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

Quantum programming is hard: Quantum programs are necessarily probabilistic and impossible to examine without disrupting the execution of a program. In response to this challenge, we and a number of other researchers have written tools to verify quantum programs against their intended semantics. *This is not enough.* Verifying an idealized semantics against a real world quantum program doesn't allow you to confidently predict the program's output. In order to have verification that works, you need both an error semantics related to the hardware at hand (this is necessarily low level) and certified compilation to the that same hardware. Once we have these two things, we can talk about an approach to quantum programming where we start by writing and verifying programs at a high level, attempt to verify properties of the compiled code, and repeat as necessary.

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

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

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.12

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, Xiaodi Wu, and Mingsheng Ying. We also thank the anonymous referees for their helpful feedback.

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

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 12; pp. 12:1–12:11

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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. *Measuring* this vector then returns an index with a probability corresponding to the weight at that index (we go into more detail in Section 2). 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, maximizing the probability that 7 or 11 is obtained. 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 (an effectful operation) and printing the returned value. This is akin to randomly and irreversibly coercing a floating point number to a nearby integer – it will give a weakly informative answer and corrupt the rest of the program. Unit tests are similarly of limited value when your program is probabilistic; repeatedly running unit tests on a quantum computer is likely to be prohibitively expensive. Simulating quantum programs on a classical computer holds some promise (and simulators are bundled with most quantum software packages) but it requires resources exponential in the number of qubits being simulated, so simulation can't help in the general case.

Where standard software assurance techniques fail us, formal verification thrives. When we reason about a quantum program, we are reasoning exclusively about vectors, and vectors don't collapse when we analyze them. Formal verification is parametric in its inputs: Instead of reasoning about a 128-qubit implementation of an algorithm, we can prove properties of that algorithm for arbitrary arities given as arguments. Using techniques like induction, algebraic reasoning and equational rewriting we can verify the correctness of a broad range of quantum programs, as we previously showed [29] using the $\mathcal{Q}$WIRE programming language and verification tool.

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

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

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

These limitations aren't uniform within a given machine, let alone across machines. On IBM's largest publicly available quantum computer, Melbourne, phase coherence times range from 22.1 to 106.5 microseconds depending on the qubit in question, and gate errors similarly vary by qubit [20].

Given the substantial challenges facing quantum computing, is formal verification even useful? We argue that it can be.

To tackle the limitations of near-term quantum computers, we will need to tailor our verification efforts to the lowest level of the quantum software stack. We will need to incorporate information about the connectivity and error rates of a given machine in order to verify that a program can be run on that machine, and to bound the error of such an execution. Such verification will be messy: It will have to take a lot of variables into account, including the ideal semantics of a given program, qubit by qubit decoherence and error rates, and specifications that may differ substantially by platform. However, we argue that this verification is necessary, and that we can provide the tools necessary to perform it.

To make this case, we begin by introducing quantum computing and the semantics of error-free quantum programs (Section 2). In Section 3, we survey the literature on formally verified quantum computing and in Section 3.2 we address certified optimizing compilers, both in an idealized, error-free setting. In Section 4 we consider the additional challenges faced by quantum programming in the near term, and how we can address them. We devote most of this discussion to the issues of errors (Section 4.1) and architectural limitations (Section 4.2). Finally, in Section 5 we reflect on how this discussion can inform the nascent field of quantum programming languages.

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 $\langle \alpha, \beta \rangle$ for complex numbers α and β, subject to the constraint that $|\alpha|^2 + |\beta|^2 = 1$.

Logical qubits obey strict rules but they are not deterministic. If we *measure* the qubit above, we will obtain one of the two *basis qubits* $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$ with probability $|\alpha|^2$ and $|\beta|^2$, respectively.

We can combine two qubits by taking their tensor product, where

$$\langle \alpha, \beta \rangle \otimes \langle \gamma, \delta \rangle = \langle \alpha\gamma, \alpha\delta, \beta\gamma, \beta\delta \rangle.$$

Measuring the quantum system above will yield one of four basis vectors, with probabilities corresponding to the given entries. With probability $|\alpha\gamma|^2$ we will obtain $\langle 1, 0, 0, 0 \rangle$; with probability $|\alpha\delta|^2$ we will obtain $\langle 0, 1, 0, 0 \rangle$; and so forth.

Besides 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 vectors. This state of affairs is called *entanglement* and is analogous to probabilistic dependence.

Let's give a simple example of entanglement in action. Imagine we start we with the simple two qubit state $\langle 1, 0 \rangle \otimes \langle 1, 0 \rangle$. We then apply the Hadamard unitary $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ to the first qubit, obtaining $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle \otimes \langle 1, 0 \rangle$, which we can also write as $\langle \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 0 \rangle$. The controlled-not unitary exchanges the third and fourth elements of the vector, yielding $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}} \rangle$. This entangled vector cannot be decomposed into the tensor product of two smaller vectors, and is known as a *Bell pair*.

(a) A circuit to produce a Bell pair.

(b) The density matrix for our Bell pair.

Figure 1 An example Bell pair.

What happens if we measure our Bell pair? As we've seen, we will obtain either $\langle 1,0,0,0\rangle = \langle 1,0\rangle \otimes \langle 1,0\rangle$ or $\langle 0,0,0,1\rangle = \langle 0,1\rangle \otimes \langle 0,1\rangle$, each with probability $\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$. After measurement, our two qubits are no longer entangled but their outcomes are correlated: Either both are $\langle 1,0\rangle$ or both are $\langle 0,1\rangle$. This correlation is an effect of entanglement, and one of the features that gives quantum computing its power.

Generally speaking, we will represent quantum programs as *circuits*. For instance, the circuit for constructing a Bell pair is shown in Figure 1, where 0 represents the vector $\langle 1,0\rangle$, H is the Hadamard matrix and the structure bridging the two wires is the controlled-not. The circuit model is standard for quantum computing: Quantum programming languages like Quipper [14], Scaffold [21] and $\mathcal{Q}$WIRE are all circuit description languages; Microsoft's recent Q# tries to move away from this model by adding some abstractions, but Q# programs can easily be read as describing circuits as well.

Conveniently, quantum 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. A density matrix for our Bell pair, obtained by multiplying $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\rangle$ by its transpose, is given in Figure 1. The $\frac{1}{2}$s in the first and fourth positions along the diagonal represent the probability of measuring both qubits as $\langle 1,0\rangle$ and $\langle 0,1\rangle$, respectively. Embedding probabilities inside density matrices saves us from having to include probabilistic transitions in our denotational semantics. Once we have a denotational semantics for quantum circuits, we can begin to prove things about them.

3 Verification Under Ideal Conditions

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 [35], Proto-Quipper [33] or our $\mathcal{Q}$WIRE [27] language. A linear type system treats a function type $A \multimap B$ as something that *consumes* an A (precisely once) and produces a B (that may itself be used precisely once). Hence, it ensures that once we've done an operation on a qubit, 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 can be thought of as scratch space for intermediate quantum computations, but they have to be regularly garbage collected. Since qubits can be entangled with one another, operating on improperly discarded ancillae can corrupt the rest of our computation. Ancillae are a common feature of quantum algorithms, and hence appear in quantum programming languages like Quipper [14] and Q# [36]. Unfortunately,

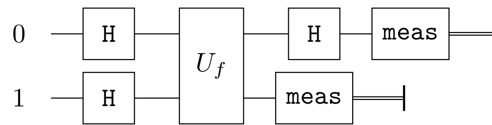

Figure 2 A circuit implementing Deutsch's algorithm.

it's quite difficult to guarantee that ancilla qubits are properly garbage collected. Inspired by the ReVerC [3] compiler for reversible programs, $\mathcal{Q}$WIRE allows the programmer to prove that ancillae are discarded correctly, while providing syntactic conditions for cases where this is trivially true [30]. In the general case, though, proving that we've appropriately disposed of ancillae requires us to reason about the behavior of complete quantum programs.

Doing whole-program analysis substantially increases the scope for formal verification, which inspired us to use $\mathcal{Q}$WIRE as a general-purpose verification tool [31]. One thing we may want to do is verify the correctness of a complete program, like Deutsch's algorithm [8, 11]. Deutsch's algorithm, shown in Figure 2, takes in an unknown function f, represented as a quantum gate U_f, and returns the 0 qubit if and only if the function is constant. Using $\mathcal{Q}$WIRE, we can express the correctness of this algorithm as follows:

```
Lemma deutsch_constant : ∀ f, constant f →
  ⟦deutsch (fun_to_gate f)⟧ I1 = |0⟩⟨0|.

Lemma deutsch_balanced : ∀ f, balanced f →
  ⟦deutsch (fun_to_gate f)⟧ I1 = |1⟩⟨1|.
```

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 one of the easier algorithms to prove correct since we can simply compute the output matrix. In practice we can verify a broad range of algorithms where this isn't possible, from families of quantum coin tossing circuits to quantum programs over arbitrary input unitaries [29].

$\mathcal{Q}$WIRE isn't the only tool that attempts to guarantee the correctness quantum programs. Amy [1] uses Feynman path integrals to check the correctness of a variety of concrete quantum circuits, including a quantum Fourier transform [10] using up to 31 qubits. A number of authors have also introduced Hoare-style logics for quantum programs and used them to verify Grover's algorithm [15, 40] and a quantum one-time pad [5, 38]. More specialized tools allow us to verify the security of quantum protocols [39] and rewrite quantum programs expressed in the ZX calculus [9, 22].

3.1 The Verification-Programming Loop

So far, we've treated verification as a task that is subsequent to programming, which guarantees that the program behaves as expected. However, in our experience, verification also plays a major role in quantum programming itself. 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 [18] recent exploration of bugs in the implementations of common quantum algorithms.)

Following Dijkstra's vision for formal verification in *A Discipline of Programming* [12], we expect quantum verification to assist in writing quantum programs. In practice, we often 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 can then go back to the original program, insert the missing Hadamard gate, and return to our verification attempt, repeating as necessary.

3.2 Compilation and Optimization

Not all errors are caused by mistakes in the high-level program. Following the success of the CompCert C compiler [23], another promising target for formal verification is the compiler that translates a high-level quantum program to a circuit capable of being run on a given architecture. This compiler should preserve the semantics of the source program and should also perform optimizations to reduce resource usage and running time. Given the difficulty of testing quantum programs and the expense involved in running them, it is especially important to verify these compilers.

A number of optimizing compilers have been designed for quantum programs to reduce resource usage [2, 16, 26]. These optimizations are often mathematically sophisticated, and thus vulnerable to programmer error. For example, in discussions with Nam et al. we learned that, while developing their optimizer, they found several bugs in their own implementation and also in the implementations that they compared against [26].

The Quantomatic tool [22] can apply verified transformations to quantum computation expressed in terms of the ZX-calculus [9, 4], a diagrammatic approach to quantum computation. Unfortunately, not every ZX diagram corresponds to a valid quantum circuit, and an optimization in ZX may not optimize a corresponding circuit. Recent work [13] optimizes a restricted subset of ZX diagrams that do represent circuits, but these are limited to a subset of quantum circuits known as Clifford circuits.

We are currently developing a new intermediate representation for quantum circuits, called sQIRe [17], to help us go further. sQIRe allows us to perform verified optimizations on circuits, with the goal of reducing the total circuit size. So far we have verified simple optimizations, like skip elimination and canceling repeated X (negation) gates.

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, these logics will be of limited use for the near future. The quantum computers that exist today and are likely to exist over the next ten to twenty years will be incapable of running arbitrary quantum circuits and will be very failure prone. Hence, for formal verification to be useful in the near term, it will need to be tied to the machines we expect, not those we hope for.

In a recent keynote address [28], 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 under 1000 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 performing a range of other tasks. In the near term, quantum programs will have to be aimed at problems for which they are uniquely well-suited (like modeling quantum systems in physics [7] and chemistry [32]) and tailored to the limitations of the available computers.

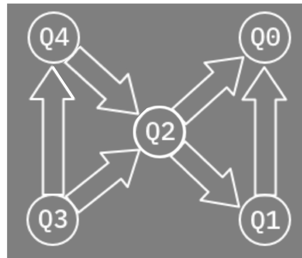

Figure 3 Two-qubit gate connections on IBM's Tenerife machine. Taken from `https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md`.

What do these limitations look like? One hard limitation is the number of qubits on the machine. Another limitation is error rates. There are multiple sources of errors in quantum circuits, stemming from *decoherence* (qubits tend to revert to their basis states with time) and errors in gate application.[1] Today's machines also have a number of architectural limitations, related to the connectivity of qubits: Instead of being complete undirected graphs, they tend to resemble sparse directed graphs. For example, consider the diagram of IBM's 5-qubit Tenerife machine shown in Figure 3. In this architecture, if you want to apply a controlled-not gate from program qubit q_1 to program qubit q_2, then you need to map those program qubits to adjacent physical qubits in the machine (e.g. $Q4$ and $Q2$). This mapping may need to be updated over the course of the program by adjusting the physical locations of program qubits. This adjustment 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 these crucial limitations of NISQ devices.

4.1 Verification in the Presence of Errors

Let us sketch out some possible approaches to dealing with the errors that are sure to arise when we run our quantum computers.

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 having the type `Qubit ⊸ Qubit`, it can have the type ∀n, (`Qubit`,n) ⊸ (`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 (Figure 2) 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.

An advantage of this approach is that it's very easy to implement and sufficiently general that we can interpret the output in a number of different ways, depending on the setting. We can also automate it, allowing a built-in type inference algorithm to calculate the errors that occur in a given circuit.[2] It is limited, though, in that errors only increase as the circuit size grows. This makes it difficult to analyze programs that include mechanisms for error mitigation, which will be important in near-term applications [24]. (In principle, we could have error-correcting gates that shrink the error, but error correction tends not to be so simple.)

We can also take ideas from our recent robustness logic [19], which draws on Carbin et al.'s Rely language for approximate classical computing [6] and provides bounds for the errors in a quantum program. While powerful, this logic suffers from the same limitations as our error wire semantics – errors only increase throughout a program. It is also high level: It uses the same quantum while language as QHL [40], which doesn't directly describe circuits that can run on near-term machines.

Ideally, the semantics for a quantum program would contain error terms, corresponding to possible failures. This would bring the advantage of allowing us to reason about and reduce error terms, through majority voting or the like. Unfortunately, it's still hard to know what the denotational model should be, without reference to the specific hardware. Hence, while doing high-level reasoning we want to leave the error term abstract (as in our robustness logic), and instantiate it for specific hardware models.

4.2 Verified Compilation to Restricted Architectures

As we've seen, in order to make effective use of near-term quantum devices, we cannot entirely abstract away low-level architectural details. However, this is not to say that the programmer needs to consider every low-level detail when writing programs. In some cases, as in classical computing, the complexity of running on a particular architecture can be handled by the compiler.

For example, we have already discussed the challenge of limited connectivity on near-term machines. There has been significant work on developing automated transformations that map arbitrary quantum circuits into circuits that satisfy a particular machine's connectivity constraints [41]. Some of this work has even looked at how to perform this mapping in a way that reduces the error of the resulting circuit [25, 37].

Another way that near-term compilers for quantum programs can help is by performing optimizations that reduce resource usage, as discussed in Section 3.2. Reducing resource usage is critical because near-term devices have access to a limited number of qubits and can support few operations before decoherence undoes the effect of any useful computation.

We would like to go further. A verified compiler for quantum circuits should take the following three things into account:

1. The connectivity of the target machine;
2. the error rates for each qubit; and
3. the fidelity of individual gates applications.

It should use this information to compile an arbitrary quantum circuit to an equivalent circuit that can run on the target machine in a way that minimizes errors. This is difficult: Each of our desirata imposes substantial constraint on the compiler and a burden on the verifier. However, each is necessary for our compiler to be both useful and reliable.

[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.

Our new sQIRe language [17] can be used to verify transformations that guarantee structural properties of circuit. For instance, the `map_to_lnn` function compiles to a linear nearest neighbor architectures, guaranteeing that every controlled-not gate is applied to adjacent qubits. However, at present, this transformation is applied manually, rather than being part of a compilation toolchain. Also, the linear nearest neighbor architecture is a toy example: The connectivity of a quantum computer can be represented by any graph, directed or undirected. Verified compilation to such machines remains a significant challenge.

5 Looking Forward

As we've seen, verification has an important role to play in quantum programming, both in the short and long term. Quantum programs are buggy and difficult to test, so if we want to have any confidence in our programs' correctness, we had better verify them. There is already substantial progress on this front, through tools like $\mathcal{Q}$WIRE [27, 29] and QHL [40].

However, in the short term these tools aren't sufficient. We need an approach to verification that deals with errors, like our recent quantum robustness logic [19]. Moreover, we need to verify error-prone programs with respect to the hardware we intend to run them on. The coherence times and error rates of quantum computers vary widely and a good error model will need to be machine specific.

Machines aren't only constrained by their error rates, but also by the number and connectivity of their qubits. This results in additional constraints that a compiler must satisfy, and satisfaction of these constraints must also be verified. The sQIRe intermediate representation for quantum circuits is a step towards verified compilation, but much work remains to be done.

In an ideal world, we would not think about circuit, let alone machine architectures, when writing quantum programs. Indeed, another ambitious project for quantum computing is developing useful abstractions for programmers. Some steps in this direction include quantum control flow [34] and amplitude amplification as a subroutine [36]. These efforts look towards a future where we have scalable quantum computers with many error-free qubits.

By contrast, we are focused on the quantum devices available today and likely to be available over the next decade. Our goal is to develop tools that will assist in developing efficient algorithms with correctness guarantees and precisely bounded errors. We aim to execute those algorithms on fundamentally limited machines, with low qubit counts and high error rates. To that end, we have to provide information about 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, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33, 2013.

3 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/.

4 Miriam Backens. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, 2014.

5 P Oscar Boykin and Vwani Roychowdhury. Optimal encryption of quantum bits. *Physical review A*, 67(4):042317, 2003.

6 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 33–52, 2013. doi:10.1145/2509136.2509546.

7 Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences of the United States of America*, 115 38:9456–9461, 2018.

8 Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 339–354. The Royal Society, 1998.

9 Bob Coecke and Ross Duncan. Interacting quantum observables. In *International Colloquium on Automata, Languages, and Programming*, pages 298–310. Springer, 2008.

10 Don Coppersmith. An Approximate Fourier Transform Useful in Quantum Factoring. *arXiv preprint*, 1994. arXiv:quant-ph/0201067.

11 David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 400, pages 97–117. The Royal Society, 1985.

12 Edsger Wybe Dijkstra. *A Discipline of Programming*, volume 1. Prentice-Hall, 1976.

13 Andrew Fagan and Ross Duncan. Optimising Clifford Circuits with Quantomatic. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.

14 Alexander 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 2013, pages 333–342, 2013.

15 Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237866.

16 Luke Heyfron and Earl T. Campbell. An Efficient Quantum Compiler that reduces T count. *Quantum Science and Technology*, 4, 2017.

17 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. Verified Optimization in a Quantum Intermediate Representation. *arXiv e-prints*, page arXiv:1904.06319, April 2019. arXiv:1904.06319.

18 Yipeng Huang and Margaret Martonosi. QDB: from quantum algorithms towards correct quantum programs. *arXiv preprint*, 2018. arXiv:1811.05447.

19 Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. Quantitative Robustness Analysis of Quantum Programs. *Proc. ACM Program. Lang.*, 3(POPL):31:1–31:29, January 2019. doi:10.1145/3290344.

20 IBM. IBM quantum experience, 2017. URL: https://quantumexperience.ng.bluemix.net/qx/devices.

21 Ali Javadi-Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE, 2012.

22 Aleks Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing.* PhD thesis, University of Oxford, 2011.

23 Xavier Leroy et al. The CompCert verified compiler. *Development available at http://compcert.inria.fr*, 2009, 2004.

24 Sam McArdle, Xiao Yuan, and Simon Benjamin. Error mitigated digital quantum simulation. *arXiv preprint*, 2018. arXiv:1807.02467.

25 Prakash Murali, Jonathan M Baker, Ali Javadi Abhari, Frederic T Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. *arXiv preprint*, 2019. arXiv:1901.11054.

26 Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018.

27 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*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009894.

28 John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. doi:10.22331/q-2018-08-06-79.

29 Robert Rand. *Formally Verified Quantum Programming.* PhD thesis, University of Pennsylvania, 2018.

30 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, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.

31 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, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. doi:10.4204/EPTCS.266.8.

32 Markus Reiher, Nathan Wiebe, Krysta Marie Svore, Dave Wecker, and Matthias Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences of the United States of America*, 114 29:7555–7560, 2017.

33 Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation.* PhD thesis, Dalhousie University, 2015.

34 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*, pages 348–364. Springer, 2018.

35 Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 2009.

36 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018.

37 Swamit S. Tannu and Moinuddin K. Qureshi. A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. *arXiv e-prints*, page arXiv:1805.10224, May 2018. arXiv:1805.10224.

38 Dominique Unruh. Quantum Hoare Logic with Ghost Variables. *arXiv preprint*, 2019. arXiv:1902.00325.

39 Dominique Unruh. Quantum Relational Hoare Logic. *Proc. ACM Program. Lang.*, 3(POPL):33:1–33:31, January 2019. doi:10.1145/3290346.

40 Mingsheng Ying. Floyd-Hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.

41 Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1135–1138, 2018. doi:10.23919/DATE.2018.8342181.