13] whereas some researchers are in favor [14].

Our project is not concerned with the low-level details of developing quantum computers. Our project focuses on developing tools to progress the theoretical side of quantum computing. We continue to introduce our project when we look at the Measurement Calculus in Section 3.1.

2.2.2 Qubits and Quantum Registers

Now that we have a definition of a Complex Vector Space, we can now define the fundamental unit for quantum computing, the *qubit*. A classical bit has a state (either 0 or 1). Similarly a qubit has a state and can be either $(1, 0)$, or $(0, 1)$, or some linear combination: $\alpha_1(1, 0) + \alpha_2(0, 1)$ where $|\alpha_1|^2 + |\alpha_2|^2 = 1$.

Definition 9 (Qubit [2])**:** *A qubit (quantum bit) is a complex vector* $\mathbf{x} = (\alpha_1, \alpha_2) \in \mathbb{C}^2$ *where* $|\alpha_1|^2 + |\alpha_2|^2 = 1$*.*

Although a qubit is an abstract *mathematical object*, it very much describes the behavior of real physical Quantum Mechanical systems. More precisely, two-level systems, that possess a state that is a *quantum superposition*, possessing simultaneously a state that is a mixture of two physically distinguishable states.

Example:
$\mathbf{x} = \frac{1}{\sqrt{2}}(1, i)$ is a qubit. To see why, we show that $|\alpha|^2 = |1/\sqrt{2}|^2 = 1/2$, $|\beta|^2 = |i/\sqrt{2}|^2 = 1/2$. Hence $|\alpha_1|^2 + |\alpha_2|^2 = 1$.

Any normalized vector in $\mathbb{C}^2$, is by definition, is a qubit. Every qubit $\mathbf{x}$ has unit length meaning that the Hermitian product of $\mathbf{x}$ with itself is 1: $\langle \mathbf{x}, \mathbf{x} \rangle = 1$. We associate linear combinations with the linear superposition phenomena. The basis vectors $(1, 0)$ and $(0, 1)$ (or any two basis vectors) can represent two physically distinguishable systems. For example:

- whether spin of an electron is up or down
- whether a photon has been polarized left or right,
- whether an electron of the hydrogen atom in the ground state or in the first excited state

Once we have fixed our basis vectors to two physically distinguishable pure states, a *superposition* of those states is represented by the linear combination $\alpha_1(1, 0) + \alpha_2(0, 1)$. There are infinitely many choices for α_1 and α_2. Compared to a classical bit that has only two states, a quantum bit has infinitely many. Because of superposition, we can store more information in a qubit giving rise to a more powerful model of computation.

Example:
All of the vectors $\in \mathbb{C}^2$ shown are qubits.

$$\begin{array}{llll} (1,0) & (0,1) & (\frac{3}{5}, \frac{4i}{5}) & \frac{1}{\sqrt{2}}(i, -i) \\ \frac{1}{\sqrt{3}}(1-i, i) & \frac{1}{2}(1+i, 1-i) & & \end{array}$$

For all of these vectors, $\mathbf{x} = (\alpha, \beta) \in \mathbb{C}^2$, it is indeed the case that $|\alpha_1|^2 + |\alpha_2|^2 = 1$ (or equivalently that $\langle \mathbf{x}, \mathbf{x} \rangle = 1$).

We now define following standard qubits that are commonly used in quantum computing literature and will be used throughout this report.

$$\begin{array}{ll} |0\rangle = (1,0) & |1\rangle = (0,1) \\ |+\rangle = (1/\sqrt{2})(1,1) & |-\rangle = (1/\sqrt{2})(1,-1) \\ |+_\alpha\rangle = (1/\sqrt{2})(1,e^{i\alpha}) & |-_\alpha\rangle = (1/\sqrt{2})(1,-e^{i\alpha}) \end{array}$$

where $\alpha \in (0, 2\pi]$. These vectors use the *Dirac Bra-ket Notation.* We will use the Dirac notation to write column vectors as $|\psi\rangle$ and the adjoint of a column vector as $\langle\psi|$. One can prove that $\{|0\rangle, |1\rangle\}$, $\{|+\rangle, |-\rangle\}$ and $\{|+_\alpha\rangle, |-_\alpha\rangle\}$ are of bases of the vector space $\mathbb{C}^2$.

We now generalise a qubit to a larger structure, the *quantum register.*

Definition 10 (Quantum Register [5])**:** *For $n > 0$, we define an n-qubit quantum register to be a normalized vector in $\mathbb{C}^{2^n}$. That is, a n-qubit register is a vector $\mathbf{x} = (\alpha_1, \alpha_2, ..., \alpha_{2^n}) \in \mathbb{C}^{2^n}$ where $|\alpha_1|^2 + |\alpha_2|^2 + ... + |\alpha_{2^n}|^2 = 1$.*

Example:
The vector $\mathbf{x} = (0,5,-0.5,0,5,-0.5)$ is an example of a 2-qubit register.
That is because: $|0.5|^2 + |-0.5|^2 + |0.5|^2 + |-0.5|^2 = 1$.

Now that we have introduced a qubit, and a quantum register, we now see that the tensor product relates the two. One property of the tensor product is that is preserves unit lengths. That is:

$$\langle \mathbf{x}, \mathbf{x} \rangle = 1 \text{ and } \langle \mathbf{y}, \mathbf{y} \rangle = 1 \text{ iff } \langle \mathbf{x} \otimes \mathbf{y}, \mathbf{x} \otimes \mathbf{y} \rangle = 1$$

This can be derived using identity 11 from Figure 2.4. This gives a useful result that the tensor product of any two qubits gives a two-qubit register.

Proposition 1: *For any $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{C}^2$, if $\mathbf{x}_1$ is a qubit and $\mathbf{x}_2$ is a qubit, then $\mathbf{x}_1 \otimes \mathbf{x}_2$ is a two-qubit quantum register.*

Proposition 2: *For any $\mathbf{x}_1 \in \mathbb{C}^{2^k}, \mathbf{x}_1 \in \mathbb{C}^{2^r}$, if $\mathbf{x}_1$ is a qubit k-qubit quantum register and $\mathbf{x}_2$ is an r-qubit, then $\mathbf{x}_1 \otimes \mathbf{x}_2$ is a $k + r$-qubit quantum register.*

A tensor product of qubits always gives a quantum register. Because of this, the tensor product is the most essential operation in quantum computing. It allows us to combine qubits and registers together to form larger registers. The tensor product acts as a glue for forming larger states for our computation.

Example:
Given qubits $\mathbf{x} = \frac{1}{\sqrt{2}}(1, i)$ and $\mathbf{y} = (\sqrt{3}/2, -1/2)$, we can use $\mathbf{x} \otimes \mathbf{y}$ to form a two-qubit register that is given by:

$$\mathbf{x} \otimes \mathbf{y} = \left(\frac{1}{\sqrt{2}} \left(\frac{\sqrt{3}}{2}, -\frac{1}{2} \right), \frac{i}{\sqrt{2}} \left(\frac{\sqrt{3}}{2}, -\frac{1}{2} \right) \right) = \left(\frac{\sqrt{3}}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{i\sqrt{3}}{2\sqrt{2}}, -\frac{i}{2\sqrt{2}} \right)$$

This is indeed a 2-qubit quantum register because $|\alpha_1|^2 + |\alpha_2|^2 + |\alpha_3|^2 + |\alpha_4|^2 = \frac{3}{8} + \frac{1}{8} + \frac{3}{8} + \frac{1}{8} = 1$, and so that is indeed a normalised vector in $\mathbb{C}^4$.

The previous example has shown that combining two qubits together gives us a 2-qubit register. Now let us use the more general result and combine this 2-qubit register with another qubit.

Example:
Let $\mathbf{z} = \frac{1}{\sqrt{2}}(1, i)$. Using the previous example, we form the tensor product $\mathbf{z} \otimes \mathbf{x} \otimes \mathbf{y}$ to find a *three*-qubit register that's given by:

$$\begin{aligned} \mathbf{z} \otimes \mathbf{x} \otimes \mathbf{y} &= \frac{1}{\sqrt{2}}(1, i) \otimes \left(\frac{\sqrt{3}}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{i\sqrt{3}}{2\sqrt{2}}, -\frac{i}{2\sqrt{2}} \right) \\ &= \left(\frac{1}{\sqrt{2}} \cdot \left(\frac{\sqrt{3}}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{i\sqrt{3}}{2\sqrt{2}}, -\frac{i}{2\sqrt{2}} \right), \frac{i}{\sqrt{2}} \cdot \left(\frac{\sqrt{3}}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{i\sqrt{3}}{2\sqrt{2}}, -\frac{i}{2\sqrt{2}} \right) \right) \\ &= \left(\frac{\sqrt{3}}{4}, -\frac{1}{4}, \frac{i\sqrt{3}}{4}, -\frac{i}{4}, \frac{i\sqrt{3}}{4}, -\frac{i}{4}, \frac{-\sqrt{3}}{4}, \frac{1}{4} \right) \end{aligned}$$

Notice that as we combine qubits together to form registers, the number of components in the vector increases *exponentially.* The vector for an n-qubit register has 2^n complex components.

2.2.3 Quantum Computation I: Unitary Matrices

We apply operations on quantum registers using unitary matrices. Quantum computation is described via matrix-vector multiplication involving an n-qubit register $\mathbf{x} \in \mathbb{C}^{2^n}$ and a unitary matrices of size $\mathbb{C}^{2^n \times 2^n}$. Applying a unitary matrix to a qubit gives another qubit. And more generally, applying a unitary matrix to a n-qubit quantum register gives another n-qubit quantum register.

Theorem 1: *For any unitary matrix* $\mathbf{U} \in \mathbb{C}^{2^n \times 2^n}$*, if* $\mathbf{x} \in \mathbb{C}^{2^n}$ *is an n-qubit quantum register then* $\mathbf{U}\mathbf{x} \in \mathbb{C}^{2^n}$ *is an n-qubit quantum register.*

Any unitary matrix can be used for computation. And so the computational effect is can be difficult to interpret - certainly when we are detailing with vectors of complex numbers. For now, we will look at the unitary matrices of Figure 2.2 whose computation effect is quite intuitive in the classical sense.

Example:
We can use the unitary matrices in Figure to perform computation on two-qubit registers. For example:
$\mathbf{M}_{not}(|1\rangle \otimes |0\rangle) = |0\rangle \otimes |1\rangle$
$\mathbf{M}_{not}(|0\rangle \otimes |0\rangle) = |1\rangle \otimes |1\rangle$
$\mathbf{M}_{swap}(|1\rangle \otimes |0\rangle) = |1\rangle \otimes |0\rangle$
$\mathbf{M}_{swap}(|0\rangle \otimes |1\rangle) = |1\rangle \otimes |0\rangle$
The $\mathbf{M}_{not}$ unitary matrix negates both the qubits in the register by mapping $|0\rangle \mapsto |1\rangle$ and $|1\rangle \mapsto |0\rangle$. The $\mathbf{M}_{swap}$ swaps the qubits in the register.

Example:
Here we look at the controlled-NOT or CNOT operation.
This operation takes the first qubit (the *control* qubit) and flips the second qubit (the *target* qubit) depending on whether the control is $|0\rangle$ or $|1\rangle$. $\mathbf{M}_{cnot}(|0\rangle \otimes |0\rangle) = |0\rangle \otimes |0\rangle$
$\mathbf{M}_{cnot}(|0\rangle \otimes |1\rangle) = |0\rangle \otimes |1\rangle$
$\mathbf{M}_{cnot}(|1\rangle \otimes |0\rangle) = |1\rangle \otimes |1\rangle$
$\mathbf{M}_{cnot}(|1\rangle \otimes |1\rangle) = |1\rangle \otimes |0\rangle$

Example:
Here we look at the CZ (controlled-Z) operation. This operation is like the controlled NOT, but we replace the NOT operation with the Pauli matrix $\mathbf{Z}$. It uses one qubit as a control and another as a target. Depending on whether the control is $|0\rangle$ or $|1\rangle$, we apply $\mathbf{Z}$ to the target.

$$\mathbf{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Example:
The CZ (controlled-Z) operation can be extended to work on a three-qubit register. Figure 2.3 gives examples of matrices $\mathbf{CZ}_{ij}$. Here i is the control qubit and j is the source qubit.
For example:
$\mathbf{CZ}_{13}(|1\rangle \otimes |\psi\rangle \otimes |1\rangle) = |1\rangle \otimes |\psi\rangle \otimes \mathbf{Z}\,|1\rangle = |1\rangle \otimes |\psi\rangle \otimes -|1\rangle$
$\mathbf{CZ}_{13}(|0\rangle \otimes |\psi\rangle \otimes |1\rangle) = |0\rangle \otimes |\psi\rangle \otimes |1\rangle$
$\mathbf{CZ}_{23}(|\psi\rangle \otimes |1\rangle \otimes |1\rangle) = |\psi\rangle \otimes |1\rangle \otimes \mathbf{Z}\,|1\rangle = |\psi\rangle \otimes |1\rangle \otimes -|1\rangle$
$\mathbf{CZ}_{23}(|\psi\rangle \otimes |0\rangle \otimes |1\rangle) = |\psi\rangle \otimes |0\rangle \otimes |1\rangle$

$$\mathbf{M}_{not} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{M}_{swap} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_{cnot} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 2.2: Quantum computation corresponds to applying unitary matrices to a qubit. The above unitary matrices perform simple operations on a two-qubit register.

$$\mathbf{CZ}_{13} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \quad \mathbf{CZ}_{12} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\mathbf{CZ}_{32} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \quad \mathbf{CZ}_{21} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

Figure 2.3: The controlled-Z operation, CZ, can be extended from 2-qubit registers to 3-qubit registers. The figure shows three examples. We use subscripts CZ_{ij} to show that i is the controller qubit and j is the target qubit. The remaining qubit in the register is not involved and is left unmodified by the operation.

Notice that $\mathbf{U}\,|\psi\rangle = \alpha_1\mathbf{u}_1 + \alpha_2\mathbf{u}_2 + ... + \alpha_n\mathbf{u}_n$. Since $\mathbf{U}$ is unitary, each column is a quantum register, and so they have a unit length: $\langle \mathbf{u}_1, \mathbf{u}_1\rangle = 1$, ..., $\langle \mathbf{u}_n, \mathbf{u}_n\rangle = 1$. Additionally, $|\psi\rangle$ is a quantum register, and so it too has a unit length: $|\alpha_1|^2 + |\alpha_2|^2 +$

$... + |\alpha_n|^2 = 1$. Hence $\mathbf{U}|\psi\rangle$ corresponds to taking a *normalized linear combination* of quantum registers. Any normalized linear combination of quantum registers is a quantum register.

Example:
Let us consider three quantum registers given by:
$\mathbf{x} = (0.5, 0, -0.5, 0.5, 0, 0, 0.5, 0)$, $\mathbf{y} = (0, 0, 1, 0, 0, 0, 0, 0, 0)$
and $\mathbf{z} = (0, i/2, 0, 0, 1/2, -i/2, 0, 1/2)$ be quantum registers. If we take a normalised linear combination of the three, using $\alpha_1 = \sqrt{3}/4$ $\alpha_2 = 3/4$ $\alpha_3 = 1/2$, we have: $(\frac{\sqrt{3}}{8}, \frac{i}{4}, \frac{-\sqrt{3}}{8} + \frac{3}{4}, \frac{\sqrt{3}}{8}, \frac{1}{4}, \frac{-i}{4}, \frac{\sqrt{3}}{8}, \frac{1}{4})$ which is indeed another quantum register.

2.2.4 Quantum Computation II: Measurements

Recall that a qubit (as a mathematical object) models a two-level quantum mechanical system. One interesting property about a quantum state is that we cannot read or *observe* a physical quantum system's state without affecting it. That is, trying to read the state will cause a write. This phenomena occurs is called the *observer effect.* In our more abstract setting, if we design an operation that reads the values of α and β, then we need to also make sure that the the vector itself is modified to match the behaviour of quantum mechanical systems. The most basic way to model this mathematically is to use the *von Neumann measurement scheme* which is also known as the *Basic Measurement Principle.*

Definition 11 (Basic Measurement Principle)**:** *When a qubit $\mathbf{x} = (\alpha_1, \alpha_2)$ is measured in the basis $\{|0\rangle, |1\rangle\}$, the result of the observation is either the state $|0\rangle$ with probability $|\alpha_1|^2$ or is the state $|1\rangle$ with probability $|\alpha_2|^2$. Furthermore, the act of measuring causes the qubit to collapse to the observed state.*

Example:
When we measure the qubit $\mathbf{x} = (1/\sqrt{2}, i/\sqrt{2})$, the probability of observing $|0\rangle$ is $1/2$ and the probability of observing $|1\rangle$ is $1/2$. Furthermore, if we observe $|0\rangle$, $\mathbf{x}$ collapses to state $|0\rangle$, and if we observe $|1\rangle$, $\mathbf{x}$ collapses to state $|1\rangle$.

Example:
When we measure the qubit $\mathbf{x} = (1/\sqrt{3}, -2i/\sqrt{3})$, the probability of observing $|0\rangle$ is $1/3$ and the probability of observing $|1\rangle$ is $2/3$. Furthermore, if we observe $|0\rangle$, $\mathbf{x}$ collapses to state $|0\rangle$, and if we observe $|1\rangle$, $\mathbf{x}$ collapses to state $|1\rangle$.

Example:
Let $(0.6, 0.8i) \otimes (0.5\sqrt{3}, -0.5) \otimes (0, -i)$ be a three-qubit quantum register. If we measure the *second* qubit of the register, then with probability $(0.5\sqrt{3})^2 = 3/4$, the register's state collapse to: $(0.6, 0.8i) \otimes |0\rangle \otimes (0, -i)$ and with probability $(-0.5)^2 = 1/4$, the register's state collapses to: $(0.6, 0.8i) \otimes |1\rangle \otimes (0, -i)$.

The Basic Measurement Principle allows us to measure qubits with respect to basis $|0\rangle$ $|1\rangle$, meaning that whenever we perform a measurement, the collapsed state is always $|0\rangle$ or $|1\rangle$. However, there is a more general form of measurement that let's us measure qubit's with respect to any basis of $\mathbb{C}^2$. This is known as the *General Measurement Principle.*

Definition 12 (General measurement principle)**:** *A general quantum measurement of a qubit with respect to basis $\{|y_1\rangle, |y_2\rangle\}$ is carried out by defining two matrices $\mathbf{M}_1 = |y_1\rangle\langle y_1|$, $\mathbf{M}_2 = |y_2\rangle\langle y_2|$. When a qubit $\mathbf{x} = (\alpha_1, \alpha_2)$ is measured in the basis $\{|y_1\rangle\, |y_2\rangle\}$, the result of the observation is either the state $|+\rangle$ with probability $p_1 = \langle \mathbf{M}_1, \mathbf{x}\rangle$ or the state $|-\rangle$ with probability $p_2 = \langle \mathbf{M}_2, \mathbf{x}\rangle$.*

Example:
We perform a general measurement of qubit $\mathbf{x} = (1/\sqrt{2}, i/\sqrt{2})$ in the basis $\{|+\rangle, |-\rangle\}$. First, we define the matrices, $\mathbf{M}_1 = |+\rangle\langle +|$, $\mathbf{M}_2 = |-\rangle\langle -|$. When we compute these matrices, we find:

$$\mathbf{M}_1 = \tfrac{1}{2}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} \quad \mathbf{M}_2 = \tfrac{1}{2}\begin{bmatrix}1 & -1\\ -1 & 1\end{bmatrix}$$

We compute $\mathbf{M}_1\mathbf{x} = \frac{1}{2\sqrt{2}}(1+i, 1+i)$ and $\mathbf{M}_2\mathbf{x} = \frac{1}{2\sqrt{2}}(1-i, i-1)$. When we measure $\mathbf{x}$, we observe state $|+\rangle$ with probability $\langle \mathbf{M}_1\mathbf{x}, \mathbf{M}_1\mathbf{x}\rangle = 1/2$ and observe the state $|-\rangle$ with probability $\langle \mathbf{M}_2\mathbf{x}, \mathbf{M}_2\mathbf{x}\rangle = 1/2$.

Example:
Let is consider the general measurement of qubit $\mathbf{x} = (3/5, 4/5)$ in the basis $\{|+_\alpha\rangle, |-_\alpha\rangle\}$ where the basis has been rotated by angle: $\alpha = \pi$. The measurement matrices are given by $\mathbf{M}_1 = |+_\pi\rangle\langle+_\pi|$, $\mathbf{M}_2 = |-_\pi\rangle\langle-_\pi|$ which evaluate to:

$$\mathbf{M}_1 = \tfrac{1}{2}\begin{bmatrix} 1 & -1 \\ -1 & 0 \end{bmatrix} \quad \mathbf{M}_2 = \tfrac{1}{2}\begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$$

Given that $\mathbf{M}_1\mathbf{x} = (-0.5, 0.5)$ and $\mathbf{M}_2\mathbf{x} = (-0.5, 0.5)$ we can say that when we measure $\mathbf{x}$, we observe state $|+_\pi\rangle$ with probability $\langle \mathbf{M}_1\mathbf{x}, \mathbf{M}_1\mathbf{x}\rangle = 1/2$ and observe the state $|-_\pi\rangle$ with probability $\langle \mathbf{M}_2\mathbf{x}, \mathbf{M}_2\mathbf{x}\rangle = 1/2$.

Example:
We can perform a general measurement to measure a qubit of a quantum register. Let $|\psi\rangle = \mathbf{x} \otimes \mathbf{y} \otimes \mathbf{z} = (1, 0) \otimes (1, -1) \otimes (1, i)$. If we measure the *second* qubit with respect to basis $|+\rangle, |-\rangle$, then the state observed as a result of the measurement is either $(1, 0) \otimes |+\rangle \otimes (1, i)$ with probability $\langle |\psi_1\rangle, |\psi_1\rangle\rangle = 1/2$. or $(1, 0) \otimes |-\rangle \otimes (1, i)$ with probability $\langle |\psi_2\rangle, |\psi_2\rangle\rangle = 1/2$. Furthermore the state of the quantum register collapses to the state observed. Here we used $|\psi_1\rangle = (\mathbf{I} \otimes \mathbf{M}_1 \otimes \mathbf{I})(\mathbf{x} \otimes \mathbf{y} \otimes \mathbf{z})$ and $|\psi_2\rangle = (\mathbf{I} \otimes \mathbf{M}_1 \otimes \mathbf{I})(\mathbf{x} \otimes \mathbf{y} \otimes \mathbf{z})$ where $\mathbf{M}_1 = |+\rangle\langle+|$, $\mathbf{M}_2 = |-\rangle\langle-|$.

3 Measurement Calculus

3.1 Introducing the Measurement Calculus

With the foundations of Quantum Computing laid down, we are now ready to introduce The Measurement Calculus [26], the main subject of this project. To do this, in Section 3.1.1 we first discuss various models of quantum computing and contrast the traditional circuit-model with the newer so-called one-way model. We then motivate the need for a measurement calculus and give its formal definition and operational semantics (Sections 3.1.5 and 3.1.6).

3.1.1 One-way Quantum Computing

The following introduction on one-way quantum computing borrows the thoughts and ideas expressed in [7] and [8]

The first explorations of quantum computing began with the *circuit model* Deutsch (1989). However several other models have been using to describe quantum computation such as *Quantum Turing machines* Deutsch (1985), Quantum Cellular Automata Although they are all proved to be equivalent models of computation but there is no agreement on what is the canonical model for exposing the key aspects of quantum computation. Finding a model that acts as a framework for physicists to perform experiments as well as a platform to give new theoretical insights.

The difficulty in finding a model comes hand-in-hand with the difficulty in finding a physical implementation of a quantum computer. In search of a canonical model, we have seen more recently the birth of *One-way Quantum Computing* or *Measurement based quantum Computer*. The one-way quantum computing is thought to be more feasibly for a physical implementation [17, 18, 19, 20, 21, 22, 23, 24, 25]. One-way quantum computation is an alternative to the traditional quantum-circuit approach that was inspired by a phenomena known as *Quantum Teleportation* [16].

The general idea is that we first *entangle* a cluster of qubits together. Then we perform a series of measurements on the qubits. Each measurement affects the qubits that it was entangled with. That is, the state of the entangle qubits *changes*. This seemingly inadvertent change is actually how one-way quantum computing performs computation. By strategically measuring the right qubits in the right order, the combined effect can leave an output state on the remaining qubits as a result of performing a computation. The name *one-way* emphasizes how we are purposely destroying qubits via measuring (an *irreversible* operation) to perform our computation. This directly contrasts the traditional circuit model where the majority of computation follows a unitary, *reversible* approach.

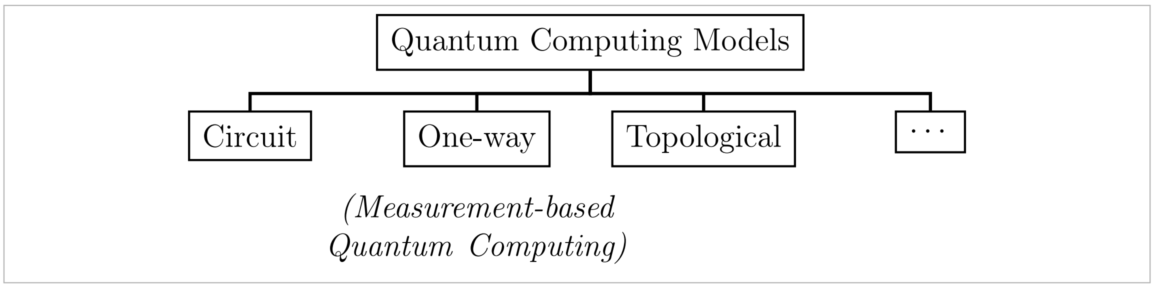


Figure 3.1: There are various models in quantum computing. All are equivalent in expressive power, but we have yet to find a canonical model that researchers from all disciplines can agree would act as the best framework for making new discoveries. While the first model, the *circuit model*, has make significant discoveries, the *one-way model* has now gained significant interest as it is thought that such a model is closer to what a physical implementation would be.

The Measurement Calculus [26] formalizes computation done in the measurement-based model. We take the key operations of one-way, namely the preparation, entanglement, measurement and correction of qubits, and develop a syntax that formally expresses how these operations are done precisely and on which qubits.

3.1.2 Existing Tools

We now briefly discuss existing tools. Currently no tools exist for the Measurement Calculus. All the tools we develop are therefore the first of their kind. A list of quantum computing simulators can be found at [4]. These simulators focus on different quantum computing models, mainly the classical *quantum circuit* model. However a previous individual project has attempted the simulation of The Measurement Calculus.

3.1.3 Introducing Patterns

The Measurement Calculus describes *patterns*. A pattern describes, step-by-step, how to perform a particular computation using the measurement-based model for quantum computation. An example of a pattern is:

$$X_2 M_1^0 E_{12} N_2$$

Informally, a pattern can be thought of a sequence of steps. The sequence is performed right-to-left. The above pattern tells us to do N_2 (an preparatio$n$), followed by E_{12} (an $e$ntanglement), followed by M_1^0 (a $m$easurement) and finally X_2 (an X-correction). Each operator has certain parameters specified via subscript/superscript. For the above example, the 2 in N_2 specifies that we are preparing qubit numbered by 2. The 12 in E_{12} specifies that we entangle together qubits 1 and 2. The 0 in step M_1^0 tells us to measure using basis $\{|+_0\rangle, |-_0\rangle\}$ and the 1 tells us to measure qubit 1.

3.1.4 Measurements

Whenever we measure a qubit, fundamentally, there are two outcomes that occur with a certain probability. Either $|+_\alpha\rangle$ or $|-_\alpha\rangle$. So any measurement operator M_i^α causes a branch in our computation with two possibilities (see Figure 3.2). If our pattern has m measurements, then there are 2^m possible branches overall. Suppose we have measurements M_1, M_2, ..., M_m appearing in our pattern, then the tuple $s = (s_1, s_2, ..., s_m)$ uniquely identifies a given branch. $s_i \in \mathbb{Z}_2$ is called an *outcome*. The tuple $s \in \mathbb{Z}_2^m$ is called a *branch*.

We can extend operations to *conditional operations*. The behaviour of conditional operations depends on the outcomes of previously made measurements. For example, we may desire that last correction C_2 is only applied if and only if $s_1 = 1$. That is, we only apply the correction if the earlier measurement of M_1^0 gave $|+_\alpha\rangle$. If the measurement gave $|-_\alpha\rangle$, then we do not apply the correction. Such a conditional correction is written as $C_2^{s_1}$.

Example:
Below shows the dependencies between instructions.

$$\begin{tikzcd} X_2^{s_1} \arrow[r, dashed, bend right] & M_1^0 & E_{12} & N_2 \end{tikzcd}$$

The value of s_1 is determined after we perform the measurement M_1. It is only when we have measured qubit 1 that we know whether $s_1 = 0$ or $s_1 = 1$.

Example:
Below shows the dependencies between instructions.

$$\begin{tikzcd}[column sep=small]
X_5^{s_4} \arrow[r, dashed, bend right] & M_4^0 & E_{45} & X_4^{s_3} \arrow[rr, dashed, bend right] & Z_4^{s_2} \arrow[rrr, dashed, bend right] & {[M_3^0]^{s_2}} \arrow[rr, dashed, bend right] & Z_3^{s_1} \arrow[rr, dashed, bend right] & {[M_2^0]^{s_1}} \arrow[r, dashed, bend right] & M_1^0 & E_{34} & E_{23} & E_{12}
\end{tikzcd}$$

Notice that an instruction must always follow the measurement instruction that it depends on. It cannot depend on an outcome that has not yet been measured.

A measurement operation behaviour can be made to be dependent on previous measurements. The measurement will always be made, but the basis used may change based on earlier measurements. Instead of always measuring using α, we can instead using a conditional basis where $\alpha' = (-1)^{s_1}\alpha + s_2\pi$. Such a conditional measurement is written as ${}^{s_1}[M_i^{\alpha}]^{s_2}$. The idea of conditioning on outcomes can be extended to conditioning on *signals*. Instead of writing $C_2^{s_1}$, we have $C_2^{s_1+s_2+s_3}$ and instead of ${}^{s_2}[M_2]^{s_1}$ we have ${}^{s_2+s_3}[M_2]^{s_1+s_2}$. Here, we allow conditioning on the *parity* of *various* outcomes. The additions shown as performed modulo-2. This allows the behavior of operations to be dependent on the outcome of multiple outcomes that were previously made. We call the modulo-2 sum of outcomes, *signals*. Notice that a signal with one summand is the same as an outcome. So conditioning on signals is a generalization of conditioning on outcomes. Also note that signals can be empty in which case the value is fixed to 1.

3.1.5 Formal Definition

Definition 13 (Pattern [27, pg 7, Definition 1])**:** *Formally, a pattern $\mathcal{P}$ is described by a four-tuple $\mathcal{P} = (V, I, O, A)$ along with two injective maps $\iota : I \to V$, $o : O \to V$, where V is the set of all qubits, I is the set of inputs, O is the set of output qubits, and A is the command sequence.*

$\iota(I) \subseteq V$ represents the set of *input qubits*[1]. Similarly, $o(O) \subseteq V$ represents the set of *output qubits*. Notice that $\iota(I)$ and $o(O)$ are not necessarily disjoint. A qubit can be both an input qubit and an output qubit. We also allow the case where a qubit is in both $\iota(I)$ and $o(O)$. The reason for having functions ι and o, is so that can index the qubits without using elements in set V. Quite often, V is often chosen to be a set of numbers, and so $\iota(I)$ and $o(O)$ allow us to bijectively map potentially more meaningful names/identifiers these numbers that belong to V.

[1] $f(S)$ denotes the image of set S under function f.

The initial state of the qubits in I is an arbitrary state chosen by the user. This represents the input of the measurement algorithm. The final state of the qubits in O are intended to hold the final quantum state, which represents the output of the algorithm. At the end we expect only the qubits in O to remain and for all the other qubits to have been measured.

Example:
Let $V = \{1,2,3,4\}$. $I = \{m,n\}$, $O = \{p,q\}$, and $\iota = \{(m,1),(n,2),(p,3)\}$. $o = \{(p,3),(q,4)\}$. Then the *input qubits* are $\iota(I) = \{1,2,3\}$, the *output qubits* are $o(O) = \{3,4\}$. The *intermediate qubits* are $\iota(I) \cap o(O) = \{3\}$.

In the tuple $\mathcal{P} = (V,I,O,A)$, A is a finite sequence of commands, $A_n, ..., A_1$ where the syntax of each command A_i follows the rule A:

$$\begin{array}{lcl} \mathsf{S} & ::= & 0 \mid 1 \mid s_i \mid \mathsf{S}+\mathsf{S} \\ \mathsf{A} & ::= & N_i \mid E_{ij} \mid {}^{\mathsf{S}}[M_i^{\alpha}]^{\mathsf{S}} \mid X_i^{\mathsf{S}} \mid Z_i^{\mathsf{S}} \end{array}$$

where $i,j \in V$ and $\alpha \in [0,2\pi)$.

We now give the implementation of patterns (Listing 3.1) Each pattern is an object constructed from the function `Pattern` that accepts an array of instructions, `instrs`, and an object `qubits`, that maps qubit names (primitive values) to a number that denotes the qubit's input/output status. We use constants `QubitIO.Neither` to denote $q \notin I, q \notin O$, `QubitIO.In` to denote $q \in I, q \notin O$, `QubitIO.Out` to denote $q \notin I, q \in O$, and `QubitIO.InOut` to denote $q \in I, q \in O$.

```
function Pattern(instrs, qubits) {
            this.instructions = instrs;
            this.qubits = qubits;
            ...
}
var hadamard = new Pattern([
           new XCorrection(2, [1]),
           new Measurement(1, x, [], []),
           new Entanglement(1, 2),
           new Prepare(2)],{1: QubitIO.In, 2: QubitIO.Out});
```

Listing 3.1: constructor for patterns

Each command N, E, M, X, Z is represented by an object constructed from functions `Prepare`, `Entanglement`, `Measurement`, `XCorrection`, `ZCorrection` respectively (Listing 3.2).

```
// Constructor functions
```

```
function Instruction(qubits, signals) {...}
function Entanglement(q1, q2) {...}
function Prepare(q) {...}
function Measurement(q, angle, sig1, sig2) {...}
function Correction(q, sig) {...}
function XCorrection(q, sig) {...}
function ZCorrection(q, sig) {...}

// Inheritance
Entanglement.prototype = new Instruction([], [], []);
Measurement.prototype = new Instruction([], [], []);
Correction.prototype = new Instruction([], [], []);
Prepare.prototype = new Instruction([], []);
ZCorrection.prototype = new Correction([], []);
XCorrection.prototype = new Correction([], []);
```

Listing 3.2: constructors for objects representing commands

Additionally, the sequence of commands A must simultaneously satisfy **D0-D4** that are defined as:

- **D0**: *no A_i depends on an outcome not yet measured.*

  For any instruction ${}^{\mathsf{S}}[M_i^\alpha]^{\mathsf{S}}$, X_i^{S}, $Z_i^{\mathsf{S}} \in A$,
  $\forall s_j \in \mathsf{S}$ there is some measurement M_j previous in sequence
  That is, no operation may depend on a measurement outcome s_q before the qubit q has been measured. This rule tells us that we cannot use the outcomes of measurements until we actually have performed the measurement. It is a causal dependency. A violation would not respect the order in which events occur. For this reason, patterns that violate D0 are sometimes called *anachronical patterns*.

- **D1**: *no command acts on a qubit already measured*
  For any instruction $A_i \in A$:
  A_i acts on qubit j implies there no M_j previous in sequence
  In other words the last instruction for a qubit that is due for destruction is the measurement instruction. Once a qubit has been measured, no command cannot act on it, for it has been destroyed. This rule also expresses that a qubit cannot be measured twice.

- **D2**: *no A_i acts on a qubit not yet prepared, except if is an input qubit*
  The first operation performed on a non-input qubit, $q \in V - I$ is a preperation N_q. This is the only preperation is that is applied to the non-input qubit (it is prepared exactly once). For input qubits $q' \in I$, we require that there are no preparation commands that act on q'. These qubits are prepared implictly by the operational semantics and so preparing them again makes no sense.

- **D3**: *a qubit $q \in V$ is measured if and only if $q \in V - O$.*
  We must measure all qubits q that satisfy $q \in I, q \notin O$, or $q \notin I, q \notin O$. Conversely, a pattern must not measure qubits q that satisfy $q \in I, q \notin O$, or $q \notin I, q \notin O$. From just the input/output structure of a pattern, we can determine exactly how many measurement instructions there must be, and, which qubits those measurements operate on.

In Appendix **??** we give functions `isValidD0`,...,`isValidD3` which check whether conditions **D0**,...,**D3** respectively are satisfied. Each function returns a boolean value to indicate whether the corresponding rule is satisfied.

Example:
Let $V = \{1, 2\}$, $I = \{1\}$ and $O = \{2\}$. The pattern with commands $X_2 M_1 N_2 E_{12}$ is *invalid* because **D2** has been violated. Instruction E_{12} attempts to operate on qubit 2 despite that fact that qubit $2 \notin I$ has not been previously prepared in the sequence.

Example:
Let $V = \{1, 2\}$, $I = \{1\}$ and $O = \{2\}$. The pattern with commands $M_2 M_1 N_2$ is *invalid* because **D3** has been violated. Instruction M_2 measures output qubit $3 \in O$. However, output qubits must never be measured.

For any given pattern, we can draw the *entanglement graph.* See Definition 24 and Figure 4.4 for two examples. Here we use a convention where input qubits have boxes, and qubits that need to be measured have filled-in circled. This notation gives us a graphical way to represent the input-output status for the qubits of a pattern and, provides one or two mnemonics to remember conditions **D2** and **D3**. As we can see from Figure 3.3, there are four possible cases that a qubit has for its input-output status. For each case, we associate a diagram involving a square and a circle. The precise colouring of the circle and square uniquely determines the input-output status. A pattern has the circle filled in if and only if it is $\in V - O$. A pattern has a square if and only if it is $\in I$. Qubits that are filled in will be measured destroyed, whereas those not filled in must not be measured. This is a mnemonic for condition **D2**. Additionally, qubits with boxes are prepared in an input state implictly, whereas those without boxes are prepared explicitly in the state $|+\rangle$ via the N command. This is a mnemonic for condition **D3**.

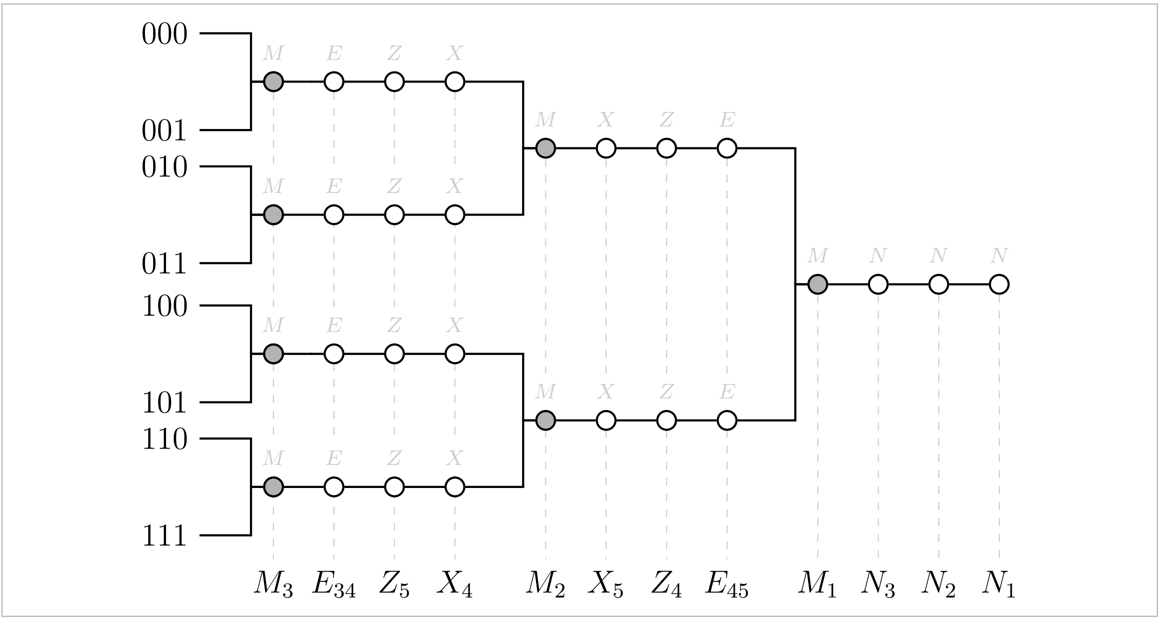

000
001
010
011
100
101
110
111
M
E
Z
X
N
M_3 E_{34} Z_5 X_4 M_2 X_5 Z_4 E_{45} M_1 N_3 N_2 N_1

Figure 3.2: Measurements in patterns cause nondeterministic branches in the computation. The result is that we have a computation *tree*. A path along from the root of the tree to a leaf corresponds to one particular computation. We call this path a *branch*. Each edge that comes out of a measurement has a bit associated (0 or 1) that we call an *outcome*. A pattern with $m = |V - O|$ measurement instructions has a tree with 2^m possible branches. The pattern shown above has three measurements with 8 branches.

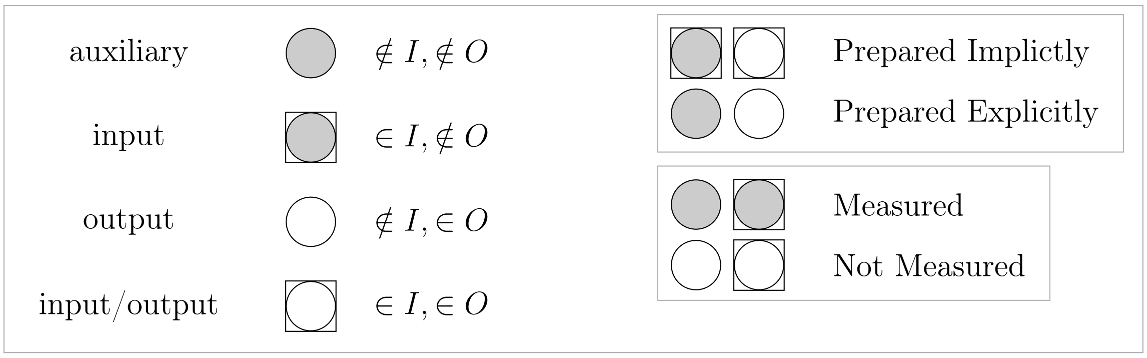


Figure 3.3: Graphical notation for the input-output status of a qubit. A square denotes $q \in I$. A white circle denotes $q \in O$. If a qubit does not have a square then it must be prepared explicitly in state $|+\rangle$ using the N command (**D2**). If a qubit has a dark circle then it must be measured (**D3**).

3.1.6 Operational Semantics

Let $A \to B$ denote the set of all *partial functions* from A to B. Then $f \in A \to B$ gives some partial function from A to B. We'll use this to distinguish from $f : A \to B$, a *total function* from A to B. We also distinguish this from: $R \subseteq A \times B$, that denotes a *relation* between A and B.

To describe the operational semantics, we define a relation:

$$\rightsquigarrow \ \subseteq \ (\mathfrak{H} \times (V \to \mathbb{Z}_2) \times 2^{V-O} \times 2^{V-O}) \to (\mathfrak{H} \times (V \to \mathbb{Z}_2) \times 2^{V-O} \times 2^{V-O})$$

where $\rightsquigarrow$ is a transition relation that describes what computation states could possible follow a given computation state. We define the state of the computation as the first two components, $\mathfrak{H}_V \times V(\to \mathbb{Z}_2)$. $q \in \mathfrak{H}$ is called the *quantum* state and $\Gamma \in (V \to \mathbb{Z}_2)$ is called the *classical* state. The third and fourth components, F and G, keep track of which qubits from have not been measured, and which have been measured, respectively at the current point of execution. We store this information[2] using sets $F, G \in 2^{V-O}$.

To begin execution, we start with $(\mathbf{q}_{in}, \Gamma_\emptyset, V - O, \emptyset)$, where $\mathbf{q}_{in}$ is an arbitrary state for the input qubits. Here $F = V - O$ and $G = \emptyset$ because we have not done any measurements yet, so all the non-output qubits are in F and are waiting to be measured. We then have a *preparation stage* where we apply all the N_i's (in any order). By performing all the N_i's that occur in the pattern we are preparing all qubits $i \in V - I$. Every N_i, is executed here regardless of where N_i is in the pattern. This gives us our initial state $\mathbf{q}_0$. Next comes the *main stage*. Here, we apply the remaining commands in the pattern that were not the preparation commands (any E, C, M commands). These are applied in sequence as they appear in the pattern from right to left. Each command in the main stage, transforms the state: $(\mathbf{q}_i, \Gamma_i, F_i, G_i)$ to $(\mathbf{q}_{i+1}, \Gamma_{i+1}, F_{i+1}, G_{i+1})$. The following sequence summarises the execution of patterns from beginning to end. $(\mathbf{q}_0, \Gamma_\emptyset, V-O, \emptyset) \xrightarrow{N_i' s} (\mathbf{q}_0', \Gamma_\emptyset, V-O, \emptyset) \xrightarrow{A_1} (\mathbf{q}_1, \Gamma_1, F_1, G_1) \xrightarrow{A_2} (\mathbf{q}_2, \Gamma_2, F_2, G_2) \xrightarrow{A_3} ... \xrightarrow{A_n} (\mathbf{q}_n, \Gamma_n, \emptyset, V - O)$

Danos et al. [27, Section 5.3, pg 11] define the small-steps semantics of $\rightsquigarrow$ by:

$$\begin{array}{rlcl}
N_i : & (\mathbf{q},\ \Gamma, F, G) & \rightsquigarrow & (\mathbf{q} \otimes |+\rangle_i\,,\ \Gamma, F, G) \\
E_{ij} : & (\mathbf{q},\ \Gamma, F, G) & \rightsquigarrow & (\mathbf{CZ}_{ij}\mathbf{q},\ \Gamma, F, G) \\
X_i^s : & (\mathbf{q},\ \Gamma, F, G) & \rightsquigarrow & (\mathbf{X}_i^{s_\Gamma}\mathbf{q},\ \Gamma, F, G) \\
Z_i^s : & (\mathbf{q},\ \Gamma, F, G) & \rightsquigarrow & (\mathbf{Z}_i^{s_\Gamma}\mathbf{q},\ \Gamma, F, G) \\
{}^t[M_i^\alpha]^s : & (\mathbf{q},\ \Gamma, F \cup \{i\}, G) & \rightsquigarrow & \begin{cases} (\langle +_{\alpha_\Gamma}|_i\,\mathbf{q},\ \Gamma[i \mapsto 0], F, G \cup \{i\}) \\ \qquad\qquad \text{or} \\ (\langle -_{\alpha_\Gamma}|_i\,\mathbf{q},\ \Gamma[i \mapsto 1], F, G \cup \{i\}) \end{cases}
\end{array}$$

We now need to make an important distinction between *qubit names*, and *qubit indices*, an in particular the difference between a tensor product, and what we call

[2]Here 2^S denotes the powerset of S.

an *associative tensor product.* Notice that at every step we have $F_i \cup G_i = V$, $F_i \cap G_i = \emptyset$. For any partial function $\Gamma \in W \to \mathbb{Z}_2$, we define the partial function $\Gamma[i \mapsto x] \in (W \cup \{i\}) \to \mathbb{Z}_2$,

$$\begin{array}{rcl} \Gamma[i \mapsto x](i) & = & i \\ \Gamma[i \mapsto x](j) & = & \Gamma(j) \quad \text{for } j \neq i \end{array}$$

The map Γ allows us to evaluate any signal whose domain qubits, J, have all already been measured. For $J \subseteq G$, if $s = \sum_{i \in J} s_j$ then $s_\Gamma = \sum_{i \in J \subseteq G} \Gamma(i)$ is the value of the signal given by the particular partial branch Γ. Here we adopt the notation that for $\mathbf{CZ}_{ij}$ operation, it is the *first* qubit listed (qubit i) that is the *control* qubit and the *second* listed (qubit j) is the *target* qubit.

Example:
Below shows an possible execution of the pattern:

$$X_2^{s_1} M_1^0 E_{12} N_2$$

where $V = \{1, 2\}$, $I = \{1\}$ and $O = \{2\}$.

$$\begin{array}{rl} (q_0, \Gamma_\emptyset, \{1,2\}, \emptyset) & \xrightarrow{N_2} (q_0 \otimes |+\rangle_2, \Gamma_\emptyset, \{1,2\}, \emptyset) \\ & \xrightarrow{E_{12}} (CZ_{12}(q_0 \otimes |+\rangle_2), \Gamma_\emptyset, \{1,2\}, \emptyset) \\ & \xrightarrow{M_1^0} (\langle +_\alpha | CZ_{12}(q_0 \otimes |+\rangle_2, \Gamma_1, \{2\}, \{1\}) \\ & \xrightarrow{X_2^{s_1}} (X_2 \langle +_\alpha | CZ_{12}(q_0 \otimes |+\rangle_2, \Gamma_1, \{2\}, \{1\}) \end{array}$$

Example:
We give an example execution of CNOT with input $q_{in[1]} = (1,0)$, $q_{in[2]} = (1,0)$.

$$
\begin{aligned}
&((1,0,0,0),\Gamma_\emptyset,\{2,3\},\emptyset)\\
&\xrightarrow{N_3} (1/\sqrt{2},1/\sqrt{2},0,0,0,0,0,0),\Gamma_\emptyset,\{2,3\},\emptyset)\\
&\xrightarrow{N_4} (1/2,1/2,1/2,1/2,0,0,0,0,0,0,0,0,0,0,0,0),\Gamma_\emptyset,\{2,3\},\emptyset)\\
&\xrightarrow{E_{23}} (1/2,1/2,1/2,1/2,0,0,0,0,0,0,0,0,0,0,0,0),\Gamma_\emptyset,\{2,3\},\emptyset)\\
&\xrightarrow{M_2^x} (1/\sqrt{2})(2,2,2,2,-2,-2,-2,-2,0,0,0,0,0,0,0,0),(2\mapsto 1),\{3\},\{2\})\\
&\xrightarrow{X_3^{s2}} (1/\sqrt{2}((2,2,2,2,-2,-2,-2,-2,0,0,0,0,0,0,0,0),(2\mapsto 1),\{3\},\{2\})\\
&\xrightarrow{E_{13}} (1/\sqrt{2})(2,2,2,2,-2,-2,-2,-2,0,0,0,0,0,0,0,0),(2\mapsto 1),\{3\},\{2\})\\
&\xrightarrow{E_{34}} (1/\sqrt{2})(2,2,2,-2,-2,-2,-2,2,0,0,0,0,0,0,0,0),(2\mapsto 1),\{3\},\{2\})\\
&\xrightarrow{M_3^x} (0,0.5,0,-0.5,0,-0.5,0,0.5,0,0,0,0,0,0,0,0),(2\mapsto 1,3\mapsto 1),\emptyset,\{2,3\})\\
&\xrightarrow{X_4^{s3}} (0,0.5,0,-0.5,0,-0.5,0,0.5,0,0,0,0,0,0,0,0),(2\mapsto 1,3\mapsto 1),\emptyset,\{2,3\})
\end{aligned}
$$

The final state is $(0,0.5,0,-0.5,0,-0.5,0,0.5,0,0,0,0,0,0,0,0)$
$= (1,0)\otimes(0,0.5,0,-0.5,0,-0.5,0,0.5)$
$= (1,0)\otimes\frac{1}{\sqrt{2}}(1,-1)\otimes\frac{1}{\sqrt{2}}(0,1,0,-1)$
$= (1,0)\otimes\frac{1}{\sqrt{2}}(1,-1)\otimes\frac{1}{\sqrt{2}}(1,-1)\otimes(0,1)$
$= (1,0)\otimes|-\rangle\otimes|-\rangle\otimes(0,1)$
As expected, result is that output qubit 4 has $(0,1) = \mathbf{X}(1,0)$.

We shall now give an implementation of the operational semantics. The simulation begins (Listing 3.3) by preparing all the input qubits in their respective input state $\in \mathbb{C}^2$ to give the overall input state $\mathbf{q} \in \mathfrak{H}_I$. We take `state` and perform the tensor product with $\mathbf{q}_{in[i]}$ (for input qubits $i \in I$), or with $|+\rangle$ (for non-input qubits $i \in V-I$).

```
this.begin = function(pattern, inputstates) {
  ...
  // add input qubits to state
  for (var q in inputstates) {
    this.state = qm.vectorTensor(this.state, [inputstates[q][0].
   clone(),inputstates[q][1].clone()] );
    this.nameToIndex[q] = this.numActiveQubits++;
    this.indexToName.push(q);
  }
  // set all non-input qubits to |+> state
  for (var x in pattern.qubits) {
    if (!qio.IsInput(pattern.qubits[x])) {
        this.prepare(x);
    }
  }
```

Listing 3.3: beginning the simulation

We have a separate function `prepare` (Listing 3.4) that prepares non-input qubits in the state $|+\rangle$. This implements the N_i command. We update `nameToIndex` track of the position a given qubit has in the tensor product. This maps a qubit name to a zero-based index. We also update `numActiveQubits` to reflect how many qubits are in the tensor product currently.

```
this.prepare = function(i) {
  this.nameToIndex[i] = this.numActiveQubits++;
  this.indexToName.push(i);
  var plus = [f.one(),f.one()];
  qm.scaleVector(1/Math.sqrt(2), plus);
  this.state = qm.vectorTensor(this.state,plus);
};
```

Listing 3.4: implementing preparation N_i

Next in Listings 3.5 we show the implementation of the entangle command E_{ij}. The operational semantics tell us that we need to apply the matrix $\mathbf{CZ}_{ij}$ to the current state. This controlled-Z operation can be expressed as:

$$\mathbf{CZ}_{ij} = |0\rangle\langle 0|_i \otimes \mathbf{I}_j + |1\rangle\langle 1|_i \otimes \mathbf{Z}_j$$

which indeed closely resembles what the controlled-Z operation performs: *if the qubit i is $|0\rangle$, then apply* $\mathbf{I}$ *to qubit j, if the qubit i is $|1\rangle$, then apply* $\mathbf{Z}$ *to qubit j.* Here `outerzero`, `outerone` is the outer products, $|0\rangle\langle 0|$, $|1\rangle\langle 1|$ respectively.

We use the function `combinedStateMatrix` to convert $|0\rangle\langle 0|_i$ to the form $\mathbf{I}\otimes\mathbf{I}\otimes ... \otimes |0\rangle\langle 0| \otimes \mathbf{I} \otimes ... \otimes \mathbf{I}$ where the position of $|0\rangle\langle 0|$ in the tensor product matches the index equal to `this.nameToIndex[i]` - ensuring that $|0\rangle\langle 0|$ is applied to qubit i and that all other qubits are left alone. This gives us $|0\rangle\langle 0|_i \otimes \mathbf{I}_j$ which we temporarily store in variable `cz`. We then a call to the function `combinedStateMatrices` converts $|1\rangle\langle 1|_i \otimes \mathbf{Z}_j$ into $\mathbf{I}\otimes\mathbf{I}\otimes ... \otimes |1\rangle\langle 1| \otimes \mathbf{I}\otimes\mathbf{I}\otimes ... \otimes \mathbf{Z}\otimes\mathbf{I}...\mathbf{I}$ where the positions of $|1\rangle\langle 1|$ and $\mathbf{Z}$ match those of `this.nameToIndex[i]` and `this.nameToIndex[j]` respectively. We add this matrix to what is stored in `cz`, take the resulting matrix and multiply it with the state.

```
this.entangle = function(i,j){
  var outerzero = [[f.one(),f.zero()],[f.zero(),f.zero()]],
      outerone = [[f.zero(),f.zero()],[f.zero(),f.one()]];
  var z = [[f.one(),f.zero()],[f.zero(),f.minusone()]];
  var cz = this.combinedStateMatrix(outerzero, this.nameToIndex[i]);
  qm.matrixAddWith(cz, this.combinedStateMatrices(outerone, this.
   nameToIndex[i], z, this.nameToIndex[j]));
  this.state = qm.matrixVectorMultiply(cz, this.state);
};
```

Listing 3.5: implementing entanglement E_{ij}

Next is the implementation of the conditional measurement command ${}^t[M_i^\alpha]^s$. In the code, `signal1` refers to s and `signal2` refers to t. We implement Γ using `this.outcomeMap` which stores the outcomes of measurements. It is a object that maps qubit names to either 0 or 1, depending on the result of a measurement. With Γ, we can find s_Γ, t_Γ, the evaluation of a signals s and t. We use two for loops and perform addition modulo 2 to find the final sums `sgamma` and `tgamma`. We can now find our angle $\alpha' = (-1)^{s_\Gamma}\alpha + t_\Gamma * \pi$ (which we call `fullangle` in the code). And thus, have the basis for our measurement $|+_{\alpha'}\rangle, |-_{\alpha'}\rangle$. We compute the measurement projection matrices $\mathbf{M}_+(\alpha')$ and $\mathbf{M}_-(\alpha')$ using `qm.plusMeasureMatrix(fullangle)` and `qm.plusMeasureMatrix(fullangle)`. We call these matrices `m1` and `m2`.

Next in Listing 3.6 we compute the unnormalised projection $\mathbf{M}_0 |\psi\rangle$ (where $|\psi\rangle$ is `this.state`). We can then, find the probability of outcome 0 (seeing i in the plus state) by using the identity:

$$Prob[outcome\ j] = \langle\psi| \mathbf{M}_j^\dagger \mathbf{M}_j |\psi\rangle = (\mathbf{M}_j |\psi\rangle)^\dagger (\mathbf{M}_j |\psi\rangle) = ||\mathbf{M}_j |\psi\rangle ||^2$$

Hence, we find the length of `projection` giving us $prob[outcome0]$ which is store in `p1`. Next we use Javascript's `Math.Random` to implement pseudo-random coin toss where our coin is biased towards heads with probability `p1`. If the coin is heads, the model this as outcome 0 (observing $|+\rangle$). Tails is outcome 1 (observing $|-\rangle$). We finally record the outcome by updating Γ (`this.outcomeMap`) and then collapsing the quantum state to $\frac{\mathbf{M}_j|\psi\rangle}{\sqrt{prob[outcome\ j]}}$.

```
this.measure = function(i, angle, signal1, signal2) {
  // 1. FIND THE ANGLE
  for (var sgamma = 0, u = 0, l = signal1.length; u < l; u++)
    sgamma = qm.mod2Add(sgamma, this.outcomeMap[u]);
  for (var tgamma = 0, v = 0, k = signal2.length; v < k; v++)
    tgamma = qm.mod2Add(tgamma, this.outcomeMap[v]);
  var fullangle // = (-1)^sgamma * angle + tgamma * pi

  // 2. PERFORM THE MEASUREMENT
  var m1 = qm.plusMeasureMatrix(fullangle);
  var m2 = qm.minusMeasureMatrix(fullangle);
  var projection = qm.matrixVectorMultiply(this.combinedStateMatrix(
   m1, this.nameToIndex[i]), this.state);
  var p1 = qm.vectorLengthSquared(projection);
  var R = Math.random() * MEASURE_RAND_RANGE;
  if (R < p1 * MEASURE_RAND_RANGE) {
    // Outcome of measurement is 0:
    this.outcomeMap[i] = 0;
    this.state = qm.scaleVector(1/Math.sqrt(p1), projection);
  } else {
    // Outcome of measurement is 1:
    this.outcomeMap[i] = 1;
    var p2 = 1 - p1;
    this.state = qm.matrixVectorMultiply(qm.scaleMatrix(1/Math.sqrt(
   p2), this.combinedStateMatrix(m2, this.nameToIndex[i])),this.
   state)
  }
};
```

Listing 3.6: implementing measurement ${}^t[M_i^\alpha]^s$

Finally in Listings 3.7, we describe `this.xcorrect`, which implements X_i^s. First we evaluate the signal to find s_Γ. Just as we did in `this.measure`, we take Γ (`this.outcomeMap`) and evaluate the signal's summation using modulo 2 arithmetic. Depending on whether the sum is zero or one, we don't apply/do apply $\mathbf{X}$ to qubit i. Again, we use the function `combinedStateMatrix` to convert $\mathbf{X}_i$ to the form $\mathbf{I} \otimes \mathbf{I} \otimes ... \otimes \mathbf{X} \otimes \mathbf{I} \otimes ... \otimes \mathbf{I}$ where the position of $\mathbf{X}$ is aligned with the position of qubit i in the vector state $|\psi\rangle$ = `this.state` (positioned at index `this.nameToIndex[i]` in the product) - ensuring that $\mathbf{X}$ is applied to qubit i and that all other qubits are left alone.

```
this.xcorrect = function(i, signal) {
  // evaluate signal
  for (var mod2sum = 0, u = 0, l = signal.length; u < l; u++)
      mod2sum = qm.mod2Add(mod2sum, this.outcomeMap[u]);
  // if signal = 1, perform correction, otherwise do nothing)
  if (mod2sum === 1) {
      var x = [[f.zero(),f.one()],[f.one(),f.zero()]];
      qm.matrixVectorMultiply(this.combinedStateMatrix(x,
        this.nameToIndex[i]), this.state);
  }
};
this.zcorrect = function(i, signal){...};
```

Listing 3.7: implementing X-correction X_i^s

We omit the listings for the implementation of Z_i^s. The function `this.zcorrect` is almost identical. The only difference is that we replace `var x = ...` with the code: `var z = ...` to compute the matrix for $\mathbf{Z}$.

3.2 Term Rewriting

Now that we have defined the operational semantics - defining precisely what a computation is - we explore how two patterns can perform the same computation. We define various notions of equivalence and then develop a term rewriting system that's both confluent and terminating, thus giving us a method of checking that two patterns are equivalent.

3.2.1 Equivalent Patterns

For each instruction we apply a matrix to the current state $\mathbf{q}_i$. So for any branch, we can find a matrix that takes us from the initial state to the final state.

Definition 14 (Branch Map [27, Section 5.3, pg 12])**:** *For any given branch s in pattern $\mathcal{P}$, the Branch Map for s, $\mathcal{T}_s^{\mathcal{P}}$, as the linear map that gives a linear transformation that transforms $\mathbf{q}_0 \in \mathfrak{h}_I \to \mathbf{q}_s \in \mathfrak{h}_O$, where $\mathbf{q}_s$ is the final state that results from the execution of branch s.*

Definition 15 (Pattern Equality [27, Section 5.5, pg 26])**:** *We say that two patterns $\mathcal{P}$ and $\mathcal{P}'$ are equal, $\mathcal{P} = \mathcal{P}'$, if and only if they have the same number of measurements, m, and more crucially, for any branch $s \in \{0,1\}^m$, we have $\mathcal{T}_s^{\mathcal{P}} = \mathcal{T}_s^{\mathcal{P}'}$.*

With our definition of equality in place, we can develop equivalence rules which preserve the equality of patterns. We write $A_k...A_1 \equiv A'_r...A'_1$ to denote that for any pattern $\mathcal{P} = (V, I, O, C_n...A_k...A_1...C_0)$, we have the pattern equality $\mathcal{P} = \mathcal{P}'$ where $\mathcal{P}' = (V, I, O, C_n...A'_r...A'_1...C_0)$.

The first equivalence rule is called the EX-rule. This rule tells us that if we X-correct qubit *immediately before* it is entangled, we can instead, perform the correction *after* the entanglement, so long as we Z-correct the other qubit involved in entanglement, and do so immediately after the entanglement. Formally,

$$\textbf{EX Rule:} \quad \begin{aligned} E_{ij}X_i^s &\equiv X_i^s Z_j^s E_{ij} \\ E_{ij}X_j^s &\equiv X_j^s Z_i^s E_{ij} \end{aligned}$$

To see intuitively why this equivalence holds[3], we can show that $\mathbf{CZ}_{ij}\mathbf{X}_i^{s_\Gamma} = \mathbf{X}_i^{s_\Gamma}\mathbf{Z}_j^{s_\Gamma}\mathbf{CZ}_{ij}$. When $s_\Gamma = 0$, all the corrections disappear and we are left with $\mathbf{CZ}_{ij} = \mathbf{CZ}_{ij}$ which

[3]We only explicitly prove the first equivalence. For the second equivalence, a similar proof for $\mathbf{CZ}_{ij}\mathbf{X}_j^{s_\Gamma} = \mathbf{X}_j^{s_\Gamma}\mathbf{Z}_i^{s_\Gamma}\mathbf{CZ}_{ij}$ can be found.

holds trivially. When $s_\Gamma = 1$, the right-hand side is $\mathbf{X}_i\mathbf{Z}_j\mathbf{CZ}_{ij} = \mathbf{X}_i\mathbf{Z}_j(|0\rangle\langle 0|_i \otimes \mathbf{I}_j + |1\rangle\langle 1|_i \otimes \mathbf{Z}_j) = \mathbf{X}|0\rangle\langle 0|_i \otimes \mathbf{Z}_j + \mathbf{X}|1\rangle\langle 1|_i \otimes \mathbf{ZZ}_j = |1\rangle\langle 0|_i \otimes \mathbf{Z}_j + |0\rangle\langle 1|_i \otimes \mathbf{I}_j$ $= (|1\rangle\langle 1|_i \otimes \mathbf{Z}_j + |0\rangle\langle 0|_i \otimes \mathbf{I}_j)\mathbf{X}_i = \mathbf{CZ}_{ij}\mathbf{X}_i$ which is the left-hand side.

The next rule is the EZ-rule. This rule states that if we Z-correct qubit *immediately before* it is entangled, we can instead, perform the correction *after* the entanglement. Formally,

$$\begin{array}{rrcl} \textbf{EZ Rule:} & E_{ij}Z_i^s & \equiv & Z_i^s E_{ij} \\ & E_{ij}Z_j^s & \equiv & Z_j^s E_{ij} \end{array}$$

To see intuitively why this equivalence holds[4], we can show that $\mathbf{CZ}_{ij}\mathbf{Z}_i^{s_\Gamma} = \mathbf{Z}_i^{s_\Gamma}\mathbf{CZ}_{ij}$. When $s_\Gamma = 0$, all the Z-correction on each side disappear and we are left with $\mathbf{CZ}_{ij} = \mathbf{CZ}_{ij}$ which holds trivially. When $s_\Gamma = 1$, the left-hand side is $\mathbf{CZ}_{ij}\mathbf{Z}_i = (|0\rangle\langle 0|_i \otimes \mathbf{I}_j + |1\rangle\langle 1|_i \otimes \mathbf{Z}_j)\mathbf{Z}_i = |0\rangle\langle 0|\mathbf{Z}_i \otimes \mathbf{I}_j + |1\rangle\langle 1|\mathbf{Z}_i \otimes \mathbf{Z}_j = |0\rangle\langle 0|_i \otimes \mathbf{I}_j - |1\rangle\langle 1|_i \otimes \mathbf{Z}_j$ $= \mathbf{Z}|0\rangle\langle 0|_i \otimes \mathbf{I}_j + \mathbf{Z}|1\rangle\langle 1|_i \otimes \mathbf{Z}_j = \mathbf{Z}_i(|0\rangle\langle 0|_i \otimes \mathbf{I}_j + |1\rangle\langle 1|_i \otimes \mathbf{Z}_j) = \mathbf{Z}_i\mathbf{CZ}_{ij}$ which is the right-hand side.

The MX-rule and MZ-rule express that correcting a qubit immediately before measuring it is the same as solely measuring it so long as we adjust the conditioning for the basis of measurement. Formally,

$$\begin{array}{rrcl} \textbf{MX Rule:} & {}^t[M_i^\alpha]^s X_i^r & \equiv & {}^t[M_i^\alpha]^{s+r} \\ \textbf{MZ Rule:} & {}^t[M_i^\alpha]^s Z_i^r & \equiv & {}^{t+r}[M_i^\alpha]^s \end{array}$$

To see why the MX-rule holds [5], we show that $|+\{(-1)^s\alpha + t\pi\}\rangle_i \mathbf{X}_i^r = |+\{(-1)^{s+r}\alpha + t\pi\}\rangle_i$. We have: $|+\{(-1)^s\alpha + t\pi\}\rangle_i \mathbf{X}_i^r = |+\{(-1)^s\alpha + t\pi\}\rangle \mathbf{X}^r{}_i = |+\{(-1)^r[(-1)^s\alpha + t\pi]\}\rangle_i$ $= |+\{(-1)^{s+r}\alpha + (-1)^r t\pi\}\rangle_i = |+\{(-1)^{s+r}\alpha + t\pi\}\rangle_i$. Here, we used the identities $|+\{\alpha\}\rangle\mathbf{X} = |+\{-\alpha\}\rangle$ and $|+\{\alpha - \pi\}\rangle = |+\{\alpha + \pi\}\rangle$
To see why the MZ-rule holds, we show that $|+\{(-1)^s\alpha + t\pi\rangle_i \mathbf{Z}_i^r = |+\{(-1)^s\alpha + (t+r)\pi\}\rangle_i$. We have: $|+\{(-1)^s\alpha + t\pi\}\rangle_i \mathbf{Z}_i^r = |+\{(-1)^s\alpha + t\pi\}\rangle \mathbf{Z}^r{}_i = |+\{(-1)^s\alpha + t\pi + r\pi\}\rangle_i$ $= |+\{(-1)^s\alpha + (t+r)\pi\}\rangle_i$ Here, we used the identity $|+_\alpha\rangle\mathbf{Z} = |+_{\alpha+\pi}\rangle$.

When instructions operate on disjoint sets of qubits, we have rules expressing commutation. These are known as the *free commutation rules* and are given below.

$$\begin{array}{rrcll} \textbf{NA Rule:} & N_iA & \equiv & AN_i & \text{if } A \neq N \text{ and } dom(A) \cap \{i,j\} = \emptyset \\ \textbf{EA Rule:} & E_{ij}A & \equiv & AE_{ij} & \text{if } A \neq E \text{ and } dom(A) \cap \{i,j\} = \emptyset \\ \textbf{AX Rule:} & AX_i^s & \equiv & X_i^sA & \text{if } A \neq C \text{ and } dom(A) \cap \{i\} = \emptyset \\ \textbf{AZ Rule:} & AZ_i^s & \equiv & Z_i^sA & \text{if } A \neq C \text{ and } dom(A) \cap \{i\} = \emptyset \end{array}$$

Here, $dom(A)$ is the function defined as: $dom(N_i) = \{i\}$, $dom(E_{ij}) = \{i,j\}$, $dom(X_i^s)$ $= dom(Z_i^s) = \{i\} \cup \{j \text{ s.t. } s_j \text{ appears in } s\}$ and $dom({}^t[M_i^\alpha]^s) = \{i\} \cup \{j \text{ s.t. } s_j \text{ appears in } s \text{ or } t\}$.

[4]Again, we only show the proof for the first equivalence. For the second equivalence, a similar proof holds for $\mathbf{CZ}_{ij}\mathbf{Z}_j^{s_\Gamma} = \mathbf{Z}_j^{s_\Gamma}\mathbf{CZ}_{ij}$.

[5]*Note*: For the following we will use the notation $|+\{\alpha\}\rangle := |+_\alpha\rangle$ to ease reading.

Definition 16 (Denotational Semantics [27, pg 13, Definition 4])**:** *A pattern $\mathcal{P}$ with m measurements realises a map, T, on density matrices, ρ, given by $T(\rho) := \sum_{s\in\{0,1\}^m} \mathbf{A}_s^\dagger \mathbf{A}_s$, where $\mathbf{A}_s$ is the branch map of $\mathcal{P}$. We define $[\![\mathcal{P}]\!] := T$, where T is the map realised by $\mathcal{P}$.*

Definition 17 (Pattern Equivalence [27, pg 27, Definition 10])**:** *We define an equivalence relation $\equiv$ on patterns by taking all the equivalences, adding the equivalence $X_1^s Z_1^t \equiv Z_1^t X_1^s$ and generating the smallest equivalence relation.*

Proposition 3: *All patterns that are equivalent by $\equiv$ are equal in the denotational semantics. That is, $\mathcal{P}_1 \equiv \mathcal{P}_2 \Rightarrow [\![\mathcal{P}_1]\!] = [\![\mathcal{P}_2]\!]$. [27, pg 27, Proposition 11]*

3.2.2 Rewrite Rules

The above rules are shown with $\equiv$, however, we can *orient* them in one way in order to establish a *term rewriting system.* These orientations are shown in Table 3.1. By fixing these orientations, we will show that it guarantees termination for process known as *standardization* (c.f. Section 3.2.3).

	Equivalence	Rewrite Rule
EX:	$E_{ij}X_i^s \equiv X_i^s Z_j^s E_{ij}$	$E_{ij}X_i^s \Longrightarrow X_i^s Z_j^s E_{ij}$
	$E_{ij}X_j^s \equiv X_j^s Z_i^s E_{ij}$	$E_{ij}X_j^s \Longrightarrow X_j^s Z_i^s E_{ij}$
EZ:	$E_{ij}Z_i^s \equiv Z_i^s E_{ij}$	$E_{ij}Z_i^s \Longrightarrow Z_i^s E_{ij}$
	$E_{ij}Z_j^s \equiv Z_j^s E_{ij}$	$E_{ij}Z_j^s \Longrightarrow Z_j^s E_{ij}$
MX:	$[M_i^\alpha]^s X_i^r \equiv {}^t[M_i^\alpha]^{s+r}$	$[M_i^\alpha]^s X_i^r \Longrightarrow {}^t[M_i^\alpha]^{s+r}$
MZ:	${}^t[M_i^\alpha]^s Z_i^r \equiv {}^{t+r}[M_i^\alpha]^s$	${}^t[M_i^\alpha]^s Z_i^r \Longrightarrow {}^{t+r}[M_i^\alpha]^s$
NA:	$N_i A_j \equiv A_j N_i$	$N_i A_j \Longrightarrow A_j N_i$
EA:	$E_{ij} A_k \equiv A_k E_{ij}$	$E_{ij} A_k \Longrightarrow A_k E_{ij}$
AX:	$A_j X_i \equiv X_i A_j$	$A_j X_i \Longrightarrow X_i A_j$
AZ:	$A_j Z_i \equiv Z_i A_j$	$A_j Z_i \Longrightarrow Z_i A_j$

Table 3.1: Orienting the equivalences to give term rewriting rules.

To implement term rewriting, we have an abstract `Rule` constructor from which concrete rule constructors `EXRule`, `EZRule`, ..., `AZRule` inherit. These constructors are shown in Listings 3.8. What is quite key to the design of this code is that we exploit the fact that the left hand side of each rewrite rule has exactly two commands, `l` and r [6].

Each rule implements `canRewrite` that has code that checks whether arguments provided in `l` and `r` pattern match the left hand side of the rule. It returns true if there is a pattern match. Each rule also implements `rewriteWith` that returns an array of commands representing the right-hand side of the rule assuming `canRewrite` is true. We also have `canRewriteInstrsAtIndex` and `rewriteInstrsAtIndex` that takes a pattern, `pat`, and an index, `idx`, and calls `canRewrite`/`rewriteWith` with the two instructions that start at position `idx` in the pattern's instruction array.

```
function Rule() {
  this.name = '';
  this.canRewrite = function (l, r) { ... };
  this.rewriteWith = function (l, r) { ... };

  this.canRewriteInstrsAtIndex = function (pat, idx) { ... };
  this.rewriteInstrsAtIndex = function (pat, idx) { ... };
}
// EZ Rule
function EZRule() {...}
EXRule.prototype = new Rule();
// ... other rules ...
```

Listing 3.8: constructors that implement rewrite rules

We also have the function `evaluatePossibleRules` in Listings 3.9 that finds all possible rewrite rules that pattern match a given command sequence for pattern. That is, it finds all rules that are applicable. Here exploit the fact that two rules cannot apply at the same index. We use a map `possible` that maps *index* $\mapsto$ *rule* if a rule at that index applies. If also fills in the inverse of this map `possibleInv`. This maps a rule's name to an array of indices. Each index in the array tells us where the rule can be applied. An empty array of indices expresses that the rule cannot be applied at all.

```
function evaluatePossibleRules(pattern, possible) {
    ...
    for (var i = 0; i < pattern.instructions.length; i++) {
        forEach(rules, function (rule) {
            if (rule.canRewriteInstrsAtIndex(pattern, i)) {
                possible[i] = rule;
                possibleInv = ....
            }
```

[6]It would be no problem to implement a rule that has more than two commands on the left-hand side. Javascript functions are very flexible support function calling with a variable argument count.

```
        });
    }
}
```

Listing 3.9: finding all possible rewrite rules that pattern match

It should come as no surprise that these rewrite rules preserve the underlying linear map of the pattern (given that the equivalences preserve the linear map).

Proposition 4: *Whenever $\mathcal{P} \Rightarrow^* \mathcal{P}'$, $[\![\mathcal{P}]\!] = [\![\mathcal{P}']\!]$ [27, Proposition 8, pg 24]*

See [27, Proposition 8, pg 24] for the proof.

3.2.3 Standard Form

Definition 18 (Local Confluence [6, pg 28, Definition 2.7.1])**:** *A term writing system is locally confluent if for all $\mathcal{P}$, if $\mathcal{P} \Rightarrow \mathcal{P}_1$ and $\mathcal{P} \Rightarrow \mathcal{P}_2$, then there is some $\mathcal{P}'$ such that $\mathcal{P}_1 \Rightarrow^* \mathcal{P}'$ and $\mathcal{P}_2 \Rightarrow^* \mathcal{P}'$*

Theorem 2 (Local Confluence of $\Rightarrow$)**:** *$\Rightarrow$ is locally confluent [27, Section 5.5, pg 26, Proof of Theorem 5].*

Proof outline: Danos et al. show local confluence in [27, Section 5.5, pg 26], by searching for *critical pairs*, cases where the left-hand sides of the rewrite rules overlap. The commands that are common to both left-hand sides form what is called the *critical term*. This critical term rewritten can always be rewritten in two ways. We often say is has two sides for it can branch in two directions when rewritten with $\Rightarrow$. If both sides of the critical pair can reduce to the same term, we say the critical term is *convergent*. The critical pair lemma states that a term rewriting system is locally confluent if all critical terms are convergent. By showing that all critical pairs of $\Rightarrow$ are convergent, Danos et al. therefore show that $\Rightarrow$ is locally confluent.

Definition 19 (Terminating [6, pg 28, Definition 2.1.3])**:** *A term rewriting system is terminating if all rewrite sequences beginning with a pattern $\mathcal{P}$ terminate after finitely many steps (there are no infinite descending chains: $\mathcal{P}_1 \Rightarrow \mathcal{P}_2 \Rightarrow ...$).*

Theorem 3 (Termination of $\Rightarrow$)**:** *$\Rightarrow$ is terminating. [27, Section 5.5, pg 26, Proof of Theorem 4]*

Proof outline: Danos et al. in [27, Section 5.5, pg 26, Proof of Theorem 4] find a well-founded lexicographic ordering $\prec$ on $\mathbb{N}^k$ for some k that is fixed for a given $\mathcal{P}$.

On each rewrite the order according to $\prec$ strictly decreases. That is $\mathcal{P} \Rightarrow \mathcal{P}'$ implies $\mathcal{P} \succ \mathcal{P}'$. Since the ordering is well-founded, there can be no infinitely decreasing chains, $\mathcal{P}_1 \succ \mathcal{P}_2 \succ \mathcal{P}_3 \succ ...$, so the rewrite proof must terminate after a finite number of rewrites.

Definition 20 (Confluence [6, pg 28, Definition 2.1.3])**:** *A term writing system is confluent if for all $\mathcal{P}$, if $\mathcal{P} \Rightarrow^* \mathcal{P}_1$ and $\mathcal{P} \Rightarrow^* \mathcal{P}_2$, then there is some $\mathcal{P}'$ such that $\mathcal{P}_1 \Rightarrow^* \mathcal{P}'$ and $\mathcal{P}_2 \Rightarrow^* \mathcal{P}'$*

Lemma 1 (Newman's Lemma [28])**:** *If $\Rightarrow$ is locally confluent and terminating, then it is confluent.*

Theorem 4 (Confluence of $\Rightarrow$)**:** *$\Rightarrow$ is confluent.*

Proof: In accordance with Newman's Lemma, by showing that $\Rightarrow$ is locally confluent and terminating, it follows that $\Rightarrow$ is confluent.

Now that we have a confluent, terminating rewrite system, it makes sense to talk about a pattern being in a *standard form.* We say that a pattern is $\mathcal{P}$ is in *standard form* (or in *canonical form* or in *normal form*) iff there is no $\mathcal{P}'$ such that $\mathcal{P} \Rightarrow \mathcal{P}'$. In other words, a pattern is in standard form if it cannot be rewritten. One property of patterns in the standard form is that they order instructions such that Ns come first, then E's, then M's and finally Xs and Zs. So instead of saying standard form (or canonical form or normal form), we may instead say a pattern is in *NEMC form.*

Definition 21 (NEMC Form [27, Section 5.5, pg 26, Definition 9])**:** *A pattern is in NEMC form if its commands occur in the order Ns first, then Es, then Ms, then finally Cs.*

Theorem 5: *For all $\mathcal{P}$, there exists a unique standard form $\mathcal{P}'$ such that $\mathcal{P} \Rightarrow \mathcal{P}'$ and $\mathcal{P}'$ is in NEMC form. [27, pg 26, Theorem 5]*

The procedure of writing a pattern to NEMC form/standard form is called *standardisation* (or normalisation) Danos et al. [27, Section 5.5, pg 28] describe the following algorithm to compute the *NEMC form* for any pattern.

Algorithm 1: *Input: A pattern $\mathcal{P}$ with command sequence $A_M, ..., A_1$.*
Output: An equivalent pattern $\mathcal{P}'$ in NEMC form.

(i) Commute all preparation commands to the right side
(ii) Commute all correction commands to the left side using EC and MC rewriting rules

(iii) *Commute all the entanglement commands to the right side after the preparation commands*

Danos et al. [27, Section 5.5, pg 28] prove that this algorithm has a worst case running time of $O(N^5)$ rewrite steps, where $N = |V|$ is the number of qubits in the pattern. Listing 3.10 gives the implementation of the algorithm. The code repeatedly called `evaluatePossibleRules`. This fills in `possible` and `possibleInv`. We then apply attempt to apply the rewrite rules that are associated for the current phase. This code also calls `performRewrite` that, in essence, adds the resulting rewritten pattern to a proof (array of patterns), along with some information about the rewrite step itself. This function is shown in Listings 3.11.

```
function standardForm(pattern) {
  var possible = {};

  // Phase (i), repeatedly apply NA
  evaluatePossibleRules();
  while (possible[NARule.name] != undefined) {
      performRewrite(possible[NARule.name][0]);
      evaluatePossibleRules(pattern, possible);
  }

  // Phase (ii), repeated apply AX, AZ, EX, EZ, MX, MZ
  evaluatePossibleRules(pattern, possible);
  var endOfPhase2 = false;
  var phase2rs = [CXRule, CZRule, EXRule, EZRule, MXRule, MZRule];
  while (!endOfPhase2) {
      // Linear search to see if any rule in phase2rs is applicable
      var i = 0; // index to phase2rs
      while (i < phase2rs.length &&
        possible[phase2rs[i].name] == undefined) {
          i++;
      }
      if (i == phase2rs.length) {
          // None of rules can be applied.
          endOfPhase2 = true;
      } else {
          // There is some rule that can be applied,
          // so apply it, then start all again
          performRewrite(possible[phase2rs[i].name][0]);
          evaluatePossibleRules(pattern, possible);
      }
  }

  // Phase (iii), repeated apply EA
  while (possible[EARule.name] != undefined) {
      performRewrite(possible[EARule.name][0]);
      evaluatePossibleRules(pattern, possible);
  }
};
```

Listing 3.10: implementing the standard form algorithm

```
function performRewrite(index) {
    // Add info about rewrite step (what rule, where it was applied)
    ...
    // Add the resulting rewritten pattern to proof
    proof.push(chosenRule.rewrittenInstrsAtIndex(currentPattern,
   index));
};
```

Listing 3.11: performing a rewrite in the standard form algorithm

3.3 Pattern Composition

We now look at how we can combine patterns together to form larger patterns. There are two ways to combine patterns, sequential composition and parallel composition. We will that with universality, we can systematically find patterns that implement any unitary operator $\mathbf{U}$*. By using two particular patterns,* $\mathcal{J}(\alpha)$ *and* $\mathcal{CZ}(i,j)$*, as building blocks, if we glue together with sequential/parallel compositions, then we can find new patterns whose computation implements* $\mathbf{U}$*.*

3.3.1 Sequential Composition

A sequential composition combines two patterns so that we perform one followed by another. We require that the output of the first pattern is equal (compatible) with the input of the second pattern in order for sequential composition to work. Furthermore, we require that the only overlap in qubits is the overlap between the output of the first and the input of the second. More formally,

Definition 22 (Sequential Composition [27, Definition 2, pg 9])**:** *Given two pattens* $\mathcal{P}_1 = (V_1, I_1, O_1, A_1)$ *and* $\mathcal{P}_2 = (V_2, I_1, O_1, A_2)$*, if* $V_1 \cap V_2 = O_1 = I_2$ *then the sequential composition is defined as a new pattern* $\mathcal{P}_2 \circ \mathcal{P}_1 = (V_2 \cup V_1, I_1, O_2, A_2A_1)$*, where* A_2A_1 *denotes the concatenation of commands* A_1 *and* A_2*.*

A composite pattern may be written as either $\mathcal{P}_2 \circ \mathcal{P}_1$, or as $\mathcal{P}_2\mathcal{P}_1$. We always follow the convention that it is the pattern on the *right* that is executed *first*. That is, it is the right pattern whose whose commands are put on the right of the concatenation and therefore executed first via the operational semantics [7].

Example:
Let $\mathcal{J}(\phi)(i,j) = (\{i,j\},\{i\},\{j\}, X_j^{s_i} M_i^{\phi} E_{ij} N_j)$.
$\mathcal{P}_1 = \mathcal{J}(0)(1,2) = (\{1,2\},\{1\},\{2\}, X_2^{s_1} M_1^0 E_{12} N_2)$.
$\mathcal{P}_2 = \mathcal{J}(0)(2,3) = (\{2,3\},\{2\},\{3\}, X_3^{s_2} M_2^0 E_{23} N_3)$.
Then we find $\mathcal{P}_2 \circ \mathcal{P}_1$ because $V_1 \cap V_2 = \{1,2\} \cap \{2,3\} = \{2\} = O_1 = I_2$.
The composite pattern is therefore defined and given by:

$$\mathcal{P}_2 \circ \mathcal{P}_1 = (V_1 \cup V_2, I_1, O_2) = (\{1,2,3\},\{1\},\{3\})$$

A visual interpretation of this composition is shown in Figure 3.4. The composition leaves the input of $\mathcal{P}_1$ alone and the output of $\mathcal{P}_2$ alone. And the identical first-output I_1 and second-input O_1 consisting of $\{2\}$ join together.
The input-output status of qubits in I_1 and O_2 remain the same. To work out the input-output status of qubit 2, we only look at I_1 and O_2. We see that $2 \notin I_1, 2 \notin I_2$, and so 2 is an auxiliary qubit (drawn with a filled in circle, without a surrounding box).

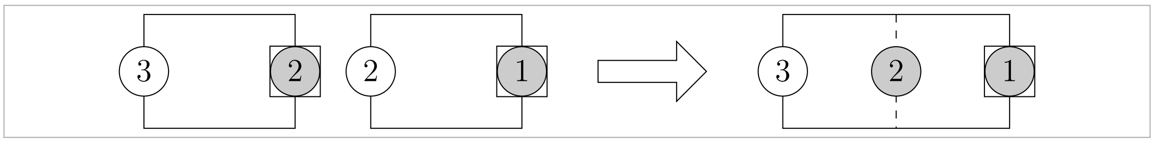


Figure 3.4: A diagram giving a visual interpretation of a composition.

Example:
Let $\mathcal{R}_x(\alpha)(1,2,3) = (\{1,2,3\},\{1\},\{3\}, X_3^{s_2} Z_3^{s_1} [M_2^{-\alpha}]^{s_1} M_1^x E_{23} E_{12} N_2 N_1)$
and $\mathcal{R}_z(\beta)(3,4,5) = (\{2,3,4\},\{3\},\{5\}, X_5^{s_4} Z_5^{s_3} M_4^x M_3^{-\beta} E_{45} E_{34} N_3 N_4)$
We can find the composite pattern $\mathcal{R}_z(\alpha)(3,4,5) \circ \mathcal{R}_x(1,2,3)$ because $O_1 = I_2 = \{3\}$ and this qubit are the only qubit that $\mathcal{R}_x$ and $\mathcal{R}_z$ have in common: $O_1 = I_2 = V_1 \cap V_2$. The composition is given by:

$$\mathcal{P}_2 \circ \mathcal{P}_1 = (V_1 \cup V_2, I_1, O_2)(\{1,2,3,4,5\},\{1\},\{5\} \quad \begin{array}{l} X_5^{s_4} Z_5^{s_3} M_4^x M_3^{-\beta} E_{45} E_{34} N_3 N_4 \\ X_3^{s_2} Z_3^{s_1} [M_2^{-\alpha}]^{s_1} M_1^x E_{23} E_{12} N_2 N_1) \end{array}$$

---

[7] This is unlike the use of $\circ$ to denote the composition of functions $f_2 \circ f_1$ where it is ambiguous as to which function is applied first

Example:
Let us consider a trivial example involving: $\mathcal{I}(i) = (\{i\}, \{i\}, \{i\}, \epsilon)$, where ϵ denotes the empty command sequence. $\mathcal{I}(1) \circ \mathcal{I}(1)$ is defined because $V_1 \cap V_2 = \{1\} = O_1 = I_2$ The composite pattern is $\mathcal{I}(1) \circ \mathcal{I}(1) = \mathcal{I}(1)$. Notice that $\mathcal{I}(2) \circ \mathcal{I}(1)$ is *not* defined because $O_1 = \{1\} \neq I_2 = \{2\}$.

Notice that when we compose patterns, we by knowing whether $q \in I_1$ and $\in O_2$, we can work out the input-output status of a qubits being merged (those in $\in O_1$ or $\in I_2$). Figure 3.5 shows all four cases. We take advantage of this property when we implement sequential composition. We define `mergemap` (Listing 3.12) that will maps $(io_2, io_1) \mapsto newio$ where io_1, io_2 are the the input-output statuses of a merge qubit in $\mathcal{P}_1, \mathcal{P}_2$ respectively and *newio* is new the input-output status of the merge qubit in the composite pattern.

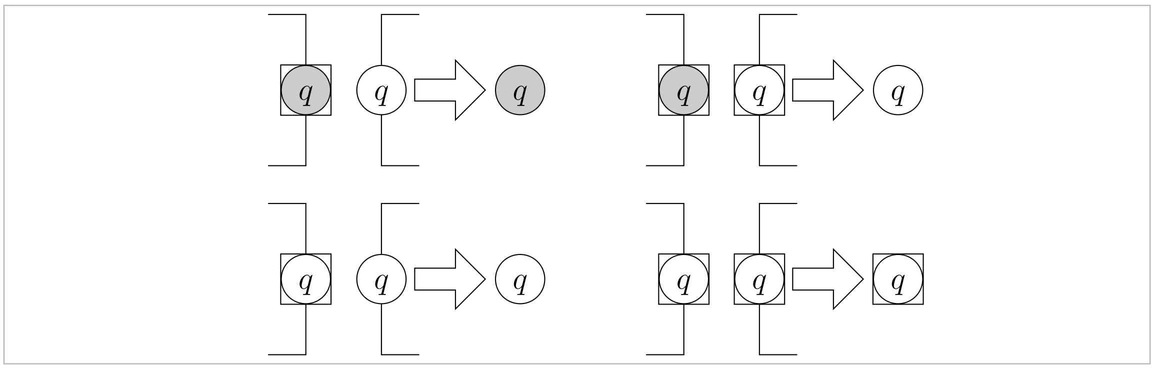


Figure 3.5: There are four cases to consider when finding the input-output status of merge qubits. We encode these four cases into `mergemap` that will maps $(io_2, io_1) \mapsto newio$.

```
var mergemap = {};
mergemap[QubitIO.In] = {}; mergemap[QubitIO.InOut] = {};
mergemap[QubitIO.In][QubitIO.Out] = QubitIO.Neither;
mergemap[QubitIO.In][QubitIO.InOut] = QubitIO.In;
mergemap[QubitIO.InOut][QubitIO.Out] = QubitIO.Out;
mergemap[QubitIO.InOut][QubitIO.InOut] = QubitIO.InOut;
```

Listing 3.12: using a map to find the input-output status of merge qubits

Once we have defined `mergemap`, finding the composition is relatively simple (Listing 3.13). We define `newqubits` that stores the input-output statuses of qubits in the composite pattern (mapping *qubitname* $\mapsto$ *iostatus*) To compute `newqubits`, we essentially (i) copy over all the statuses for qubits not being merged (ii) use the `mergemap` to determine the statuses for qubits being merged.

```
function evaluate() {
  // check outputs of l = inputs of r
  // check V1 intersect V2 = O1
  ...
  var newqubits = {};
  for (var q in p1.qubits) {
      var io1 = p1.qubits[q];
      if (io1 == QubitIO.In || io1 == QubitIO.Neither) {
          newqubits[q] = io1;
      } else {
          var io2 = p2.qubits[q];
          newqubits[q] = mergemap[io2][io1];
      }
  }

  for (var q in p2.qubits) {
      var io2 = p2.qubits[q];
      if (io2 == QubitIO.Out || io2 == QubitIO.Neither) {
          newqubits[q] = io2;
      }
  }
  return new pattern.Pattern(p2.instructions.concat(p1.instructions
   ), newqubits);
});
```

Listing 3.13: evaluating sequential composition

3.3.2 Parallel Composition

We now introduce a second notion of composition, known as *parallel composition*. Parallel composition doesn't have the input-output merge constraint that sequential composition does. In rather the opposite fashion, we require that there is independence between the two patterns by making sure neither pattern operates on the other's qubits ($V_1 \cap V_2 = \emptyset$).

Definition 23 (Parallel Composition [27, Definition 3, pg 10])**:** *Given two pattens* $\mathcal{P}_1 = (V_1, I_1, O_1, A_1)$ *and* $\mathcal{P}_2 = (V_2, I_1, O_1, A_2)$*, if* $V_1 \cap V_2 = \emptyset$ *then the parallel composition is defined as a new pattern* $\mathcal{P}_2 \otimes \mathcal{P}_1 = (V_2 \cup V_1, I_2 \cup I_1, O_2 \cup O_1, A_2A_1)$*, where* A_2A_1 *denotes the concatenation of commands* A_1 *and* A_2*.*

Despite being called a *parallel* composition, we still end up having a concatenation of commands which seems somewhat sequential. The operational semantics will still execute each pattern one after the other. This may seem unintuitive. To clarify, note that the parallelism here comes from the *independence of the qubits*. If we were able to physically implement The Measurement Calculus, then the notion of parallelism expressed here suggests that we could perform the execution of both patterns at the time on the physical system.

Example:
Let $\mathcal{H}(1,2) = (\{1,2\},\{1\},\{2\},X_2^1 M_1^x E_{12} N_2)$,
and $\mathcal{H}(3,4) = (\{3,4\},\{3\},\{4\},X_4^3 M_3^x E_{34} N_3)$. Since the qubits are disjoint, $\{1,2\} \cap \{3,4\} = \emptyset$, the parallel composition is defined and so we can find $\mathcal{H}(3,4) \otimes \mathcal{H}(1,2) = (\{1,2,3,4\},\{1,3\},\{2,4\},X_4^3 M_3^x E_{34} N_3 X_2^1 M_1^x E_{12} N_2)$. A sketch of this composition is given in Figure

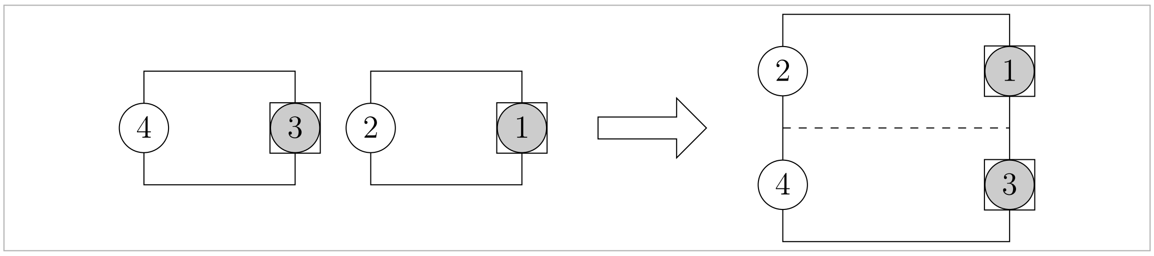


Figure 3.6: A diagram giving a visual interpretation of a parallel composition.

Notice that we can also find $\mathcal{H}(1,2) \otimes \mathcal{H}(3,4)$. Instead, the order in which we concatenation the commands has swapped, so we have parallel composition of $\mathcal{H}(1,2) \otimes \mathcal{H}(3,4) = (\{1,2,3,4\},\{1,3\},\{2,4\},X_2^1 M_1^x E_{12} N_2 X_4^3 M_3^x E_{34} N_3)$.

Example:
Let $\mathcal{I}(1) = (\{1\},\{1\},\{1\},\epsilon)$, where ϵ denotes the empty command sequence and $\mathcal{H}(i,j) = (\{2,3\},\{2\},\{3\},X_3^{s_2} M_2^0 E_{23} N_3)$ Since the qubits are disjoint, $\{2,3\} \cap \{1\} = \emptyset$, we can find both $\mathcal{P}_2 \otimes \mathcal{P}_1$ and $\mathcal{P}_1 \otimes \mathcal{P}_2$, which in this case, are both equal to $(\{1,2,3\},\{1,2\},\{1,3\},X_3^{s_2} M_2^0 E_{23} N_3)$.

The implementation for finding the parallel composition (Listing 3.14) is simpler in comparison to the implementation for sequential composition. Again, we define `newqubits` that stores the input-output statuses of qubits in the resulting composite pattern. To compute `newqubits`, we copy over the statuses for all the qubits in $\mathcal{P}_1$, then repeat for the qubits in $\mathcal{P}_2$.

```
function evaluate() {
    // check V1 intersect V2 = emptyset
    ...

    // join the two disjoint qubit maps together
    var newqubits = {};
    for (var q in p2.qubits) {
        newqubits[q] = p2.qubits[q];
    }
    for (var q in p1.qubits) {
        newqubits[q] = p1.qubits[q];
    }
    return new pattern.Pattern(p2.instructions.concat(p1.
   instructions), newqubits);
});
```

Listing 3.14: evaluating parallel composition

3.3.3 Universality

Theorem 6 (Compositional Denotational Semantics [27, Definition 2, pg 9])**:** *The denotational semantics is compositional.* $[\![\mathcal{P}_1 \circ \mathcal{P}_2]\!] = [\![\mathcal{P}_2]\!][\![\mathcal{P}_1]\!]$ *and* $[\![\mathcal{P}_1 \otimes \mathcal{P}_2]\!] = [\![\mathcal{P}_2]\!] \otimes [\![\mathcal{P}_1]\!]$

We now consider two patterns that act as *generators*, in the sense that with sequential composition and parallel composition, they can generate patterns that implement *all* unitary matrices. These two patterns are the *one-parameter-family* generator $\mathcal{J}(0)$ and *the controlled-Z* operator $\mathcal{CZ}$ defined by:

$$\begin{array}{lll} \mathcal{J}(\alpha)(i,j) & := & (\{i,j\},\{i\},\{j\},X_j^{s_i}M_i^{-\alpha}E_{ij}N_j) \\ \mathcal{CZ}(i,j) & := & (\{i,j\},\{i,j\},\{i,j\},E_{ij}) \end{array}$$

These two patterns implement, respectively, the two unitary operators given explicitly below

$$\mathbf{J}(\alpha) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & e^{i\alpha} \\ 1 & -e^{i\alpha} \end{bmatrix} \quad \mathbf{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Theorem 7 (Universality [27, Section 5.4, pg 19, Theorem 2])**:** *The set* $\{\mathcal{J}(\alpha)\}$ *with* $\circ$ *generates patterns for all one-qubit unitary matrices* $\mathbf{U} \in \mathbb{C}^{2\times 2}$. *The set* $\{\mathcal{J}(\alpha), \mathcal{CZ}\}$ *with* $\circ$ *generates patterns for all two-qubit unitary matrices* $\mathbf{U} \in \mathbb{C}^{4\times 4}$.

The set $\{\mathcal{J}(\alpha), \mathcal{CZ}\}$ *with* $\circ$ *and* $\otimes$ *generates patterns for all unitary matrices* $\mathbf{U} \in \mathbb{C}^{n\times n}$.

Proof Sketch: Danos et al. state in [27, Section 5.4, pg 17] that the set $\{\mathbf{J}(\alpha), \mathbf{CZ}\}$ generate all unitary matrices. They manage to find patterns $\mathcal{J}(\alpha), \mathcal{CU}$ that implement $\mathbf{J}(\alpha), \mathbf{CU}$ and therefore implement $\mathbf{J}(\alpha), \mathbf{CZ}$. Danos et al. give $\mathcal{J}(\alpha), \mathcal{CU}$ written as a composition of patterns from $\{\mathcal{J}(\alpha), \mathcal{CZ}\}$. Hence we can conclude that $\{\mathcal{J}(\alpha), \mathcal{CZ}\}$ generate patterns that implement all unitary matrices.

Unitary Matrix	Pattern
$\mathbf{J}(\alpha)$	$\mathcal{J}(\alpha)$
$\mathbf{P}(\alpha) = \mathbf{J}(0)\mathbf{J}(\alpha)$	$\mathcal{J}(0) \circ \mathcal{J}(\alpha)$
$\mathbf{X} = \mathbf{J}(\pi)\mathbf{J}(0)$	$\mathcal{J}(\pi) \circ \mathcal{J}(0)$
$\mathbf{Z} = \mathbf{J}(0)\mathbf{J}(\pi)$	$\mathcal{J}(0) \circ \mathcal{J}(\pi)$
$\mathbf{H} = \mathbf{J}(0)$	$\mathcal{J}(0)$

Table 3.2: Fundamental unitary matrices and the corresponding patterns giving their implementation. The Universality theorem gives the result that unitary matrices can be implemented with a pattern expressed compositions of $\{\mathcal{J}(\alpha), \mathcal{CZ}\}$. Indeed, all the above patterns are build up from just $\{\mathcal{J}(\alpha), \mathcal{CZ}\}$.

Example:
We can represent the CNOT gate $\mathbf{CX} = (\mathbf{I} \otimes \mathbf{H})(\mathbf{CZ})(\mathbf{I} \otimes \mathbf{H})$ by using the pattern $(\mathcal{I}(1) \otimes \mathcal{H}(3,4)) \circ \mathcal{CZ}(1,3) \circ (\mathcal{I}(1) \otimes \mathcal{H}(2,3))$

Example:
We can represent the Pauli-Y gate $\mathbf{Y} = \mathbf{XZ}$ using the pattern $\mathcal{J}(\pi)(4,5) \circ \mathcal{J}(0)(3,4) \circ \mathcal{J}(0)(2,3) \circ \mathcal{J}(\pi)(1,2)$.

3.4 Flow Analysis

> *We previously discussed parallelism when looking at Parallel composition. We now look at the concept of flow which formally depicts how parallel our pattern is formalising strategies for performing measurements and corrections in parallel.*

3.4.1 Causal Flow

Definition 24 (Open Graph [33, Definition 1])**:** *An open graph is a triplet (G, I, O) where $G = (V, E)$ is a undirected graph, and $I, O \subseteq V$ are respectively called input and output vertices.*

We define a partial order that tells us what order we should measure qubits $i \in V(G) - O$ for a particular strategy. We prefer strategies whose order $\prec$ gives rise to a smaller depth $d^{\prec}$ so that we can perform more corrections in parallel.

Definition 25 (Layers)**:** *For a given open graph (G, I, O) and a given strict partial ordering $\prec$ over $V(G)$ let*

$$V_k^{\prec} = \begin{cases} \max_{\prec}(V(G)) & if\ k = 0 \\ \max_{\prec}(V(G) - V_{k-1}^{\prec}) & if\ k > 0 \end{cases}$$

where $\max_{\prec}(X) = \{u \in X\ s.t.\ \forall v \in X, \neg(u \prec v)\}$ is the set of the maximal elements of X with respect to $\prec$.

The layers V_k induce a partition of V. Qubits in a given layer are all incomparable with respect to $\prec$. This means that there is no constraint on the order, and so we can measure all of them in parallel.

Example:
Example of computing V_k

Definition 26 (Causal Flow [31])**:** *$(g, \prec)$ is a Causal flow of (G, I, O) where $g : (V(G) - O) \longrightarrow (V(G) - I)$ and $\prec$ is a strict partial order over $V(G)$, if and only if*

1. *$i \prec g(i)$*
2. *if $j \in N(g(i))$ then $j = i$ or $i \prec j$*
3. *$i \in N(g(i))$*

where $V(G), E(G)$ are the vertices, edges of G, and $N(v)$ is the neighborhood of v, defined by: $N(v) = \{\ u \mid (v, u) \in E(G)\}$.

Example:
For the open graph $(G, \{1, 2\}, \{5, 6, 7\})$ shown in Figure 4.4(a), there is a causal flow where $g = (1 \mapsto 3,\ 2 \mapsto 4,\ 3 \mapsto 5,\ 4 \mapsto 7)$ and $\{1, 2\} \prec \{3, 4\} \prec \{5, 6, 7\}$

Example:
For the open graph $(G, \{1,2,3\}, \{7,8,9\})$ shown in Figure 4.4(b), there is a causal flow where $g = (1 \mapsto 4,\ 2 \mapsto 5,\ 3 \mapsto 5,$
$(4 \mapsto 7,\ 5 \mapsto 8,\ 6 \mapsto 9$, and $1 \prec 2 \prec 3 \prec \{4,5,6\} \prec \{7,8,9\}$

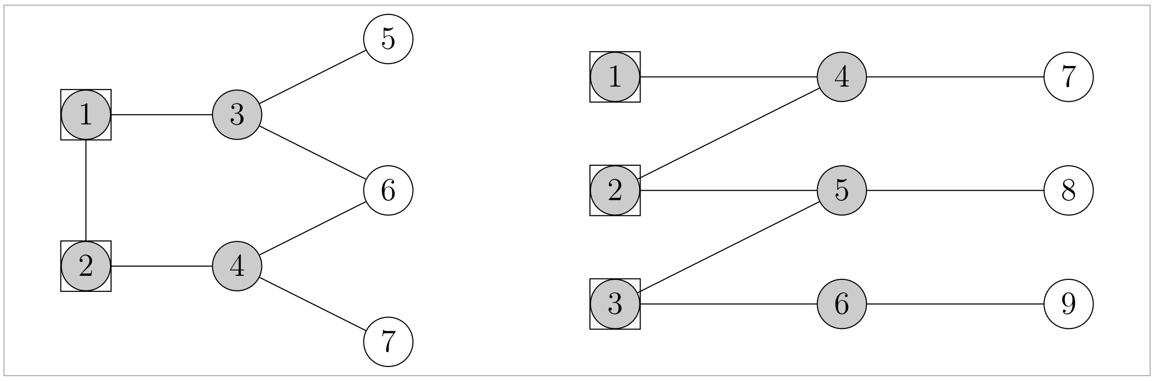


Figure 3.7: Two open graphs. Left: (a) a graph with seven nodes. Right: (b) a graph with nine nodes. Both of these graphs have a causal flow.

We can compute the causal flow for any open graph.

Theorem 8: *There exists a polynomial time algorithm that decides whether a given open graph has a causal flow, and that outputs a causal flow if it exists. [30, p862, Theorem 1]*

Mhalla and Perdrix [30, Section 3, pg 5] give a polynomial time algorithm for computing the causal flow. This is shown in Listing 3.15. Our implementation stores functions l, g using an objects `l`,`g`. Note that our implementation doesn't use the In parameter of Flowaux. This is because the algorithm doesn't modify In in any recursive calls and so can be safely removed.

To make the first recursive call, we need to work out the sets O, $C = O - I$. We create arrays `initialc`, `initialout`, which will store the names of qubits in O and $O - I$ respectively. We use a for loop to iterate over all the qubits, check their input-output status add them accordingly to `initialc`, `initialout`. For all qubits the in O, we perform `l[q] = 0` to implement the assignment $l(v) := 0$ as seen in lines 5-7 of algorithm A.1. We then make our initial recursive call to the auxiliary function `flowAux(initialout, initialc,1)`.

```
function flow(qubits, adj) {
  var g = {}, l = {};

  // we compute numQubits = |V|
  // along with the initial values for out and C
  var numQubits = 0;
  var initialc = [], initialout = [];

  for (var q in qubits) {
    numQubits++;
    if (qio.IsOutput(qubits[q])) {
        l[q] = 0;
        initialout.push(q);
    }
    if (qubits[q] == qio.Out) {
        initialc.push(q);
    }
  }
  return [flowAux(initialout, initialc, 1),g,l];
  ...
```

Listing 3.15: flow implementation

We begin `flowAux` (Listing 3.16) by implementing $Out' := \emptyset;\ C' := \emptyset$ with `outprime := []`, `cprime := []`. We then iterate over all qubits $v \in C$ with a for loop over `c` giving us `v = c[i]`. We implement **if** $\exists u\ s.t.\ N(v) \cap (V - Out) = \{u\}$. by computing the intersection $N(v) \cap (V - Out)$ and storing the result in `intersect`. We then check `intersect.length`. If the length is 1, then there is one element in the intersection and then we indeed have $\exists u\ s.t.\ N(v) \cap (V - Out) = \{u\}$. The body of the if statement has a one-to-one translation.

```
...
function flowAux(out, c, k) {
  var outprime = [], cprime = [];

  for (var i = 0, n = c.length; i < n; i++) {
      var v = c[i];
      // compute N(v) n (V-Out)
      var intersect = [];
      for (var j = 0, n2 = adj[v].length; j < n2; j++) {
          var u = adj[v][j];
          if (out.indexOf(u) == -1) {
              intersect.push(u);
          }
      }
      // if there is some u s.t. N(v) n (V-Out) = {u}
      if (intersect.length == 1) {
          u = intersect[0];
          g[u] = v; l[u] = k;
          outprime.push(u);
          cprime.push(v);
      }
  }
```

Listing 3.16: flowAux implementation, the first half

The code shown in Listing 3.17 translates the remainder of `flowAux`. The first 8 lines translate directed from the pseudo-code. The majority of the work is in calculating the recursive call to `flowAux` in the outer else case. We compute $Out \cup Out'$ using `initialout = out.concat(outprime)` and compute $(C - C') \cap (Out' \cap V - In)$ using two for loops and storing the result `newc`. The first for loop works out $(C - C')$, the second work out $(Out' \cap V - In)$. We finally make the recursive call with this newly calculated parameters: `flowAux(newout, newc, k+1)`.

```
  ...
  if (outprime.length == 0) {
      if (out.length == numQubits) {
          return true;
      } else {
          return false;
      }
  } else {
      var newout = out.concat(outprime);
      var newc = [];
      for (i = 0, n = c.length; i < n; i++) {
          v = c[i];
          if (cprime.indexOf(v) == -1) {
              newc.push(v);
          }
      }
      for (i = 0, n = outprime.length; i < n; i++) {
          v = outprime[i];
          if (!qio.IsInput(qubits[v])) {
              newc.push(v);
          }
      }
      return flowAux(newout, newc, k+1);
  }
}
```

Listing 3.17: flowAux implementation, the remaining half

3.4.2 General Flow

The concept of a causal flow can be extended to a *general flow*. Now our successor function g maps a qubit to a *set of qubits*. Let $\mathcal{P}(S)$ denote the powerset of S.

Definition 27 (General Flow [32])**:** *$(g, \prec)$ is a gflow (generalized flow) of (G, I, O) where $g : (V(G) - O) \longrightarrow \mathcal{P}(V(G) - I)$ and $\prec$ is a strict partial order over $V(G)$, if and only if*

1. *$i \prec g(i)$*
2. *if $j \in Odd(g(i))$ then $j = i$ or $i \prec j$*
3. *$i \in Odd(g(i))$*

where $Odd(K)$ denotes vertices of G that have a odd K-neighbourhood, that is, the set of vertices that have an odd number of neighbours in K, $Odd(K) = \{u \in G \mid$ cardinality of $N(G) \cap K$ is odd$\}$.

Theorem 9: *There exists a polynomial time algorithm that decides whether a given open graph has a gflow, and that outputs a gflow if it exists. [30, p865, Theorem 2]*

Mhalla and Perdrix [30, Section 4, pg 8] give a polynomial time algorithm for computing the general flow. This is shown in algorithm X. We give an implementation of this algorithm with code in Listings 3.18 Note that again, we use slightly different parameters for the recursion in the code. Like the casual flow, we don't use In because it is left unchanged by a recursive call. For the same reason, we don't use Γ parameter in our code. `out` is now an object instead of an array and we add an additional parameter called `outsize` that measures how many qubits have been added to `out`. Also note that the general flow algorithm has one less parameter than the casual flow - we no longer have the C parameter.

Like before, our implementation stores the outputs l, g using an objects `l`,`g`. But now `g` maps a qubit name to *an array* of qubit names. We compute the parameters for the first recursive call - computing Out using object `initialout`. as well as the number of qubits added, `initialoutsize`.

```
function gflow(qubits, adj) {
  var g = {}, l = {};
  var numQubits = 0;
  var initialOut = {}, initialOutSize = 0;
  for (var q in qubits) {
      numQubits++;
      if (qio.IsOutput(qubits[q])) {
          initialOutSize++;
          initialOut[q] = true;
```

```
            l[q] = 0;
        }
    }
    ...
```

Listing 3.18: glow implementation

In `gflowAux`, we begin (Listing 3.19) with `c = []` that implements $c := \{\}$. Next we compute the adjacency matrix $\Gamma_{Out-In,V-Out}$. To do this, we first compute $Out - In$, $V - Out$ storing it in `Gamma`. Then for each qubit $i \in Out - In$, we assign an row index to so that each qubit has a unique row in Γ. Similarly, we assign for each qubit $j \in V - Out$, we asisgn a unique column index. These are stored in the maps `nameToRowIndex` and `nameToColIndex`. At the same time, we compute how many rows and columns $\mathbf{\Gamma}_{Out-In,V-Out}$ and store these values in `numRows` and `numCols`.

We translate **forall** $u \in V - Out$ **do** into a for loop. We store $\mathbb{I}_u$ in `rhs` and perform a gaussian elimination over $\mathbb{F}_2$ by calling `qm.gaussianOverF2(gamma,rhs,...)`. This attempts to solve the linear system $\mathbf{\Gamma}_{Out-In,V-Out}\mathbf{x} = \mathbb{I}_u$. If there is a solution, a return value of `true` is assigned to `success` and array `x` contains the solution.

```
function gflowAux(out, outsize, k) {
  var c = [];
  // compute Out - In, nameToColIndex, numCols
  // compute V - Out, nameToRowIndex, numRows
  // ...
  if (numRows > 0 && numCols > 0) {
    // compute gamma
    for (var u in nameToRowIndex) {
      // rhs = the indicator vector for {u}, I_{u}
      var rhs = Array.apply(null, new Array(numRows))
                            .map(Number.prototype.valueOf,0);
      rhs[nameToRowIndex[u]] = 1;
      // gaussian elimination
      var x = Array.apply(null, new Array(numRows))
                  .map(Number.prototype.valueOf,0);
      var success = qm.gaussianOverF2(gamma, rhs,
                                         numRows, numCols, x);
      if (success) {
        c.push(u);
        // g[u] = the set X_0
      }
      l[u] = k;
    }
  }
  ...
```

Listing 3.19: gflowAux implementation, the first half

The remaining code of `gflowAux` (Listing 3.20) has a direct translation. At the end we compute $Out := Out \cup C$ by adding all the qubits in array C to map Out (if they have not already been added). Then we make the recursive call: `return gflowAux(out ,outsize,k+1);`.

```
  ...
  if (c.length == 0) {
      if (outsize == numQubits) {
          return true;
      } else {
          return false;
      }
  } else {
      for (var r = 0, len = c.length; r < len; r++) {
          if (out[c[r]] == undefined) {
              out[c[r]] = true;
              outsize++;
          }
      }
      return gflowAux(out,outsize,k+1);
  }
}
```

Listing 3.20: gflowAux implementation, the remaining half

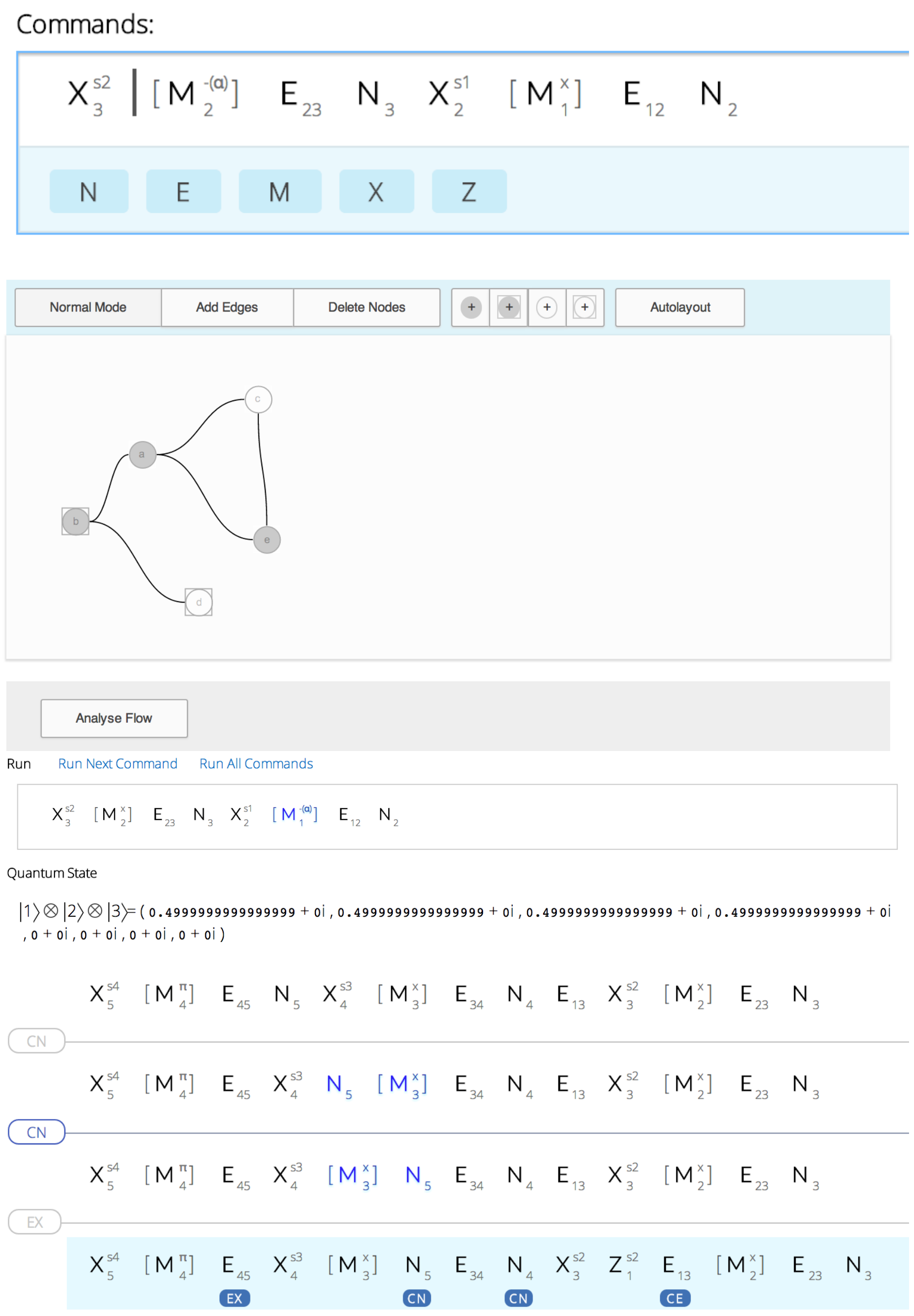
3.5 Graphical User Interface

The main reason we choose Javascript as our language is because its compatibility with web browsers makes it easy to produce highly interactive, attractive user interfaces. Here we summarise the features of our user interface.

- *Pattern Input*: The user can type keys on his keyboard and enter patterns. The user can also directly modify patterns. The backspace key deletes commands and the arrow keys can be used to navigate through a pattern. The user can select from pre-existing patterns. The user interface allows the editing of command sequences and trees of compositions. We wrote an LL(1) top-down descent parser to support the input of arbitrary angle expressions.
- *Proof assistant*: A proof assistant allows the user to input a pattern and construct a rewrite proof. At each state it shows which rewrite rules apply, and at what index. Simply by clicking, a user can perform a rewrite. For a given step, the assistant is able to highlight which commands were involved in the step. The assistant has a button that executes the standard form algorithm.

- *LaTeX Export*: We have a LaTeX exporter that can take a pattern and export it to LaTex. This reduces errors being made in typing up patterns. The exporter can also export rewrite proofs so that the user can be confident the proof is correct.
- *Simulator*: A simulator gives an interface that shows intermediate quantum states of a pattern execution. It shows the measurement outcomes, shows the branch tree and where the current execution is in the tree. The simulator is able to show the entanglement graph (see Definition 24) and show how the graph changes as execution is performed.
- *Flow Analyser*: We developed tools to allow a user to input an open graph and with a click of a button, the program runs the glow and gflow algorithms. It shows the user the layers of a graph induced by the flow and gives a visual representation of the flow. This work is presented in Section 3.4

A collection of screenshots highlighting these features are shown in Figure 3.8



X_3^{s2} | $[M_2^{-(\alpha)}]$ E_{23} N_3 X_2^{s1} $[M_1^{x}]$ E_{12} N_2

X_3^{s2} $[M_2^{x}]$ E_{23} N_3 X_2^{s1} $[M_1^{-(\alpha)}]$ E_{12} N_2

$|1\rangle \otimes |2\rangle \otimes |3\rangle =$ (0.4999999999999999 + 0i , 0.4999999999999999 + 0i , 0.4999999999999999 + 0i , 0.4999999999999999 + 0i , 0 + 0i , 0 + 0i , 0 + 0i , 0 + 0i)

X_5^{s4} $[M_4^{\pi}]$ E_{45} N_5 X_4^{s3} $[M_3^{x}]$ E_{34} N_4 E_{13} X_3^{s2} $[M_2^{x}]$ E_{23} N_3

X_5^{s4} $[M_4^{\pi}]$ E_{45} X_4^{s3} N_5 $[M_3^{x}]$ E_{34} N_4 E_{13} X_3^{s2} $[M_2^{x}]$ E_{23} N_3

X_5^{s4} $[M_4^{\pi}]$ E_{45} X_4^{s3} $[M_3^{x}]$ N_5 E_{34} N_4 E_{13} X_3^{s2} $[M_2^{x}]$ E_{23} N_3

X_5^{s4} $[M_4^{\pi}]$ E_{45} X_4^{s3} $[M_3^{x}]$ N_5 E_{34} N_4 X_3^{s2} Z_1^{s2} E_{13} $[M_2^{x}]$ E_{23} N_3

Figure 3.8: Screenshots of the User Interface

4 Evaluation

The following benchmark results were found by testing on Google Chrome version 35.0.1916.114 (Official Build 270117) running V8 3.25.28.16 on a MacBook Pro, 2.3 GHz Intel Core i7, 16GB 1600 MHz DDR3 with OSX 10.9.3 (13D65). Tests are run using `benchmark.js` [1] v1.0.0 using a timer with a resolution of 1 microsecond [2]. via `Performance.now()`. The tests are run repeatedly until a percentage uncertainty of 1% is achieved. Times are expressed as $\bar{x} \pm p$ where $\bar{x}$ is the mean and p is the 95% confidence interval given by: $p = (\bar{x} \pm t_{99,0.975}(s_{n-1}/\sqrt{N}))/\bar{x}$.

4.1 Term Rewriting

> *We first begin our evaluation with the implementation of term writing. We check rules by performing a linear search, checking whether a rule can be applied at the current instruction index. Consequently, we expect $O(n)$ growth where n is the number of instructions.*

4.1.1 Possible Rules

To evaluate our implementation of term rewriting, we perform several tests to ensure that our rewrite system correct detects where rules apply and correctly rewrites rules. Although we cannot formally prove our implementation is correct, by performing a thorough feature test, we can be confident that the implementation is reliable. Checking to rules and performing rewrites are very fast and so we omit any severe performance testing. To ensure that we have $O(n)$ performance, we time how long it takes to find the possible rules for $\mathcal{H}(n)$ (see Section 4.1.3 for the definition of $\mathcal{H}(n)$).

4.1.2 Standardisation

To evaluate our implementation of the standard form algorithm, we have several test cases shown in Table 4.1 and find the average time to find the standard form in Table 4.2. Here we are testing the patterns given in the Examples section of the Measurement Calculus paper [27, Section 5.6]. We have unit tests to test our rules

[1] benchmarkjs.com
[2] https://developer.mozilla.org/en-US/docs/Web/API/Performance.now()

more thoroughly than what is shown here, but Table 4.1 should give an idea. We ensure that the standard form that our program finds has a perfect match with the forms given in the paper.

Name	Pattern	NEMC Form
Teleportation	$X_3^{s_2} M_2^{-\beta} E_{23} N_2 X_2^{s_1} M_1^{\alpha} E_{12} N_3 N_2$	$X_3^{s_2} Z_3^{s_1} [M_2^{-\beta}]^{s_1} M_1^{\alpha} E_{23} E_{12} N_2 N_3 N_2$
X-Rotation	$X_3^{s_2} M_2^{-\alpha} E_{23} N_3 X_2^{s_1} M_1^{x} E_{12} N_2$	$X_3^{s_2} Z_3^{s_1} M_2^{s_1} M_1 E_{23} E_{12} N_3 N_2$
Z-Rotation	$X_3^{s_2} M_2 E_{23} N_3 X_2^{s_1} M_1 E_{12} N_2$	$X_3^{s_2} Z_3^{s_1} M_2^{s_1} M_1 E_{23} E_{12} N_3 N_2$
CNOT	$X_4^{s_3} M_3 E_{34} E_{13} X_3^{s_2} M_2 E_{23} N_4 N_3$	$X_4^{s_3} Z_4^{s_2} Z_1^{s_2} M_3^{s_2} M_2 E_{34} E_{13} E_{23} N_4 N_3$
Y-Rotation	$X_5^{s_4} M_4^{\pi} E_{45} N_5 X_4^{s_3} M_3^{0} E_{34} N_4$	$X_5^{s_4} Z_5^{s_3 s_2} [M_4^{\pi}]^{s_3 s_1} [M_3^{0}]^{s_2} M_2^{s_1} M_1^{\pi}$
	$X_3^{s_2} M_2^{0} E_{23} N_3 X_2^{s_1} M_1^{\pi} E_{12} N_2$	$E_{45} E_{34} E_{23} E_{12} N_5 N_4 N_3 N_2$

Table 4.1: Test cases for standardisation.

	Rewrite Steps	Mean Time^{-1} (runs/sec)	No. of Runs	S.d. (sec)
Teleportation	8	7,986±2.63%	78	0.000014855008895826105
X-Rotation	8	10,908±3.23%	83	0.000013751778984893029
Z-Rotation	26	9,745±2.13%	79	0.0000168950210012417
CNOT	9	5,903±2.60%	83	0.00002047401324488813
Hadamard	0	90,965±2.65%	76	0.000001296366332074945
Controlled-U	163	107±3.01%	71	0.0012087848984146466

Table 4.2: Standard Form Average Runtime Results.

From our runtime results, we can indeed see that rules with more rewrite rules have a longer execution time. In theory we should have a polynomial growth as the number of rewrite rules increases. The results in the previous table don't show this trend very clearly. To see whether we have polynomial growth, we shall next (Section 4.1.3) find the standard form of a n-way Hadamard, $\mathcal{H}(n)$ and see how the execution time varies as we increase n.

4.1.3 Standard Form of $H^{\otimes n}$

We now look at how the standard form behaves for a pattern as we increase the number of instructions. We will use the n-way Hadamard, $\mathcal{H}(n)$ as our pattern.

$$\mathcal{H}(n) := \mathcal{H}(2n-1, 2n) \otimes ... \otimes \mathcal{H}(3,4) \otimes \mathcal{H}(1,2)$$

For n, this pattern has $4n$ instructions and uses $2n$ qubits. We give the code for testing this chain in listings 4.1 (see B.3 for implementation of $\mathcal{H}(n)$).

```
...
var suite = new Benchmark.Suite;
var pattern = hadamard(n);
suite.add('chain', function () {
    rewrite.findStandardForm(pattern, []);
})
...
```

Listing 4.1: the code to time the standardisation of $\mathcal{H}(n)$

Results are shown in Table B.1 and have been plotted in Figure 4.1. From the results shown in Figure 4.1, we can see that the execution time increases at a *polynomial rate* as n increases. This is expected as the standard form algorithm Danos et al give is $O(N^5)$, where $N = 2n$ is the total number of qubits in the pattern.

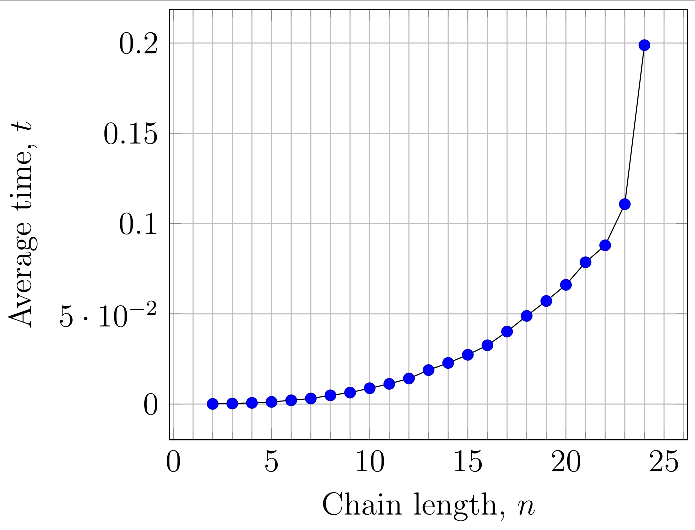


Figure 4.1: Standard Form of $H^{\otimes n}$ Runtime Plot.

4.2 Simulation

We now evaluate our simulation. First we feature test our simulator in Section 4.2.1 and 4.2.2. Then we make a performance test our simulator in We suspect that memory will be our main limiting factor. By holding our state vector in memory and doing all our matrix vector operations in memory, we will no doubt run out of memory. With n qubits, our state has 2^n components and we will be working with matrices of size $2^n \times 2^n$. We will look at how fast our simulation runs and how many qubits it takes to hit a memory limit.

4.2.1 Simulation Correctness

First we gain reassurance of our simulation's correctness, we test each individual instruction type in a modular fashion.

- *Testing Preparations:* We use various patterns that have varying of inputs and outputs that have just N commands. We want to test we have a correct input state by ensuring qubits $\notin I$ are prepared in $|+\rangle$.
- *Testing Entanglements:* We use the pattern $(\{1, 2\}, \{1, 2\}, \{1, 2\}, E_{12})$. By having both qubits in I, we can choose an arbitrary input state. By having both qubits in O, we do not need to measure them.
- *Testing Measurements:* We use the pattern $(\{1\}, \{1\}, \emptyset, M_1^{\alpha})$. By having $1 \in I$, we can choose an arbitrary input state for qubit 1. By having $1 \notin O$, we must measure 1. We choose various values for α, thus adjusting the basis for our general measurement. We then use a debugger to inspect the probability of output 0, and compare it to the expected theoretical probability.
- *Testing Corrections:* We use patterns $(\{1\}, \{1\}, \{1\}, X_1^1)$ and $(\{1\}, \{1\}, \{1\}, Z_1^1)$. We choose arbitrary states for the input state of qubit 1, $\mathbf{q}_{in[1]}$ and ensure that the output state is indeed $\mathbf{Xq}_{in[1]}$ and $\mathbf{Zq}_{in[1]}$ for the respective patterns.

The test cases given in Tables 4.3 and 4.4 are the simplest in that they involve the smallest number of qubits and the least amount of instructions. These test cases have been repeated with more qubits in the system. We can also extend the test cases to handle more signals.

$q_{in[1]}$	$q_{in[2]}$	Exp. Output	Actual Output
$\|0\rangle$	$\|0\rangle$	$(1,0,0,0)$	$(1,0,0,0)$
$\|0\rangle$	$\|1\rangle$	$(0,1,0,0)$	$(0,1,0,0)$
$\|1\rangle$	$\|1\rangle$	$(0,0,0,-1)$	$(0,0,0,-1)$
$\|+\rangle$	$\|+\rangle$	$(0.5,0.5,0.5,-0.5)$	$(0.5,0.5,0.5,-0.5)$
$\|0\rangle$	$-\|0\rangle$	$(-0.5,-0.5,-0.5,0.5)$	$(-0.5,-0.5,-0.5,0.5)$

Table 4.3: Some test cases for Entanglement: E_{12}.

$q_{in[1]}$	α	Basis	Exp Prob.	Actual Prob
$\|+\rangle$	0	$\|+_0\rangle, \|-_0\rangle$	1	0.9999999999999998
$\|-\rangle$	0	$\|+_0\rangle, \|-_0\rangle$	0	0
$(1,0)$	π	$\|+_\pi\rangle, \|-_\pi\rangle$	1/2	0.5
$(1,0)$	$\pi/2$	$\|+_{\pi/2}\rangle, \|-_{\pi/2}\rangle$	1/2	0.5

Table 4.4: Some test cases for Measurement: M_1^α.

4.2.2 Randomness of Measurements

The purpose of the following test is to show that computation branches are indeed uniformly distributed. Recall that if a pattern has m measurements, we expect there to be 2^m branches where a given computation follows a particular branch with probability 2^m. Consider the following pattern that prepares n qubits in the state $|+\rangle$, then measures all of them.

$$\mathcal{M}(n) := (\{1,...,n\}, \emptyset, \{1,...,n\}, M_n^{\pi/2}...M_1^{\pi/2}N_n...N_1)$$

We expect that if we were to run this pattern over and over, then the probability of observing a particular signal s, at the end of execution, is $1/2^n$. We give the code for testing this chain in listings 4.2. (see 4.2 for the code that implements $\mathcal{M}(n)$)

```
function randomMeasure(n) {
    var pat = measure(n);
    var bins = {};
    for (var j = 0; j < N; j++) {
        sim.begin(pat, {}, {}, null);
        sim.runAll();
        if(bins[sim.signal] == undefined)
            bins[sim.signal] = 1;
        else
            bins[sim.signal]++;
    }
}
```

	$q_{in[1]}$	Exp Output.	Actual Output
X	$\|+\rangle$	$(1, 1, 1, 1)$	$(1, 1, 1, 1)$

Table 4.5: Some test cases for Corrections: X_1^1,Z_1^1.

Listing 4.2: implementation of $\mathcal{M}(n)$

The results are shown in Table 4.6. From these results can indeed see that the first branch is visited approximately $1/2^n$ amount of the time. When we performed these tests, we also inspected all other branches and found no abnormal spikes or anomalies for large N. So we conclude that the each branch is indeed visited with a pseudo-random probability of $1/2^m$.

n	N	$N/2^n$	First Bin
1	100	50	47
	1000	500	468
	10000	5000	5046
	100000	50000	49811
2	100	25	22
	1,000	250	251
	10,000	2500	2433
	100,000	25000	24938
...	...	...	...
8	100	0.390625	1
	1,000	3.90625	3
	10,000	39.0625	37
	100,000	390.625	394

Table 4.6: Results for measurement bins.

4.2.3 Execution Runtime

We now look at the execution time for simulation. For simplicity, if any pattern has an input qubit $i \in I$, then we assign it to the state $\mathbf{q}_{in[i]} = |0\rangle$. The results are shown in Table 4.7.

	Mean Time^{-1}	No. of Runs (runs/sec)	S.d. (sec)
Teleportation:	506±2.46%	79	0.00022037772672865925
X-Rotation:	576±3.19%	74	0.0002427537386046713
Z-Rotation:	586±2.94%	78	0.00022635596233192714
CNOT:	131±2.85%	68	0.000919838566836595
Hadamard:	3,362±3.56%	73	0.000046193262956885646
$\mathcal{J}(\pi/2,1,2)$	1,127±3.21%	78	0.00012822982275891766
$\mathcal{CZ}(1,2)$	6,655±3.07%	77	0.00002067219329236154

Table 4.7: Average Run Time of Pattern.

4.2.4 Entanglement Chains

We now consider a chain C_n.

$$\mathcal{C}(n) := (\{1,...,n\}, \emptyset, \{1,...,n\}, E_{n,n-1}...E_{i,i+1}...E_{1,2}N_n...N_1)$$

This pattern represents a linear chain of entanglements of n qubits. Note that is pattern NEMC form of the pattern $\mathcal{CZ}(n,n-1) \circ \mathcal{CZ}(n-1,n-2) \circ ... \circ \mathcal{CZ}(2,1)$. The purpose of testing this chain is to get a rough idea how many qubits we can entangle. By finding an upper bound for n, we find a rough quantitative measure of how large our graph state can be, and in general, see how many qubits our simulator can handle. We give the code for testing this chain in Listings 4.3 (the implementation of $\mathcal{C}(n)$ is in Listings B.2).

```
var suite = new Benchmark.Suite;
var pattern = chain(n);
suite.add('chain', function () {
    sim.begin(pattern, {}, {}, null);
    sim.runAll();
})
```

Listing 4.3: code to run the entanglement chain runtime test

The results are shown in Table B.2 and have been plotted in Figure 4.3. What we can see from the graph in Figure 4.3 is that as we increase the n linearly, the execution time increases *exponentially*. From a high-level, we can argue that with a length n chain, our tensor product state has $O(2^n)$ components and acts as the dominating factor of the execution time. All matrices would need be $\in \mathbb{C}^{d \times d}$ where $d = O(2^n)$.

For $n \geq 10$, `benchmark.js` caused our browser to run out of memory. Google Chrome runs Javascript using the V8 engine which has a maximum limit of around 1.4GB for its heap. According to , running Google Chrome with the command line argument `--max-old-space-size=`⟨*num megabytes*⟩ can increase this limit. However we were still unable to succeed in running our tests with the argument `--max-old-space-size=10000`.

We tried other browsers such as Mozilla Firefox v30.0 and Safari v7.0.4 (9537.76.4). When we tried running the code normally and running the code with `benchmark.js`, we were unable to succeed in running the entanglement test for more than nine qubits. With these results, we conclude that 9 qubits is the maximum our simulator can handle on modern browsers.

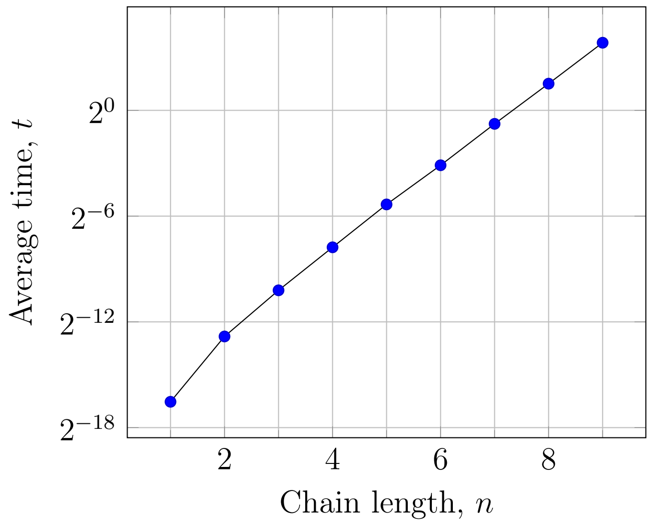


Figure 4.2: Average Runtime of $\mathcal{C}(n)$ Plot.

4.2.5 Execution Time of $H^{\otimes n}$

To supplement the previous tests for $\mathcal{C}(n)$, we perform tests to time the n-way Hadamard. $\mathcal{H}(n)$. Recall from Section 4.1.3 the pattern that computes the n-way Hadamard:

$$\mathcal{H}(n) := \mathcal{H}(2n-1, 2n) \otimes ... \otimes \mathcal{H}(3,4) \otimes \mathcal{H}(1,2)$$

We time how long it takes to execute this pattern and seek the largest n that our simulator can handle. Listings 4.4 shows the code needed to reproduce this test. In order to run a simulation of $\mathcal{H}(n)$, we need to give the input qubits of the pattern a state. We choose to give each $q \in I$ the state $q_{in[i]} = |0\rangle$ for simplicity.

```
var suite = new Benchmark.Suite;
var pattern = hadamard(n);
```

```
suite.add('chain', function () {
    sim.begin(pattern, {}, {}, null);
    sim.runAll();
})
...
```

Listing 4.4: the code to run the execution time of $H^{\otimes n}$

The results are given in Table B.3 and plotted in Figure 4.3. As we can see from the graph in Figure 4.3, we again have an exponential growth in average time as we increase the number of qubits linearly. Note that for a given n, $\mathcal{H}(n)$ has $2n$ qubits.

After running our tests we found that an n-way Hadamard with $n = 5$ (10 qubits) causes V8 to run out of memory. Again, trying to use perform a simulation with more than nine qubits is not possible. This is consistent with the results of the previous section.

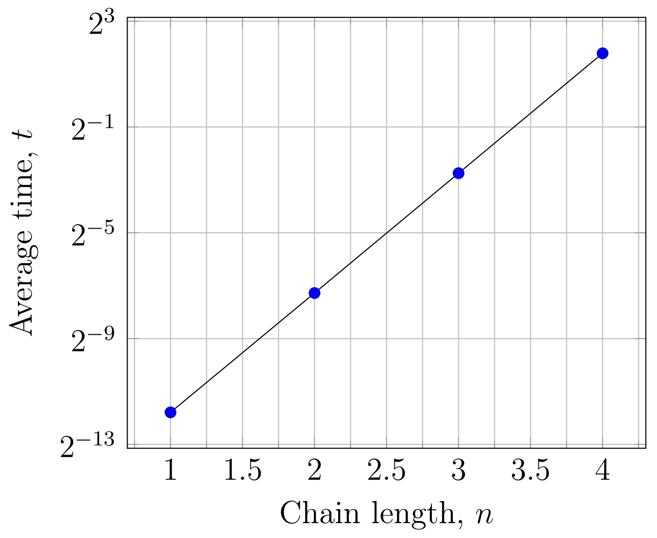


Figure 4.3: Average Runtime of $\mathcal{H}(n)$ Plot.

4.3 Flow Analysis

We now evaluate our implementation of the causal flow and general flow algorithms. We look at how it performs with certain test cases and also time its computation. We hope to see that our implementation gives the correct output and behaves according to the complexity bounds given in the literature.

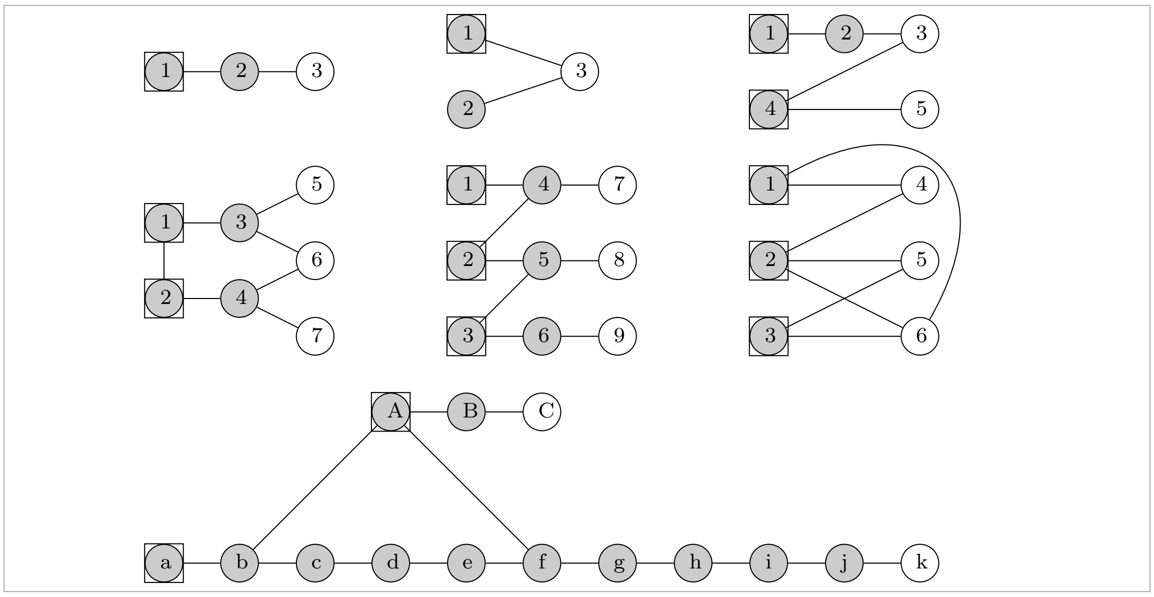


Figure 4.4: Open graphs OG_1,...,OG_7 numbered left-to-right, top-to-bottom with OG_1 on the top left, OG_3 on the top right and OG_7 at the bottom. These are some of the open graphs used to test the causal flow and general flow algorithms.

Like before, we cannot give a formal verification that our algorithm is correct. However to gain reassurance, have taken several open graphs from various papers and used them as test cases. We know precisely whether or not these graphs have a flow/gflow. Should a graph have a flow, we expect our algorithm to report there is a flow and to output some flow (but not necessarily the same as the one given in the respective paper). Should a graph have no flow, then we expect our algorithm to report that there is no flow. Proving a graph doesn't have a flow is not trivial but we rely on the proofs given by others. The test cases we used are shown in Tables 4.9 and 4.9.

We used several techniques to generate more test cases. One can take a *quantum circuit* and apply an algorithm given in [27, Section 5.12.1]. This constructs an open graph that always has a flow. The open graphs constructed from several quantum

Input	Flow?	g	$\prec$	Valid?
OG_1	Yes	$(1,2),(2,3)$	$1 \prec 2 \prec 3$	Yes
OG_2	No	-	-	Yes
OG_3	Yes	$(1,2),(2,3),(4,5)$	$1 \prec 2 \prec 4 \prec \{3,5\}$	Yes
OG_4	Yes	$(1,3),(2,4),(3,5),(4,7)$	$\{1,2\} \prec \{3,4\} \prec \{5,6,7\}$	Yes
OG_5	Yes	$(1,4),(2,5),(3,6)$	$1 \prec 2 \prec 3 \prec \{4,5,6\} \prec \{7,8,9\}$	Yes
		$(4,7),(5,8),(6,9)$		
OG_6	No	-	-	Yes
OG_7	Yes	(A,B), (B,C), (a,b), (b,c)	$a \prec b \prec c \prec d \prec e$	Yes
		(c,d), (d,e), (e,f), (f,g)	$\prec f \prec g \prec h$	
		$(g,h),(h,i),(i,j),(j,k)$	$\prec \{A,i\} \prec \{B,j\} \prec \{C,k\}$	

Table 4.8: Testing the results of `flow`.

circuits gave us the majority of the test cases used in our unit tests. We can also use the star pattern $\mathcal{S}(\alpha, n)$ as defined in [31, Figure 6] and ensure that there is always a flow given by a single arrow from the input to one of the outputs and that there is a two level partial order. As far as we can see from our tests, the implementation is reliable but we were unable to test the largest graph our program could handle due to the difficulty in finding the open graph itself.

```
var suite1 = new Benchmark.Suite,
    suite2 = new Benchmark.Suite;
var qubits = ...,
var adj = ....;
suite1.add('causal flow', function () {
   flowAnalyser.flow(qubits, adj);
})
...
suite2.add('gflow', function () {
   flowAnalyser.gflow(qubits, adj);
})
...
```

We also give the average runtime for these algorithms working on our test case graphs in Tables 4.10 and 4.11. From these results, we see that, as the theory would suggest, patterns with a larger depth (number of layers in the partial order $\prec$) will exhibit a longer execution time (as we need to make more recursive calls).

	gFlow?	g	$\prec$	Valid?
OG_1	Yes	$(1,\{2\}),(2,\{3\})$	$1 \prec 2 \prec 3$	Yes
OG_2	No	$-$	$-$	Yes
OG_3	Yes	$(1,\{2\}),(2,\{3,5\}),(4,\{5\})$	$1 \prec \{2,4\} \prec \{3,5\}$	Yes
OG_4	Yes	$(1,\{3\}),(2,\{4\})$ $(3,\{5\}),(4,\{5,6\})$	$\{1,2\} \prec \{3,4\} \prec \{5,6,7\}$	Yes
OG_5	Yes	$(1,\{4,5,6\}),(2,\{5,6\})$ $(3,\{6\}),(4,\{7\})$ $(4,\{7\}),(5,\{8\}),(6,\{9\})$	$\{1,2,3\} \prec \{3,4,5\}$ $\prec \{7,8,9\}$	Yes
OG_6	Yes	$(1,\{5,6\}),(2,\{4,5,6\})$ $(3,\{4,6\})$	$\{1,2,3\} \prec \{4,5,6\}$	Yes
OG_7	Yes	$(A,\{B\}),(B,\{C\})$ $(a,\{\}),(b,\{\})$ $(c,\{\}),(d,\{\})$ $(e,\{\}),(f,\{\})$ $(g,\{h\}),(h,\{i\})$ $(i,\{j\}),(j,\{k\})$	$a \prec b \prec c \prec d$ $e \prec f \prec g \prec h \prec$ $\{A,i\} \prec \{B,j\} \prec \{C,k\}$	Yes

Table 4.9: Testing the results of `gflow`.

4.4 Evaluation Remarks

We feel that our program is reliable and works reasonable well in modern browsers. However there is much room for improvement performance-wise. We could improve execution time by using Web Workers. However, using web workers wouldn't overcome out main limitation of running out of memory. The biggest problem to overcome

Even if we could implement a sensible scheme for in-place operations, another potential issue would be addressing components of the vector. In Javascript, the maximum positive integer that can be represented is 2^{53}. [29, Section 8.5]. Unless we create/use a library for handling arbitary length integers (*big numbers*), we would be constrained to address a maximum number 53 qubits in our state vector.

We could improve the Gaussian Elimination over $\mathbb{F}_2$ by using a binary search to locate 1s we want to eliminate with our pivot element - as proposed in [34]. Instead of an arithmetic complexity of $O(n^3)$ we could achieve $O(n^2 \log(n))$. Furthermore, if we were to parallelise, then [34] gives a parallel algorithm that achieves $2m^2 + m \log_2(n) - 2m$ bit operations to triangularize a $n \times m$ matrix on a bit-array with n processors and $m + 2 + \lceil \log_2 n \rceil$ bits of vertical memory per processor.

	Qubits	Mean Time^{-1} (runs/sec)	No. of Runs	S.d. (sec)
OG_1	x	`88,654`$\pm$`2.20`	87	`0.0000011797226831808897`
OG_2	x	`138,755`$\pm$`2.87`	83	`9.603079734780232e-7`
OG_3	x	`72,501`$\pm$`2.10`%	89	`0000013949694224995928`
OG_4	7	`65,566`$\pm$`2.52`%	84	`0.0000017966207873590255`
OG_5	9	`47,835`$\pm$`2.14`%	87	`0.00000212994199073697`
OG_6	6	`109,377`$\pm$`2.37`%	85	`0.0000010202119343244366`
OG_7	14	`32,368`$\pm$`2.39`%	90	`0.000003572486944217387`

Table 4.10: Average Runtime for causal flow, `flow`.

	Qubits	Mean Time^{-1} (runs/sec)	No. of Runs	S.d. (sec)
OG_1	3	`15,526`$\pm$`2.57`	87	`0.000007887101728242508`
OG_2	3	`23,860`$\pm$`2.73`	83	`0.000005318657050769652`
OG_3	5	`11,302`$\pm$`2.72`	84	`0.000011245628861711882`
OG_4	7	`8,145`$\pm$`2.37`	87	`0.000013868158196588812`
OG_5	9	`4,975`$\pm$`2.69`	82	`0.000024971930873043455`
OG_6	6	`11,928`$\pm$`2.60`	85	`0.000010256450647638566`
OG_7	14	`817`$\pm$`2.77`	87	`0.00016128587323041782`

Table 4.11: Average Run Time for general flow, `gflow`.

Running Javascript across several machines is difficult. If we wanted to seriously maximise the performance by using packages like CUDA, OpenCL, MPI etc. - on a cluster of machines or GPU's we believe we would need to rewrite out quantum mathematics library in a different language. However we then face the difficulty of translating results to a GUI. Overall we feel that Javacript was a good language choice based on our user interface and evaluation results. We feel that there would not have been enough time to develop a user interface in a different language.

5 Conclusion

To conclude, we summarise our contributions and discuss possible extensions to the project.

- We have implementing term rewriting of the measurement calculus. Our program can successfully find the standard form of an arbitary pattern in polynomial time. Our user interface allows user to easily explore term rewriting by automatically detecting valid rule applications and performing the rewriting.
- We have a simulator and established that can simulate up to nine qubits. We provide a user interface for a researcher to input patterns for simulation and see their intermediate state.
- We have implemented flow and gflow algorithms. The user can input an open graph and see whether or not there is a flow/gflow. Furthermore, the interface will compute the layers.
- We have a renderer that can visually show any pattern using HTML/CSS. Furthermore users can create and edit arbitrary patterns (both sequences of commands and composition trees involving $\circ, \otimes$).

Potential extensions could be:

- A lot of our computation involves matrices and vectors. There so much scope for parallelization yet all our work was done on a single processor using single-threaded Javascript.
- The user interface could be polished to a professional standard. Currently the user interface has only been tested on Google Chrome. Support for other browsers could be introduced.
- Investigate how we can optimise our implementation and attempt to reduce [35] provides new optimisation that could be implemented.
- Compute the branch map [27, Definition 4] for a given computation. We could then determine whether a pattern is *deterministic* [27, pg 15] or *strongly deterministic* [27, Proposition 6].
- Provide a user interface for the user to input quantum circuits and for the program to find a corresponding pattern. That is, perform the construction as described in [27, 5.12.1]. We could similarly implement the patterns to circuits algorithm [27, 5.12.2].
- Implement other models of the Measurement Calculus such as the *Phase Model* [27, 5.7.1], the *Pauli Model* [27, 5.7.2] or the *Teleportation Model* [27, 5.7.3].

Bibliography

[1] Krysia Broda, Jiefei Ma, Gabrielle Sinnadurai, and Alexander Summers. Pandora: A reasoning toolbox using natural deduction style. Logic journal of the IGPL, 15(4):293âĂŞ304, 2007.

[2] B. Schumacher. *Quantum coding.* Physical Review A 51 (4): 2738-2747. Bibcode:1995PhRvA..51.2738S. doi:10.1103/PhysRevA.51.2738. 1995

[3] Simon Bone and Matias Castro. A Brief History of Quantum Computing. `http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol4/spb3/`

[4] `http://www.quantiki.org/wiki/List_of_QC_simulators`

[5] Andreas de Vries. *Quantum Computation: An Introduction for Engineers and Computer Scientists* 2012, pg 21, Equation 2.22

[6] *Term Rewriting and All That.* Franz Baader and Tobias Nipkow, Cambridge University Press, 1998

[7] R. Jorza. *An Introduction to Measurement-Based Quantum Computation.* quant-ph/0508124v2, 2005

[8] D. E. Browne, H. J. Briegel. One-way Quantum Computation - a tutorial introduction. arXiv:quant-ph/0603226, 2006

[9] http://mathjs.org/

[10] http://sylvester.jcoglan.com/

[11] http://www.dwavesys.com/

[12] http://www.qudev.ethz.ch/content/QSIT14/QSITpresMarcandGustavoFV.pdf

[13] How âĂŸQuantumâĂŹ is the D-Wave Machine?, Seung Woo Shin, Graeme Smith, John A. Smolin, and Umesh Vazirani. Jan 2014.

[14] Distinguishing quantum and classical models for the D-Wave device. Vinci, W., Albash, T., Mishra, A., Warburton, P. A. Lidar, D. A. arXiv:1403.4228 [quant-ph]. (Mar 2014)

[15] R. Raussendorf, D. E. Browne, and H. J. Briegel. *Measurement-Based Quantum Computation on Cluster States.* quant-ph/0301052v2, 2004

[16] D. Gottesman and I. Chuang, Quantum teleportation as a universal computational primitive, Nature 402, 390-393, 1999. arXiv:quant-ph/9908010.

[17] M. A. Nielsen. Optical quantum computation using cluster states. Physical Review Letters, 93:040503, 2004.

[18] S. R. Clark, C. Moura Alves, and D. Jaksch. Efficient generation of graph states for quantum computation. New Journal of Physics, 7:124, 2005.

[19] D. E. Browne and T. Rudolph. Resource-efficient linear optical quantum computation. Physical Review Letters, 95:010501, 2005

[20] M. S. Tame, M. Paternostro, M. S. Kim, and V. Vedral. Toward a more economical cluster state quantum computation. quant-ph/0412156, 2004.

[21] M. S. Tame, M. Paternostro, M. S. Kim, and V. Vedral. Natural three-qubit interactions in one-way quantum computing. Physical Review A, 73:022309, 2006.

[22] P. Walther, K. J. Resch, T. Rudolph, E. Schenck, H. Weinfurter, V. Vedral, M. Aspelmeyer, and A. Zeilinger. Experimental one-way quantum computing. Nature, 434:169âĂŞ176, March 2005.

[23] A. Kay, J. K. Pachos, and C. S. Adams. Graph-state preparation and quantum computation with global addressing of optical lattices. Physical Review A, 73:022310, 2006.

[24] S. C. Benjamin, J. Eisert, and T. M. Stace. Optical generation of matter qubit graph states. New Journal of Physics, 7:194, 2005.

[25] Q. Chen, J. Cheng, K.-L Wang, and J. Du. EfïňĄcient construction of two-dimensional cluster states with probabilistic quantum gates. Physical Review A, 73:012303, 2006.

[26] V. Danos, D. E. Kashefi, and P. Panangaden. *The Measurement Calculus.* quant-ph/0412135, 2004

[27] V. Danos, D. E. Kashefi, P. Panangaden and S. Perdrix. *Extended Measurement Calculus.* Semantic Techniques in Quantum Computation, Cambridge University Press, 2010

[28] M. H. A. Newman. *On theories with a combinatorial definition of equivalence. Annals of Mathematics*, 43, Number 2, pages 223-243, 1942.

[29] *ECMA-262 ECMAScript Language Specification* http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, Section 4.3.19

[30] M. Mhalla, S. Perdrix. *Finding Optimal Flows Efficiently.* arXiv:0709.2670v1 [quant-ph], 2007

[31] V. Danos and E. Kashefi. *Determinism in the one-way model.* Phys. Rev. A., 74, 2006

[32] D. Browne, E. Kashefi, M. Mhalla and S. Perdrix. *Generalized flow and determinism in measurement-based quantum computation.* New J. Phys. 9, 250, 2007

[33] Ross Duncan and Simon Perdrix. *Rewriting measurement-based quantum computations with generalised flow.* Automata, Languages and Programming. Springer Berlin Heidelberg, 2010

[34] Koç, Çetin K., and Sarath N. Arachchige. *A fast algorithm for gaussian elimination over GF(2) and its implementation on the GAPP.* Journal of Parallel and Distributed Computing 13.1 (1991): 118-122.

[35] Nikahd, E.; Houshmand, M.; Zamani, M.S.; Sedighi, M., OWQS: One-Way Quantum Computation Simulator, Digital System Design (DSD), 2012 15th Euromicro Conference on , vol., no., pp.98,104, 5-8 Sept. 2012 doi: 10.1109/DSD.2012.100

[36] J. Allcock, Emulating circuit-based and measurement-based quantum computation. Department of Computing, MSc Thesis: Imperial College London, 2010.

Appendix A

A.1 Causal Flow Algorithm

We give the general flow algorithm in Listings A.1.

A.2 General Flow Algorithm

We give the general flow algorithm in Listings A.2.

```
input  : An open graph
output: A causal flow
Flow($G, I, O$) =
begin
    for all $v \in O$ do
        $l(v) := 0$
    end
    Flowaux($G, I, O, O - I, 1$)
end

Flowaux($G, In, Out, C, k$) =
begin
    $Out' := \emptyset$;
    $C' := \emptyset$;
    for all $v \in C$ do
        if $\exists u$ s.t. $N(v) \cap (V - Out) = \{u\}$ then
            $g(u) := v$;
            $l(v) := k$;
            $Out' := Out' \cup \{u\}$;
            $C' := C' \cup \{v\}$;
        end
    end
    if $Out' = \emptyset$ then
        if $Out = V$ then
            true;
        else
            false;
        end
    else
        FlowAux($G, In, Out \cup Out', (C - C') \cup (Out' \cap V - In), k + 1$);
    end
end
```

Algorithm A.1: Causal Flow Algorithm

```
input  : An open graph
output: A generalized flow
gFlow($\Gamma, In, Out$) =
begin
    for all $v \in Out$ do
        $l(v) := 0$
    end
    gFlowaux($\Gamma, In, Out, 1$)
end

gFlowaux($G, In, Out, k$) =
begin
    $Out' := Out - In$;
    $C := \emptyset$;
    for all $u \in V - Out$ do
        Solve in $\mathbb{F}_2 : \Gamma_{V-Out,Out'}\mathbb{I}_X = \mathbb{I}_{\{u\}}$;
        If there is a solution $X_0$ then $C := C \cup \{u\}$ and $g(u) := X_0$;
        $l(u) := k$;
    end
    if $C = \emptyset$ then
        if $Out = V$ then
            true;
        else
            false;
        end
    else
        gFlowAux($\Gamma, In, Out \cup C, k + 1$);
    end
end
```

Algorithm A.2: General Flow Algorithm

Appendix B

B.1 Evaluation Helper Functions

```
function measure(n) {
    var qubits = {}, instrs = [];
    for (var i = n; i > 0; i--) {
        qubits[i] = QubitIO.Out;
        instrs.push(new pattern.Measurement(i, piOverTwo, [], []))
    }
    for (i = n; i > 0; i--) {
        instrs.push(new pattern.Prepare(i))
    }
    return new pattern.Pattern(instrs,qubits);
}
```

Listing B.1: implementation of $\mathcal{M}(n)$

```
function chain(n) {
    var qubits = {}, instrs = [];
    for (var i = n; i > 1; i--) {
        instrs.push(new pattern.Entanglement(i, i-1))
    }
    for (i = n; i > 0; i--) {
        qubits[i] = QubitIO.Out;
        instrs.push(new pattern.Prepare(i))
    }
    return new pattern.Pattern(instrs, qubits);
}
```

Listing B.2: implementation of $\mathcal{C}(n)$

```
function hadamard(n) {
    var hs = new ct.LeafNode(predefinedPatterns.h(1,2));
    for (var i = 2; i <= n; i++) {
        hs = new ct.ParallelNode()
            .left(new ct.LeafNode(predefinedPatterns.h(2*i-1, 2*i)))
            .right(hs);
    }
    return hs.evaluate();
}
```

Listing B.3: implementation of $\mathcal{H}(n)$

B.2 Tables of Results

n	Mean Time^{-1} (runs/sec)	No. of Runs	S.d. (sec)
2	10,418±3.63%	71	0.000014996283440623804
3	3,261±3.02%	85	0.00004357925145277791
4	1,494±2.91%	84	0.00009110013185581718
5	821±2.80%	84	0.00015964641216242216
6	478±3.03%	84	0.0002960192549109096
7	323±2.54%	79	0.00035668972173847924
8	207±2.89%	77	0.0006253234863303502
9	157±2.69%	74	0.0007505882390148032
10	114±2.43%	74	0.0009307894710848719
11	89.26±2.43%	72	0.0013435169287912146
12	70.56±2.85%	73	0.0017609153653732005
13	53.14±3.90%	58	0.0028493868313660144
14	43.88±3.40%	59	0.003040662473660319
15	36.63±2.84%	54	0.0029081026671853565
16	30.70±2.72%	55	0.0033572927681549446
17	24.89±2.96%	46	0.004120669401421932
18	20.47±4.58%	39	0.007135722572530794
19	17.51±3.60%	48	0.007262664779322228
20	15.14±4.14%	43	0.009155492160707414
21	12.73±4.23%	37	0.010307457671449709
22	11.37±4.57%	33	0.01178201291030333
23	9.03±5.18%	28	0.01478330350585626
24	5.03±5.96%	27	0.016935400672911615

Table B.1: Standard Form of $H^{\otimes n}$ Runtime Results.

n	Mean Time^{-1} (runs/sec)	No. of Runs	S.d. (sec)
1	94775±3.12%	76	0.000001465046949701225
2	7262±3.19%	80	0.00002001783415300274
3	1183±2.82%	84	0.0001114996545281742
4	218±2.70%	72	0.0005351496140581391
5	40.39±3.82%	56	0.003609741531322303
6	8.66±3.05%	26	0.008708444173198637
7	1.70±2.89%	9	0.022129229057728548
8	0.35±1.84%	5	0.042600782469245216
9	0.07±5.28%	5	0.5703432571979602
10	OOM	–	–

Table B.2: Entanglement Chain Average Runtime Results.

n	Mean Time^{-1} (runs/sec)	No. of Runs	S.d. (sec)
1	3,533±3.34%	76	0.000042034978965494774
2	155±3.12%	68	0.0008486062139536539
3	6.71±5.66%	22	0.01901450558924536
4	0.29±8.47%	5	0.23679136126744663
5	OOM	–	–

Table B.3: n-way Hadamard Average Runtime Results.