Research Statement:
Languages, Verification and Compilation for the Quantum Era

Robert Rand
University of Maryland

Executive Summary: The Age of Quantum Programming

Quantum computers are here. Google's recent demonstration of *quantum supremacy* [1] experimentally validated the central claim of quantum complexity theory: Quantum computers can provide exponential speedup over classical computers on certain tasks.

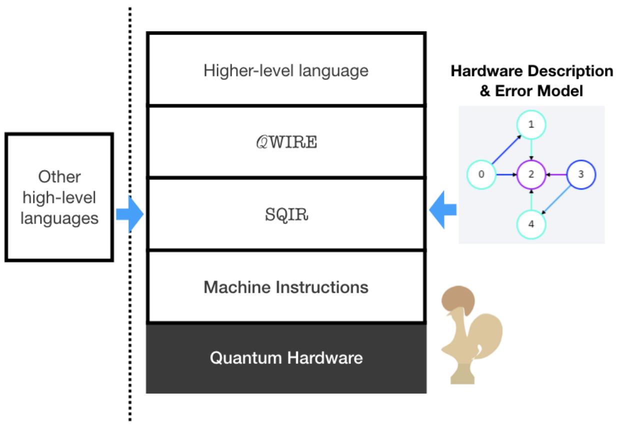

Figure 1: The verified quantum stack.

Now we face the daunting task of proving that we can do something *practical* with them. That's an engineering challenge but it's also a software development challenge. Quantum computing is conceptually hard: It amounts to applying a restricted set of matrices to complex-valued vectors and probabilistically querying that vector. Even once we've developed a quantum algorithm, it's hard to make sure we've gotten it right. Not only are quantum programs probabilistic and error-prone, we cannot observe their intermediate states without causing a quantum collapse. As a result, standard software assurance techniques will fail in a quantum setting. And even once we have a correct program, we need to get it working on the very limited quantum hardware we can expect over the coming years. All this demands new and novel quantum programming languages, verification tools, and optimizing compilers to run our programs on constrained devices. If we want useful quantum programs, we need to give programmers the tools to **design**, **analyze**, and **execute** them.

In the pursuit of programming languages support for quantum programming I:

- Co-developed QWIRE, a core language for quantum circuits with a type system to **guide development** and ensure that quantum programs don't go wrong [POPL17].
- Embedded QWIRE in the Coq proof assistant, used it to formally verify the **correctness** and other **safety properties** of quantum programs [QPL17; QPL18], and later added **error-tracking and correction** [PLanQCa].
- Laid out a vision for a **verified quantum compilation stack** that ensures that high-level quantum programs are reliably compiled to quantum hardware [CCC19; SNAPL19].
- Worked on the verified compiler VOQC for **optimizing** and **mapping** quantum circuits onto restricted hardware [QPL19; VOQC] and advised a Ph.D. student on verified translation between IBM's popular OpenQASM target language and our SQIR intermediate representation [PLanQCb].

Finally, I've worked to build a community around quantum programming and verification, which will be necessary for an effort of this scale. This has been my focus for several years, in which I spoke and led discussions at a Dagstuhl seminar on quantum programming languages [Dagstuhl], and helped write a Computing Community Consortium report on practical quantum programming [CCC19]. This January, I'll be giving a series of lectures on quantum programming languages at WiSQCE, the Winter School on Quantum Computing at Emory. After that I'll head to New Orleans for *Programming Languages for Quantum Computing* (PLanQC 2020), a workshop I founded to introduce programming languages researchers to quantum computing and facilitate interaction between the two communities. Following PLanQC, I'll present a tutorial at Principles of Programming Languages on *Verified Quantum Computing*, using my online textbook of the same name [VQC]. This text, which I first taught in my formal verification course at Maryland, is designed to introduce formal verification researchers to quantum computing through the use of the Coq proof assistant. I've designed these workshops and classes to draw programming languages and systems researchers into quantum computing, which can sorely use their skills.

Quantum computing is here and I'm committed to developing the tools and bringing together the community to make it work.

Quantum Programming Languages

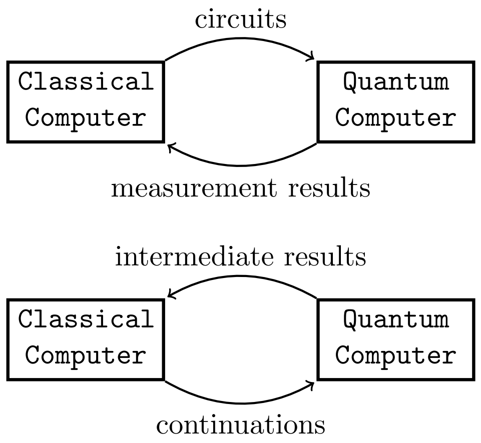

Figure 2: Two models of classical-quantum communication.

Before we start designing a quantum programming language, we have to think about the computational model for quantum computing. Popular quantum programming tools, like IBM's Qiskit [2] and Rigetti's PyQUIL [3], assume that quantum programs are simply circuits, consisting of a sequence of gate operations to some set of registers. Hence, these tools are essentially Python libraries for describing circuits in a basic instruction set language, and then running or simulating these circuits. However, quantum computers will rarely, if ever, be used in isolation. In designing the $\mathcal{Q}$WIRE programming language [POPL17; Thesis], we adopted the QRAM model [4], which treats conventional (or *classical*) and quantum computers as co-processors. Generally, the classical processor will describe a circuit for the quantum processor to execute and report back results. However, the quantum computer can also initiate communication by sending the classical computer partial results and asking it to compute a continuation circuit. $\mathcal{Q}$WIRE is embedded in classical host language to maximize the expressivity of its classical fragment while providing a *quantum core* for describing circuits and requesting continuations.

$\mathcal{Q}$WIRE's type system reflects the distinction between the quantum and classical co-processors. The quantum circuit language is *linearly typed*, meaning that every qubit must be used exactly once. This ensures that every circuit respects the *no-cloning* principle of quantum mechanics. The classical language, by contrast, is *dependently typed*. This allows it to describe *families* of circuits parameterized by some natural number. Such families appear everywhere in quantum programs, so it's important to be able to give them precise types that allow for safe composition.

I augmented $\mathcal{Q}$WIRE with support for *ancillae*, temporary qubits used for a computation and then returned to their original configuration and discarded [QPL18]. These ancillae play a core role in the classical subroutines of popular quantum algorithms, like Shor's algorithm. Unlike other languages, QWIRE allows us to prove that our ancillae are used correctly, either manually or by adhering to a common template. This rules out an important class of human error in writing quantum programs. I also provided a compilation function from a small Boolean language to QWIRE and used it to implement a quantum adder. This provides a means for implementing a range of classical routines in a quantum device.

In ongoing work [PLanQCa], Kesha Hietala and I implemented a type system for tracking errors throughout a quantum program, which is bound to be useful for any programmer intending to run a program on a near term machine. We also added an *error correction* gate to QWIRE that would correct an error in a *logical qubit* composed of multiple physical qubits. Use of logical qubits in quantum programs is a longer-term goal, but necessary to implement a variety of important algorithms.

I'm also exploring another use-case for more powerful types [WIP]. At the moment, the richest type system for quantum programs has the type `Qubit` for quantum bits and `Bit` for qubits that display no quantum effects. But quantum resource theories [6; 5] provide a rich language for characterizing *coherence* and *entanglement* among qubits, much of which we can bake into our type systems. This gives the programmer a better sense of the type of data she is working with and helps prevent programming mistakes.

Quantum programming languages have a long way to go. Even a circuit description language augmented with circuit families, ancilla management, compilation of classical functions, and rich types is too low-level for most programmers. To address this, I've been analyzing quantum algorithms for common patterns to distill into future programming language abstractions.

I should also note that the prevalent quantum circuit model is in no way fundamental: Quantum computers run on laser pulses or microwave resonators, not circuits. So I've been investigating alternative models from the ZX-calculus [7] to quantum control flow (multiple programs running in superposition) [8] to measurement-based quantum computing [9]. Only time will tell which models catch on in the quantum computing community, but each contains enough of interest to be worth formalizing and building upon over the coming years. I expect these to play a central role in my research agenda over the medium- to long-term.

Verifying Quantum Programs

There's a standard story we tell our students: Programming can be easy but testing is still important. In quantum computing, programming is never easy and testing may well be impossible.

Quantum programs are fundamentally about linear algebra: A *quantum states* is a vector of 2^n complex numbers, where n is the number of qubits being acted upon. We operate on such a state via a restricted class of matrices called unitaries. Probability enters the equation when we *measure* our quantum state, that is, when we attempt to turn it into classical data. Herein lies the difficulty: If we can only observe classical data, how can we inspect our program when it goes wrong? Generally speaking, we can't, not without perturbing the program. On top of that, unit testing is of limited use when programs have probabilistic outputs, and doubly so when they're so expensive to run (as we can expect to be the case for a while). In the extreme case, as in Google's recent experiment, you sample from a distribution that no computer, quantum or classical, can represent analytically and you cannot determine whether or how it went wrong.

Where unit tests and debugging flounder, formal verification thrives. Here quantum programs have an advantage: a standard denotational semantics in terms of *density matrices*, which represent probability distributions over quantum states. In the Coq implementation of QWIRE, I built up this semantics from scratch, starting with a matrix library with lightweight dependent types [CoqPL18] for ease of programming and verification. I used 𝒬WIRE to prove the correctness of a variety of simple quantum algorithms, like Deutsch's algorithm and a variety of random number generating circuits. Since then my collaborators and I have added more complicated examples, like the generalized Deutsch-Jozsa algorithm and circuits for producing W States and GHZ states. (We later simplified these proofs for the SQIR language [QPL19].) I've also proven 𝒬WIRE itself correct at multiple levels [Thesis]: Its unitary matrices are shown to satisfy the conditions of unitarity, which in turn is proven to preserve quantum states. The denotations of its well-typed circuits are shown to be valid functions between density matrices and its input matrices are shown to be *pure* (deterministic) or *mixed* (probabilistic) quantum states, as appropriate.

I've taken the lessons of verification in 𝒬WIRE and SQIR and distilled them in my online textbook, **Verified Quantum Computing** [VQC], modeled after the popular Software Foundations series. VQC introduces students to quantum computing and verification in the Coq proof assistant, using simplified versions of the libraries underlying 𝒬WIRE and SQIR. It also introduces several of the automation techniques used in 𝒬WIRE and SQIR and helps get students and researchers into verifying quantum programs. Towards this purpose, I've already taught from VQC at Maryland and will be teaching an updated version at POPL's TutorialFest this January.

Tools for verifying quantum programs should be not only accessible but easy to use. This is a core motivation for *program logics*, which provide precise and intuitive deductive systems for proving properties about programs. In the area of quantum computing, we can use program logics to prove properties of communications protocols [WIP], quantum cryptography [10] and error rates [11], as well as more general properties of quantum programs [12; 13; 14]. Building such systems inside a proof assistant not only makes them more useful but helps in their development, as I discovered when working on probabilistic [MFPS15] and nondeterministic [PPS16] Hoare logics. My collaborators and I are currently looking into building logical systems around SQIR, while embedding features like recursion inside the circuit language itself.

Compiling Quantum Circuits for Tomorrow's Machines

Current quantum computers have an abundance of limitations: Small qubit counts, limited connectivity between qubits, high (and highly variable) error rates, and quick decoherence. All of these factors mean that we will need to compile our programs to very limited machines and anticipate some level of error. To handle this, I proposed a verified quantum stack that would allow us to program and prove properties of programs at a high level and then compile these programs to a format designed to be run on a quantum computer [CCC19, §8; SNAPL19]. We would also be able to prove properties of the lower-level representation, particularly about the error rates. Finally, this entire process would be verified, guaranteeing that compilation itself doesn't introduce errors (as it so often does, especially in the quantum setting).

The first step in this process was designing a small quantum intermediate representation, which I called SQIR [QPL19]. I designed SQIR not only as a practical low-level language but for ease of verification. With my collaborators at Maryland, I quickly reproduced most of the proofs from the 𝒬WIRE library and then some. Together we built VOQC [VOQC], a verified circuit compiler that outperforms IBM's Qiskit [2] and is competitive with leading compilers. VOQC can also take in a graph representing a quantum computer's architecture and produce an equivalent SQIR program that respects the constraints on that graph: However, we have not yet attempted to match the performance of the best circuit mappers.

I have yet to realize to core parts of the VOQC vision. VOQC is not a full compilation stack: It doesn't yet do verified translation from 𝒬WIRE to SQIR, instead expecting source programs in SQIR or the popular OpenQASM target language [15]. I intend to verify both the translation from 𝒬WIRE to SQIR and a bidirectional translation between SQIR and OpenQASM [PLanQCb], using the OpenQASM semantics provided by Amy [16]. I also intend to add support for errors. These error rates should be specific to the device on hand and represented via SQIR's own density matrix semantics. From there I will start looking at the extremes of the quantum stack: Can we directly compile to laser pulses, as described by IBM's OpenPulse [17], in a way that allows us to characterize outputs even more precisely? Can we verifiably compile higher-level language abstractions to SQIR to guarantee that they are quantum-mechanically sound? What about libraries for quantum chemistry and other application domains? As the quantum computing stack expands, the uses for verification will multiply and VOQC will continue to grow.

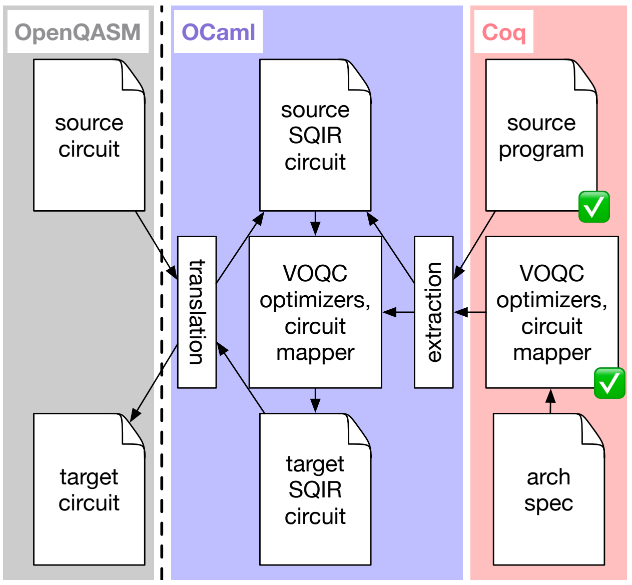

Figure 3: The VOQC architecture. The source program can be written in SQIR using Coq or OCaml, or in another high level language and compiled to OpenQASM. The leftmost slice of the diagram is unverified.

Conclusion

Quantum programming is hard and will remain hard for the foreseeable future. My goal is to make it less hard. I'm taking a three-pronged approach: designing programming languages that make it easier to write and understand quantum programs; crafting verification tools to guarantee that our programs do what we want them to do while minimizing machine errors; and building a software stack to compile our quantum programs to limited and error-prone quantum devices. As quantum computing rapidly moves from theory to practice, the programming languages and systems communities need to move with it. I'm off running and I hope you'll all run with me.

Publications

[MFPS15]	**Robert Rand** and Steve Zdancewic. “VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs”. In: *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics*. 2015.
[PPS16]	**Robert Rand** and Steve Zdancewic. “Models for Probabilistic Programs with an Adversary”. In: *Workshop on Probabilistic Programming Semantics*. 2016.
[POPL17]	Jennifer Paykin, **Robert Rand**, and Steve Zdancewic. “QWIRE: A Core Language for Quantum Circuits”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017.
[QPL17]	**Robert Rand**, Jennifer Paykin, and Steve Zdancewic. “QWIRE Practice: Formal Verification of Quantum Circuits in Coq”. In: *Proceedings 14th International Conference on Quantum Physics and Logic*. 2017.
[QPL18]	**Robert Rand**, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. “ReQWIRE: Reasoning about Reversible Quantum Circuits”. In: *Proceedings of the 15th International Conference on Quantum Physics and Logic*. 2018.
[CoqPL18]	**Robert Rand**, Jennifer Paykin, and Steve Zdancewic. “Phantom Types for Quantum Programs”. In: *Coq for Programming Languages*. 2018.
[SNAPL19]	**Robert Rand**, Kesha Hietala, and Michael Hicks. “Formal Verification vs. Quantum Uncertainty”. In: *Summit on Advances in Programming Languages*. 2019.
[QPL19]	Kesha Hietala, **Robert Rand**, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. “Verified Optimization in a Quantum Intermediate Representation”. In: *arXiv preprint arXiv:1904.06319* (2019). Presented at the 16th International Conference on Quantum Physics and Logic.
[CCC19]	Margaret Martonosi et al. “Next Steps in Quantum Computing: Computer Science’s Role”. In: *arXiv preprint arXiv:1903.10541* (2019). Computing Community Consortium Catalyst Report.
[VQC]	**Robert Rand**. *Verified Quantum Computing*. Online at `http://www.cs.umd.edu/~rrand/vqc`. 2019.
[Dagstuhl]	**Robert Rand**. “Verified Quantum Programming in QWIRE: Optimization and Error Correction”. In: *Quantum Programming Languages (Dagstuhl Seminar 18381)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. Chap. 3.12.
[PLanQCa]	Kesha Hietala, **Robert Rand**, and Michael Hicks. “Tracking Errors through Types in Quantum Programs”. In: *Programming Languages for Quantum Computing*. To appear. 2020.
[PLanQCb]	Kartik Singhal, **Robert Rand**, and Michael Hicks. “Verified translation between low-level quantum languages”. In: *Programming Languages for Quantum Computing*. To appear. 2020.
[WIP]	**Robert Rand**, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. *A Type System for Quantum Resources*. Work In Progress. 2020.
[VOQC]	Kesha Hietala, **Robert Rand**, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. *A Verified Optimizer for Quantum Circuits*. In stage of submission.
[Thesis]	**Robert Rand**. “Formally Verified Quantum Programming”. PhD thesis. University of Pennsylvania, 2018.

References

[1] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510.

[2] Gadi Aleksandrowicz et al. *Qiskit: An Open-source Framework for Quantum Computing.* 2019. DOI: 10.5281/zenodo.2562110.

[3] Rigetti Computing. *Pyquil Documentation.* URL: http://pyquil.readthedocs.io/en/latest/.

[4] Emmanuel H. Knill. *Conventions for Quantum Pseudocode.* Tech. rep. LAUR-96-2724. Los Alamos National Laboratory, 1996.

[5] Ryszard Horodecki, Paweł Horodecki, Michał Horodecki, and Karol Horodecki. "Quantum entanglement". In: *Rev. Mod. Phys.* 81 (2 2009), pp. 865–942. DOI: 10.1103/RevModPhys.81.865.

[6] T. Baumgratz, M. Cramer, and M. B. Plenio. "Quantifying Coherence". In: *Phys. Rev. Lett.* 113 (14 2014), p. 140401. DOI: 10.1103/PhysRevLett.113.140401.

[7] Bob Coecke and Ross Duncan. "Interacting quantum observables". In: *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP).* Lecture Notes in Computer Science. 2008.

[8] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. "From symmetric pattern-matching to quantum control". In: *International Conference on Foundations of Software Science and Computation Structures.* Springer. 2018, pp. 348–364.

[9] Hans J Briegel, David E Browne, Wolfgang Dür, Robert Raussendorf, and Maarten Van den Nest. "Measurement-based quantum computation". In: *Nature Physics* 5.1 (2009), p. 19.

[10] Dominique Unruh. "Quantum relational hoare logic". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 33.

[11] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. "Quantitative robustness analysis of quantum programs". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 31.

[12] Mingsheng Ying. "Floyd–hoare logic for quantum programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.6 (2011), p. 19.

[13] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. "Invariants of quantum programs: characterisations and generation". In: *ACM SIGPLAN Notices.* Vol. 52. 1. ACM. 2017, pp. 818–832.

[14] Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. "A theorem prover for quantum Hoare logic and its applications". In: *arXiv e-prints* (2016). arXiv: 1601.03835.

[15] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. "Open quantum assembly language". In: *arXiv e-prints* (2017). arXiv: 1707.03429.

[16] Matthew Amy. "Sized Types for Low-Level Quantum Metaprogramming". In: *Reversible Computation.* Ed. by Michael Kirkedal Thomsen and Mathias Soeken. Cham: Springer International Publishing, 2019, pp. 87–107. ISBN: 978-3-030-21500-2.

[17] David C McKay et al. “Qiskit backend specifications for OpenQASM and OpenPulse experiments”. In: *arXiv e-prints* (2018). arXiv: `1809.03452`.