Qurry: A quantum programming language

Lucas Saldyt, *Arizona State University*

***Abstract*—The core philosophy of Qurry is that simple language features, in aggregate, can make quantum programming significantly easier, by offering lightweight abstractions in the spirit of modern C++. This allows users to implement quantum algorithms cleanly, without glossing over necessary lower-level details of quantum computing, such as the abstract topology of a particular computer. Qurry takes a top-down approach, moving from the abstract goal of creating higher-order functions and datatypes to the level of individual language features. While the desired semantics of a true quantum programming language are not yet completely crystalized, the creation of simple language abstractions will elevate the level at which quantum programs are thought about, potentially enlightening the creation of a true quantum programming language. Many existing quantum programming languages are truly circuit languages, with few features above the level of gates, and sometimes rudimentary functions or macros, and there is currently a tendency not to stray very far from this model. Lastly, Qurry is not just a language, but a software ecosystem which is meant to catalyze the development of quantum programming languages.**

***Index Terms*—Quantum Computing, Programming Languages**

I. Introduction

[TODO: General motivations, cite Rigetti, cite Haskell/LISP/Rich Hickey, cite Probabilistic programming languages]

INNOVATION in near-term quantum programming requires the use of lightweight abstractions. Lightweight abstractions, as defined by Bjarne Stroustrup [TODO: Cite], are abstractions that lower the cognitive load on the user, without sacrificing understanding of the underlying processes behind particular code. Layers of abstraction are a fundamental idea in all of computer science, and quantum computing is no different. Currently, quantum computing operates on the abstraction that is the gate-level, where programs are defined by gates acting sequentially on particular qubits, instead of, for instance, specific microwave pulses (or another implementation-specific low-level mechanism). The universal gate model of quantum computing generally allows a quantum programmer to ignore many details of the quantum computer they are running on: Sources of error aside, modeling the bond energy of molecular hydrogen should be the same on an ion-trap quantum computer as on a superconducting quantum computer. However, some have argued for the importance of hardware, as in Google's phrase "hardware aware, not hardware agnostic". [TODO: Cite] Many aspects of hardware are particularly important, for instance, topology, which will potentially result in a programmer needing to modify a quantum algorithm for it to run on two separate computers.

Lastly, the matrix operator model of quantum computing actually lends itself to functional programming paradigms quite nicely, because quantum programs and quantum operators are functions in a sense. Additionally, since quantum states are fixed once measured, and are generally measured at the end of a program, in a sense memory is not truly mutated (even though it appears to be). [i.e. $result = Pv$, not $result = 0; Pv$]. A quantum program itself is simply a higher order function, which operates on an initial state vector. In turn, a particular quantum program is composed further of simpler matrix-functions, which operate on their own vectors, or are composed with other matrices. For instance, consider a bell state program. As a whole, we may call the bell state program which creates the $+$ state B, and know that $B0$ [TODO: Dirac notation] is the application of the program B to a two-element zero vector. However, this program will further be composed as a Hadamard operator, H, and entanglement operator, $CNOT$, where H will operate on one qubit, and then $CNOT$ will operate on both qubits. [TODO: Math writeup]

In a traditional circuit language, these operators are composed by simply listing which qubits they operate on, and ordering them correctly in a circuit definition file. However, with higher-order functions, quantum operators can be composed in mariad helpful ways, as is common in function languages like Haskell or LISP (from which Qurry draws many influences).

In addition to being a prototype quantum programming language, Qurry defines a software stack surrounding the language, which is intended to make development more pleasant. For instance, this software stack makes it exceptionally easy to add new language features and libraries to Qurry. This allows one to rapidly test new ideas in quantum programming and let the language evolve on its own as opposed to architecting a top-down "perfect" language.

Since quantum computers are simply special probabilistic computers, Qurry also attempts to create a classical statistical library for high-level modeling. This is particularly useful in the same way that a classical probabilistic programming language is, namely for modeling anything statistical, and especially for bayesian machine learning. For instance, the R. Tucci and H. Dekant's group have shown uses for this through their software, Bayesforge [TODO: Cite]. Qurry includes simple statistical packages for creating states, but no inference engine. [However, Qurry might allow one to interface with Bayesforge]

II. Features

Qurry as a language is simply a circuit language with an overlay of higher-order functions. [Explain circuit languages and the functionality Qurry includes here, through pyquil]

The simplest overlay is quite trivial. It is the map function, which allows an arbitrary quantum operator to be applied to several qubits.

At time of writing, Qurry contains the following constructs:

- bernoulli
- block
- cascade
- clear
- cond
- datatype
- defcircuit
- define
- do
- gaussian
- macro
- map
- multinomial
- uniform

[TODO: Elaborate on each]

III. Standard Library

Explain how the statistics library can be easily implemented.

IV. Comparisons

Draw examples from Nielsen and Chuang, and the general literature.

V. Conclusion

In the creation of a Qurry and its corresponding framework, it is hoped that this will aid the development of quantum algorithms, as algorithm designers will have a new, richer, more abstract vocabulary with which to express themselves. To recap, this goal is approached in the following N ways. By introduction of lightweight abstractions from the C++ school of thought, efficient and transparent programming interfaces are created. Through specialized libraries, Qurry can claim to be a generalized library, while still offering powerful sub-frameworks for specific tasks. With functional programming paradigms, Qurry can move towards higher levels of abstraction as the semantics of quantum programming become better understood. Lastly, by creating a rapid prototyping framework, new language features can be developed in a bottom-up style, which will allow Qurry to be created naturally, instead of artificially.

Acknowledgment

The author would like to thank Dr. Will Zeng of Rigetti computing, an organizer of the Unitary Fund, Dr. Ajay Bansal of Arizona State University, PLoS and other donators to the unitary fund, and ASU's FURI program.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

Lucas Saldyt is currently a student researcher at Arizona State University, and has previously worked for Sandia National Laboratories and Los Alamos National Laboratories as a student intern in the quantum computing department.