Lambda Calculus for Quantum Computing

Introduction

This is a report on my attempt at understanding the paper "A Lambda Calculus for Quantum Computation" [1] by André' van Tonder as part of the final project for the course CSCI-B629 Quantum programing at Indiana University.

Classical Lambda Calculus

The lambda calculus is a minimal programing language which is turning complete. It has three components namely "variable", "function" and "application". We can build numbers and Boolean logic just using these three components. The following is a program in scheme to verify if a given expression is a valid classical lambda calculus expression.

```
λexpr :== y                   ; Variable
       | (λ (x) λexpr)        ; Function
       | (λexpr λexpr)        ; Application

(define parse
  (λ (e)
    (match e
      [`,y #:when (symbol? y) y]
      [`(λ (,x) ,body) #:when (symbol? x) `( λ (,x) ,(parse body))]
      [`(,rator ,rand) `(,(parse-match rator) ,(parse rand))]
      [else (error "Given expression not a valid lambda calculus expression")])))
```

Quantum Lambda Calculus

To extend the same concept of minimal programming language for Quantum computing the minimal set above is not sufficient as in Quantum computing we must take into consider the following things which are particular to Quantum world.

- Reversibility of the computation
- No cloning theory
- Superposition
- Interference
- …

To design such a system for Quantum computing we must extend the components lambda calculus to the following [1]

```
λexpr :== y                 ; Variable
        | (λ (x)  λexpr)    ; Function
        | (λexpr λexpr)     ; Application
        | c                 ; Constant
        | !t                ; Nonlinear term
        | (λ! (x)  λexpr)   ; Nonlinear function
c :== Constants:
        0 | 1 | H | S | R3 | cnot | X | Y | Z | . . .
```

Paper summary

The classical lambda calculus cannot be directly used in quantum settings as it would violate some of the fundamental properties of quantum mechanics, also it lacks certain features that makes quantum computing more powerful that classical computing.

The author discusses each of the problem in detail and how the quantum lambda calculus described above solves it.

- **Reversibility:** One of the powerful feature of quantum computing is reversibility, to implement reversibility in lambda calculus, we have to keep track of the history of operations so we can apply them in reverse order to get the initial state, but in storing the history if we also save the arguments to the operation we will leave the current state in superposition of history which doesn't make sense and apply the operations in reverse order will then result in non-normalized initial state, the author proposes a solution for this problem i.e. we don't save the state in history and at the end of the paper the author shows that we don't need to keep track of history at all if we are implementing quantum lambda calculus as prescribed by author.
- **No Cloning:** In quantum world the quantum state cannot be copied, the classical lambda calculus doesn't restrict copying the qubit, for example we can do something like (λ (x) (+ x x)) which is valid in classical computing, but this requires a using the variable x twice which in-turn requires copying x which cannot be done with quantum state. To restrict this behavior the author proposes to use of constraints from typed linear calculi.
- **Superposition:** The paper discusses about two types of variables the quantum variables which are handled by quantum system and classical variables which are called non-linear variables, so the superposition is handled by default by the type of variables and quantum system.
- **Evaluation:** At the end of the paper the author gives beta reduction rules for quantum lambda calculus, which can be applied systemically to the lambda calculus expression to reduce it to its final state.

Understanding the constructs of the language

I believe the best way to understand the constructs of a new language is to implement it, So I chose to implement the parser for Quantum lambda calculus, and I chose racket as the programing language. Before we implement a parser for quantum lambda calculus, we need a representation for qubits and we should define representation of language primitives (like H, S, R3, cnot etc)

Representation of basis

We use |0>, |1> basis for the implementation. These basis states are represented as structure in racket which have a filed for amplitude of that state. We construct the state as below

```
> (0> 1)
> (1> 1)
```

Representation of quantum state

To design a representation for quantum state we must consider the fact that a quantum state can be in superposition of multiple state. The quantum state is represented as structure in racket which has a field for quantum state. Below is the representation that I chose.

```
> (ψ (list (vector (0> _)/(1> _) ….) …. (vector (0> _)/(1> _)  ….)))
```

Here the state is represented as a nested structure list of vectors. Below are some example

```
> (ψ (list (vector (0> 1))))                                  ≡  |0>
> (ψ (list (vector (1> 1))))                                  ≡  |1>
> (ψ (list (vector (0> 1/√2)) (vector (1> 1/√2))))            ≡  1/√2(|0> + |1>)
> (ψ (list (vector (0> 1/√2)) (vector (1> -1/√2))))           ≡  1/√2(|0> - |1>)
> (ψ (list (vector (0> 1) (0> 1))))                           ≡  |00>
> (ψ (list (vector (0> 1/√2) (0> 1/√2))
        (vector (1> -1/√2) (1> -1/√2))))                      ≡  1/√2(|00> - |11>)
```

From the above examples we can get an idea of the representation of state, the vectors contain the basis of n-bit qubit, if there are more than one element in the list all those are vectors are in superposition state.

Gates

If we implement one of the universal gates set as primitives for the language it is sufficient, because any program using any other gate can be translated to be implemented in universal gate set.

The gates are also implemented as structs in racket, which doesn't have any data fields but have an implementation of `evalq` which takes in a quantum state and does the operation on it and returns a new quantum state. The following are some usage examples evalq function

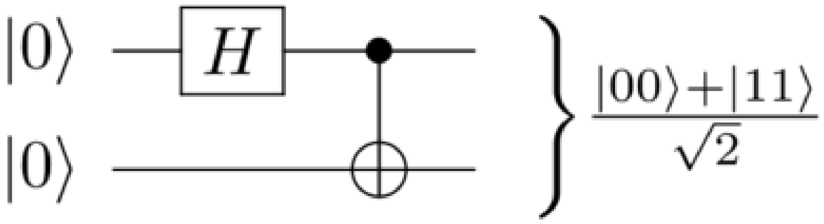
```
; let |0> = (ψ (list (vector (0> 1))))
> (evalq (H) |0>)
(ψ (list (vector (0> 0.7071067811865476))
           (vector (1> 0.7071067811865476))))                    ≡  1/√2(|0> + |1>)

; let |10> = (ψ (list (vector (1> 1) (0> 1))))
> (evalq (cnot) |10>)
(ψ (list (vector (1> 1) (1> 1))))                                ≡ |11>
```

The evalq function can only act on one state at a time, to act on superposition of states we need to use evalsp function below is an example

```
; let |+> =  (ψ (list (vector (0> 0.7071067811865476)) (vector (1> 0.7071067811865476))))
> (evalsp (H) |+>)
(ψ (list (vector (1> 1))))                                       ≡ |0>
```

EPR pair generation

Using above implementations of H and cnot we can generate EPR pair, below are series of steps to generate EPR pair



```
; prepare initial state
> (define |00> (ψ (list (vector (0> 1) (0> 1)))))

; apply Hadamard on first qubit
> (define ψ1 (G@ (H) 0 |00>))
(ψ (list (vector (0> 0.7071067811865476) (0> 1))
         (vector (1> 0.7071067811865476) (0> 1))))               ≡  1/√2(|00> + |10>)

; apply cnot on the resultant state
> (evalsp (cnot) ψ1)
(ψ (list (vector (0> 0.7071067811865476) (0> 1))
         (vector (1> 0.7071067811865476) (1> 1))))               ≡  1/√2(|00> + |11>)
```

Implementing the parser

We use racket's match to do patter match the quantum lambda calculus expression and we check if the expression follows the syntax rules of the language. Below is the implementation of the parser note that because we have implemented all our primitives as structs, we can define a construct like `constant?` which checks if the element is of type primitives.

```
(define syntax-parse
  (λ (e)
    (match e
      [`,c #:when (constant? c) c]
      [`,x #:when (linear? x) x]
      [`,!t #:when (symbol? !t) !t]
      [`(λ (,x) ,body) #:when (symbol? x) `(λ (,x) ,(syntax-parse body))]
      [`(λ! (,x) ,body) #:when (symbol? x) `(λ! (,x) ,(syntax-parse body))]
      [`(,rator ,rand) `(,(syntax-parse rator) ,(syntax-parse rand))]
      [else (error "Given expression not a valid lambda calculus expression")])))
```

The complete implementation of the in racket can be found at https://github.com/v-chanikya/quantum_programing/blob/master/quantum_lambda_calculus.rkt

Bibliography

[1] A. v. Tonder. [Online]. Available: https://arxiv.org/pdf/quant-ph/0307150.pdf.