

Formal Verification vs. Quantum Uncertainty

Robert Rand 

University of Maryland, College Park, USA

<http://www.cs.umd.edu/~rrand/>


rrand@cs.umd.edu

Kesha Hietala 

University of Maryland, College Park, USA

<https://www.cs.umd.edu/people/khietala>

kesha@cs.umd.edu

Michael Hicks 

University of Maryland, College Park, USA

<http://www.cs.umd.edu/~mwh/>

mwh@cs.umd.edu

Abstract

Programming a quantum computer is difficult and writing a program that will execute successfully on quantum devices that exist today (or are likely to exist in the near future) is a daunting task. Not only is quantum computing inherently uncertain, the quantum computers that we have introduced a variety of novel errors that are difficult to predict or work around. Techniques from formal verification will allow us to quantify and mitigate these errors if we can bridge the gap between high level languages and machine specifications. In this paper, we review existing approaches to quantum program verification and propose a new approach focused not only on long term quantum programming, but on the quantum programs we can run today.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Hardware → Quantum error correction and fault tolerance

Keywords and phrases Formal Verification, Quantum Computing, Programming Languages, Quantum Error Correction, NISQ

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding All authors are funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040

Robert Rand: Also partly funded by a Victor Basili Postdoctoral Fellowship

Acknowledgements We would like to acknowledge our co-authors on work reviewed here, including Jennifer Paykin, Dong-Ho Lee, Steve Zdancewic, Shih-Han Hung, Shaopeng Zhu, Mingsheng Ying, and Xiaodi Wu.

1 Introduction

Writing quantum programs is hard. Fundamentally, a quantum program corresponds to applying a limited set of operations to vectors of complex numbers, with the goal of producing a vector with the majority of its weight in a few meaningful indices. For instance, if you are trying to factor 77, you might want the 7th or 11th entry to contain a number close to 1 while the other indices contain numbers close to 0. As a result, designing a quantum program requires a fair amount of effort and mathematical sophistication. This difficulty is compounded by the fact that quantum programs are very difficult to test or debug.

Consider two standard techniques for debugging programs: breakpoints and print statements. In a quantum program, printing the value of a quantum bit entails measuring it and



© Robert Rand, Kesha Hietala and Michael Hicks;
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:9



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 printing the returned value. This is akin to randomly and irreversibly coercing a floating
 46 point number to a nearby integer—it will give a weakly informative answer and corrupt
 47 the rest of the program. Examining a value will have the exact same effect. Unit tests
 48 are similarly of limited value when your program is probabilistic; repeatedly running unit
 49 tests on a quantum computer is likely to be prohibitively expensive. Simulating quantum
 50 programs on a classical computer holds some promise (and simulators are bundled with most
 51 quantum software packages) but it requires resources exponential in the number of qubits
 52 being simulated, so simulation can't help us in the general case.

53 Where standard software assurance techniques fail us, formal verification thrives. When
 54 we reason about a quantum program, we are reasoning exclusively about vectors, and
 55 vectors don't collapse when we analyze them. Formal verification is parametric in its
 56 inputs: Instead of reasoning about a 128-qubit implementation of an algorithm, we can
 57 prove properties of that algorithm for arbitrary values of n . Using techniques like induction,
 58 algebraic reasoning and equational rewriting we can verify the correctness of a broad range
 59 of quantum programs, as we previously showed [17] using the *QWIRE* programming language
 60 and verification tool.

61 Unfortunately, the challenges of measurement and simulation complexity are only the tip
 62 of the iceberg when it comes to near-term quantum computing.

63 For the foreseeable future, useful quantum computing will face a broad range of obstacles.
 64 The two major competing architectures for quantum computers are the superconducting
 65 qubit model used by IBM, Google, and Rigetti, and the trapped-ion model of IonQ and
 66 a number of academic labs. To varying extents (and the variance matters), each of these
 67 models suffers from the following issues:

- 68 1. Coherence times: Qubits can only maintain their state for a certain amount of time before
 69 they *decay*, effectively resetting themselves.
- 70 2. Gate errors: Quantum gates introduce errors and these errors vary with the gates being
 71 applied.
- 72 3. Connectivity: In general, you can't apply an operation to two or more qubits unless those
 73 qubits are physically adjacent to one another. We can use quantum gates to swap the
 74 values of qubits, and thereby bring two or more qubits together, but these operations
 75 take time and introduce additional errors.

76 These limitations aren't even uniform within a given machine, let alone across machines.
 77 On IBM's largest publicly available quantum computer, Melbourne, phase coherence times
 78 range from 22.1 to 106.5 microseconds depending on the qubit in question, and gate errors
 79 similarly vary by qubit [11].

80 Software alone cannot solve these problems, but it can help mitigate their impact. Formal
 81 verification allows us to prove that a given quantum program meets its specification: This
 82 allows us to rule out software errors and focus on likely sources of hardware errors. We
 83 can also attempt to model and quantify hardware errors, though these vary substantially
 84 by machine. Once we have verifiable programs and an error model we can use verified
 85 optimizations to minimize the error rate. Finally, we can compile the program to the target
 86 architecture, taking into account the connectivity, coherence times and error rates of the
 87 machine's physical qubits.

88 This paper proposes an approach to formal verification that begins with high level quantum
 89 programs and ends with low-level instructions tailored to the quantum computer meant to
 90 run the program. We begin by reviewing existing approaches to verifying quantum programs
 91 in Coq using the *QWIRE* tool [15, 19] developed by the first author and his collaborators at
 92 the University of Pennsylvania. We then discuss reasoning about errors, following in the

footsteps of the quantum robustness logic developed by the second and third authors and their collaborators at the University of Maryland and the University of Technology Sydney. We propose our vision for extending *QWIRE* or similar tools to reason about every level of the quantum software stack so that formal verification becomes relevant to quantum computing today.

2 Logical Qubits

The approach to quantum computation taken by most textbooks, as well as quantum complexity theorists and (most) quantum algorithms designers assumes the existence of *logical qubits*, quantum bits which are not error-prone and behave according to a strict mathematical model. To be precise, a qubit corresponds to a two element vector of the form $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ for complex numbers α and β , subject to the constraint that $|\alpha|^2 + |\beta|^2 = 1$.

Our logical qubits obey strict rules but they are not deterministic. If we *measure* our qubit above, we will obtain one of the two *basis qubits* $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ with probability $|\alpha|^2$ and $|\beta|^2$, respectively.

We can combine two qubits by taking their tensor product, where $\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}$.

In this case, measuring the system ϕ returns one of four basis vectors with the 1 appearing in the i^{th} position with probability $|\phi_i|^2$.

Besides for measuring (systems of) qubits, we can modify them by applying *unitary gates* which correspond to multiplication on the left by a restricted set of matrices called *unitaries*, which preserve the property that the sum-of-squares adds up to one. It's worth noting that if we apply certain gates (such as the controlled-not gate) to two or more qubits, it may no longer be possible to represent the outcome as the tensor product of multiple distinct 2-dimensional vectors. This state of affairs is called *entanglement* and is analogous to probabilistic dependence.

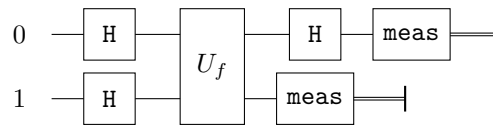
Generally speaking, we will represent quantum programs as *circuits*, like that in Figure 1, though they can equally well be thought of as a simple sequences of gate and measurement applications. This is the simplest model for quantum computation and it doesn't take errors into account. Conveniently, these circuits have a straightforward denotational semantics: They correspond precisely to functions over complex vectors. We can also represent them as functions over *density matrices* which in turn correspond to distributions over complex vectors. This saves us from having to explicitly include probabilities in our denotational semantics. Once we have a denotational semantics for quantum circuits, we can begin to prove things about them.

3 Programming Languages and Verification Under Ideal Conditions

Numerous tools exist for verifying properties of quantum programs, from Ying's quantum Hoare logic [24] to Amy's path integral calculator [1]. Our tool, *QWIRE* [15, 19], is fully general in that it allows us to write high level quantum programs, translate them to functions over density matrices (or vectors, in the measurement-free case), and verify arbitrary properties of those functions within the Coq proof assistant.

One of the simplest things to verify about a quantum program is that it doesn't attempt to duplicate qubits, which would violate the *no cloning* theorem of quantum mechanics. Non-duplication can be enforced by a linear type system, like that employed by the quantum lambda calculus [22], Proto-Quipper [21] or *QWIRE*. A linear type system treats a function type $A \multimap B$ as something that *consumes* an A and produces a B . Hence, it ensures that

23:4 Formal Verification vs. Quantum Uncertainty



■ **Figure 1** A circuit implementing Deutsch's algorithm

once we've done an operation on a qubit (presumably getting a bit or qubit as a result) the original qubit can no longer be used.

It's rather more complicated to ensure that a quantum program uses *ancillae* safely. Ancillae are spare qubits that are used in the computation of some result and then returned to their original state and discarded. They are a common feature of quantum algorithms, and hence appear in quantum programming languages like Quipper [8] and Q# [23]. Unfortunately, it's difficult to guarantee that qubits are indeed returned to their original state – this problem is complete for the complexity class QMA. Inspired by the REVERC [2] compiler for reversible programs, QWIRE allows the programmer to prove that ancillae are discarded correctly, while providing syntactic conditions for the simple cases, such as when a sequence of classical operators is applied and then inverted [18]. In the general case, though, providing ancilla correctness requires us to reason about the behavior of complete quantum programs.

Doing whole-program analysis substantially increases the scope for formal verification. One thing we may want to do is verify the correctness of a complete program, like Deutsch's algorithm [7, 5]. Deutsch's algorithm takes in an unknown function f , represented as a quantum gate, and returns the 0 qubit if and only if the function is constant. Using QWIRE, we can express the correctness of this algorithm as follows:

```

154
155 Lemma deutsch_constant : ∀ f, constant f →
156   [[deutsch (fun_to_gate f)]] I1 = |0⟩⟨0|.
157
158 Lemma deutsch_balanced : ∀ f, balanced f →
159   [[deutsch (fun_to_gate f)]] I1 = |1⟩⟨1|.
160

```

Here $|0\rangle\langle 0|$ represents a 0 qubit in density matrix form, and similarly for $|1\rangle\langle 1|$. I_1 is a 1×1 identity matrix, representing that the circuit has no input qubits (akin to `unit` or `void`). Deutsch's algorithm is relatively easy to prove correct by computation, Rand's thesis [17] demonstrates a variety of correctness proofs that require algebraic reasoning or induction.

One powerful use of verification tools is to show that two small circuits are equivalent and use those for circuit optimizations. Optimizations can substantially reduce the number of gates in a circuits, thereby reducing both the running time and error rate of those programs. For instance, Nam *et al.* [14] show a variety of circuit equivalences and use them to optimize Quipper circuits. Unfortunately, these equivalences aren't formally verified and (as we learned from talking to Nam *et al.*) in practice the optimizations introduced a number of bugs. Using a verified optimizer guarantees that the circuit equivalences are correct and that the rewriting is performed correctly.

4 Verification in the Real Quantum World

So far we have an interesting story and perhaps even a compelling one: Quantum programs are hard to write and debug and hence provide an excellent target for the techniques of formal verification, from program logics to proof assistants. However, in the short term, machine errors are likely to be so commonplace that programs proved correct for an idealized

semantics will produce the wrong answer if run on an actual machine. Rather than assume we can make quantum machines highly reliable, in the near term it probably makes more sense to expect to the programmer to expect failure.

4.1 NISQ: The Era of Errors

In a recent keynote address [16], John Preskill coined the term Noisy Intermediate-Scale Quantum Computing (NISQ) to refer to quantum computing over the coming five or ten years. Preskill, like many in the field, suspects that quantum computing will soon have its first practical applications on computers with on the order of 100 physical qubits. On the other hand, it will take hundreds or thousands of physical qubits to construct one error-corrected logical qubit, and many thousands of logical qubits to beat classical computers at factoring numbers or a range of other tasks. In the term term, quantum programs will have to be aimed at problems for which they are uniquely well-suited (like modeling quantum systems themselves, in physics [4] and chemistry [20]) and tailored to the limitations of the available computers.

What do these limitations look like? One hard limitation is the number of qubits on the machine. However, even if you have enough qubits to run your program in principle, you have to account for errors that creep into your program, both through *decoherence* (qubits tend to revert to their basis states with time) and errors in gate application.¹ These difficulties are compounded by the architectural limitations of current machines: Instead of being complete undirected graphs, they tend to resemble sparse directed graphs. That means that if you wish to apply a controlled not gate from qubit q_1 to q_2 , you need to move their values to some qubits q_3 and q_4 , with an edge from q_3 to q_4 , and then apply the gate. (You may then need to return the values to the original qubits.) Transporting values in this way is both computationally expensive and error prone.

If formal verification is to guarantee the correctness of quantum programs in practice, it will need to account for the variety of errors that impact NISQ devices.

4.2 Verification in the Presence of Errors

One straightforward approach to verification in the presence of errors is to simply aggregate errors along a quantum circuit's wire. Instead of the Hadamard gate (H) having the type $\text{Qubit} \multimap \text{Qubit}$, it can have the type $\forall n, (\text{Qubit}, n) \multimap (\text{Qubit}, n+1)$, meaning that the gate adds a single error to its wires. We could equally well include error probabilities along the wires, though those would assume we knew the error rate for each gate and they were consistent across qubits. Multiple-qubit gates are a bit trickier, as they take in and produce multiple wires: We can either output the sum of all error terms along each output wire plus the additional error introduced by the gate, take the max of the two wires, or (particularly in the case of probabilities) use a more complex function over multiple inputs. A circuit then, would likewise have an error term corresponding to the aggregated output of its wires. For a simple example, depending on whether U_f takes the max or sum of its inputs' errors, Deutsch's algorithm would produce 2 or 3 errors, plus the errors introduced by U_f , assuming measurement doesn't introduce errors itself.

¹ The presence of these gate errors actually allows us to comfortably ignore a more fundamental issue in quantum computing. The so-called "universal gate sets" implemented by general purpose quantum computers are not actually universal in the sense that NAND gates are universal for classical computation: They only allow us to *approximate* arbitrary quantum operations. Unfortunately, in the near term, all gates will only loosely approximate their specified behaviors.

218 An advantage of this approach is that it's very easy to implement and sufficiently general
 219 that we can interpret the output in a number of different ways, depending on the setting.
 220 We can also automate it, allowing a built in type inference algorithm to calculate the errors
 221 that occur in a given circuit.² It is also limited, though, in that errors only increase as the
 222 circuit size grows. This makes it difficult to analyze programs that include mechanisms for
 223 error mitigation, which will be important in near-term applications [12]. (In principle, we
 224 could have error-correcting gates that shrink the error, but error correction tends not to be
 225 so simple.)

226 We can also take ideas from our recent robustness logic [10], which draws on Carbin *et*
 227 *al.*'s Rely [3] and provides bounds for the errors in a quantum program. While powerful,
 228 this logic suffers from the same limitations as our error wire semantics—errors only increase
 229 throughout a program. It is also high level: It uses the same quantum while language as
 230 QHL [24], which doesn't directly describe circuits. By contrast, the ideal near-term logic
 231 would describe quantum circuits under a precise error model and allow us to apply general
 232 reasoning techniques to those circuit.

233 Ideally, the semantics for a quantum program would contain error terms, corresponding
 234 to possible failures. Given that the density matrix semantics already describes a distribution
 235 over states, we wouldn't even need to change the type of our denotation function. This would
 236 bring the advantage of allowing us to reason about and reduce error terms, through majority
 237 voting or the like. Unfortunately, it's still hard to know what the denotational model should
 238 be, without reference to the specific hardware. Hence, while doing high-level reasoning we
 239 want to leave the error term abstract (as in our robustness logic), and instantiate it for
 240 specific hardware models.

241 4.3 A Verified Quantum Stack

242 This brings us to the hardware level. As we noted in the previous section, optimizations play
 243 a big role in reducing circuits to the size where they be executed by a quantum computer.
 244 Similarly, hardware specific compilation can both reduce circuit size for a given quantum
 245 computer and attempt to minimize error. Recent work by Murali *et al.* [13] uses an SMT
 246 solver to reduce the error rate of circuits mapped to IBM's 16 qubit quantum computer
 247 by a factor of three (on average) compared to IBM's default mapper. Unfortunately, this
 248 doesn't do gate level optimization, replacing gate sequences with equivalent, lower error
 249 sequences. Nor does it report an error rate back to the programmer, allowing them to modify
 250 the program if necessary. Ideally, gate level optimization and circuit mapping would be done
 251 in tandem in order to minimize the program's error rate. The compiled program would also
 252 be made available to the programmer so they could modify the compiled as needed.

253 Conveniently, *QWIRE* already contains multiple levels of abstraction. High-level *QWIRE*
 254 programs use Coq variables to refer to qubits or even collections of qubits and make use
 255 of Coq datatypes and recursive calls. However, in order to convert a *QWIRE* circuit to
 256 a function on density matrices (which order their qubits), we must first compile it to an
 257 intermediate representation which explicitly applies gates to numbered qubits. These circuits
 258 (which resemble those of IBM's "quantum assembly language", OpenQASM [6]) can then
 259 be converted to the desired functions. Once we have this representation, it makes sense
 260 to provide a `compile` function that takes an assembly level circuit and a description of the

² It's worth noting that checking linearity, though harder, is an orthogonal problem, so we can infer the error terms and check linearity separately. This is somewhat surprising, since without linearity we would have to be concerned about adding errors to terms without using them up.

desired hardware and maps the circuit onto the given hardware. This function should both map the circuit to the specified gate set and qubit placement while optimizing to reduce the gate count and expected error. We should also verify that this function produces an equivalent circuit to the input at the high level, as described in Section 3.

Once we have a compiled circuit and a hardware error model, it becomes possible to verify the precise behavior of the circuit on the given architecture, taking error into account. This would be a big step forward for reliable quantum computing but it raises the question of its impact on the quantum software development cycle.

5 The Verification-Programming Loop

Throughout this exposition, we've treated verification as a task that is subsequent to programming, which guarantees that the program behaves as expected. However, this doesn't quite reflect our intended use for formal verification in the quantum setting. In our experience, verification plays a major role in quantum programming itself, even ignoring machine errors. Even reproducing well-known quantum algorithms proves to be difficult, given the low level at which they are written. (For examples, see Huang and Martonosi's [9] recent exploration of bugs in the implementations of common quantum algorithms.) In practice, we find ourselves writing quantum programs, attempting to prove their correctness, failing, and revising the original program. Often the failures can be quite informative: If the Coq proof assistant asks us to prove that $\frac{1}{\sqrt{2}} = 1$, it's a sure sign that we've either left out a Hadamard gate or put in one too many (since $\frac{1}{\sqrt{2}}$ appears throughout the Hadamard matrix).

We expect error-aware quantum verification to similarly assist us in writing quantum programs. On occasion it may make sense to simply write a quantum program, send it off to an optimizer, and execute it on a noisy quantum machine. However, if you reach a certain threshold where your program output is indistinguishable from noise, optimizations probably aren't going to make it work. At that point, you need to take a second look at your program, just as if you found that your running time was exponential in your inputs. High error rates are a sign that you're doing something wrong, and the sooner you're made aware of that, the better off you'll be.

We should note that all this is far from ideal: In an ideal world we would write quantum programs in a high level language with appropriate abstractions, and never even think about quantum circuits. Unfortunately, we don't live in such a world and aren't likely to live in it for many years. Instead, our goal is develop tools that will assist in developing efficient algorithms with correctness guarantees and precisely bounded errors. To that end, we have to expose everything down to the individual qubit decoherence to the programmer, so they can handle that decoherence. Providing these tools will allow us to do more with near term quantum devices than we could possibly do today.

References

- 1 Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018*, June 2018.
- 2 Matthew Amy, Martin Roetteler, and Krysta M. Svore. Verified compilation of space-efficient reversible circuits. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer, July 2017. URL: <https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/>.

- 306 **3** Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for
 307 programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN*
 308 *International Conference on Object Oriented Programming Systems Languages & Applications*,
 309 pages 33–52, 2013. doi:10.1145/2509136.2509546.
- 310 **4** Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. Toward the
 311 first quantum simulation with quantum speedup. *Proceedings of the National Academy of*
 312 *Sciences of the United States of America*, 115 38:9456–9461, 2018.
- 313 **5** Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms
 314 revisited. In *Proceedings of the Royal Society of London A: Mathematical, Physical and*
 315 *Engineering Sciences*, volume 454, pages 339–354. The Royal Society, 1998.
- 316 **6** Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum
 317 assembly language. *arXiv preprint arXiv:1707.03429*, 2017.
- 318 **7** David Deutsch. Quantum theory, the church-turing principle and the universal quantum
 319 computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and*
 320 *Engineering Sciences*, volume 400, pages 97–117. The Royal Society, 1985.
- 321 **8** Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron.
 322 Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM*
 323 *SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013,
 324 pages 333–342, 2013.
- 325 **9** Yipeng Huang and Margaret Martonosi. Qdb: from quantum algorithms towards correct
 326 quantum programs. *arXiv preprint arXiv:1811.05447*, 2018.
- 327 **10** Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi
 328 Wu. Quantitative robustness analysis of quantum programs. *Proc. ACM Program. Lang.*,
 329 3(POPL):31:1–31:29, January 2019. URL: <http://doi.acm.org/10.1145/3290344>, doi:10.
 330 1145/3290344.
- 331 **11** IBM. IBM quantum experience, 2017. URL: [https://quantumexperience.ng.bluemix.net/](https://quantumexperience.ng.bluemix.net/qx/devices)
 332 [qx/devices](https://quantumexperience.ng.bluemix.net/qx/devices).
- 333 **12** Sam McArdle, Xiao Yuan, and Simon Benjamin. Error mitigated digital quantum simulation.
 334 *arXiv preprint arXiv:1807.02467*, 2018.
- 335 **13** Prakash Murali, Jonathan M Baker, Ali Javadi Abhari, Frederic T Chong, and Margaret
 336 Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers.
 337 *arXiv preprint arXiv:1901.11054*, 2019.
- 338 **14** Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated
 339 optimization of large quantum circuits with continuous parameters. *npj Quantum Information*,
 340 4(1):23, 2018.
- 341 **15** Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum
 342 circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming*
 343 *Languages*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. doi:10.1145/
 344 3009837.3009894.
- 345 **16** John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.
 346 URL: <https://doi.org/10.22331/q-2018-08-06-79>, doi:10.22331/q-2018-08-06-79.
- 347 **17** Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania,
 348 2018.
- 349 **18** Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning
 350 about reversible quantum circuits. In *Proceedings of the 15th International Conference on*
 351 *Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.
- 352 **19** Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of
 353 quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics*
 354 *and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. URL:
 355 <https://doi.org/10.4204/EPTCS.266.8>, doi:10.4204/EPTCS.266.8.

- 356 20 Markus Reiher, Nathan Wiebe, Krysta Marie Svore, Dave Wecker, and Matthias Troyer.
357 Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy*
358 *of Sciences of the United States of America*, 114 29:7555–7560, 2017.
- 359 21 Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie
360 University, 2015.
- 361 22 Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie,
362 editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University
363 Press, 2009.
- 364 23 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina
365 Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#:
366 Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings*
367 *of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018.
- 368 24 Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming*
369 *Languages and Systems (TOPLAS)*, 33(6):19, 2011.