Qurry: A prototype quantum programming language

Lucas Saldyt

September 18, 2019

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.

1 Introduction

Innovation in near-term quantum programming requires the use of lightweight abstractions, which allow users to easily exploit the power of quantum computing while still understanding its fundamental mechanisms [**?**]. In 1981, Richard Feynman noted that quantum physics appears to be impossible to simulate using a classical computer, but that quantum computers appeared to be perfectly capable of simulating quantum physics [**?**]. Effectively, quantum computation potentially allows new problems to be computed efficiently: in particular, this includes literal simulations of the physical world, but also abstract algorithms which may receive a superpolynomial change in time complexity. Qurry allows quantum programmers to access the power of quantum computers, without sacrificing performance or understanding of the underlying mechanisms.

1.1 Motivation

Stephen Jordan, of Microsoft, keeps a nearly exhaustive list of quantum algorithms and the speedups that they offer [**?**]. According to this list at time of writing, there are thirty-five distinct quantum algorithms which offer a potential superpolynomial speedup. This famously includes Peter Shor's factoring and discrete log algorithms, as well as, fundamentally, quantum simulation. Interestingly, many quantum algorithms such as the Deutsch-Jozsa algorithm are matched (at least practically, and sometimes theoretically) by probabilistic algorithms, and even some algorithms with superpolynomial speedups are based on older probabilistic versions. For instance, machine learning does not appear to have superpolynomial improvements at time of writing. The most promising application of quantum computing in the near term is in molecular simulation. As the comparisons section will demonstrate, Qurry offers unique programming language features which implementing each of these algorithms significantly easier.

1.2 Background

The absolute basics of quantum computing are not nearly as intimidating as they are sometimes made out to be. The main requirement is linear algebra, but complex numbers and probability are also helpful. Conventionally, quantum data is represented on qubits. When measured, qubits will be in *either* of two states: $|0\rangle$ or $|1\rangle$. However, more generally, qubits are in a combination of these two states, which is

known as a superposition. A particular active qubit's state is described by two complex numbers, α and β, which are collected in a vector. This is written in the simple equation:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

. However, the state vector α, β is not directly examinable. Instead, when a qubit is measured, one measures $|0\rangle$ with probability $|\alpha|^2$, and 1 with probability $|\beta|^2$, which must sum to $|1\rangle$. For single qubits, a state is evolved in the model simply by multiplying the state vector for a qubit by a 2 by 2 unitary matrix, known as a one-qubit gate. This is written as Av.

For n qubits, the state is simply a complex vector of length 2^n, and a n qubit gate is a 2^n by 2^n matrix:

$$|\psi\rangle = \alpha_i|s_i\rangle \text{ for } s_i \in \{\{0,1\}^n\}, 0 < i < 2^n - 1$$

. Once a given n-qubit quantum state is measured, the outcome is a single bit-string, s_i, with probability $|\alpha_i|^2$. If it is possible to repeat this measurement (by preparing the quantum state multiple times), then a collection of measurements is a multinomial distribution defined by the states s_i and probabilities α_i, but this distribution has 2^n states, and measuring a quantum state only gives a single sample.

Importantly, past a single qubit, quantum states can be entangled. Two states are entangled when the measurement outcome of one qubit is correlated with the measurement outcome of other qubits. The simplest, most famous example of this is the Bell states, also known as EPR pairs. For instance, in one Bell state, two qubits are either measured both as $|0\rangle$, or both as $|1\rangle$ (even when measured independently), but there is no probability for them to differ. This is the basis for all interesting quantum algorithms.

To summarize, superposition is the fundamental state model for quantum computers, but a given quantum state cannot be measured directly, only sampled once. Then, entanglement allows the correlation of measurements, which is a crucial ingredient in any useful quantum algorithm. Any quantum program is simply a combination of linear operators which affect the superposition. Given an initial state, conventionally the zero vector, a quantum program P operates on the state vector, and then the vector is sampled, by measuring a subset of its qubits. Importantly, a quantum program P is itself a linear operator, potentially further composed of other linear operators. The data in a quantum program, at this level of abstraction, will only ever be a vector of qubits, but at higher levels of abstraction, this vector can be subsectioned into semantic datatypes, no differently than how C++'s fundamental memory model is a sequence of bytes.

At the same time, in hybrid classical-quantum computation, classical control can setup, run, and measure the outcomes of quantum sub-programs. In this way, some quantum program P will be embedded in a hybrid program.

1.3 Circuit Languages

At time of writing, Rigetti pyquil contains the following quantum gates and operations:

Single qubit gates and operations:

- RESET, I, X, Y, Z, H, S, T

Qubit gates taking an angle as the the parameter and qubit as the second:

- RX, RY, RZ, PHASE

Swap operators, where each takes two qubits, and PSWAP takes an additional angle as a first argument:

- SWAP, ISWAP, and PSWAP

Controlled operators:

- CZ, CNOT
- CSWAP
- CPHASE00, CPHASE01, CPHASE10, CPHASE

And of course the hybrid measurement instruction, which takes a qubit as the first argument, and a classical register as the second:

- MEASURE

And also contains the following classical operations:

- TRUE, FALSE, NOT, NEG
- AND, OR, MOVE, EXCHANGE, IOR, XOR
- ADD, SUB, MUL, DIV
- EQ, GT, GE, LE, LT
- LOAD
- STORE
- CONVERT

Most importantly, though, QUIL actually already has a notion of higher-level functions, which it calls "Modifiers". These are the following:

- DAGGER
- CONTROLLED

1.4 Classical Probabilistic Languages

Classical probabilistic programming languages are a recent innovation from the MIT cognitive science community. Essentially, they create a way for non-expert programmers to access the power of Bayesian inference. Users can create simple probabilistic models in standard code, and then run them through an expert-created inference backend. Famously, this has resulted in dramatically reduced code complexity, with a famous case where a 50-line probabilistic program could compete with traditional approaches to face recognition [TODO: Cite].

Qurry is inspired by the effects these languages have had, and in fact there is some overlap between quantum programming and classical probabilistic languages. For instance, quantum bayesian inference has been conceptualized since the 70s-90s [Cite]. There are 10-billion cool libraries that use quantum bayesian inference [Cite].

1.5 Parallels to C++ and its role in the classical software ecosystem

Lightweight abstractions, as defined by Bjarne Stroustrup, are abstractions that lower the cognitive load on the user, without sacrificing understanding of the underlying processes behind particular code [**?**].

In Bjarne's words: "The aim [of C++] is to allow a programmer to work at the highest feasible level of abstraction by providing a simple and direct mapping to hardware and zero-overhead abstraction mechanisms"

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, and users conventionally do not need to be aware about every hardware detail of their quantum computer. 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, a quantum program on an ion-trap computer should behave in the same way as the same program on a computer which uses superconducting qubits, in much the same way that a conventional C++ program does not need to worry about the CPU architecture of the computer it is running on, because (generally), these details are handled by the compiler. It is precisely abstractions like these which make programming possible in the first place.

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. Additionally, the error generation of a quantum computer is actually a crucial detail, even though quantum programmers might desire to ignore it. In the current state of quantum computing, many details of hardware cannot yet be ignored. However, it is obvious that *eventual* hardware independence is desirable — Consider the power of Java in the classical computing world. Lightweight abstractions are precisely the scaffold that will catalyze this transition.

2 Features

Creating abstractions in quantum programming languages comes down to the creation of higher-order functions and higher-order datatypes. Language features that allow the creation and composition of both higher-order functions and higher-order datatypes set Qurry apart from lower-level circuit languages.

2.1 Higher Order Functions

The simplest illustration of a higher order function is trivial, but surprisingly neglected from any existing quantum language: it is the tensor operator ($U^{\otimes n}$), known to functional programmers as the higher-order function, *map*. Many quantum algorithms will begin with a change of basis, effectively a state preparation. Commonly, this is to the Hadamard basis, and is done by applying the Hadamard operator to a block of qubits. In a conventional circuit language, this is done as the following:

```
H 0
H 1
...
H n
```

However, textbooks and papers will write this as $H^{\otimes n}$, and in Qurry, it can be written:

```
(define workspace (block n qubit))
(map H workspace)
```

Implicitly, this example also introduces Qurry's assignment statement *define*, and its representation for qubit arrays, the *block* command, which takes a size and a type, and allocates space and handles mapping to actual qubit indices. More impressively, Qurry supports automatic currying of functions:

```
(define initialize_basis (map H))
(define workspace (block n qubit))
(initialize_basis workspace)
```

Of course, Qurry utilizes the two existing higher-order functions defined by QUIL: DAGGER, and more importantly, CONTROLLED. For brevity, Qurry redefines CONTROLLED to CU. Effectively, this allows the creation of arbitrary controlled operators, for instance the redefinition of CNOT:

```
(define custom_cnot (CU X))
(H 0)
(custom_cnot 0 1)
```

Qurry calls functions like CU controlled higher level operators. This list includes:

- CU
- CNU
- Cascade, CascadeU
- ReverseCascade, ReverseCascadeU
- Collect, CollectU
- Expand, ExpandU
- SimU

Interestingly, many of these are defined through composition of simpler gates. For instance, CNU takes a block of control qubits and a block of work qubits, and entangles pairs of control and work qubits, and finally entangles a target qubit, which a unitary is controlled by: [TODO: Circuit Diagram] A "Cascade" is simply a chain of shifted CNOT gates, and a "CascadeU" is simply a "Cascade" which in turn controls the operation of a unitary gate. [Circuit Diagram] ReverseCascade: [Circuit Diagram] Now consider the circuit used for the simulation of a hamiltonian: [Circuit Diagram] Clearly, this circuit contains repeated information, which can be abstracted into the form of another function: "SimU", which in turn is a "collect" operation, an "expand" operation, and a controlled unitary in between them. [Circuit Diagram]

2.2 Higher Order Datatypes

Qurry's memory model is simple: an array of n qubits, and m classical bits. However, in programmer-space, these arrays are cut up and defined using semantic datatypes.

The previous section discussed the *block* type, and the *define* command. The *block* command will automatically select appropriate qubits in the array, and map to these when used.

In terms of datatypes, not much more is needed, except for the *datatype* command, which mimics C++'s *struct* or *class*. For simplicity, elegance, and robustness, Qurry does not implement encapsulation or inheritance, but instead uses public access by default (in the spirit of Python, since after all, Qurry has a Python interface), and relies on composition instead of inheritance. A *datatype* is nothing more than a contiguous collection of other Qurry datatypes, with names for each field. Like *block* objects, *datatype*s automatically map to qubits and bits in Qurry's memory model. These can be *block*s, single qubits, single bits, and other defined *datatype* objects, allowing for recursive types. Fields within a datatype are simply accessed with the dot operator:

```
(datatype entanglion
          (a qubit)
          (b qubit))
(define e (entanglion))
(H e.a)
(CNOT e.a e.b)
```

Recursively composed higher-order datatypes, in combination with recursively composed higher-order functions are the foundation for creating a more abstract programming language.

2.3 Other Features

- clear
- cond
- do
- macro

3 Comparisons

This section is currently commented out.

4 Software Ecosystem

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.

5 Standard Library

Qurry contains mechanisms which enable easy inclusion of qurry code in the form of libraries. As an example, Qurry's standard library is implemented in this fashion.

Explain how the statistics library can be easily implemented.

At time of writing, Qurry contains the following constructs:

- gaussian
- bernoulli
- multinomial

- uniform

Similarly to in a classical probabilistic programming language, these enable the creation of classical probabilistic states, which can then be used in quantum programs. For instance, it is possible to create a multi-dimensional gaussian distribution, and then entangle an auxillary qubit with the state of the gaussian distribution.

6 Extension

Making additions to Qurry is particularly easy: For instance, the *map* feature is defined using the following python code:

```
from ..compiler.utils import named_uuid

def map(operator, blockname, kernel=None):
    '''
    Apply a single-qubit operator to every qubit in a block
    (map H blocka)
    '''
    try:
        block = kernel.definitions[blockname]
    except KeyError:
        raise ValueError('The block {} is not defined'.format(blockname))
    return '\n'.join('{} {}'.format(operator, i)
            for i in range(block.start, block.end + 1))
```

7 Statistical Libraries

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]

8 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.

9 Appendix one

Appendix content

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] Bjarne Stroustrup. Foundations of c++.

[2] Richard Feynman. Simulating physics with computers, May 1981.

[3] Stephen Jordan. Quantum algorithm zoo.