Due: May 13th, 2016
Date: June 8, 2016

Midterm

Raleigh Foster, Dustin Reid

1 Outline

We have done work in programming in quantum programming languages, and studying quantum lambda calculi.

2 Quantum Programming with Microsoft Liquid

2.1 Overview

We have implemented Deutsch's algorithm on a single Qubit. We also tried scaling up their implementation of Shor's algorithm. Factoring 1023 (10 qubits) took 1 hour 8 minutes. We can do more performance tests later if desired. For our final, we will probably also be implementing other algorithms.

2.2 Implementing Deutch's Algorithm

First, for convenience, we define each possible function $f : \{0,1\} \to \{0,1\}$. This will make testing our implementation for each f easy to manage. For simplicity our functions map from $\{0,1\} \to \{true, false\}$, this does not affect the simulation.

```
    let f0 (n:int) = false

let f1 (n:int) = not (n = 0)

let f2 (n:int) = (n = 0)

let f3 (n:int) = true
```

Next, an oracle for the unitary U_f acting on two qubits is defined. In Liquid, unitary gates are defined as functions which take qubit lists as arguments. We need $U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$, for any $f : \{0,1\} \to \{0,1\}$. Of course, the actual entries of U_f are dependant on the function f.

For this reason, we have defined U_f as function which takes not only a qubit list, but a function f as well. The values of $f(0)$ and $f(1)$ are determined, and then the corresponding unitary operator is contrsucted. In the cases where $f(0) = f(1) = 0$ and $f(0) = 0$, and $f(1) = 1$, the built-in Identity and CNOT gates are used respectively. Otherwise, new gates must be defined.

```
let Uf (qs:Qubits) f = let (x, y) = ((f 0), (f 1))
                       match (x, y) with
                           | (false, false) -> I qs
                           | (false, true) -> CNOT qs
                           | (true, false) ->
                              let gate =
                                  new Gate(
                                         Mat = ( CSMat(4, [(0,1,1.,0.); (1,0,1.,0.);
                                                           (2,2,1.,0.); (3,3,1.,0.)])),
                                         )
                              gate.Run qs
                           | (true, true) ->
                              let gate =
                                  new Gate(
                                         Mat = ( CSMat(4, [(0,1,1.,0.); (1,0,1.,0.);
                                                           (3,2,1.,0.); (2,3,1.,0.)])),
                                         )
                              gate.Run qs
```

Our method of defining U_f is not scalable for the general case of the Deutsch-Jozsa algorithm where the function f is not limited to 1 bit, i.e. $f : \{0,1\}^n \to \{0,1\}$. For n bits, there are $2^{\{2^n\}}$ possible functions $f : \{0,1\}^n \to \{0,1\}$. The desired oracle U_f must map $|x\rangle |y\rangle$ to $|x\rangle |y \oplus f(x)\rangle$. If we applied our current method to the general case, this would require matching all $2^{\{2^n\}}$ cases of f, to build the corresponding U_f.

Determining the phase "kickback" effect of a unitary U_f for a given function f could provide an easier implementation. Considering the case where $U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$, we could just loop through the 2^n possible input states $|x\rangle$, calculate $f(x)$ for each $|x\rangle$, and then multiply the amplitude of $|x\rangle$ by $(-1)^{f(x)}$. Liquid unfortunately does not provide any convenient method for setting arbitrary amplitudes in a multi-qubit state vector. Amplitudes can only be set for individual qubits. However, this cannot work for cases where U_f produces an entangled state, since then the result cannot be expressed as a product state of single qubit vectors.

Following is the code for Deutsch's Algorithm.

```
[<LQD>]
let deutsch() =
logOpen "deutsch_test.log" false

let qt = QubitTimer()

//Create a 2 qubit vector
let ket = Ket(2)
qt.Show "Created 2 qubits"

//Get list of qubits in the vector
let qs = ket.Qubits

//Initialize second qubit to 1
(ket.Item 1).StateSet(0.,0.,1.,0.)

//The >< operator applies the hadamard gate to all qubits in the list qs
H >< qs
qt.Show("Step 1 - Hadamard both qubits", qs.Length)
for q in qs do show "q[%d]=%s" q.Id (q.ToString())

//Apply the oracle for the function f
//Pass f0, f1, f2, or f3 to test different functions
Uf qs f3

qt.Show("Step 2 - Apply Uf", qs.Length)
for q in qs do show "q[%d]=%s" q.Id (q.ShowMag())

qt.Show("Step 3 - Apply hadamard to first qubit")
H qs
for q in qs do show "q[%d]=%s" q.Id (q.ToString())

qt.Show("Step 4 - Measure first qubit")
M qs

for q in qs do show "q[%d]=%s" q.Id (q.ToString())
```

First a Ket object is created. This represents a 2-qubit state vector. Next, the second qubit is set to the state $|1\rangle$, since this is necessary for Deutch's algorithm.

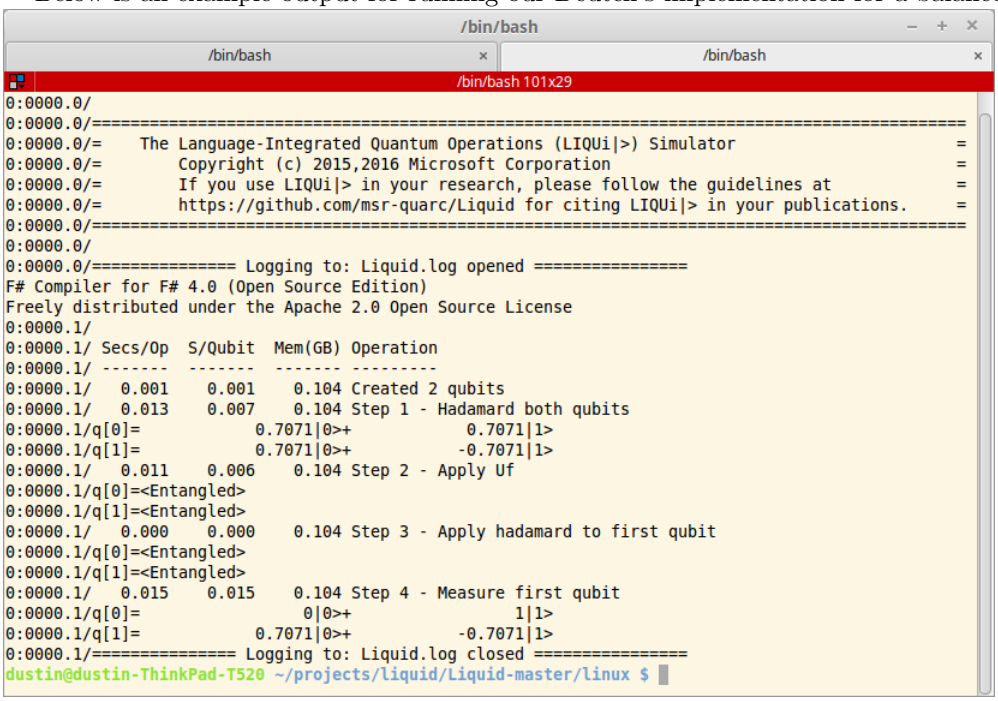
The first gate applied is the hadamard gate, to both qubits. In Liquid, gates are applied as functions which take qubit lists as arguments. The hadamard gate is built-in as the function H.

Gate functions can be used in two ways. If qs is a qubit list, calling Hqs will apply a hadamard gate to the first qubit. If we want to apply a gate to the entire list, we can use the $><$ operator: $H >< qs$. This will call Hq for every qubit q in the list qs, and accomplishes the first step of Deutsch's algorithm.

Next, we apply the U_f gate to both qubits. We also pass one of the four functions $f : \{0, 1\} \to \{0, 1\}$, so that the proper gate can be constructed.

Lastly, a final hadamard gate is applied to the first qubit, and then it is measured. Measurement in Liquid is achieved by applying the built in gate M. Of course, this collapses the measured to qubit(s) to a classical state.

Below is an example output for running our Deutch's implementation for a balanced function f.



Liquid supports printing and logging information as scripts are run. In our Deutsch's example we output the state vector for each qubit after each gate is applied. The state vectors are output in bra-ket notation, unless the state is entangled, in which case the string "entangled" is displayed.

Looking at the first qubit, at the end of the script, the state $|0\rangle$ has amplitude 0, and the state $|1\rangle$ has amplitude 1, as expected for a balanced function.

3 Quantum Programming Language Theory

Preface and Overview of classical lambda calculi:

In the classical setting, various lambda calculi exist with different type systems.

These can be divided into the untyped lambda calculus, and a large number of important typed lambda calculi.

Untyped lambda calculus:

The untyped lambda calculus is Turing complete: we can provide encodings of objects we care about in the form of terms in the untyped lambda calculus, and the computational power of the untyped lambda calculus allows us to map inputs to outputs in terms of these encodings in arbitrary computable ways.

The problem with using untyped lambda calculi for computation lies in the great number of terms for which we cannot attach computational meaning, as well as the difficulty in reasoning about the behavior of programs, such as determining if they output a value of the form we are interested in.

Typed lambda calculi:

Typed lambda calculi, the basis for typed programming languages, address this by providing a mechanism for rejecting certain programs which have undesirable properties, the most common of which being terminating computation in a term which does not correspond to a valid output of the program.

Type systems in general can do even more, and can verify more or less arbitrary properties of programs.

This is useful for the purpose of verifying the correctness of code, and for generation of code from a specification. This is closely tied to applications in universal artificial intelligence and other important fields.

We are, however, limited by Rice's Theorem, so type systems cannot decidably differentiate arbitrary properties with certainty.

The type systems in the classical case therefore typically tackle this problem from two directions.

Either they start with a total language, which has only terminating programs, and then add functionality to increase the amount of computable functions in the language, or they start with a Turing complete language, and then add limitations on programs to reject more and more invalid programs.

In extreme cases, languages push the limits on Rice's theorem by allowing the programmer to assist the compiler in type checking programs by providing proofs that they are of the correct type.

Quantum lambda calculi:

We are interested in how this pictures relates to the quantum case. Before attempting to study quantum lambda calculi, however, we need to have a model of the untyped quantum lambda calculus.

An Untyped Quantum lambda Calculus:

This is the main theoretical work for this report. We focused on the paper "A Lambda Calculus for Quantum Computation" for this, It is possible that there are other ways of creating an untyped quantum lambda calculus. We found this presentation to be rather elegant. When creating the quantum untyped lambda calculus from the previous section, we must satisfy many properties, including:

Our quantum untyped lambda calculus:

1) should be Turing complete.

2) should resemble as closely as possible the classical untyped lambda calculus.

3) should have a universal set of quantum primitives in order to perform optimal quantum computation

4) should maintain invertibility of quantum computation.

5) should not terminate in a superposition that isn't properly normalized

There are a few more technical properties as well that we might care about.

Since we start with the classical untyped lambda calculus, property 1 is satisfied, and while 2 is subjective it appears to be reasonably satisfied as well. One complaint is that our calculus works by recording histories, which may be too wasteful of space. To achieve (3), we add any universal set of quantum primitives. For instance, the paper we examine suggests Rotation by $\pi/8$, H, and CNot.

To achieve (4), we record histories of our computation. That is, every small-step in our computation consists in transforming lambda expression "h1; h2; ... hn; x" to "h1; h2; ... hn; x ; y" where y is the term we would obtain under standard reduction of x. In other words, in addition to normalizing terms as we typically do in lambda calculus, we generate product states that repeatedly append lambda terms to represent the entire history of computation.

While we will later modify this definition slightly, for now, our plan is for $|H0\rangle$ to become $1/\sqrt{2}(|H0;0\rangle + |H0;1\rangle)$

Note that we now have invertibility, but adding histories causes problems with cancellation of terms with different histories.

In particular, computing $|H(H0)\rangle$ does not result in $|0\rangle$ in the computational register. The issue when computing this value, is that we cannot cancel a term with H 1 in its history with one that has H 0 in the same position in its history: $|H(H0)\rangle$ normalizes to $\frac{1}{2}|H(H0)\rangle \bigotimes (|H0;0\rangle + |H0;1\rangle + |H1;0\rangle - |H1;1\rangle)$

To fix this problem, a rather deep result is that the super position doesn't affect the structure of the computation: If we replace all 0 and 1 kets in the history with underscores, everything factors nicely, and $|H(H0)\rangle$ evaluates to 0. The amazing thing about this is that everything is still invertible!

Linearity:

Other than the potential inefficiency associated with copying histories, there is only one problem remaining with the above presentation. Any function that discards its argument loses information, and a function whose argument appears twice or more in its body can perform cloning.

We need a mechanism for specifying which terms are linear, meaning they must be used exactly once, and which are non-linear, or classical, meaning they can be copied or discarded at will. We allow for terms to be linear or non-linear, and we allow for the argument to lambda abstractions to be linear or non-linear. What remains is a lot of technical details to show that linear terms aren't promoted to non-linear terms when they should not be, etc.

We can think of linear terms as representing arbitrary quantum computation, while non-linear terms represent classical computation. One interesting question is whether this can be formulated in a non-conservative manner. That is, a calculus that allows linear terms to be promoted to non-linear terms exactly when they are classical.

Quantum Computing in Typed Languages:

As we have seen above, quantum programming languages can be achieved by taking a classical programming language, and adding a small set of quantum primitives, and then enforcing restrictions that avoid the violations of the no-cloning theorem, and potentially other restrictions that we may wish to enforce.

Further, this appears to be the universal approach taken in developing type theory with quantum computation. As an example, while Liquid provides very limited documentation with regards to its type system, it appears to apply the above methodology starting with the Hindley Milner type system.

The Curry Howard Isomorphism and Future Work:

The Curry Howard Isomorphism is a relationship between programs and proofs. Namely, types are propositions, and programs are proofs.

This correspondence applies to systems like the Hindley Milner type system, but only in trivial ways. More advanced classical programming languages which support dependent types allow the programmer to exploit the relationship between programs and proofs much more directly. While not the final word on type systems, they allow us to write formally verified programs.

The next question I would like to answer, is whether it is possible to use the above strategy for dependent types: That is, can we take dependently typed systems, and very slightly change the type system to support qubits, and then produce formally verified quantum algorithms with essentially known tools and techniques from the classical setting?