Quantum Computing Project
Report for CMPS299 – Software Graduation Project

Rafeh, Bayan [*] Saleh, Adel[†] Zakhour, George [‡]

June 2015

Abstract

We present in this report a summary of our work for CMPS299 – Software Graduation Project. A pure quantum programming language with quantum control dubbed Braqet was developed whose syntax and semantics and its compiler are described in this report. Alongside is a tool that was developed to simulate and visualize a Quantum Turing Machine. To avoid using a hybrid system and to make the quantum programming experience more realistic we have also developed a quantum virtual machine that runs as a server. This report also details on the development process and the optimization techniques used to perform the computations.

[*] `bayan.rafeh92@gmail.com`

[†] Adel was not able to register the course, however his many contributions to the project this past year are of great importance and must be included

[‡] `g1995z@gmail.com`

To my wonderful co-authors.
Without whom this work would have been half as decent.

– *The Authors*

Contents

Chapter 1

Project Introduction

1.1 Problem Description

Quantum algorithms are intrinsically different than classical algorithms, for many reasons, the most relevant ones being that the CPU no longer performs fetch-execute cycles, but more of a prepare-manipulate-measure process. Moreover the linearity of the Schrödinger equation imposes that all operations be written as a unitary invertible matrix. It is thus natural for the programming environment of the Quantum Computer to be much different and sometimes orthogonal to the environment of the classical computer, for example there cannot be any branching (function calls through branching, loops, conditional breaking, etc...). The framework in which researches work with is very similar to the functional programming, hence the reason why a functional design as opposed to an imperative one would make more sense.

1.2 Our Contribution

Our project is to thus build this environment, a "functional" programming language based on a quantum lambda calculus with quantum control that is inspired from the quantum lambda calculus with the popular classical control defined in [SV05]. Since no such language has been developed then we must also develop a virtual machine that executes these programs, this also necessite the development of a quantum assembly language. Moreover the virtual machine executing a quantum algorithm will be run independently from the classically machine executing a classical algorithm communicating both ways through an interface defined in the virtual machine.
What has been developed is mainly imperative procedural programming languages that combine classical and quantum code together that runs on q hybrid quantum-classic virtual machine. Splitting this system makes more sense and makes the environment more modular and allows quantum code to

be ported easily to a quantum computer if it were built. Also this split gives the programmer more flexibility when working with classical algorithms (as long as the language supports socket programming then it can be used).

1.3 Logistics

Our project consists of two major projects one of them split into two smaller projects.

1. Creating a functional programming language, George Zakhour is responsible to develop the language and build a compiler that compiles the code a byte code that the virtual machine can understand.
2. Writing a virtual machine
   (a) Constructing the virtual machine, Bayan Rafeh is responsible to develop the backend of the virtual machine, mainly memory management, and parsing the byte code.
   (b) Writing the ALU and performing computations, Adel Saleh is responsible to develop optimized algorithms that perform quantum computation.

It should also be noted that we also juggle roles in order to maintain compatibility across the departments and to share our experience and ideas.

Chapter 2

A Review of Quantum Computation

Richard Feynman in [Fey82] showed that using a classical probabilistic Turing machine to simulate a simple physical system takes an exponential amount of time and memory. Quoting him "If doubling the volume of space and time means I'll need an exponentially larger computer, I consider that against the rules (I make up the rules, I'm allowed to do that)". He then proposed combining quantum mechanics with the theory of computation to simulate physical phenomena faster. David Deutsch in [Deu85] formalized the concept of a quantum Turing machine and redefined the Church-Turing thesis to include quantum phenomena to be efficiently computable. What gave quantum computation its boost is Shor's seminal paper [Sho99] that showed that prime factorization and the discrete logarithm problem were in **BQP**, the class of bounded-error quantum polynomial time algorithms.

2.1 The Qubit and Superposition

The basic piece of information in the classical world is the bit, in the quantum realm it is the quantum bit; *qubit.* We say a qubit lives in a 2 dimensional Hilbert space $\mathcal{H}$ whose basis, dubbed the *computational basis* is $\{|0\rangle, |1\rangle\}$. One does not necessarily need to work with the bra/ket notation and can build an isomorphism between $\mathcal{H}$ and $\mathbb{R}^2$ by identifying $|0\rangle$ with $\begin{bmatrix}1\\0\end{bmatrix}$ and $|1\rangle$ with $\begin{bmatrix}0\\1\end{bmatrix}$. $\langle x|$ (pronounced *bra x*) will denote the complex transpose of $|x\rangle$ (pronounced *ket x*). $\langle x|y\rangle$ will denote the inner product between $|x\rangle$ and $|y\rangle$. $|x\rangle\langle y|$ will denote the projection matrix that projects $|y\rangle$ to $|x\rangle$ and the rest to 0 (not to be confused with $|0\rangle$) if $|x\rangle$ and $|y\rangle$ were basis vectors.

A qubit ψ is expressed as follows:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \qquad \alpha, \beta \in \mathbb{C}$$

with the condition that $|\alpha|^2 + |\beta|^2 = 1$. This condition follows from the postulates of quantum mechanics. Once a qubit is observed then its state collapses to either $|0\rangle$ with probability $|\alpha|^2$ or to $|1\rangle$ with probability $|\beta|^2$. If $\alpha \neq 0$ and $\beta \neq 0$ then we say that the qubit is in a superposition of the two states.

2.2 Kronecker Product or Tensor Product

To manipulate an n-qubit register that lives in a 2^n-dimensional Hilbert space $\mathcal{H}^{\otimes n}$, we use the tensor product $\otimes$ that is defined on the matrices $A_{n\times m}$ and $B_{p\times q}$ to give a matrix $C_{np\times mq}$ as follows

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nm}B \end{bmatrix}$$

In the special case when A and B are n and m dimensional vectors then

$$A \otimes B = \begin{pmatrix} a_1 b_1 \\ \vdots \\ a_1 b_m \\ \vdots \\ a_n b_1 \\ \vdots \\ a_n b_m \end{pmatrix}$$

Using the bra/ket notation we use the following shortcuts

$$\begin{aligned} |a\rangle \otimes |b\rangle &= |a\rangle\,|b\rangle \\ &= |a, b\rangle \\ &= |ab\rangle \end{aligned}$$

The Kronecker product is left and right distributive, associative. and non-commutative that satisfies these identities

- $(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$
- $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$ where $\cdot$ is matrix multiplication.
- $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$
- If B_1 is a basis for $\mathcal{H}_1$ and B_2 is a basis for $\mathcal{H}_2$ then $\{|e_i^1, e_j^2\rangle \ : \ |e_i^1\rangle \in B_1,\ |e_i^2\rangle \in B_2\}$ is a basis for $\mathcal{H}_1 \otimes \mathcal{H}_2$.

2.3 Operations need to be unitary and invertible

Theorem 1. *All operations acting on qubits need to be square matrices that are unitary, i.e. if U is an operator then $UU^\dagger = U^\dagger U = \mathbb{I}$.*

Proof. A matrix transforming a quantum state should also transform it into a valid quantum state, i.e. $\langle\psi|\psi\rangle = 1$. We therefore require, for every valid quantum state $|\psi\rangle$ that $\langle U\psi|U\psi\rangle = \langle\psi|U^\dagger U|\psi\rangle = 1$, hence $\langle\psi|U^\dagger U|\psi\rangle - \langle\psi|\psi\rangle = \langle\psi|(U^\dagger U - \mathbb{I}|\psi\rangle = 0$. Since $|\psi\rangle$ was arbitrary, it follows that $U^\dagger U = \mathbb{I}$. Hence U is square and unitary. □

Theorem 2. *All unitary matrices U are invertible and $U^{-1} = U^\dagger$.*

Proof. Let U be a unitary matrix, then $UU^\dagger = \mathbb{I}$, hence $|UU^\dagger| = |U| \cdot |U^\dagger| = |\mathbb{I}| = 1$, therefore $|U| \neq 0$ from which follows that U is invertible. Multiplying by U^{-1} on the left of the both sides of the condition of unitarity we find that $U^{-1} = U^\dagger$. □

2.4 No-Cloning Theorem

Theorem 3. *There does not exist a unitary operator U that can reliably clone any given qubit whereby cloning we mean the application of a transformation that given $|a, 0\rangle$ produces $|a, a\rangle$.*

Proof. Assume such an operator U exist, then $U|a, 0\rangle = |a, a\rangle$ and $U|b, 0\rangle = |b, b\rangle$ for arbitrary $|a\rangle$ and $|b\rangle$. Let $|c\rangle = \alpha|a\rangle + \beta|b\rangle$ then $U|c, 0\rangle = |c, c\rangle = \alpha^2|a, a\rangle + \alpha\beta|a, b\rangle + \alpha\beta|b, a\rangle + \beta^2|b, b\rangle$. But $U|c, 0\rangle = U(\alpha|a, 0\rangle + \beta|b, 0\rangle) = \alpha U|a, 0\rangle + \beta U|b, 0\rangle = \alpha|a, a\rangle + \beta|b, b\rangle$. This is only possible when $\alpha = 1, \beta = 0$ or $\alpha = 0, \beta = 1$. However if our qubit is in a superposition (i.e. not in a classical state) then $\alpha \neq 0, \beta \neq 0$ hence it is clear that we reached a contradiction. Therefore no such operator exists. □

One could simply argue that if cloning were to be possible then Heisenberg's uncertainty principle no longer holds!

2.5 Quantum Algorithms

2.5.1 Deutsch-Jozsa Algorithm

The Deutsch-Jozsa Algorithm described in [DJ92] is one of the first quantum algorithms to be conceived. The problem is that given a function $f : \mathbb{Z}_{2^n} \to \mathbb{Z}$ determine whether it's balanced or constant. Classically this requires $\Omega(2^{n-1})$ calls to the blackbox f, quantum mechanically this problem can be solved with just one query.

Starting with a $|0\dots0\rangle|1\rangle$, update the state by applying $H^{\otimes n+1}$ to it, then feed it to the blackbox using the operator U_f and then reapply $H^{\otimes n+1}$ and measure the first n-qubit register. If the result is $|0\rangle$ then the function is balanced, otherwise it's constant.

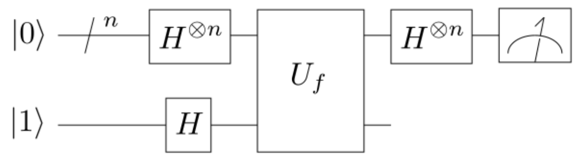

Figure 2.1: Deutsch-Jozsa Algorithm

2.5.2 Simon's Algorithm

Simon's algorithm described in [Sim94] solves the following problem: given a function $f : \mathbb{Z}_{2^n} \rightarrow \mathbb{Z}_{2^m}$ with $m \geq n$ such that for every $x, y \in \mathbb{Z}_{2^n}$ $f(x) = f(y)$ if and only if $y = x$ or $y = x \oplus a$ where $a \in \mathbb{Z}_{2^n}$ and $\oplus$ the addition modulo 2^n to find the characterizing a. Classically this requires $\Omega(2^{n/2})$ queries to f. Using Simon's algorithm this could be accomplished in $O(n)$ queries.

An iteration is described in the quantum circuit below. After executing an iteration a number y will be measured that satisfies $y \cdot a = y_0 a_0 \oplus \cdots \oplus y_n a_n = 0$. Repeating this process on average $O(n)$ times we find a system of n equations with different ys which we could then solve for a.

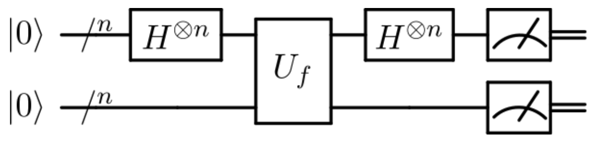

Figure 2.2: Simon's Algorithm

2.5.3 Grover's Algorithm

Grover's algorithm described in [Gro96] can search for $|\omega\rangle$ in an unsorted database in $O(\sqrt{n})$. Starting with a superposition of all possible entries in the database the algorithm iteratively rotates the superposition in the $(|\omega\rangle^{\perp}, |\omega\rangle)$ plane in the positive direction. The state comes closest to $|\omega\rangle$ in $O(\sqrt{n})$ rotations.

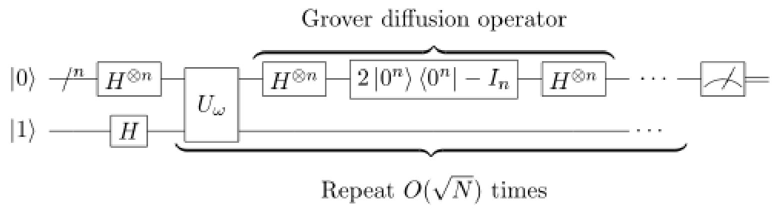
Grover diffusion operator
$|0\rangle$
$|1\rangle$
n
$H^{\otimes n}$
H
U_ω
$2|0^n\rangle\langle 0^n| - I_n$
Repeat $O(\sqrt{N})$ times

Figure 2.3: Grover Algorithm

2.6 The Quantum Turing Machine

We present below the quantum Turing machine and a special version, the stationary rotational quantum Turing machine as defined in [BV93] and [Oza02].

A quantum Turing machine M is a triplet (Q, Σ, δ) with Q being a sub-space of a Hilbert space with a basis $\{|q_0\rangle, \ldots, |q_f\rangle\}$; q_0 being the start state and $|q_f\rangle$ the final halting state. Σ the alphabet of the machine with # a special symbol indicating the blank symbol[1], Σ is also a sub-space of a Hilbert space spanned by $\{|0\rangle, |1\rangle, |\#\rangle\}$. δ is the transition function that is defined as the following: $\delta : Q \times \Sigma \to \tilde{\mathbb{C}}^{Q\times\Sigma\times\{-1,1\}}$ or equivalently as $\delta : Q \times \Sigma \times Q \times \Sigma \times \{-1, 1\} \to \tilde{\mathbb{C}}$ where $\tilde{\mathbb{C}}$ is the set of efficiently computable complex numbers, $\delta(q, \sigma, p, \tau, d)$ can be read as the probability amplitude of the transition when M is in the state q reading σ to the state p writing a τ and moving in the direction d. The configuration of a quantum Turing machine is a superposition of basis vectors in the Hilbert space $\mathcal{Q}\otimes\mathcal{Z}\otimes\Sigma^{\otimes\infty}$ where $\mathcal{Q}$ is the Hilbert space of the states, $\mathcal{Z}$ is the infinite Hilbert space that is spanned by $\{|0\rangle, |1\rangle, |-1\rangle, \ldots\}$. The reason the tape is bilaterally infinite is a necessity, consider what to do when the head is at 0 and you would want to move to the left, one can easily see how the transition function won't be reversible and hence violating Theorem 1. A configuration is written as $|q, \xi, T\rangle$ and the transition matrix is defined as

$$M_\delta |q, \xi, T\rangle = \sum_{p\in\mathcal{Q},\sigma\in\Sigma,d\in\{-1,1\}} \delta(q, T(\xi), p, \sigma, d) |p, \xi + d, T_\xi^\sigma\rangle$$

where $T(\xi)$ means the ξ-th symbol on the tape, and

$$T_\xi^\sigma(m) = \begin{cases} T(m) & m \neq \xi \\ \sigma & m = \xi \end{cases}$$

in other words replacing the ξ-th symbol of the tape by σ. Obviously the transition matrix needs to be unitary. In [ON98] the authors set conditions on δ rather than M_δ which makes working with QTM easier.

[1]In our examples later on we won't use the blank symbol and instead work on a model with $\Sigma = \{0, 1\}$

2.6.1 The Halting Problem of Quantum Turing Machines

One can ask how to figure out when a quantum Turing machine halts, if it were to halt. In the classical case one observes the final state of the machine after every transition, if it's in q_f then the machine halts. In the case of a quantum Turing machine one cannot observe the state, the superposition will be destroyed and there goes interference and entanglement! What gives quantum computing its power! A more complex halting schema needs to be developed, we explore a variation of QTMs called SR-QTMs whose halting schema is quite trivial and then discuss its limitation. Later we present the halting schema presented by Deutsch and proved in [Oza02].

2.7 The Stationary Rotational Quantum Turing Machine – SR-QTM

Definition 1. *A Quantum Turing Machine M is said to be stationary if its head behaves classically without ever going into a superposition. Moreover if the configuration reaches a final state then the tape head is at the first cell. In symbols, for every $t \in \mathbb{N}$ then M is stationary if and only if:*

$$U^t \left|q_0, \zeta, T\right\rangle = \left|q\right\rangle \left|\xi\right\rangle \left|\Gamma\right\rangle$$

where $\left|\xi\right\rangle$ is a basis vector and if $\langle q|q_f\rangle > 0$ then $\left|\xi\right\rangle = \left|0\right\rangle$.

Definition 2. *A Quantum Turing Machine M is said to be unidirectional if every each state can be entered from only one direction. In symbols, if $\delta(p_1, \sigma_1, q, \tau_1, d_1) \neq 0$ and $\delta(p_2, \sigma_2, q, \tau_2, d_2) \neq 0$ then $d_1 = d_2$.*

Definition 3. *A Quantum Turing Machine M is said to be rotational if it is unidirectional and satisfies the condition, if $\delta(p, \sigma, q_1, \tau_1, d_1) \neq 0$ and $\delta(p, \sigma, q_2, \tau_2, d_2) \neq 0$ then $q_1 = q_2$. In other words a QTM M is rotational if it is unidirectional and given a state p and a symbol σ then the machine cannot transition to a configuration with a superposition of different states.*

Definition 4. *A Stationary Rotational Quantum Turing Machine M is a machine that is stationary, rotational and whose branches halt at the same time. In symbols, if $\left|q_n, \xi_n, T_n\right\rangle = M_\delta^n \left|q_0, \xi_0, T_0\right\rangle$ where $\left|q_0, \xi_0, T_0\right\rangle$ is the initial configuration then there exist an $N \in \mathbb{N}$ such that for every $t \in \mathbb{N}$ if $t \geq N$ then $\langle q_t|q_f\rangle = 1$, and if $t < N$ then $\langle q_t|q_f\rangle = 0$. N is said to be the run-time of the SR-QTM.*

An SR-QTM has many advantages, it is easily visualized and its halting scheme is trivial. Since the position of the head is deterministic and when the branch halts it goes back to the first cell indexed 0, then if we design the machine such that its head is at the first cell only in the initial configuration and the final configuration, then the halt status of the machine can be determined by simply observing the position of the machine's head.

All of the previous definitions allow us to write a compact version of δ which we define by $\delta' : Q \times \Sigma \times \Sigma \to \tilde{C}$ with two helper functions, $s : Q \times \Sigma \to Q$ and $\Delta : Q \to \{-1, 1\}$.

$$\delta'(p, \sigma, \tau) = \delta(p, \sigma, \tau, s(p, \sigma), \Delta \circ s(p, \sigma))$$

2.8 State Transition Diagram for an SR-QTM

In [LY13] a state transition diagram is introduced which we have altered. We present some examples and explain how the diagram works

$$\begin{tikzcd}
\text{start} \arrow[r] & \overrightarrow{q_s} \arrow[r, "H"] & \overrightarrow{q_1} \arrow[r, "I"] & \overleftarrow{q_2} \arrow[r, "H"] & \overrightarrow{q_3} \arrow[r, "I"] & \overleftarrow{q_f} \arrow[llll, bend left, "I"]
\end{tikzcd}$$

Figure 2.4: A machine that applies $H^{\otimes 2}$ to the first two qubits

The start state is indicated by q_s and the final state by q_f, since each state can be entered in one direction, this direction is indicated with the arrow above the state name. The matrices labeling the transitions are the matrices to apply on the qubit under the head. The transition from q_f to q_s is only there to make the matrix unitary. All transitions that are not possible to happen are omitted, but they must still be present when expressing the matrix of the machine.

$$\begin{tikzcd}
\text{start} \arrow[r] & \overrightarrow{q_s} \arrow[r, "H"] & \overrightarrow{q_1} \arrow[r, "I"] & \overleftarrow{q_2} \arrow[r, "H"] & \overrightarrow{q_3} \arrow[r, "I"] & \overleftarrow{q_f}
\end{tikzcd}$$

Figure 2.5: The previous machine with the impossible transitions omitted

The next example is a more interesting one. It applies the Hadamard operator H on the second qubit if the first qubit is 0. Again we display the machine full with the transition q_f to q_s removed, and we further remove useless transitions. Moreover we will adopt in the transition diagram the following notation U_n where U is any matrix and n the n-th element of the basis of the Hilbert space of U to mean $U_n = U |n\rangle \langle n|$. For example $I_0 = |0\rangle \langle 0|$, $I_1 = |1\rangle \langle 1|$, $X_0 = |1\rangle 0$ and $X_1 = |0\rangle 1$.

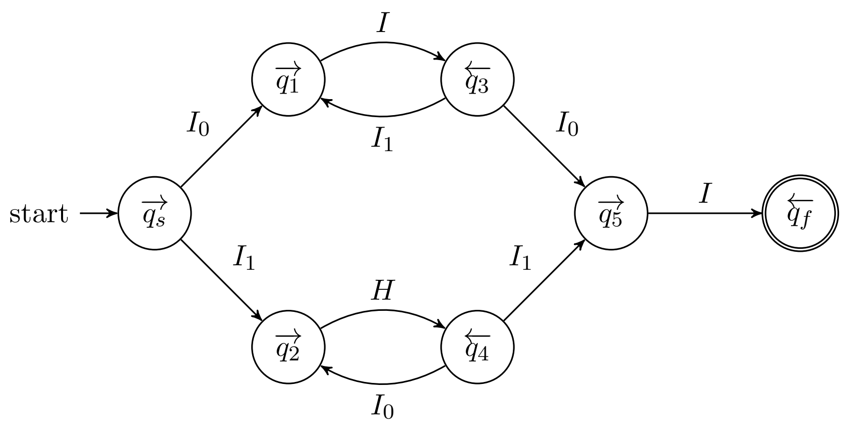

Figure 2.6: A machine that applies $\bigwedge H$ on the first two qubits

One notices that the matrices coming out of q_s and into q_5 are not unitary. This is due to the conditional breaking. It can be easily shown that the sum of the outgoing matrices and the sum of the incoming matrices of any given state must be unitary. Moreover the transition from q_3 to q_1 is impossible to take since the input cannot be 1 (we branched on a 0), same goes for the transition from q_4 to q_2. The reduced machine is

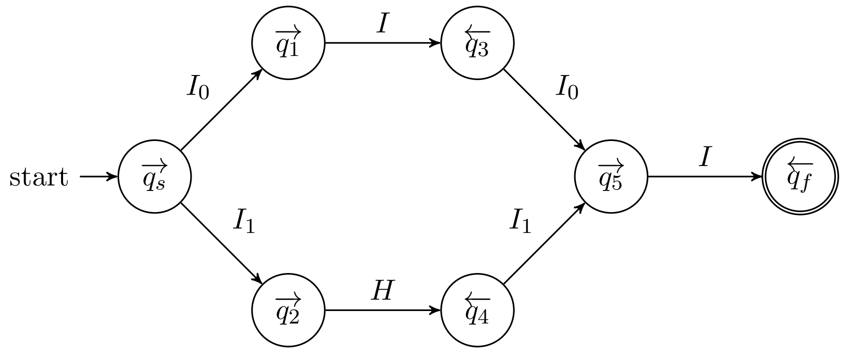

Figure 2.7: The previous machine with useless transitions removed

2.9 A Middle Ground Machine and the Halting Scheme

The previous model makes creating quantum Turing machines very simple, however the fact that all computational branches need to halt at the same time makes conditional looping a very complicated task to accomplish. We therefore defined a Middle Ground Quantum Turing Machine (MG-QTM) to be a unidirectional rotational quantum Turing machine whose branches may not necessarily halt at the same time. And we introduce to the configuration a halt qubit that lives in a Hilbert space spanned by $\{|0\rangle, |1\rangle\}$. The halting scheme is presented and proved in [Oza02].

Every valid quantum algorithm should set the halting qubit to 1 once its halt. After every transition we observe the halting qubit, if we find it to be 1 then the machine is considered to halt. If it were 0 then we apply another transition and the process is repeated. One might wonder whether at some point during the computation the halting qubit will be entangled with the rest of the configuration. Matters of fact it might, this measurement might appear to spoil the computation, but the probability distribution of the output (the tape) is not spoiled. This method is referred to as Quantum Nondemolition Monitoring. Below we present a sketch of the proof.

Proof. (Sketch) Let $|C\rangle = |q, h, \xi, T\rangle$ be a given configuration with $|h\rangle$ being the halt qubit. Let $\hat{P} = \mathcal{I}_Q \otimes |1\rangle \langle 1| \otimes \mathcal{I}_Z \otimes \mathcal{I}_{\Sigma*}$ be the projection on the space of the halt qubit. And let $\hat{Q}_j = \mathcal{I}_Q \otimes \mathcal{I}_H \otimes \mathcal{I}_Z \otimes |T_j\rangle \langle T_j|$ be the projection of a tape output T_j. Then what we wish to prove is

$$Pr(T_j|monitored) = Pr(T_j|\neg monitored)$$

i.e. the probability of measuring T_j using the halting scheme is the same as the probability of the measuring T_j using N transitions. It is simple to see that

$$Pr(T_j|\neg\text{monitored}) = ||\hat{P}\hat{Q}_j U^N |C\rangle ||^2$$

$$Pr(T_j|\text{monitored}) = \sum_{n=0}^{N} ||\hat{P}\hat{Q}_j (U\hat{P}^{\perp})^n |C\rangle ||^2$$

The remainder of the proof can be found in [Oza02] or in [Oza98] presented in a more elaborate fashion. □

Chapter 3

Braqet – A Functional Quantum Programming Language

3.1 Current Status

As mentioned previously, George Zakhour is responsible for this part of the project. Many obstacles needed to be overcome because of the lack of contribution to the field of high-level programming languages for “pure” (i.e. independent of a classical machine) quantum computers, much less functional languages. All usual programming constructs used classically need to be converted to valid quantum ones. The difficulty here lies in the underlying nature of Nature: every quantum operator needs to be represented by a unitary invertible matrix.

We have successfully converted classical arithmetical operations performed on a classical computer to their quantum counterparts, these include addition, multiplication, subtraction, division and the modulo operation (all operations being made modulo 2^n where n is the number of qubits being acted upon). Because jumps violate reversibility we had to account for conditional breaks, function calls (or operators as they shall be called in the quantum context) and looping. The first two problems alongside the problem of performing counting loops were solved in the Summer of 2014. And recently, in early April, we were able to solve the problem of conditional looping.

We chose the syntax of the language to be an S-expression syntax (i.e. LISP-like syntax), we developed the operational semantics and extended the one developed for a quantum lambda calculus with classical control[1] in [SV05].

3.2 Vision of the programming language

Below are some examples that we hope could be compiled verbatim and can illustrate our vision of what the programming language will look like

3.2.1 Deutsch–Jozsa Algorithm

The Deutsch–Jozsa algorithm figures out whether a given black-box function $f : \{0,1\}^n \to \{0,1\}$ is a constant function or a balanced function (for half its input it is 0 and for the other half it is 1). Classically this requires $O(2^n)$ calls to the black-box, the Deutsch–Jozsa algorithm performs this check deterministically in $O(n)$ calls to the black-box function.

```
; Create a new operator that performs H^2 on two qubits
(define (H2 a b)
        ((H a) (H b)))

; Create the Deutsch operator, given a the blackbox number and the qubits
(define (deutsch n in out)
        ((lambda (x y) ((H x) (H2 y)))
            (fCnot n (H in)
                    (H2 out))))

; Create the Deutsch function, given the number of its blackbox
(define (main-deutsch n)
        (measure (deutsch n (qubit a) (qubit b 2))))

(main-deutsch 0)
(dump)
```

Which will be called in the following fashion

```
> qvm ./deutsch -b 127.0.0.1:5000
```

Where `qvm` is the executable used to run the virtual machine, `./deutsch` is the compiled `qbin` byte-code of the previous source code. Since this algorithm uses an external black-box then it must be provided. This black-box will be hosted and will be queried on the following address `127.0.0.1:5000`

[1]Classical control is the use of the classical control flows, for example when the

which is passed to the virtual machine using the `-b` flag. We hope the output would be:

```
a = 0
b = 0
[DUMP] 3 qubits, |000>
```

3.2.2 Quantum Teleportation

Bellow is a program that teleports the content of the qubit `a` to the another qubit `b`.

```
; Assumes |a> = |b> = |0> and transforms them into the entangled state
; 0.707 |00> + 0.707 |11>
(define (entangle a b)
        (Cnot (H a) b))

; Creates the necessary transformations before measuring
(define (pre-teleport a b c)
        ((entangle a b) c))

; Teleport the content of the qubit from a to b
(define (teleport a b)
        ; Perform the corresponding correction on the third
        ; qubit based on the observations made on the first two
        ((lambda (x y z)
                 (ifelse (lambda (y z)
                                 (ifelse Z X y z))
                         (lambda (y z)
                                 (ifelse I X z y))
                         x (y z)))
            ; measure the first two qubits
            ((lambda (x y z)
                     ((measure (x y)) z))
                ; perform the pre-teleportation transformation
                (pre-teleport a
                              ; entangle the given qubit with a new one
                              (entangle (qubit y) b)))))


; Perform teleportation from a to b
(qubit a)
```

```
(qubit b)
(T a) ; T is any transformation defined
(dump)
(teleport a b)
(dump)
```

Would produce the output

```
[DUMP] 2 qubits, 0.8660254 |00> + 0.5 |10>
a = 0
y = 1
[DUMP] 3 qubits, 0.8660254 |000> + 0.5 |001>
```

3.3 Elaboration on Solutions

3.3.1 Working with 1s and 0s on a finite tape

In the examples presented bellow we work with 0s and 1s on a finite tape. Here we present a proof that this can be easily translated to a QTM with a bilateral countably infinite tape with a special blank symbol #.

Lemma 1 (**Finite Completion Lemma**). *Given a finite QTM M with alphabet $\Gamma = \{0, 1\}$ without the blank symbol #, then it can be translated to a QTM T with a bilateral infinite tape whose alphabet includes #.*

3.3.2 Function Calls

A function, or an operator, will be an alias to a successive application of atomic operations (being the ones predefined, for example the Pauli matrices I, X, Y, Z etc...). For example, if we are given an operator $U : \mathcal{H}^{\otimes n} \to \mathcal{H}^{\otimes n}$ that transforms an n-qubit register. Then we can write U as $U = \prod_{i=m}^{i=0}(\mathbb{I}^{\otimes k-1} \otimes U_i \otimes \mathbb{I}^{n-k})$ where U has m instructions and U_i is the i-th instruction in the definition of U that acts on the k-th qubit of the input.

3.3.3 Conditional Breaking

We were inspired by the Cnot operator that is defined by

$$Cnot\,|x, y\rangle = |x, x \oplus y\rangle$$

which can be thought of as a conditional application of the swap on $|y\rangle$ if $|x\rangle = |1\rangle$. In general these sorts of gates are written as $\bigwedge U$ where U is the operator to apply to the second (and later) qubits if the first is 1. In matrix form, we could express it as

$$\bigwedge U = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & U \end{bmatrix}$$

where 0 and $\mathbb{I}$ are the 0 matrix and the identity whose dimensions match U's.

To create an if-else-like operator then we use the notation $A \bigwedge B$ to if the first qubit is 0 then A is applied to the rest of the qubits, and if the first qubit is 1 then B is applied to the rest of the qubits. One could express it in matrix form

$$A \bigwedge B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}$$

where we assume the dimensions of A, B and the 0 matrix to match.

Theorem 4. *$A \bigwedge B$ is a unitary matrix if and only if A and B are unitary.*

Proof. $(A \bigwedge B)(A \bigwedge B)^\dagger = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix} \begin{bmatrix} A^\dagger & 0 \\ 0 & B^\dagger \end{bmatrix} = \begin{bmatrix} AA^\dagger & 0 \\ 0 & BB^\dagger \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & \mathbb{I} \end{bmatrix}$ if and only if A and B are unitary. □

3.3.4 Counting Loops

Counting loops are very simple and can also be expressed in matrix form, very much like the two cases above. Let U be an operator that is applied to a qubit register n times, then its equivalent matrix would very simply be U^n. One can also see this from the matrix form of a function whose instructions are n applications of U on a given input.

3.3.5 Conditional Loops

To present conditional loops we will need to use our MG-QTM model, it is then very easy to construct a looping quantum Turing machine. We were able to design a 70 state quantum Turing machine that acts on two 3-qubit registers to perform the following algorithm

```
# Given two 3-bit numbers x and k
while |x-k| < 4 do
    x = x + 1
end
```

The general idea of performing loops on a quantum computer is having a transition from a state back to the start state. We present in Figure (3.1)

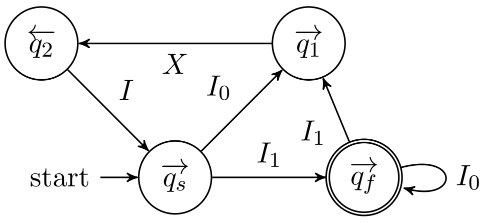

Figure 3.1: A machine that increments its tape content if its first qubit was 0

an example of a machine that intrigued us that performs either 0 or 1 iterations, i.e. visits the start state once or twice.

The reason this machine is interesting because it is a decider and if we consider as input the set of 2-qubit registers then for no input does the machine produce the output $|00\rangle$. Naturally this raises the question of what the inverse machine (the machine that reverses the computations of a given machine) behaves like with input $|00\rangle$? The machine loops forever on this input! This has lead us to conjecture the following

Conjecture 1. *If M is a GM-QTM that decides a function $f : \{0,1\}^n \to \{0,1\}^n$, and if M^{-1} is the inverse machine then M^{-1} does not halt on any input in $\{0,1\}^n / f(\{0,1\}^n)$.*

3.4 Syntax and Semantics

The syntax of Braqet is LISP-like given by the following Context-Free Grammar

$$\begin{aligned} Program \to &\quad Term \quad Program \quad | \quad \varepsilon \\ Term \to &\quad (\quad Program \quad) \quad | \quad Atom \\ Atom \to &\quad IDENTIFIER \quad | \quad NUMBER \end{aligned}$$

The semantics of the language are exactly the same as the ones mentioned in [SV05] with the difference that the conditional break is adapted to exploit the $\bigwedge$ operator relation. Moreover we have the conditional looping based on the halting scheme previously described.

3.5 Braqet v0.1

For the project we wrote a compiler that implements most of the features mentioned earlier. Some of the features that were not implemented were conditional looping since the assembly code did not feature conditional looping, anonymous functions, and non-unitary (i.e. non-duplicable or use-once) functions.

Below is the help and usage dialog of the compiler that features a small tutorial, and a working example

```
Command Line Options:
  -o                Specify the output file
  --asm             Compile to assembly language only
  --symboltable     Print the symbol table to the standard output
  -h, --help        Displays this message dialog

Programming Language:
Braqet is a pure quantum language with quantum control However not all
constructs are currently supported. Currently the language lacks conditional
looping, the definition of functions as opposed to operators and lambda
functions.

The language is in the LISP family. Below is an introduction on how to write
with braqet

* creation of a qubit |x>:
  (qubit x)
* creation of a 5-qubit register |x5>:
  (qubit x5 5)
* application of the hadamard operator on |x>:
  (H x)
* application of the Controlled-Not operator on 2 qubits |x,y>:
  (Cnot x y)
* measurement of a qubit |x>:
  (measure x)
* measurement of a qubit register |x5>:
  (measure x5)
* Creating the H^2 operator:
  (define (H x y) (H x) (H y))
* Creating the entangling operator:
  (define (entangle a b) (Cnot (H a) b))
* Performing counting loops, perform X 4 times on |a>:
  (repeat X a 4)
* Performing Y conditionally on |x> based on |a>:
  (if Y x a)
* Performing Y conditionally on |x> based on |a>, otherwise X:
  (ifelse Y X x a)
* Use a blackbox labeled 0 with |x> as input and |y> as output handler:
  (fcnot 0 x y)
```

```
All defined functions must be unitary operators. Therefore no qubit creation,
qubit measurement or function creation inside a function definition.

Bellow is an example of the Deutsch-Jozsa algorithm performed on a function
f: {0..2^4-1} -> {0, 1} provided as a blackbox labeled 0


(define (H4 a b c d)        ; Create the H^4 operator
        (H a) (H b)
        (H c) (H d))
(define (H5 a b)            ; Create the H^5 operator
        (H4 a) (H b))
(measure                    ; Apply the Deutsch-Jozsa algorithm
    (H5 (fcnot 0
            (H4 (qubit a 4))
            (H (X (qubit b))))))
```

The example mentioned above will compile into either assembly code or byte code. The output of the assembly version of the code above where a is a 2 qubit register is the following

```
fn H2
load a_0
load b_0
on a_0
apply H
on b_0
apply b_0
endfn
fn H3
load a_0
load a_1
load b_0
on a_0
on a_1
apply H2
on b_0
apply H
endfn
qubit a_0
qubit a_1
on a_0
on a_1
apply H2
```

```
qubit b_0
on b_0
apply X
on b_0
apply H
on a_0
on a_1
on b_0
fcnot 0
on a_0
on a_1
on b_0
apply H3
measure a_0
measure a_1
measure b_0
```

3.6 MG-QTM Simulator

We have also written a program that can simulate and visualize a given MG-QTM. The graphs showcased in this document are actually generated from this program. To visualize we are using a force-layout graph drawing algorithm. The program will be also packaged with the final software. Bellow is the help page of the program.

```
Usage: srqtm [options...] filename

Command Line Options:
  -a, --auto Simulate the machine until it halts
            default behavior is manual stepping, initiated by pressing Enter
  -t, --tape=N Sets the length of the tape to N, default is 5
  -i, --input=N Writes to the tape the binary representation of N
  -h, --help Displays this message dialog
  -v, --verbose Be verbose

Graphical Interface Options:
  <click>Selects a node (if clicked on it) or agitates the system
  d  Deletes the force between two selected nodes but keeps edge
  h  Hides the edge between two selected nodes, the force is not affected
  x  Hides the edge and deletes the force between two selected nodes
  t  Writes the Tikz representation to the standard output
  ESC Quits the application
To select a node click on it, nodes are stored in pairs and order matters.
A node can be selected multiple times, for example consider the following graph:
        --(A)->
  q1 _/       \_ q2
      \<_(B)__/
And you wish to delete both edges, then you would click on q1 then q2
to select (A) followed by clicking on q2 then q1 to select (B) and
you would press x
```

3.6.1 Force-Layout Graph Drawing Algorithm

Force-layout graph drawing algorithms are a class of algorithms that considers a graph to represent a physical system whose internal forces move it around to reach a state of equilibrium. More often than not this state of equilibrium produces aesthetically pleasing drawings of graph. We have used a model where nodes are considered charged particles (electrons for example) repelling each other. The edges between two nodes are considered to be springs pulling these nodes together. Refer to [Tam07] for more detailed explanation.

Given a pair of nodes (A, B) where $d(A, B)$ represents the distance between A and B, then the repulsive force applied on A by B is

$$F_r(A, B) = -\frac{30000}{d(A, B)^2}$$

while the attractive force applied on A by B is

$$F_a(A, B) = 0.03d(A, B)$$

Therefore, if we denote by V the set of all nodes of a graph and $N(A) \subseteq E$ the set of all neighbors of A, then for any given node A, the force acting on the particle representing A is

$$F(A) = \sum_{v \in V/\{A\}} \frac{\overrightarrow{Av}}{\left|\overrightarrow{Av}\right|} F_r(A, v) + \sum_{v \in N(A)} \frac{\overrightarrow{Av}}{\left|\overrightarrow{Av}\right|} F_a(A, v)$$

From Newton's laws of motion one can derive a good enough approximation of the position at a new time index t given by

$$x_{t+1}(A) = x_t(A) + F(A)\Delta t^2$$

3.6.2 Simulating an MG-QTM

The simulator accepts a `.qtm` file that describes a quantum Turing machine in the following fashion. For every $\delta(q, \sigma, p, \tau, \{-1, 1\}) \neq 0$ then there is a line in the file that follows this pattern

```
q p<DIR>(a11, a12, a21, a22);
```

where `q` and `p` are string representations of q and p respectively, `<DIR>` is `<` if $d = -1$ otherwise `>` and `anm` are the nm-th entry of the projection matrix $|\tau\rangle\langle\sigma|$. If q has other transitions then they could be written on the same line as follows:

```
q p1<DIR1>(a11, a12, a21, a22) p2<DIR2>(b11, b12, b21, b22);
```

There must be `_s` and `_f`, the start and final state respectively, the machine has to be unitary, i.e. there cannot be two tokens of the form `q>(` and `q<(` in the file, and the sum of the incoming and outgoing matrices should be unitary. If a branch halts before an another, or if the machine halts and its head is not at the first cell or if the head is no longer in a deterministic position then an error is raised and the simulation stops.

Machine that increments the first 2 qubits if the first qubit is 0

The state transition diagram of this machine is given in Figure 3.1. The following is the code and the output of the simulator.

```
_s _f>(0,0,0,1) q1>(1,0,0,0);
q1 q2<(0,1,1,0);
q2 _s>(1,0,0,1);
_f q1>(0,0,0,1) _f>(1,0,0,0);
```

The output of the simulation with default tape length and tape content is

```
+1 |_s,0,00000>
+1 |q1,1,00000>
+1 |q2,0,01000>
+1 |_s,1,01000>
+1 |_f,2,01000>
```

Machine that computes the conditional Hadamard gate $\bigwedge H$

The state transition diagram of this machine is given in Figure 2.3-2.4. Bellow is the code and the output of the simulator.

```
_s q1>(1,0,0,0) q2>(0,0,0,1);
q1 q3<(1,0,0,1);
q2 q4<(0.707,0.707,0.707,-0.707);
q3 q5>(1,0,0,0) q1>(0,0,0,1);
q4 q5>(0,0,0,1) q2>(1,0,0,0);
q5 _f<(1,0,0,1);
_f _s>(1,0,0,1);
```

The output of this simulation with tape length of 2 and input 3 ($|11\rangle$) is

```
+1 |_s,0,11>
+1 |q2,1,11>
+0.707107 |q4,0,10> -0.707107 |q4,0,11>
+0.707107 |q5,1,10> -0.707107 |q5,1,11>
+0.707107 |_f,0,10> -0.707107 |_f,0,11>
```

The output of this simulation with tape length of 2 and input 1 ($|01\rangle$) is

```
+1 |_s,0,01>
+1 |q1,1,01>
+1 |q3,0,01>
+1 |q5,1,01>
+1 |_f,0,01>
```

Chapter 4

Quantum Virtual Machine

4.1 Component Descriptions

The project is split into 3 different components:

- The `qlib` library which contains utilities needed in order to read and execute quantum programs
- The QVM Assembler which converts an assembly file into a byte-code format.
- The QVM which executes quantum programs.

4.2 Qlib

This is the library which will contain most of the utilities needed by the virtual machine. The goal of this library is to allow different implementations of the virtual machine without having to write the boilerplate such as parsing files and communicating over the network.

Since `qlib` is going to contain abstractions for low level operations, modules and classes are going to be added to it depending on what is needed. Bellow is what is already implemented

qlib.collections contains some useful data structures for the virtual machine.

qlib.qbin contains abstractions needed for reading and writing binaries.

qlib.blackbox contains the abstractions needed for communicating with black boxes.

4.2.1 qlib.qbin

A qbin file is the input formatted using the byte-code specification of the Virtual Machine. It contains the instructions to be executed, as well as any custom operators that might be defined by the user. The rest of this section is an overview of the qbin file format.
A qbin file consists of a sequence of sections. A section is either an identifier section and a function section. Each section starts with two signature bits and a 16 bit unsigned integer each section interpreting it differently.

An identifier section, whose signature is 01, contains a string of arbitrary length, which can represent a qubit, a function or anything with a name really. The index of the n-th identifier read is n, the index of the identifier is not actually stored in the file, it is implied by the order in which the identifiers are read by the parser.

A function section, whose signature is 00, stores a sequence of instructions, each instruction is defined later in the assembler section. Each file contains at least 1 function: main, which as the name implies is the first function executed by the Virtual Machine.

In the byte-code, each instruction is split into five sections:

1. Opcode (4 bits) which is the opcode of the instruction to be executed.
2. Qubit (7 bits) which represents the index of the identifier of a qubit in the state.
3. Operators (7 bits), each instruction consists of 1 or 2 operators; 7 bits each, which represent operators to be applied on the qubits in the queue.
4. Number (unsigned 8 bits) which is useful for storing external metadata such as a black box index or the number of loops of a counting loop.
5. Line Number (16 bits) The line number either in the source code of a compiled language or in the assembly code this instruction is associated with. Stored for debugging and error reporting purposes.

The bit lengths are completely arbitrary and can be changed with little to no effort later. Due to the limitations of the format obviously it allows a maximum of 2^7 qubits and $2^7 - k$ user defined operators, k being the number of operators defined by the Virtual Machine. Considering the fact that this Virtual Machine is meant to be run on home computers these numbers should be enough.

4.2.2 qlib.collections

This is the main module the Virtual Machine will be dealing with. The most important class in this module is `Program`, a class that acts as an iterator over the instructions to be executed, as well as providing an interface to handle control flow. It also includes some helper functions in order to read and write qbin files.

4.2.3 qlib.blackbox

This module is currently in development, it will contain the implementation of a defined network protocol that is used to query black boxes, as well as a wrapper for the server in order to allow users to implement black boxes in D. The user supplies the address of the black box to the Virtual Machine, and it, in turn queries the black box.

A black box is defined as a function f that takes an n-bit string as input, and outputs an m-bit string. In symbols, $f : \{0,1\}^n \to \{0,1\}^m$. The size of the resulting $fCnot$ matrix will be of size $2^{m+n} \times 2^{m+n}$, fortunately it is sparse so it can be represented with $O(2^{m+n})$ space.

Each black box server must provide the following information to the Virtual Machine.

- The number of black box functions implemented on this server.
- Given the index of the black box, it must provide the size of the size of the input and the size of the output.
- Given the index, the input and it's length in bits, it must provide an output whose length is expected. If the input is of incorrect size it should return an error message.

The black box is needed by the $Ccnot$ operator, which queries the black box in order to generate an operator that is defined as follows. If we are given $f : \{0,1\}^n \to \{0,1\}^m$, $|x\rangle$ an n-qubit register, $|y\rangle$ an m-qubit register, then $fCnot\,|x, y\rangle = |x, y \oplus f(x)\rangle$, where addition is made modulo 2^m.

4.3 Quantum Virtual Machine Assembler

This component is an executable that simply converts an assembly code into byte-code. No semantic checking is done at this level, it is performed during execution. The assembler only fails if there is a syntax error in the code, otherwise the code is converted into the byte-code format we defined (refer to later section). This component is currently fully implemented and tested.

4.3.1 Assembly language definition

The assembly language is a sequence of instructions which may contain custom operators defined by the user. Any instructions read in the assembly language which are not contained in a function are considered part of main. Any user defined function must begin with `fn` and ends with `endfn`.

The valid instructions are defined below

qubit q Adds a qubit to the current state and initializes it to $|0\rangle$, q is added to the symbol table refering to the index of the aforementioned qubit in the state.

if cq op1 If the control qubit is $|1\rangle$ then this instruction applies the operator op1 on the qubits in the queue, otherwise it applies the identity. I.e. this instruction performs the $\bigwedge Op_1$ gate.

ifelse cq op1 op2 If the control qubit is $|1\rangle$ then this instruction applies the operator op1 to the qubits in the queue, otherwise it applies op2. I.e. this instruction performs the $Op_2 \bigwedge Op_1$ gate.

measure q Measures the qubit q and renormalizes the state.

loop op1 n Applies the operator op1 n times on the qubits in the queue.

on q push the qubit q on the queue, this instruction is often used right before the application of a gate.

apply op Applies the operator op on the qubits in the queue. After the application of the qubit the queue is emptied, even if not all qubits were transformed.

load q Unloads a qubit from the queue and associates it with the name q. Used in user defined functions.

srec name Starts a classical timer named name..

erec name Ends a classical timer named name.

qsrec name Starts a quantum timer named name.

qerec name Ends a quantum timer named name.

print name Prints the content of a classical variable, for now this action is only possible when name refers to some time counter.

dump Prints the full state of the machine. This is only useful for debugging and illustrative purposes.

fcnot n Applies the $fCnot$ operator as defined earlier, f is queried at the address provided by the user, n being the index of the black box.

fn name Starts a new function definition.

endfn Ends a function definition.

4.3.2 Example program

The following program entangles two qubits creating the bell state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

```
qubit x
qubit y

fn bellStates
   load g
   load h
   on g
   apply H
   on g
   on h
   apply cnot
endfn

on x
on y
apply bellStates
measure x
measure y
```

4.3.3 Limitations

Unfortunately the only thing missing from this language is conditional looping. Once the language is complete we will modify the byte-code and the Virtual Machine to support this feature. Obviously all programs currently halt making the language non-Turing complete. Once conditional looping is supported then the language will become Turing complete.
Moreover it is not advised to exceed 15-20 qubits since the amount of memory required to store the state and the operators is exponentially large.

4.4 QVM v0.1

The quantum virtual machine has not been fully implemented, the only feature that is still missing is the application of the `fcnot` instruction even though most of the underlying infrastructure to complete this task has been developed.

Chapter 5

The ALU – Performing the computations

We will discuss in this chapter the problem of performing computations in the Virtual Machine and what are the algorithmic limitations of each solution.

5.1 Overview

Consider the n-qubit register $|\Psi\rangle$, then

$$|\Psi\rangle = \alpha_1 |\Phi_1\rangle + \ldots + \alpha_{2^n} |\Phi_{2^n}\rangle$$

where each $|\Phi_i\rangle$ represents the i-th basis vector in the Hilbert space $\mathcal{H}^{\otimes n}$. If one wishes to store this state then 2^n complex numbers must be stored, i.e 2^{n+1} floating numbers. Our goal is to simulate 32 qubits which requires at most 64GB of RAM. In order to evolve the state of the machine $|\Psi\rangle$, we need to apply a linear operator to the state, which is a $2^n \times 2^n$ matrix. This is obviously very expensive (time-wise and space-wise). Luckily, there are solutions to these problems in which one need not matrix multiplication to apply the operator. We present these solutions in the upcoming sections.

5.2 State Representation and Shortcuts

Given what is said above, superposition forces us to have at most 2^n possible states with complex coefficients. This worst case occurs almost always in any quantum algorithm[1] in quantum algorithms and needs to be dealt with, thus good memory management and compression is required.

[1]Since quantum interference is what makes quantum computing so efficient, almost all algorithms start by applying the $H^{\otimes n}$ operator on the state to get a uniform distribution of all the possible states

5.2.1 Exploiting The Tensor Product and Factorization

The first aid to reduce memory consumption comes from the tensor product which can be used to factor states. The quantum state might be represented by the tensor product of independent quantum sub-states (will be referred to by *clusters*). For example, consider the application of $H^{\otimes 4}$ on a 4-qubit register $|\Psi\rangle$ initialized to the classical $|0\rangle$,

$$|\Psi\rangle = \frac{1}{4}\sum_{i\in\mathbb{N}_{16}} |i\rangle \qquad (|i\rangle \text{ is a basis vector of } \mathcal{H}^{\otimes 4}) \tag{5.1}$$

We denote by $|\Psi_i\rangle$ the i-th qubit in $|\Psi\rangle$. Explicitly expanding the state $|\Psi\rangle$ in many ways, we obtain

$$\begin{aligned}
|\Psi\rangle &= |\Psi_1\rangle \otimes |\Psi_2\rangle \otimes |\Psi_3\rangle \otimes |\Psi_4\rangle && (5.2)\\
&= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) && (5.3)\\
& && (5.4)\\
|\Psi\rangle &= |\Psi_1\Psi_2\rangle \otimes |\Psi_1\Psi_2\rangle && (5.5)\\
&= \frac{1}{2}(|0\rangle + |1\rangle + |2\rangle + |3\rangle) \otimes \frac{1}{2}(|0\rangle + |1\rangle + |2\rangle + |3\rangle) && (5.6)\\
& && (5.7)\\
|\Psi\rangle &= |\Psi_1\rangle \otimes |\Psi_2\Psi_3\rangle \otimes |\Psi_4\rangle && (5.8)\\
&= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{2}(|0\rangle + |1\rangle + |2\rangle + |3\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) && (5.9)
\end{aligned}$$

Observe from (5.3) that if one stores each cluster independently than the other (i.e. without expanding them) then one needs to store only 2 complex numbers per cluster resulting in just 8 complex numbers as opposed to 16 if one would store the state as in (5.1). The advantages of this representation are not only restricted to state memory management, it has direct effect on the application of an operator on a state and the storage of said operator, these are mentioned in a later section.

The only problem is that most of the time the state is not fully factorisable as in (5.3) but may be partially factorisable. If an n-qubit register is fully factored, i.e. the number of clusters is n then it is sufficient to store $2n$ complex numbers as opposed to 2^n. If, however, the state is not fully factorisable then an improvement is also noticeable!

5.2.2 Compression

Assuming a worst case scenario where the state is in no way factorisable, one would have to find another solution for memory management; compressing the actual coefficients. We are currently working on implementing a lossless compression algorithm for 64-bit floating-point values that is fast enough to support software-based real-time compression and decompression [RKB06].

5.3 Quantum Operations

As previously mentioned, quantum computations are done by applying an operator to the state vector. Luckily, since the quantum state will always be written as a superposition of the basis of $\mathcal{H}^{\otimes n}$, one work around to reduce the $O(n^2)$ time complexity for matrix-vector multiplication.

5.3.1 The Application super-operator

Let U be an operator that acts on d qubits, then applying U on $|\Phi_m, \ldots, \Phi_{m+d-1}\rangle$ of some n-qubit register $|\Phi\rangle$ is performed by applying the following super-operator on $|\Phi\rangle$

$$\Lambda_{n,d} : L(\mathcal{H}^{\otimes d}) \times \mathbb{N}_{n-d-1} \to L(\mathcal{H}^{\otimes n})$$
$$\Lambda_{n,d}(U, m) = \mathbb{I}^{\otimes m-1} \otimes U \otimes \mathbb{I}^{\otimes n-m-d+1}$$

where $L(V)$ is the set of all linear transformations from V to V. For example, applying the *Cnot* operator on the 4-th qubit of the 10-qubit register $|1004\rangle$, is equivalent to applying $\Lambda_{10,2}(Cnot, 4)$ on $|1004\rangle$.

5.3.2 Using Clusters

Let $|\Psi\rangle$ be an n-qubit register that is factorised into k clusters

$$|\Psi\rangle = \bigotimes_{i=1}^{k} |\Psi^i\rangle$$

where $|\Psi^i\rangle$ refers to the i-th cluster. Applying an operator U to $|\Psi_i, \ldots, \Psi_{i+d}\rangle$, i.e. the register made up of the i-th qubit to the $i + d$-th qubit, is trivially the application of $\Lambda_{n,d}(U, i)$ to $|\Psi\rangle$. However if the i-th qubit to the $i + d$-th qubit were in the same cluster $|\Psi^k\rangle \in \mathcal{H}^{\otimes m}$ at index j, then applying U to these qubits gets reduced to the application of $\Lambda_{m,d}(U, j)$ on $|\Psi^k\rangle$, hence reducing the trivial matrix-vector multiplication runtime from $O(2^{2n})$ to $O(2^{2m})$ where $m \leq n$ is the number of qubits in $|\Psi^k\rangle$.

However if the i-th qubit to the $i + d$-th qubit were to range over l clusters from the k_1-th cluster $|\Psi^{k_1}\rangle \in \mathcal{H}^{\otimes m_1}$ to the k_l-th cluster $|\Psi^{k_l}\rangle \in \mathcal{H}^{\otimes m_l}$ at index j then we expand these clusters, i.e. performing the computation

$|\Psi^{k_1}\rangle \otimes \cdots \otimes |\Psi^{k_l}\rangle$, and apply $\Lambda_{m_1+\cdots+m_l,d}(U,j)$.

If measure the i-th qubit in the k-th cluster $|\Psi_i^k\rangle \in \mathcal{H}^{\otimes m}$ then we split the k-th cluster to 3 clusters

$$|\Psi^{k_1}\rangle = |\Psi_1^k, \ldots, \Psi_{i-1}^k\rangle\,, \quad |\Psi^{k_2}\rangle = |\Psi_i^k\rangle\,, \quad |\Psi^{k_3}\rangle = |\Psi_{i+1}^k, \ldots, \Psi_m^k\rangle$$

5.3.3 Using the Basis Vectors of the Hilbert Space

After designing a method for compressing and reducing both the operators and the states, we will design an algorithm to perform the matrix-vector multiplication. This algorithm is linear, hence faster than the trivial one if half of the probability amplitudes are 0. From basic linear algebra, if the i-th basis vector of a vector space is multiplied by a matrix U with respect to the same basis, then the result is the i-th column vector of U. Hence if $|\phi_i\rangle$ is the i-th column vector of U, then $U(\alpha_i\,|i\rangle) = \alpha_i\,|\phi_i\rangle$. The resulting vector is then written as linear combination of the basis vectors of $\mathcal{H}^{\otimes k}$. If however the number of probability amplitudes not zero are numerous, then we will add up all the basis vectors to get

$$|\phi_k\rangle = \alpha_0\,|0\rangle + \cdots + \alpha_{2^n-1}\,|2^n-1\rangle$$

and then multiply it by U and rewrite the output as a linear combination of the basis.

5.4 ALU of QVM v0.1

The ALU of the QVM is complete. It can split the state of a machine into clusters, apply operators on these clusters either individually when possible or by expanding them when needed. Moreover the ALU can perform measurement and factor when computationally possible.

Bibliography

[BV93] Ethan Bernstein and Umesh Vazirani, *Quantum complexity theory*, in Proc. 25th Annual ACM Symposium on Theory of Computing, ACM, 1993, pp. 11–20.

[Deu85] D. Deutsch, *Quantum theory, the church-turing principle and the universal quantum computer*, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences **400** (1985), no. 1818, pp. 97–117.

[DJ92] David Deutsch and Richard Jozsa, *Rapid solution of problems by quantum computation*, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences **439** (1992), no. 1907, 553–558.

[Fey82] RichardP. Feynman, *Simulating physics with computers*, International Journal of Theoretical Physics **21** (1982), no. 6-7, 467–488.

[Gro96] Lov K. Grover, *A fast quantum mechanical algorithm for database search*, ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, ACM, 1996, pp. 212–219.

[LY13] M. Liang and L. Yang, *On a class of quantum Turing machine halting deterministically*, Science China Physics, Mechanics, and Astronomy **56** (2013), 941–946.

[ON98] Masanao Ozawa and Harumichi Nishimura, *Local transition functions of quantum turing machines*, RAIRO Theor. Inform. Appl (1998), 379–402.

[Oza98] M. Ozawa, *Quantum Nondemolition Monitoring of Universal Quantum Computers*, Physical Review Letters **80** (1998), 631–634.

[Oza02] Masanao Ozawa, *Halting of quantum turing machines*, Unconventional Models of Computation, Lecture Notes in Computer Science, vol. 2509, Springer Berlin Heidelberg, 2002, pp. 58–65.

[RKB06] P. Ratanaworabhan, J. Ke, and M. Burtscher, *Fast lossless compression of scientific floating-point data*, Data Compression Conference, 2006. DCC 2006. Proceedings, March 2006, pp. 133–142.

[Sho99] Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Review **41** (1999), no. 2, pp. 303–332.

[Sim94] Daniel R. Simon, *On the power of quantum computation*, SIAM Journal on Computing **26** (1994), 116–123.

[SV05] Peter Selinger and Benot Valiron, *A lambda calculus for quantum computation with classical control*, In Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (tlca), Volume 3461 of Lecture Notes in Computer Science, Springer, 2005, pp. 354–368.

[Tam07] Roberto Tamassia, *Handbook of graph drawing and visualization (discrete mathematics and its applications)*, Chapman & Hall/CRC, 2007.