A Brief Survey Of Quantum Programming Languages

Peter Selinger

Department of Mathematics, University of Ottawa
Ottawa, Ontario, Canada K1N 6N5
selinger@mathstat.uottawa.ca

Abstract. This article is a brief and subjective survey of quantum programming language research.

1 Quantum Computation

Quantum computing is a relatively young subject. It has its beginnings in 1982, when Paul Benioff and Richard Feynman independently pointed out that a quantum mechanical system can be used to perform computations [11, p.12]. Feynman's interest in quantum computation was motivated by the fact that it is computationally very expensive to simulate quantum physical systems on classical computers. This is due to the fact that such simulation involves the manipulation is extremely large matrices (whose dimension is exponential in the size of the quantum system being simulated). Feynman conceived of quantum computers as a means of simulating nature much more efficiently.

The evidence to this day is that quantum computers can indeed perform certain tasks more efficiently than classical computers. Perhaps the best-known example is Shor's factoring algorithm, by which a quantum computer can find the prime factors of any integer in probabilistic polynomial time [15]. There is no known classical probabilistic algorithm which can solve this problem in polynomial time. In the ten years since the publication of Shor's result, there has been an enormous surge of research in quantum algorithms and quantum complexity theory.

2 Quantum Programming Languages

Quantum physics involves phenomena, such as superposition and entanglement, whose properties are not always intuitive. These same phenomena give quantum computation its power, and are often at the heart of an interesting quantum algorithm. However, there does not yet seem to be a unifying set of principles by which quantum algorithms are developed; each new algorithm seems to rely on a unique set of "tricks" to achieve its particular goal.

One of the goals of programming language design is to identify and promote useful "high-level" concepts — abstractions or paradigms which allow humans to think about a problem in a conceptual way, rather than focusing on the details of its implementation. With respect to quantum programming, it is not yet clear what a useful set of abstractions would be. But the study of quantum programming languages provides a setting in which one can explore possible language features and test their usefulness and expressivity. Moreover, the definition of prototypical programming languages creates a unifying formal framework in which to view and analyze existing quantum algorithm.

2.1 Virtual Hardware Models

Advances in programming languages are often driven by advances in compiler design, and vice versa. In the case of quantum computation, the situation is complicated by the fact that no practical quantum hardware exists yet, and not much is known about the detailed architecture of any future quantum hardware.

To be able to speak of "implementations", it is therefore necessary to fix some particular, "virtual" hardware model to work with. Here, it is understood that future quantum hardware may differ considerably, but the differences should ideally be transparent to programmers and should be handled automatically by the compiler or operating system. There are several possible virtual hardware models to work with, but fortunately all of them are equivalent, at least in theory. Thus, one may pick the model which fits one's computational intuitions most closely.

Perhaps the most popular virtual hardware model, and one of the easiest to explain, is the *quantum circuit model*. Here, a quantum circuit is made up from quantum gates in much the same way as a classical logic circuit is made up from logic gates. The difference is that quantum gates are always reversible, and they correspond to unitary transformations over a complex vector space. See e.g. [3] for a succinct introduction to quantum circuits. Of the two basic quantum operations, unitary transformations and measurements, the quantum circuit model emphasizes the former, with measurements always carried out as the very last step in a computation.

Another virtual hardware model, and one which is perhaps even better suited for the interpretation of quantum programming languages, is the *QRAM model* of Knill [9]. Unlike the quantum circuit model, the QRAM models allows unitary transformations and measurements to be freely interleaved. In the QRAM model, a quantum device is controlled by a universal classical computer. The quantum device contains a large, but finite number of individually addressable quantum bits, much like a RAM memory chip contains a multitude of classical bits. The classical controller sends a sequence of instructions, which are either of the form "apply unitary transformation U to qubits i and j" or "measure qubit i". The quantum device carries out these instruction, and responds by making the results of the measurements available.

A third virtual hardware model, which is sometimes used in complexity theory, is the *quantum Turing machine*. Here, measurements are never performed, and the entire operation of the machine, which consists of a tape, head, and finite control, is assumed to be unitary. While this model is theoretically equivalent to the previous two models, it is not generally considered to be a very realistic approximation of which a future quantum computer might look like.

2.2 Imperative Quantum Programming Languages

The earliest proposed quantum programming languages followed the *imperative* programming paradigm. This line of languages was started by Knill [9], who gave a set of conventions for writing quantum algorithms in pseudo-code. While Knill's proposal was not very formal, it was very influential in the design of later imperative quantum programming languages. More complete imperative languages were defined by Ömer [10], Sanders and Zuliani [13], and Bettelli et al. [2].

A common feature of these imperative quantum programming languages is that a program is viewed as a sequence of operations which operate by updating some global state. These languages can be directly compiled onto (or interpreted in) the QRAM virtual hardware model. Quantum states in this paradigm are typically realized as *arrays* of qubits, and run-time checks are needed to detect certain error conditions. For instance, out-of-bounds checks are necessary for array accesses, and distinctness checks must be used to ensure $i \neq j$ when applying a binary quantum operation to two qubits i and j. As is typical for imperative programming languages, the type system of these languages is not rich enough to allow all such checks to be performed at compile-time. Also, typically these languages do not have a formal semantics, with the exception of Sanders and Zuliani's language, which possesses an operational semantics.

The various languages in this category each offer a set of advanced programming features. For instance, Ömer's language QCL contains such features as, automatic scratch space management, and a rich language for describing user-defined operators [10]. It also offers some higher-order operations such as computing the inverse of a user-defined operator.

The language of Bettelli et al. emphasizes practicality. It is conceived as an extension of C++, and it treats quantum operators as first-class objects which can be explicitly constructed and manipulated at run-time [2]. One of the most powerful features of this language is the on-the-fly optimization of quantum operators, which is performed at run-time.

Finally, Sanders and Zuliani's language qGCL is of a somewhat different flavor [13]. Based on Dijkstra's guarded command language, qGCL is as much a specification language as a programming language, and it supports a mechanism of stepwise refinement which can be used to systematically derive and verify programs.

2.3 Functional Quantum Programming Languages

In the functional programming style, programs do not operate by updating a global state, but by mapping specific inputs to outputs. The data types associated with purely functional languages (such as lists, recursive types) are more amenable to compile time analysis than their imperative counterparts (such as arrays). Consequently, even in very simple functional programming languages, many run-time checks can be avoided in such languages in favor of compile-time analysis.

The first proposal for a functional quantum programming language was made in [14]. In this paper, a language QFC is introduced, which represents programs via a functional version of flow charts. The language also has an alternative, text-based syntax. Both unitary operations and measurements are directly built into the language, and are handled in a type-safe way. Classical and quantum features are integrated within the same formalism. There are no run-time type checks or errors. The language can be compiled onto the QRAM model, and it also possesses a complete denotational semantics, which can be used to formally reason about programs. The denotational semantics uses complete partial orders of superoperators, and loops and recursion are interpreted as least fixpoints in the way which is familiar from domain-theoretic semantics.

The basic quantum flow chart language of [14] is functional, in the sense of being free of side-effects. However, functions are not themselves treated as data, and thus the language lacks the higher-order features typical of most functional programming languages such as ML or Haskell. It is however possible to extend the language with higher-order features. The main technical difficulty concerns the proper handling of linearity; here, one has to account for the fact that quantum information, unlike classical information, cannot be duplicated due to the so-called “no-cloning property”. Van Tonder, in a pair of papers [16, 17], has described a linear lambda calculus for quantum computation, with a type system based on Girard’s linear logic [7].

Van Tonder’s calculus is “purely” quantum, in the sense that it does not incorporate classical data types, nor a measurement operation. If one further extends the language with classical features and a measurement primitive, then a purely linear type system will no longer be sufficient; instead, one needs a system with linear and non-linear types. Such a language, with intuitionistic linear logic as its type system, will be presented in a forthcoming paper by Benoît Valiron.

We should also mention that there is some interesting work on using functional languages to *simulate* quantum computation. For instance, Sabry [12] shows how to model quantum computation in Haskell.

3 Semantics

The basic flow chart language of [14] has a satisfactory denotational semantics, but it lacks many language features that would be desirable, including higher-order features and side-effects. In trying to add new language features, one may either work syntactically or semantically. Semantic considerations, in particular, may sometimes suggest useful abstractions that are not necessarily apparent from a syntactic point of view. We briefly comment on some semantic projects.

Girard [8] recently defined a notion of quantum coherent spaces as a possible semantics for higher-order quantum computation. The class of quantum coherent spaces has good closure properties, for instance, it forms a *-autonomous category. However, the model is still incomplete, because the interpretation of quantum languages in this category is currently limited to the “perfect”, or purely linear, fragment of linear logic. This means that classical data is subject to the same non-duplication restriction as quantum data in this model.

A different approach to a semantics for higher-order quantum computation is given by Abramsky and Coecke [1]. This work is more qualitative in nature, and relies on entanglement and quantum measurement to model higher-order functions and their applications, respectively.

Also on the semantic side, there have been a number of works on possible connections between quantum theory and domain theory. For instance, Edalat [5] gives a domain-theoretic interpretation of Gleason’s theorem in the presence of partial information. Coecke and Martin [4] give a domain-theoretic treatment of the von Neumann entropy of a quantum state.

3.1 Topological Quantum Computation

An radically different direction in the semantics of quantum computation, and one which might lead to the discovery of new conceptual paradigms for quantum computation, is the work of Freedman, Kitaev, and Wang [6]. This line of work seeks to exploit connections between quantum computation and topological quantum field theories (TQFT’s). In a nutshell, in topological quantum computation, a quantum state is represented by a physical system which is resistant to small perturbations. Thus, quantum operations are determined only by global topological properties, e.g., linking properties of the paths traversed by some particles. This leads to a potentially very robust model of quantum computation. It also suggests that there is a more discrete, combinatorial way of viewing quantum computation, which might in turns suggest new quantum algorithms. These topological approaches to quantum computation are currently limited to a description of unitary operators; measurements are not currently considered within this model.

4 Challenges

There are many remaining challenges in the design and analysis of quantum programming languages. One such challenge is to give a sound denotational semantics for a higher-order quantum programming language, including classical features and measurement. While there has been recent progress on this issue, both on the syntactic side and on the semantic side, the connection between syntax and semantics remains tenuous at this point, and typically covers only fragments of the language. A related question is how to model infinite data types, particularly types which include an infinite amount of “quantum” data.

Another challenge is to formulate a theory of “quantum concurrency”. This is not far-fetched, as one can easily imagine networks of quantum processes which communicate by exchanging classical and quantum data. There is a considerable body of work in quantum information theory and quantum cryptography, which suggests some potential applications for quantum concurrent systems.

Another interesting research area is the implementation of quantum programming languages on imperfect hardware. Unlike the idealized "virtual machine" models of quantum computation, one may assume that real future implementations of quantum computation will be subject to the effects of random errors and decoherence. There are known error correction techniques for quantum information, but it is an interesting question to what extent such techniques can be automated, for instance, by integrating them in the compiler or operating system, or to what extent specific algorithms might require customized error correction techniques.

References

1. S. Abramsky and B. Coecke. Physical traces: Quantum vs. classical information processing. In R. Blute and P. Selinger, editors, *Proceedings of Category Theory and Computer Science, CTCS'02*, ENTCS 69. Elsevier, 2003.
2. S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. arXiv:cs.PL/0103009 v2, Nov. 2001.
3. R. Cleve. An introduction to quantum complexity theory. In C. Macchiavello, G. Palma, and A. Zeilinger, editors, *Collected Papers on Quantum Computation and Quantum Information Theory*, pages 103–127. World Scientific, 2000.
4. B. Coecke and K. Martin. A partial order on classical and quantum states. Technical report, Oxford University Computing Laboratory, 2002. PRG-RR-02-07.
5. A. Edalat. An extension of Gleason's theorem for quantum computation. http://www.doc.ic.ac.uk/∼ae/papers.html, 2003.
6. M. H. Freedman, A. Kitaev, and Z. Wong. Simulation of topological field theories by quantum computers. arXiv:quant-ph/0001071/ v3, Mar. 2000.
7. J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50:1–102, 1987.
8. J.-Y. Girard. Between logic and quantic: a tract. Manuscript, Oct. 2003.
9. E. H. Knill. Conventions for quantum pseudocode. LANL report LAUR-96-2724, 1996.
10. B. Ömer. A procedural formalism for quantum computing. Master's thesis, Department of Theoretical Physics, Technical University of Vienna, July 1998. http://tph.tuwien.ac.at/∼oemer/qcl.html.
11. J. Preskill. Quantum information and computation. Lecture Notes for Physics 229, California Institute of Technology, 1998.
12. A. Sabry. Modeling quantum computing in Haskell. In *ACM SIGPLAN Haskell Workshop*, 2003.
13. J. W. Sanders and P. Zuliani. Quantum programming. In *Mathematics of Program Construction*, Springer LNCS 1837, pages 80–99, 2000.
14. P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*. To appear.
15. P. Shor. Algorithms for quantum computation: discrete log and factoring. In *Proceedings of the 35th IEEE FOCS*, pages 124–134, 1994.
16. A. van Tonder. A lambda calculus for quantum computation. arXiv:quant-ph/0307150/ v4, Dec. 2003.
17. A. van Tonder. Quantum computation, categorical semantics and linear logic. arXiv:quant-ph/0312174/ v1, Dec. 2003.