Quantum Programming Languages

Kesha Hietala
University of Maryland
December 9, 2016

Abstract

Quantum algorithms are often presented as a mixture of mathematical equations, unitary matrices, circuit diagrams, and narrative text, which describe what the algorithm does and argue for its correctness. This approach does not scale well to complex algorithms and does not lend itself to formal verification or generation of instructions for an actual machine. For practical quantum computation, we need a formalism that allows us to describe quantum computations in a precise and intuitive way, and in a way that can be translated easily into "code" that can be analyzed and run on a (hypothetical) system for quantum computation. In other words, we need programming languages for quantum computers. The ideal time to explore different quantum programming language designs is now, before a quantum computer has been realized, so that when a quantum computer is actually built we will be able to take full advantage of its resources. This paper discusses some of the work that has begun this exploration.

1 Introduction

From the earliest days of computing, programming languages have bridged the semantic gap between humans and machines. Programming languages allow programmers to specify algorithms in a high-level form suited to human understanding and facilitate translation of this specification into a low-level form understandable by a machine. Structuring complex interactions with computers would not be possible without programming languages.

The situation for quantum computing is the same. In order to make effective use of an (eventual) quantum computer, we need high level constructs that allow intuitive specification of complex quantum algorithms. Although quantum computing has not yet made the leap from theory to practice, we can make informed guesses about the interface by which we may interact with a quantum computer and use this as a starting point for developing quantum programming languages. In fact, now is the best time to begin exploring design options for quantum programming languages because that way, when a quantum computer is built, we will have the mechanisms available to be able to take full advantage of it.

In this paper we discuss some of the work that has begun this search for an ideal quantum programming language. The hope is that this paper serves as a basic introduction to the field as well as a useful set of references for the interested reader. Section 2 discusses some early work on quantum programming languages, including the QRAM model of computation and quantum pseudocode. Section 3 briefly discusses how existing programming language theory, in particular linear types, applies to quantum programming languages. Section 4 provides an example of a current quantum programming language, and Section 5 concludes.

2 First Steps Toward a Quantum Programming Language

There were early attempts made to formalize the presentation of particular quantum algorithms (e.g. [19]), but the first general approach was proposed by Knill in 1996 [13]. Knill introduced a quantum analog of the classical random access machine (RAM) model of computation and a general pseudocode for quantum programming. Knill's model of computation was a break from earlier work, which focused on formal definitions of quantum Turing machines, and provided insight into how programmers might actually interact with a quantum computer — namely, through a classical machine. His pseudocode has also been influential in the design of many imperative[1] quantum programming languages [21].

2.1 The Quantum RAM Model

There are several popular models for quantum computation including the quantum Turing machine [7], the quantum circuit model [8], and the quantum random access machine (QRAM) [13]. Each of these models are equivalent in terms of their computational power and each has its own merits. The quantum Turing machine is useful for proving complexity bounds of quantum algorithms, and the quantum circuit model provides a straight-forward way to present quantum algorithms (provided that the quantum circuits don't get too large!). However, it is tedious to write programs for the Turing machine and circuit models, and neither model incorporates classical control, which is an important part of many quantum algorithms [15]. The QRAM model has a central role when designing programming languages because it most naturally models how programmers can interact at a high-level with a (hypothetical) quantum machine. For this reason, in this paper we limit our focus to the QRAM model of computation.

In the classical RAM model, computation occurs on a machine with an unlimited number of memory registers, which can be accessed both directly and indirectly (i.e. using an offset from a fixed location) via indices. A RAM program is a finite sequence of instructions that interact with the memory registers. A variant the RAM model where the program is stored in the registers along with data (the "random-access stored-program machine") is one of the closest abstract models to the common notion of a computer. The QRAM model is just an extension of the classical RAM model that, in addition to being able to perform any classical computation, also has access to quantum registers and is able to perform quantum operations (e.g. unitary transformations and measurements) on those registers.

The QRAM model is built on the assumption that realistic quantum computing will take place on a classical computer that has access to a quantum computer. The idea is that the classical computer will provide the quantum computer with (classical) descriptions about starting states and operations to perform and the quantum machine will return (classical) measurement results (see Figure 1). Of course, this is a simplified description to facilitate discussion of high-level programming languages that abstract from hardware details. See e.g. [9, 16] for a description of the implementation of a realistic QRAM-style system.

2.2 Quantum Pseudocode

A good convention for writing pseudocode is essential for describing and formally analyzing algorithms and data structures. Good pseudocode should be easy to write, easy to understand, and result in programs that can be reasonably implemented on a physical machine. But even if the pseudocode is not precise enough to implement on a machine directly, it is an important step beyond ad hoc narrative descriptions of algorithms [10].

In Knill's design, quantum registers are distinguished from classical registers using underlines. Quantum registers can be introduced by applying a unitary operator to a classical register or by calling a subroutine that returns a quantum state. For example, "$\underline{a} \leftarrow a$" converts the classical register a to a quantum register $\underline{a}$ and "$\underline{b} \leftarrow UNIFORMSUPERPOSITION(d)$" creates a

[1]Most programming languages can be categorized as either *imperative* or *functional*. The distinction is that imperative languages treat computation as the execution of statements that update a system's state, while functional languages treat computation as the evaluation of mathematical functions. Examples of imperative languages include C, Java, and Python. Examples of functional languages include Haskell, Lisp, and Clojure.

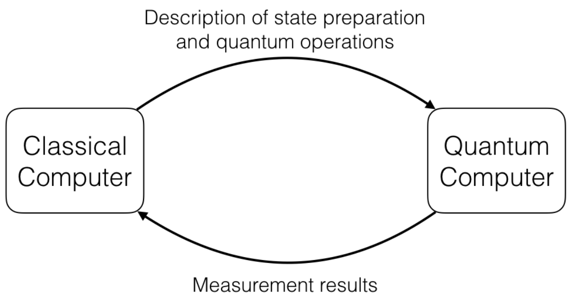


Figure 1: The classical computer sends a description of the states to prepare and the unitary operations and measurements to perform on those states. The quantum computer returns the result of measurements. This loop may have many iterations — each time the quantum computer returns a measurement result, the classical computer can use this result to determine what instructions to send to the quantum computer next.

quantum register $\underline{b}$ as the d-qubit uniform superposition. Quantum registers can be converted back into classical registers via measurements. The application of unitary operators can be controlled by classical or quantum registers using the familiar "**if** X **then** Y **else** Z" notation (or "$\underline{\textbf{if}}$ X $\underline{\textbf{then}}$ Y $\underline{\textbf{else}}$ Z" in the case of quantum control, assuming control in the standard basis). Standard for-loops and while-loops can also be used to control program flow.

As an example, consider the following pseudocode for an implementation of the quantum Fourier transform (QFT) mod 2^d.

FOURIER($\underline{a}$,d)
Input: A quantum register $\underline{a}$ with d qubits. The most significant qubit has index $d-1$.
Output: The amplitudes of $\underline{a}$ are Fourier transformed over $\mathbb{Z}_{2^d}$. The most significant bit in the output has index 0, that is the ordering is reversed.

```
for $i = d-1$ down to $i = 0$
        $H(\underline{a_i})$
        for $j = i-1$ down to $j = 0$
                $\underline{\textbf{if}}$ $\underline{a_j}$ $\underline{\textbf{then}}$ $R_{1-j+i}(\underline{a_i})$
```

Note that above we define H and R_k as $H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$, $R_k = \begin{bmatrix}1 & 0\\ 0 & e^{2\pi i/2^k}\end{bmatrix}$. In the case of $d = 3$, we can unroll the for-loops in the pseudocode above as follows:

```
        $H(\underline{a_2})$
        $\underline{\textbf{if}}$ $\underline{a_1}$ $\underline{\textbf{then}}$ $R_2(\underline{a_2})$
        $\underline{\textbf{if}}$ $\underline{a_0}$ $\underline{\textbf{then}}$ $R_3(\underline{a_2})$
        $H(\underline{a_1})$
        $\underline{\textbf{if}}$ $\underline{a_0}$ $\underline{\textbf{then}}$ $R_2(\underline{a_1})$
        $H(\underline{a_0})$
```

We can see that this code matches the behavior of the QFT circuit shown in Figure 2.

Knill also presents several rules for when assignments in the pseudocode are valid (an assignment is made using the syntax $X \leftarrow ...$ to define a quantum or classical register X). These rules represent

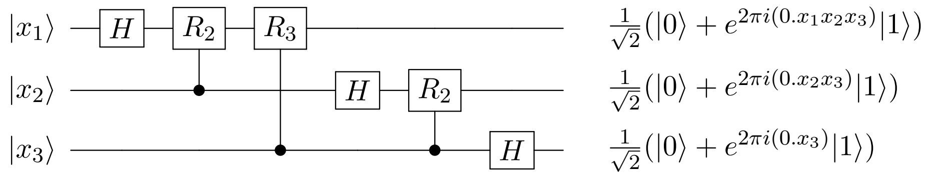

$|x_1\rangle$
$|x_2\rangle$
$|x_3\rangle$
H
R_2
R_3
H
R_2
H
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(0.x_1x_2x_3)}|1\rangle)$
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(0.x_2x_3)}|1\rangle)$
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(0.x_3)}|1\rangle)$

Figure 2: Circuit for a 3-qubit quantum Fourier transform. The top wire corresponds to $\underline{a_2}$, the middle wire corresponds to $\underline{a_1}$, and the bottom wire corresponds to $\underline{a_0}$ in the pseudocode presented.

the main break of quantum pseudocode from classical pseudocode, and reflect physcial properties of quantum systems (e.g. the quantum "no cloning" rule).

- No register can appear in its quantum form on both sides of an assignment.
- A register appearing only on the left must either be classical (in which case the original contents are lost), or not previously declared.
- A register appearing only on the right can experience side-effects during the operation. In other words, registers are assumed to be passed by reference.
- A register appearing in its quantum form on the right and classical form on the left is measured during the operation.

These rules mean, for example, that the line "$\underline{b} \leftarrow ADD(\underline{a}, c)$", which adds the contents of the classical register c to $\underline{a}$ and places the result into $\underline{b}$, is only valid if the quantum register $\underline{b}$ has not yet been defined (i.e. we cannot "reassign" $\underline{b}$), and we need to take into account that during the call to ADD, $\underline{a}$ may have been modified. These rules also mean that in the line "$a \leftarrow DOSOMETHING(\underline{a}, \underline{b}, c)$", a must have been measured, so we cannot access the quantum register $\underline{a}$ at any point after the call to $DOSOMETHING$. Most importantly, these rules suggest that quantum programming languages, like quantum pseudocode, will need to follow different rules from classical programming languages. In the next section we discuss what form these rules might take, and how they can be enforced.

3 Quantum Programming Language Theory

As we saw at the end of the last section, because quantum computers are fundamentally different from classical computers, quantum programming languages (and quantum pseudocode) have certain requirements that classical programming languages typically do not. For example, because quantum programs can be viewed as specifications of quantum circuits, they must obey the rules of quantum circuits [17]:

- Quantum circuits must be n-to-n.
- Only n-to-n gates are allowed.
- Inputs cannot be forked.
- No "dead ends" are allowed.[2]

These rules come from the fact that there is no unitary operator or measurement capable of copying or erasing a qubit. These rules mean that, for example, it does not make sense to have a circuit with 3 inputs and 2 outputs, or to apply a 2-qubit gate to two copies of the same input. If we pretend that circuits are functions, then these rules say that a function with 3 inputs and 2 outputs is not possible and that the call to the 2-input function f with two copies of the qubit k, $f(k, k)$, is not allowed. In most classical programming languages neither of these would be an issue, so clearly quantum programming languages need to have restrictions that classical languages do not. It turns out that the restrictions imposed by the rules above can be described by *linear logic* and enforced using *linear type systems*, and this is the primary focus of this section.

But before moving onto our discussion of linear logic, we mention that a proper formal presentation of quantum programming language theory would require *category theory*, which is a field of mathematics that strives to formulate all of mathematics in terms of abstract objects and functions between those objects. Category theory allows for precise mathematical presentation of quantum algorithms and formal reasoning about those algorithms, replacing ad hoc calculations involving bras and kets, normalizing constants, and unitary matrices with conceptual definitions and proofs [2]. Category theory has had a central role in the development of functional quantum programming languages such as QPL [22] and QML [3]. Linear logic is also closely tied to category theory as linear logic can be seen as the internal logic of any "closed symmetric monoidal category" [6].

3.1 Linear Logic

Linear logic can be thought of as *resource-aware* logic. In traditional intuitionistic logic, from $A \to B$ (read "A implies B") and A, we can deduce A and B. This is written as

$$A \to B, A \vdash A \times B. \tag{1}$$

In linear logic, from the corresponding $\langle A \multimap B\rangle$ (read "consuming A yields B") and $\langle A\rangle$, we can deduce A or we can deduce B, but not both. That is,

$$\langle A \multimap B\rangle, \langle A\rangle \nvdash A \otimes B \tag{2}$$

where $A \otimes B$ is read as "both A and B". We write angle brackets around the assumptions to mark them as linear, meaning that they must be used *exactly once*. This is different from traditional logic, where assumptions can be reused and discarded freely.

To see the merit of a resource-aware logic, consider the example presented in [24]. Let A be the proposition "I have ten dollars", B be the proposition "I have a pizza", and C be the proposition "I have a cake". Now the axioms $\langle A\rangle \vdash B$ and $\langle A\rangle \vdash C$ express that for ten dollars I can buy a pizza, and for ten dollars I can buy a cake. From these axioms we can derive $\langle A\rangle, \langle A\rangle \vdash B \otimes C$, which says that for twenty dollars I can buy both a pizza and a cake. We can also derive $\langle A\rangle \vdash B \& C$, which says that with ten dollars I can buy whichever I choose from a pizza and a cake (note that in this context $B \& C$ does not mean "B and C" but rather that B and C are both available, and that I may choose one of them). However, we *cannot* derive $\langle A\rangle \vdash B \otimes C$, meaning that with ten dollars I can buy both a pizza and a cake. Using traditional logic it would be possible to derive this. So in this case, linear logic better models how we expect the world to work — given ten dollars, I am limited in what I can buy.

In the context of quantum computing, our resource is qubits. Much like ten dollars, once a qubit is "spent" in a unitary operation or measurement, we cannot access the original state of that qubit again. Its state has been changed. Linear logic enforces this principle and also ensures that each qubit is accounted for at the end of the computation (a result of the requirement that each assumption be used *exactly* once). The relationship between quantum computing and linear logic was realized as early as the 1990's [18, 25], and has been influential in the design of many quantum programming languages. Its influence is primarily seen in the form of linear type systems.

[2]At a conceptual level, programmers may choose to ignore the final state of certain qubits (e.g. ancillary qubits), but this is not the same as physically deleting a qubit.

3.2 Linear Type Systems

The "Curry-Howard correspondence" says that propositions of logic correspond to types in functional programs [24]. Essentially, this mean that proofs can be viewed as programs that compute/prove the type of an expression. Consider proof (1) from the previous section. One program that corresponds to this proof is

$$f : A \to B, x : A \vdash (x, f(x)) : A \times B$$

where $e : t$ mean that expression e has type t. This says that if function f has type $A \to B$ (meaning that f takes input of type A and produces output of type B) and x has type A, then the pair $(x, f(x))$ must have type $A \times B$ (meaning that the first element of the pair has type A and the second element of the pair has type B).

One program that corresponds to the non-proof (2) from the previous section is

$$\langle f : A \multimap B\rangle, \langle x : A\rangle \nvdash \langle x, f\langle x\rangle\rangle : A \otimes B.$$ [3]

This says that given the (linear) assumptions that f has type $A \multimap B$ and x has type A, the expression $\langle x, f\langle x\rangle\rangle$ does *not* have type $A \otimes B$. It has a *type error*. The reason for this is that the expression $\langle x, f\langle x\rangle\rangle$ uses x twice. Much like how linear logic requires that each assumption be used exactly once, linear type systems require that each variable be used exactly once.

In modern programming languages, program types are checked for correctness either at compile time or at run time. Type-checking at compile time is generally preferred because it limits run time overhead. However, most compilers for classical languages are not equipped with linear type systems so quantum languages that extend existing classical languages often resort to checking for linearity at run time. One interesting design challenge for type systems for quantum languages is that only the quantum part of the language requires a linear type system — the part of the language that handles only classical computation should not have the same restrictions.

4 Quantum Programming Languages

Up to this point we have introduced some of the approaches taken to formalize the presentation of quantum algorithms, and discussed some theoretical considerations when designing a quantum programming language. In this section we discuss quantum programming languages themselves.

Proposed requirements for a quantum programming language include [5, 15]:

- **Completeness:** The language must allow the programmer to code every valid quantum algorithm and, conversely, every piece of code must correspond to a valid quantum algorithm.
- **Extensibility:** The language must also be able to encode high-level classical computations, which is important because many quantum algorithms require non-trivial classical computation.
- **Separability:** The quantum and classical parts of the language should be separated, allowing execution of classical computation on a classical machine without access to quantum resources.
- **Expressivity:** The language must provide high level constructs that facilitate the intuitive expression of complex quantum algorithms.
- **Independence:** The language must be independent from any particular quantum architecture. It should be possible to compile programs written in the language for different architectures without changing the program's source code.

[3] We use angle bracket and $\multimap$ notation here to stay consistent with [24], but what we essentially want to say is that, under a linear type system, if f has type $A \to B$ and x has type A (as in the first example), then the pair $(x, f(x))$ does have type $A \times B$. Instead, it is an invalid expression.

A variety of quantum programming languages have been proposed that address these requirements in different ways, but one common theme is to develop quantum programming languages as extensions of classical programming languages by introducing specialized commands and data types for the quantum part of the computation. This neatly handles the issues of separability and extensibility. Independence is achieved by incorporating Knill's QRAM model, where the quantum computer acts as black box that takes descriptions of operations as input and returns measurement results as output. One direction of completeness is addressed by run time and compile time checks that verify (e.g. using linear logic) that programs written in the language correspond to valid quantum operations. However, many existing languages are limited in the types of quantum operations that they can describe. For example, several proposed languages do not have support for computation with mixed states. Expressivity is a difficult requirement to address because it is largely qualitative. The hope is that by building quantum programming languages off of popular classical languages and using high-level data structures to describe quantum data, programmers will find themselves in familiar territory and will not be limited in their ability to express complex quantum computations.

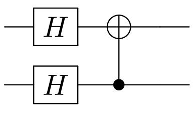
There are many existing quantum programming languages. Some of the most influential include QCL [17], Q Language [5], qGCL [20], QPL [22], QML [3], and the quantum lambda calculus [23]. In this section we focus on just one language, Quipper, to give a flavor of what quantum programming may look like. We choose to discuss Quipper because it is a recent language that shows great promise in its ability to implement quantum algorithms at a real-world scale. However, at this stage of quantum computing, it is difficult to assess the merit of any particular language because we do not know what the final "winning" quantum architecture will be and what languages will be best-suited to that architecture in terms of efficiency and ease of expression. For a more comprehensive review of existing quantum programming languages see [10, 15, 21].

4.1 Quipper

Quipper [11, 12] is a recent functional quantum programming language embedded in Haskell. Quipper is notable for of its scalability – it has been used to implement complex real-world quantum algorithms such as those chosen by IARPA as a representative cross-section of quantum algorithms in the context of its Quantum Computer Science (QCS) program [1]. This is in contrast to many current quantum programming languages, which are still in the early stages of development and can only scale to toy algorithms or proofs of concept.

The basic abstraction provided by Quipper is that quantum operations are functions that take quantum data as input, perform unitary operations and measurements on that data, and then output the updated quantum data/classical results of measurement. As an example, consider the following program from [12] that constructs a small circuit that performs two Hadamard gates and a controlled-not gate (the constructed circuit is on the right).[4]

```
mycirc ::  Qubit -> Qubit -> Circ(Qubit, Qubit)
mycirc a b = do
     a <- hadamard a
     b <- hadamard b
     (a,b) <- controlled_not a b
     return (a,b)
```

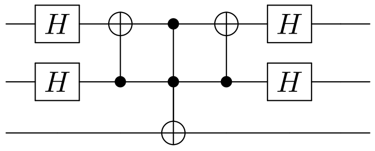


Subroutines can be used build up complex circuits from simpler ones. For example, the following piece of code constructs a circuit that performs `mycirc`, a doubly-controlled not, and then `mycirc`

[4]We have chosen to preserve Quipper example code in its original syntax. We refer readers unfamiliar with Haskell's syntax to Appendix A.

again in reverse.

```
mycirc2 ::  Qubit -> Qubit -> Qubit
   -> Circ(Qubit, Qubit, Qubit)
mycirc2 a b c = do
      mycirc a b
      qnot c ‘controlled’ (a,b)
      reverse_simple mycirc (a,b)
      return (a,b,c)
```



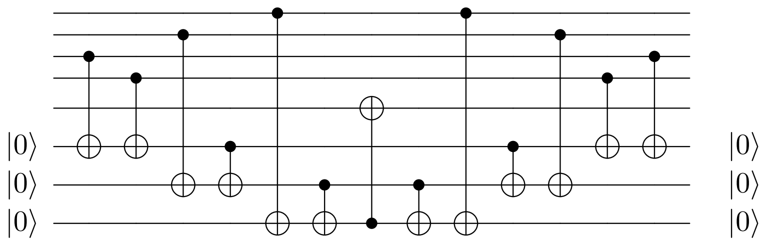
Note that controlled is the infix version of a built-in function that lets a block of gates be controlled by a qubit and reverse_simple is a built-in operation that returns the inverse of a quantum function. For more interesting examples of how to use Quipper see [11], which is a tutorial introduction to the language that includes examples of how to code quantum teleportation, the quantum Fourier transform, and a quantum circuit for addition.

Quipper can be used to describe very large circuits, such as in [12], where Quipper is used to describe circuits with over 30 trillion gates. An important application of Quipper is to estimate the physical resources (e.g. number of gates, number of qubits, number of ancilla) required for quantum algorithms.

Above we specified quantum circuits at the level of quantum gates, but Quipper also provides a way to automatically generate quantum circuits from high-level classical descriptions. This is useful for constructing quantum oracles, which are the classical "black-boxes" found in many quantum algorithms. For example, Shor's factoring algorithm uses a quantum oracle to compute the modular exponentiation $f(x) = a^x (modN)$. [12] discusses the example of a simple oracle that takes as input a list of booleans and returns their parity. The classical program to do this is easy to write:

```
f ::  [Bool] -> Bool
f bs = case bs of
      [] -> False
      [h] -> h
      h:t -> h ‘bool_xor’ f t
```

However, constructing a circuit to do this is less straight-forward. Fortunately, Quipper can take over from here. Quipper's build_circuit and classical_to_reversible operations can convert this classical code into a reversible quantum circuit. The circuit generated for this particular oracle when applied to a list of 4 qubits is below.



The top four qubits are the inputs, the fifth qubit is the output, and the remaining qubits are ancillas that must begin and end in the $|0\rangle$ state. Quipper's process of "circuit lifting" has been used to implement oracles containing millions of gates.

Running Quipper programs is a three-stage process. First, the source code is compiled using a standard Haskell compiler. Next, the compiled code is executed on the classical machine to generate descriptions of quantum circuits. Haskell does not have a linear type system, so Quipper checks that circuits are well-formed at this stage (but note that development of a compile time type-checker for Quipper programs is in progress [12]). In the third stage, these circuit descriptions are sent to the quantum machine. The quantum machine carries out the computations specified, and returns the results of measurements. The compiled Quipper program may then use these measurement results to generate new circuit descriptions.

5 Future Directions

A great deal of work has gone into designing and developing quantum programming languages already, and surely this field will become even more important as quantum computers come closer to actuality. This section contains some thoughts on how quantum programming languages will develop moving forward.

Although not discussed here due to space constraints, one of the most important components of any programming language is not the language itself, but rather its compiler. Because the compiler is the interface between the high-level programs that people write and the low-level instructions that machines read, it is responsible for many complex optimizations required to generate efficient code. Compilers for quantum programming languages will need to generate circuits that are extremely efficient in their use of physical qubits (because we can expect that physical qubits will always be a limited resource), and also minimal in the number of time slices required to compute the circuit. Before one technology emerges as the "best", compilers for quantum programming languages will also need to be extremely portable in the sense that they should able to generate circuits for whatever physical quantum architecture is finally built.

Other advances in quantum programming languages may come from exploration of the overlap between the QRAM model and classical heterogeneous computing. *Heterogeneous computing* refers to the use of multiple types of processors in a single system. This can be useful in the case where the different processors are specialized for different computational tasks. For example, graphics processing units (GPUs) are specialized for massively parallel arithmetic computations and so are often used together with CPUs for calculation-intensive tasks such as physics-based simulations (including simulations of quantum systems [4]), machine learning, and image and video processing. To highlight the similarities between the QRAM model and heterogeneous computing, we explore this example of CPU/GPU systems. The GPU and CPU are physically separate devices with different instruction sets and different memory layouts, and different styles of programming are appropriate for each. To write code for a GPU, programmers must go through the CPU. In the general-purpose GPU programming model, the CPU launches a GPU "kernel", and then the GPU performs the specified operations and returns results once the computation has finished. In the meantime, the CPU can continue its own processing. When it receives results from the GPU, the CPU can perform operations based on those results and may launch additional kernels.

Certainly the difference between GPUs and CPUs is not as great as the difference between a classical computer and a quantum computer, but the intended programming style is similar, and many of the issues that come up when designing languages for general-purpose GPU programming are also issues when designing languages for quantum computers. For example, how does the language separate data types intended for different architectures? How does the compiler apply different optimization strategies to parts of the code intended for different architectures? How can the programmer write code that is well-suited to a particular architecture while still abstracting away low-level details?

A Notes on Haskell Syntax

This appendix contains some basic notes on reading Haskell code as a supplement to Section 4.1, which uses Haskell syntax to present Quipper examples. For a more comprehensive (but very approachable) introduction to Haskell see [14].

1. In Haskell it is common to provide type signatures for the functions you write. As an example consider the two type signatures below:

   ```
   square ::  Int -> Int
   map ::  (a -> b) -> [a] -> [b]
   ```

   The first signature says that the function `square` takes an `Int` as input and produces an `Int` as output. The second says that the function `map` takes takes any function of type `(a -> b)` and a list of `a`'s as input, and produces a list of `b`'s as output. Here `a` and `b` can be any type. For example, you could call `map` with a function of type `Int -> Char` and a list of type `[Int]`. Then this type signature tells us that the output will have type `[Char]`.

   The type signature `mycirc ::  Qubit -> Qubit -> Circ(Qubit, Qubit)` from Section 4.1 says that if the function `mycirc` is provided with two arguments of type `Qubit`, then it will produce a result of type `Circ(Qubit,Qubit)`, which represents a circuit with two wires.

2. Pattern matching is a staple of any functional programming language. Pattern matching is used to take different actions based on the structure of the input argument. For example, the map function, which applies a function `f` to each element in a list `l` can be defined as follows:

   ```
   map f l = case l of
        [] -> []
        [h] -> [f h]
        h:t -> (f h):(map f t)
   ```

   In the case where `l` is an empty list, the `map` function returns an empty list. In the case where `l` is a list with a single element `h`, the `map` function returns a list with a single element `f h` (`f` applied to `h`). Technically we don't need this second case, but we include it for similarity with the oracle example in Section 4.1. In the case of a list with one or more elements, `map` applies `f` to the first element of the list `h` (which stands for "head") and recursively calls the `map` function on the remainder of the list `t` (which stands for "tail"). The results of applying `f` to `h` and of calling `map` with `t` as an argument are connected using the "cons" operator `:`, which adds an element to the front of a list.

3. Haskell's `do` notation technically comes a Haskell construct called a "monad", but for our purposes we can think of it as providing a list of imperative-style instructions. The meaning of the following code:

   ```
   do x1 <- action1
      x2 <- action2
      action3 x1 x2
   ```

   is essentially "run `action1` and bind its result to `x1`, then run `action2` and bind its result to `x2`, then run `action3` with `x1` and `x2` as inputs". Sometimes `do` blocks end with an expression like "`return x`". This has the expected behavior of "returning" `x` as the value of the `do` block.

References

[1] IARPA Quantum Computer Science Program. Broad Agency Announcement IARPA-BAA-10-02, Apr 2010. Available at https://www.fbo.gov/notices/637e87ac1274d030ce2ab69339ccf93c.

[2] Samson Abramsky and Bob Coecke. A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 415–425, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 249–258, 2005.

[4] A. Amariutei and S. Caraiman. Parallel quantum computer simulation on the gpu. In *System Theory, Control, and Computing (ICSTCC), 2011 15th International Conference on*, pages 1–6, Oct 2011.

[5] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics*, 25(2):181–200, 2003.

[6] R. Blute and P. Scott. Category theory for linear logicians. In *Linear Logic in Computer Science*, pages 1–52. Cambridge University Press, 2004.

[7] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 400(1818):97–117, 1985.

[8] D. Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 425(1868):73–90, 1989.

[9] X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels. A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 323–330, New York, NY, USA, 2016. ACM.

[10] Simon J. Gay. Quantum programming languages: Survey and bibliography. *Mathematical. Structures in Comp. Sci.*, 16(4):581–600, August 2006.

[11] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In *Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings*, pages 110–124, 2013.

[12] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, New York, NY, USA, 2013. ACM.

[13] E. Knill. Conventions for quantum pseudocode, 1996.

[14] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[15] Jaroslaw Adam Miszczak. Introduction to models of quantum computation and quantum programming languages. *CoRR*, abs/1012.6035, 2010.

[16] Rajagopal Nagarajan, Nikolaos Papanikolaou, and David Williams. Simulating and compiling code for the sequential quantum random access machine. *Electron. Notes Theor. Comput. Sci.*, 170:101–124, March 2007.

[17] Bernhard Ömer. Quantum Programming in QCL. Master's thesis, Technical University of Vienna, 2000.

[18] Vaughan Pratt. Linear logic for generalized quantum mechanics. In *In Proc. Workshop on Physics and Computation (PhysComp'92*, pages 166–180. IEEE, 1993.

[19] David P. DiVincenzo Richard Cleve. Schumachers quantum data compression as a quantum computation, 1996.

[20] J. W. Sanders and P. Zuliani. Quantum programming. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, MPC '00, pages 80–99, London, UK, UK, 2000. Springer-Verlag.

[21] Peter Selinger. *A Brief Survey of Quantum Programming Languages*, pages 1–6. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[22] Peter Selinger. Towards a quantum programming language. *Mathematical. Structures in Comp. Sci.*, 14(4):527–586, August 2004.

[23] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 06 2006. Copyright - 2006 Cambridge University Press; Last updated - 2015-08-15.

[24] Philip Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, MFCS '93, pages 185–210, London, UK, UK, 1993. Springer-Verlag.

[25] Martin Wehr. Quantum computing: A new paradigm and it's type theory, 1996.