

A Verified Optimizer for Quantum Circuits

KESHA HIETALA, University of Maryland, USA

ROBERT RAND, University of Chicago, USA

SHIH-HAN HUNG, University of Maryland, USA

XIAODI WU, University of Maryland, USA

MICHAEL HICKS, University of Maryland, USA

We present voqc, the first fully *verified optimizer for quantum circuits*, written using the Coq proof assistant. Quantum circuits are expressed as programs in a simple, low-level language called sqir, a *simple quantum intermediate representation*, which is deeply embedded in Coq. Optimizations and other transformations are expressed as Coq functions, which are proved correct with respect to a semantics of sqir programs. sqir uses a semantics of matrices of complex numbers, which is the standard for quantum computation, but treats matrices symbolically in order to reason about programs that use an arbitrary number of quantum bits. sqir's careful design and our provided automation make it possible to write and verify a broad range of optimizations in voqc, including full-circuit transformations from cutting-edge optimizers.

CCS Concepts: • **Hardware** → *Quantum computation; Circuit optimization*; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Formal Verification, Quantum Computing, Circuit Optimization, Certified Compilation, Programming Languages

ACM Reference Format:

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (January 2021), 36 pages. <https://doi.org/10.1145/3434318>

1 INTRODUCTION

Programming quantum computers will be challenging, at least in the near term. Qubits will be scarce and gate pipelines will need to be short to prevent decoherence. Fortunately, optimizing compilers can transform a source algorithm to work with fewer resources. Where compilers fall short, programmers can optimize their algorithms by hand.

Of course, both compiler and by-hand optimizations will inevitably have bugs. As evidence of the former, [Kissinger and van de Wetering \[2020\]](#) discovered mistakes in the optimized outputs produced by the circuit optimizer of [Nam et al. \[2018\]](#), and [Nam et al.](#) themselves found that the optimization library they compared against ([Amy et al. \[2013\]](#)) sometimes produced incorrect results. Likewise, [Amy \[2018\]](#) discovered an optimizer they had recently developed produced buggy results [[Amy et al. 2018](#)]. Making mistakes when optimizing by hand is also to be expected: as put well by [Zamdzhev \[2016\]](#), quantum computing can be frustratingly unintuitive.

Authors' addresses: Kesha Hietala, University of Maryland, USA, kesha@cs.umd.edu; Robert Rand, University of Chicago, USA, rand@uchicago.edu; Shih-Han Hung, University of Maryland, USA, shung@cs.umd.edu; Xiaodi Wu, University of Maryland, USA, xwu@cs.umd.edu; Michael Hicks, University of Maryland, USA, mwh@cs.umd.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).

2475-1421/2021/1-ART37

<https://doi.org/10.1145/3434318>

Unfortunately, the very factors that motivate optimizing quantum compilers make it difficult to test their correctness. Comparing runs of a source program to those of its optimized version is often impractical due to the indeterminacy of typical quantum algorithms and the substantial expense involved in executing or simulating them. Indeed, resources may be too scarce, or the qubit connectivity too constrained, to run the program without optimization!

An appealing solution to this problem is to apply rigorous *formal methods* to prove that an optimization or algorithm always does what it is intended to do. For example, CompCert [Leroy 2009] is a compiler for C programs that is written and proved correct using the Coq proof assistant [Coq Development Team 2019]. CompCert includes sophisticated optimizations whose proofs of correctness are verified to be valid by Coq’s type checker.

In this paper, we apply CompCert’s approach to the quantum setting. We present voQC (pronounced “vox”), a *verified optimizer for quantum circuits*. voQC takes as input a quantum program written in a language we call sQIR (“squire”). sQIR is designed to be a *small quantum intermediate representation*, but it is suitable for source-level programming too: it is not very different from languages such as Quil [Smith et al. 2016] or OpenQASM [Cross et al. 2017], which describe quantum programs as circuits. sQIR is deeply embedded in Coq, similar to how Quil is embedded in Python via PyQuil [Rigetti Computing 2019a], allowing us to write sophisticated quantum programs. voQC applies a series of optimizations to sQIR programs, ultimately producing a result that is compatible with a specified quantum architecture. For added convenience, voQC provides translators between sQIR and OpenQASM. (Section 2.)

We designed sQIR to make it as easy as possible to reason about the semantics of quantum programs, which are significantly different from the semantics of classical programs. For example, while in a classical program one can reason about different variables independently, the phenomenon of quantum entanglement requires us to reason about a global quantum state, typically represented as a large vector or matrix of complex numbers. This means that reasoning about quantum states involves linear algebra, and often trigonometry and probability too. As a result, existing approaches to program proofs in Coq tend not to apply in the quantum setting. Indeed, we first attempted to build voQC using QWIRE [Paykin et al. 2017], which is also embedded in Coq and more closely resembles a classical programming language, but found proofs of even simple optimizations to be non-trivial and hardly scalable.

To address these challenges, sQIR’s design has several key features (Section 3). First, it uses natural numbers in place of variables so that we can naturally index into the vector or matrix state. Using variables directly (e.g., with higher-order abstract syntax [Pfenning and Elliott 1988], as in QWIRE and Quipper [Green et al. 2013]) necessitates a map from variables to indices, which we find confounds proof automation. Second, sQIR provides two semantics for quantum programs. We express the semantics of a general program as a function between *density matrices*, as is standard (e.g., in QPL [Selinger 2004] and QWIRE), since density matrices can represent the *mixed states* that arise when a program applies a measurement operator (Section 5). However, measurement typically occurs at the end of a computation, rather than within it, so we also provide a simpler *unitary semantics* for (sub-)programs that do not measure their inputs. In this case, a program’s semantics corresponds to a restricted class of square matrices. These matrices are often much easier to work with, especially when employing automation. Other features of sQIR’s design, like assigning an ill-typed program the denotation of the zero-matrix, are similarly intended to ease proof. Pleasantly, unitary sQIR even turns out to be effective for proving quantum programs correct. This paper presents a proof of correctness of *GHZ state preparation* [Greenberger et al. 1989]; in concurrent work [Hietala et al. 2020], we have proved the correctness of implementations of *Quantum Phase Estimation* (a key component of Shor’s prime factoring algorithm [1994]), *Grover’s search algorithm* [1996], and *Simon’s algorithm* [1994].

At the core of voQC is a framework for writing transformations of sqIR programs and verifying their correctness. To ensure that the framework is suitably expressive, we have used it to develop verified versions of a variety of optimizations. Many are based on those used in an optimizer developed by Nam et al. [2018], which is the best performing optimizer we know of in terms of total gate reduction (per experiments described below). We abstract these optimizations into a couple of different classes, and provide library functions, lemmas, and automation to simplify their construction and proof. We have also verified a circuit mapping routine that transforms sqIR programs to satisfy constraints on how qubits may interact on a specified target architecture. (Section 4.)

We evaluated the quality of the optimizations we verified in voQC, and by extension the quality of our framework, by measuring how well it optimizes a set of benchmark programs, compared to Nam et al. and several other optimizing compilers. The results are encouraging. On a benchmark of 28 circuit programs developed by Amy et al. [2013] we find that voQC reduces total gate count on average by 17.8% compared to 10.1% for IBM’s Qiskit compiler [Aleksandrowicz et al. 2019], 10.6% for CQC’s t|ket) [Cambridge Quantum Computing Ltd 2019], and 24.8% for the cutting-edge research optimizer by Nam et al. [2018]. On the same benchmarks, voQC reduces T -gate count (an important measure when considering fault tolerance) on average by 41.4% compared to 39.7% by Amy et al. [2013], 41.4% by Nam et al., and 42.6% by the PyZX optimizer. Results on an even larger benchmark suite (detailed in Appendix A) tell the same story. In sum, voQC and sqIR are expressive enough to verify a range of useful optimizations, yielding performance competitive with standard compilers. (Section 6.)

voQC is the first fully verified optimizer for general quantum programs. Amy et al. [2017] developed a verified optimizing compiler from source Boolean expressions to reversible circuits and Fagan and Duncan [2018] verified an optimizer for ZX-diagrams representing Clifford circuits; however, neither of these tools handle general quantum programs. In concurrent work, Shi et al. [2019] developed CertiQ, which uses symbolic execution and SMT solving to verify circuit transformations in the Qiskit compiler. CertiQ is limited to verifying correct application of local equivalences and does not provide a way to describe general quantum states (a key feature of sqIR), which limits the types of optimizations that it can reason about. This also means that it cannot be used as a tool for verifying general quantum programs. Smith and Thornton [2019] presented a compiler with built-in translation validation via QMDD equivalence checking [Miller and Thornton 2006]. However, QMDDs represent quantum state concretely, which means that the validation time will increase exponentially with the number of qubits in the compiled program. In contrast to these, sqIR represents matrices *symbolically*, which allows us to reason about arbitrary quantum computation and verify interesting, non-local optimizations, independently of the number of qubits in the optimized program. (Section 7.)

Our work on voQC and sqIR are steps toward a broader goal of developing a full-scale verified compiler toolchain. Next steps include developing certified transformations from higher-level quantum languages to sqIR and implementing optimizations with different objectives, e.g., that aim to reduce the probability that a result is corrupted by quantum noise. All code we reference in this paper can be found online at <https://github.com/inQWIRE/SQIR>.

2 OVERVIEW

We begin with a brief background on quantum programs, focusing on the challenges related to formal verification. We then provide an overview of voQC and sqIR, summarizing how they address these challenges.

2.1 Preliminaries

Quantum programs operate over *quantum states*, which consist of one or more *quantum bits* (a.k.a. *qubits*). A single qubit is represented as a vector of complex numbers $\langle \ ; \ \rangle$ such that $| \ |^2 + | \ |^2 = 1$. The vector $\langle 1; 0 \rangle$ represents the state $|0\rangle$ while vector $\langle 0; 1 \rangle$ represents the state $|1\rangle$. A state written $| \ \rangle$ is called a *ket*, following Dirac’s notation. We say a qubit is in a *superposition* of $|0\rangle$ and $|1\rangle$ when both $\ \$ and $\ \$ are non-zero. Just as Schrodinger’s cat is both dead and alive until the box is opened, a qubit is only in superposition until it is *measured*, at which point the outcome will be 0 with probability $| \ |^2$ and 1 with probability $| \ |^2$. Measurement is not passive: it has the effect of collapsing the state to match the measured outcome, i.e., either $|0\rangle$ or $|1\rangle$. As a result, all subsequent measurements return the same answer.

Operators on quantum states are linear mappings. These mappings can be expressed as matrices, and their application to a state expressed as matrix multiplication. For example, the *Hadamard* operator H is expressed as a matrix $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. Applying H to state $|0\rangle$ yields state $\langle \frac{1}{\sqrt{2}}; \frac{1}{\sqrt{2}} \rangle$, also written as $|+\rangle$. Many quantum operators are not only linear, they are also *unitary*—the conjugate transpose (or adjoint) of their matrix is its own inverse. This ensures that multiplying a qubit by the operator preserves the qubit’s sum of norms squared. Since a Hadamard is its own adjoint, it is also its own inverse: hence $H|+\rangle = |0\rangle$.

A quantum state with n qubits is represented as vector of length 2^n . For example, a 2-qubit state is represented as a vector $\langle \ ; \ ; \ ; \ \rangle$ where each component corresponds to (the square root of) the probability of measuring $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, respectively. Because of the exponential size of the complex quantum state space, it is not possible to simulate a 100-qubit quantum computer using even the most powerful classical computer!

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

n -qubit operators are represented as $2^n \times 2^n$ matrices. For example, the *CNOT* operator over two qubits is expressed as the matrix shown at the right. It expresses a *controlled not* operation—if the first qubit (called the *control*) is $|0\rangle$ then both qubits are mapped to themselves, but if the first qubit is $|1\rangle$ then the second qubit (called the *target*) is negated, e.g., $CNOT|00\rangle = |00\rangle$ while $CNOT|10\rangle = |11\rangle$.

n -qubit operators can be used to create *entanglement*, which is a situation where two qubits cannot be described independently. For example, while the vector $\langle 1; 0; 0; 0 \rangle$ can be written as $\langle 1; 0 \rangle \otimes \langle 1; 0 \rangle$ where \otimes is the tensor product, the state $\langle \frac{1}{\sqrt{2}}; 0; 0; \frac{1}{\sqrt{2}} \rangle$ cannot be similarly decomposed. We say that $\langle \frac{1}{\sqrt{2}}; 0; 0; \frac{1}{\sqrt{2}} \rangle$ is an entangled state.

An important non-unitary quantum operator is *projection* onto a subspace. For example, $|0\rangle\langle 0|$ (in matrix notation $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$) projects a qubit onto the subspace where that qubit is in the $|0\rangle$ state. Projections are useful for describing quantum states after measurement has been performed. We sometimes use $|i\rangle_q \langle i|$ as shorthand for applying the projection $|i\rangle \langle i|$ to qubit q and an identity operation to every other qubit in the state.

2.2 Quantum Circuits

Quantum programs are typically expressed as circuits, as shown in Figure 1(a). In these circuits, each horizontal wire represents a *qubit* and boxes on these wires indicate quantum operators, or *gates*. Gates can either be unitary operators (e.g., Hadamard, *CNOT*) or non-unitary ones (e.g., measurement). In software, quantum circuit programs are often represented using lists of instructions that describe the different gate applications. For example, Figure 1(b) is the Quil [Smith et al. 2016] representation of the circuit in Figure 1(a).

In the *QRAM model* [Knill 1996] quantum computers are used as co-processors to classical computers. The classical computer generates descriptions of circuits to send to the quantum

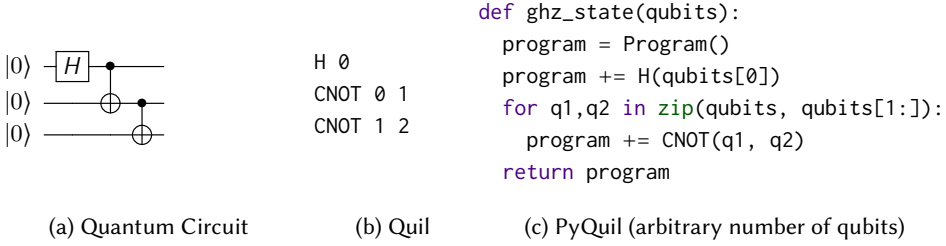


Fig. 1. Example quantum program: GHZ state preparation

computer and then processes the measurement results. High-level quantum programming languages are designed to follow this model. For example, Figure 1(c) shows a program in PyQuil [Rigetti Computing 2019a], a quantum programming framework embedded in Python. The `ghz_state` function takes an array `qubits` and constructs a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [Greenberger et al. 1989], which is an n -qubit entangled quantum state of the form

$$|\text{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n});$$

Calling `ghz_state([0, 1, 2])` returns the Quil program in Figure 1(b), which produces the quantum state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$. The high-level language may provide facilities to optimize constructed circuits, e.g., to reduce gate count, circuit depth, and qubit usage. It may also perform transformations to account for hardware-specific details like the number of qubits, available set of gates, or connectivity between physical qubits.

2.3 sqir: A Small Quantum Intermediate Representation Supporting Verification

What if we want to formally verify that `ghz_state`, when passed an array of indices $[0, \dots, n-1]$, returns a circuit that produces the quantum state $|\text{GHZ}^n\rangle$? What steps are necessary?

First, we need a way to formally define quantum states as matrices of complex numbers. Indeed, we need a way to define indexed *families* of states— $|\text{GHZ}^n\rangle$ is a function from an index n to a quantum state. Second, we need a formal language in which to express quantum programs; to this language we must ascribe a mathematical semantics in terms of quantum states. A program like `ghz_state` is a function from an index (a list of length n) to a circuit (of size n), and this circuit’s denotation is its equivalent (unitary) matrix (of size 2^n). Finally, we need a way to mechanically reason that, for arbitrary n , the semantics of `ghz_state([0, 1, \dots, n-1])` applied to the zero state $(|0\rangle^{\otimes n})$ is equal to the state $|\text{GHZ}^n\rangle$.

We designed `sqir`, a *small quantum intermediate representation*, to do all of these things. `sqir` is a simple circuit-oriented language deeply embedded in the Coq proof assistant in a manner similar to how Quil is embedded in Python via PyQuil. We use `sqir`’s host language, Coq, to define the syntax and semantics of `sqir` programs and to express properties about quantum states. We developed a library of lemmas and tactic-based automation to assist in writing proofs about quantum programs; such proofs make heavy use of complex numbers and linear algebra. These proofs are aided by isolating `sqir`’s *unitary core* from primitives for measurement, which require consideration of probability distributions of outcomes (represented as *density matrices*); this means that (sub-)programs that lack measurement can have simpler proofs. Either way, in `sqir` we perform reasoning *symbolically*. For example, we can prove that every circuit generated by the `sqir`-equivalent of `ghz_state` produces the expected state $|\text{GHZ}^n\rangle$ when applied to input lists of length n , for any n .

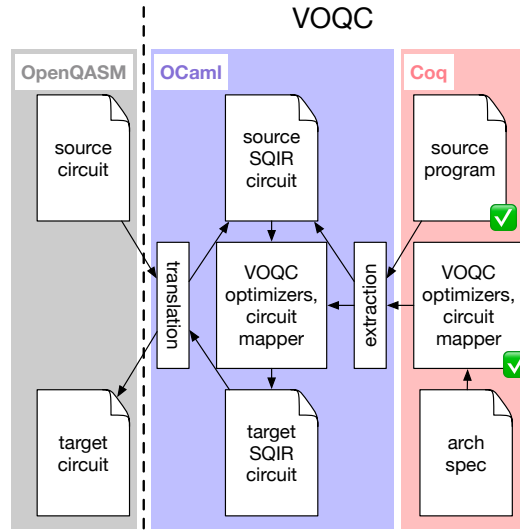


Fig. 2. The voqc architecture. Input circuits can be described in OpenQASM, OCaml, or Coq (top). The verified voqc optimizer is extracted to an executable OCaml optimizer (middle), which can produce an optimized sqir or OpenQASM circuit (bottom). The dashed line indicates the separation between the standard quantum compiler stack (left) and our contribution (right).

sqir is implemented in just over 3500 lines of Coq. We started with Coq libraries for complex numbers and matrices developed for the *QWIRE* language [Paykin et al. 2017]; over the course of our work we have extended these libraries with around 3000 lines of code providing more automation for linear algebra and better support for complex phases. We present sqir’s syntax and semantics along with an example program and verified property of correctness in Section 3.

2.4 voqc: A Verified Optimizer for Quantum Circuits

While sqir is suitable for proving correctness properties about source programs like `ghz_state`, its primary use has been as the intermediate representation of voqc, our verified optimizer for quantum circuits, and the signature achievement of this paper. An optimizer is a function from programs to programs, with the intention that the output program has the same semantics as the input. In voqc, we prove this is always the case: a voqc optimization f is a Coq function over sqir circuit C , and we prove that the semantics of input circuit C is always equivalent to the semantics of the output $f(C)$.

The voqc approach stands in contrast to prior work that relies on translation validation [Amy 2018; Kissinger and van de Wetering 2020; Smith and Thornton 2019], which may fail to identify latent bugs in the optimizer, while adding compile-time overhead. By proving correctness with respect to an explicit semantics for input/output programs (i.e., that of sqir), voqc optimizations are flexible in their expression. Prior work has been limited to peephole optimizations [Shi et al. 2019], leaving highly effective, full-circuit optimizations we have proved correct in voqc out of reach. Such global (non-peephole) proofs are aided by the design of sqir (notably, the isolation of a unitary core) and accompanying proof automation.

The structure of voqc is summarized in Figure 2. The voqc transformations themselves are shown at the middle right, and are described in Sections 4 and 5.3. In addition to performing circuit optimizations, voqc also performs *circuit mapping*, transforming a sqir program to an equivalent

one that respects constraints imposed by the target architecture. Once again, we prove that it does so correctly (Section 4.7).

Using Coq’s standard code extraction mechanism, we can extract voqc into a standalone OCaml program. This program takes as input a sqir program in an OCaml representation. This input can be extracted from a Coq-hosted (and proved correct) sqir program (upper right), or from a program expressed in OpenQASM [Cross et al. 2017], a standard representation for quantum circuits (upper left). Since a number of quantum programming frameworks, including Qiskit [Aleksandrowicz et al. 2019], t|ket) [Cambridge Quantum Computing Ltd 2019], Project Q [Steiger et al. 2018] and Cirq [The Cirq Developers 2019], can output OpenQASM, this allows us to run voqc on a variety of generated circuits, without requiring the user to program in OCaml or Coq.

voqc is implemented in about 7200 lines of Coq, with roughly 2100 lines for circuit mapping, 2100 lines for general-purpose sqir program manipulation, 2200 lines for unitary program optimizations, and 800 lines for non-unitary program optimizations. We use about 400 lines of standalone OCaml code for running voqc on our benchmarks in Section 6. We started work on sqir and voqc in March 2019 and concluded work for this paper in May 2020. The majority of the optimizations were implemented and verified within a span of four months.

3 SQIR: A SMALL QUANTUM INTERMEDIATE REPRESENTATION

Here we present the syntax and semantics of sqir, a *small quantum intermediate representation*. The sqir language is composed of two parts: a core language of unitary operators and a full language that incorporates measurement. This section focuses on the former; Section 5 presents the latter.

As we will show, the semantics of a unitary sqir program is expressed directly as a matrix, in contrast to the full sqir, which treats programs as functions over density matrices. This matrix semantics greatly simplifies proofs, both of the correctness of unitary optimizations (the bulk of voqc) and of source programs, many of which are essentially unitary (measurement is the very last step). Other aspects of sqir’s design also make proofs easier, as we will discuss and demonstrate with an example at the end of the section.

3.1 Unitary sqir: Syntax

A unitary sqir program U is a sequence of applications of gates G to qubits q .

$$U := U_1; U_2 \mid G \ q \mid G \ q_1 \ q_2$$

Qubits are referred to by natural numbers that index into a *global register* of quantum bits. Each sqir program is parameterized by a set of unitary one- and two-qubit gates (from which G is drawn) and the dimension of the global register (i.e., the number of available qubits). In Coq, a unitary sqir program U has type `ucom g n`, where g identifies the gate set and n is the size of the global register.

As an example, consider the program to the right, which is equivalent to PyQuil’s `ghz_state` from Figure 1(c). The Coq function `ghz` recursively constructs a sqir program, i.e., a Coq value of type `ucom base n`. This program, when run, prepares the GHZ state. When n is 0, `ghz` produces a sqir program that is just the identity gate I applied to qubit 0. When n is 1, the result is the Hadamard gate H applied to qubit 0. When n is greater than 1, `ghz` constructs the program $U_1; U_2$, where U_1 is the `ghz` circuit on n' (i.e., $n - 1$) qubits, and U_2 is the appropriate *CNOT* gate. The result of `ghz 3` is equivalent to the circuit shown in Figure 1(a).

```
Fixpoint ghz (n : N) : ucom base n :=
  match n with
  | 0 => I 0
  | 1 => H 0
  | S n' => ghz n'; CNOT (n'-1) n'
  end.
```

$$\begin{aligned}
nU_1; U_2 0_d &= nU_2 0_d \times nU_1 0_d \\
nG_1 q 0_d &= \begin{cases} \text{appl}_1(G_1; q; d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases} \\
nG_2 q_1 q_2 0_d &= \begin{cases} \text{appl}_2(G_2; q_1; q_2; d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 3. Semantics of unitary `sqir` programs, assuming a global register of dimension d . The appl_k function maps a gate name to its corresponding unitary matrix and extends the intended operation to the given dimension by applying an identity operation on every other qubit in the system.

3.2 Semantics

Suppose that M_1 and M_2 are the matrices corresponding to unitary gates U_1 and U_2 , which we want to apply to a quantum state vector $|\psi\rangle$. Matrix multiplication is associative, so $M_2(M_1|\psi\rangle)$ is equivalent to $(M_2M_1)|\psi\rangle$. Moreover, multiplying two unitary matrices yields a unitary matrix. As such, the semantics of `sqir` program $U_1; U_2$ is naturally described by the unitary matrix M_2M_1 .

This semantics is shown in Figure 3. There are two things to notice. First, if a program is not *well-typed* its denotation is the zero matrix (of size $2^d \times 2^d$). A program U is well-typed if every gate application is *valid*, meaning that its index arguments are within the bounds of the global register, and no index is repeated. The latter requirement enforces linearity and thereby quantum mechanics' *no-cloning theorem*, which says that it is impossible to create a copy of an arbitrary quantum state.

Otherwise, the program's denotation follows from the composition of the matrices that correspond to each of the applications of its unitary gates, G . The only wrinkle is that a full program consists of many gates, each operating on 1 or 2 of the total qubits; thus, a gate application's matrix needs to apply the identity operation to the qubits not being operated on. This is what appl_1 and appl_2 do. For example, $\text{appl}_1(G_u; q; d) = I_{2^q} \otimes U \otimes I_{2^{(d-q-1)}}$ where U is the matrix interpretation of the gate G_u and I_k is the $k \times k$ identity matrix. The appl_2 function requires us to decompose the two-qubit unitary into a sum of tensor products: for instance, *CNOT* can be written as $|0\rangle\langle 0| \otimes I_2 + |1\rangle\langle 1| \otimes X$ where $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. We then have

$$\text{appl}_2(\text{CNOT}; q_1; q_2; d) = I_{2^{q_1}} \otimes |0\rangle\langle 0| \otimes I_{2^r} \otimes I_2 \otimes I_{2^s} + I_{2^{q_1}} \otimes |1\rangle\langle 1| \otimes I_{2^r} \otimes X \otimes I_{2^s}$$

where $r = q_2 - q_1 - 1$ and $s = d - q_2 - 1$, assuming $q_1 < q_2$.

In our development we define the semantics of `sqir` programs over the gate set $G \in \{R_{\theta; \phi; \chi}; \text{CNOT}\}$ where $R_{\theta; \phi; \chi}$ is a general single-qubit rotation parameterized by three real-valued rotation angles and *CNOT* is the standard two-qubit controlled-not gate. This is our *base set* of gates. It is the same as the underlying set used by OpenQASM [Cross et al. 2017] and is *universal*, meaning that it can approximate any unitary operation to within arbitrary error. The matrix interpretation of the single-qubit $R_{\theta; \phi; \chi}$ gate is

$$\begin{pmatrix} \cos(\theta/2) & -e^{i\phi} \sin(\theta/2) \\ e^{i\chi} \sin(\theta/2) & e^{i(\chi+\phi)} \cos(\theta/2) \end{pmatrix}$$

and the matrix interpretation of the *CNOT* gate is given in Section 2.1.

Common single-qubit gates can be defined in terms of $R_{\theta; \phi; \chi}$. For example, the two single-qubit gates used in our GHZ example—identity I and Hadamard H —are respectively defined as $R_{0;0;0}$ and $R_{\pi/2;0;0}$. The Pauli X ("NOT") gate is $R_{\pi;0;0}$ and the Pauli Z gate is $R_{0;0;\pi}$. We can also define

more complex operations as `sqir` programs. For example, the *SWAP* operation, which swaps two qubits, can be defined as a sequence of three *CNOT* gates.

3.3 Design for Proofs

We designed `sqir`'s unitary core to simplify formal proofs, in three ways.

Zero Matrix for Ill-typed Programs. By giving a denotation of the zero matrix to ill-typed gate applications (and thereby ill-typed programs), we do not need to explicitly assume or prove that a program is well-typed in order to state a property about its semantics, thereby removing clutter from theorems and proofs. For example, in our proof of the `ghz` program below we do not need to explicitly prove that `ghz n` is well-typed (although this is true).

Phantom Types for Matrix Indices. We do not use dependent types to represent matrices in the semantics. Following [Rand et al. \[2017, 2018a\]](#), we define matrices as functions from pairs of natural numbers to complex numbers.

Definition `Matrix (m n : N) := N → N → C.`

The arguments `m` and `n`, which are the dimensions of the matrix, are *phantom types*—they do not appear in the definition. These phantom types are useful to define certain operations on matrices that depend on these dimensions, such as the tensor product and matrix multiplication. However, there is no proof burden internal to the matrices themselves. Instead, it is possible to show a matrix is well-formed within its specified bounds by means of an external predicate:

Definition `WF_Matrix {m n} (M : Matrix m n) : P := ∀ i j, i ≥ m ∨ j ≥ n → M i j = 0.`

Phantom types occupy a convenient middle ground in allowing information to be stored in the types, while pushing the majority of the work to external predicates. For instance, we can define $|i\rangle^{\otimes n}$ (for $i \in \{0; 1\}$) recursively as $|i\rangle \otimes |i\rangle \otimes \dots \otimes |i\rangle$, with n repetitions. Coq has no way of inferring a type for this, so we declare that it has type `Vector 2^n`, and Coq will allow us to use it in any context where a vector is expected. However, before using it in rewrite rules like $|_{2^n} \times |i\rangle^{\otimes n} = |i\rangle^{\otimes n}$ (which says that multiplication by the identity matrix is an identity operation), we will need to show that $|i\rangle^{\otimes n}$ is a well-formed vector of length 2^n , for which we provide convenient automation.

Qubits are Concrete, Not Abstract, Indices. When we first set out to build `voqc`, we thought to do it using `QWIRE` [[Paykin et al. 2017](#)], another formally verified quantum programming language embedded in Coq. However, we were surprised to find that we had tremendous difficulty proving that even simple transformations were correct. This experience led to the development of `sqir`, and raised the question: Why does `sqir` seem to make proofs easier, and what do we lose by using it rather than `QWIRE`?

The fundamental difference between `sqir` and `QWIRE` is that `sqir` programs use *concrete (numeric) indices into a global register* to refer to qubits. As such, the semantics can naturally map qubits to rows and columns in the denoted matrix. In addition, qubit disjointness in a `sqir` program is obvious— $G_1 m$ operates on a different qubit than $G_2 n$ when $m \neq n$. Both elements are important for easily proving equivalences, e.g., that gates acting on disjoint qubits commute (a property that allows us to reason about gates acting on different parts of the circuit in isolation).

In `QWIRE`, variables are implemented using higher-order abstract syntax [[Pfenning and Elliott 1988](#)] and refer to *abstract* qubits. This approach eases programmability—larger circuits can be built by composing smaller ones, connecting inputs and outputs by normal variable binding, indifferent to the physical identity of a qubit. This approach is also used in the language `Quipper` [[Green et al. 2013](#)]. However, we find that this approach complicates formal proof. To denote the semantics of a program that uses abstract qubits requires deciding how abstract qubits will be represented

concretely, as rows and columns in the denotation matrix. Reasoning about this translation can be laborious, especially for recursive circuits and those that allocate and deallocate qubits (entailing de Bruijn-style index shifting [Rand 2018]). Moreover, notions like disjointness are no longer obvious— $G_1 x$ and G_2 for variables $x \neq$ may not be disjoint if x and could be allocated to the same concrete qubit.

From a proof-engineering standpoint all of the above benefits have been pivotal in allowing our proofs to scale up. Appendix B presents a detailed comparison of sqir and QWIRE, exploring the tradeoffs of concrete versus abstract qubits at a lower level.

3.4 Source-program Proofs

This paper focuses on sqir’s use in proving circuit optimizations correct, but sqir was designed to support source-program proofs too. As an illustration, we present a sqir proof of correctness for *GHZ state preparation*. We close with some discussion of ongoing efforts to prove more sophisticated algorithms correct in sqir.

```

Definition GHZ (n : N) : Vector (2 ^ n) :=
  match n with
  | 0   => I 1
  | S n' =>  $\frac{1}{\sqrt{2}}$  * |0>⊗n +  $\frac{1}{\sqrt{2}}$  * |1>⊗n
  end.

```

GHZ Proof. As an example of a proof we can carry out using sqir, we show that ghz, sqir’s Greenberger-Horne-Zeilinger (GHZ) state [Greenberger et al. 1989] preparation circuit given in Section 3.1, correctly produces the GHZ state. The GHZ state is an n -qubit entangled quantum state of the form $\frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$. This vector can be defined in Coq as shown. Like our definition of $|i\rangle^{\otimes n}$ discussed above, we declare that this expression has type $\text{Vector } 2^n$, which will allow us to use it in any context where Coq expects a vector, deferring the proof that it is well-formed.

Our goal is to show that for any $n > 0$ the circuit generated by $\text{ghz } n$ produces the corresponding GHZ n vector when applied to $|0\rangle^{\otimes n}$.

```

Lemma ghz_correct :  $\forall n : N,$ 
   $n > 0 \rightarrow \text{nghz } n 0_n \times |0\rangle^{\otimes n} = \text{GHZ } n.$ 

```

The proof proceeds by induction on n . The $n = 0$ case is trivial as it contradicts our hypothesis. For $n = 1$ we show that H applied to $|0\rangle$ produces the $|+\rangle$ state. In the inductive step, the induction hypothesis says that the result of applying $\text{ghz } n'$ to the input state $n\text{ket } n' |0\rangle$ is the state $(\frac{1}{\sqrt{2}} * |0\rangle^{\otimes n'} + \frac{1}{\sqrt{2}} * |1\rangle^{\otimes n'}) \otimes |0\rangle$. By applying CNOT $(n' - 1) n'$ to this state, we show that $\text{ghz } (n' + 1) = \text{GHZ } (n' + 1)$. Our use of concrete indices allows us to easily describe the semantics of CNOT $(n' - 1) n'$. If we had instead used abstract wires (e.g. variables x and y), then to reason about the semantics of CNOT $x y$ we would also need to reason about the conversion of x and y to concrete indices, showing that in the inductive case x refers to a qubit in the GHZ state prepared by the recursive call and y references a fresh $|0\rangle$ qubit.

Further Proofs. It turns out that with the right abstractions, sqir is capable of verifying a range of quantum algorithms, from Simon’s [1994] and Grover’s [1996] algorithms to quantum phase estimation, a key component of Shor’s factoring algorithm [1994]. All in all, the sqir development contains about 3500 lines of example proofs and programs including GHZ state preparation, superdense coding, quantum teleportation, the Deutsch-Jozsa algorithm, Simon’s algorithm, Grover’s algorithm, and quantum phase estimation. As this paper’s focus is voqc, we refer the interested reader to a separate paper [Hietala et al. 2020] for detailed discussion of these source-program proofs and proof techniques. We summarize some of the key takeaways here.

The textbook proofs of the algorithms listed above argue correctness by considering the behavior of the program on a basis vector of the form $|i_1 i_2 \dots i_n\rangle$ for $i_k \in \{0; 1\}$, or a weighted sum over such vectors. To match this style of reasoning, we developed a `sqir` framework for describing quantum states as vectors. We also found that we needed more sophisticated math lemmas when reasoning about quantum source programs. For example, though the trigonometric lemmas in Coq’s standard library are sufficient for verifying `voqc` optimizations, our proof of quantum phase estimation relies on the Coq Interval package [Melquiond 2020] to prove bounds on trigonometric functions.¹

These extensions aside, we were able to reuse much of what we developed for `voqc`. For example, we directly use the `sqir` unitary semantics presented in this section, benefiting from its various language simplifications. We also benefit from automation for matrices and complex numbers added to `QWIRE` as part of our work on `voqc`. In particular, we were able to re-purpose the `gridify` tactic we developed for proving low-level matrix equivalences (described in Section 4.5) to prove statements about the effects of different gates on vector states.

4 OPTIMIZING UNITARY SQIR PROGRAMS

This section and the next describe `voqc`, our verified optimizer for quantum circuits. `voqc` primarily implements optimizations inspired by the state-of-the-art circuit optimizer of Nam et al. [2018]. As such, we do not claim credit for the optimizations themselves. Rather, our contribution is a framework that is sufficiently flexible that it can be used to prove such state-of-the-art optimizations correct. This section focuses on `voqc`’s optimizations for unitary `sqir` programs and mapping to connectivity-constrained architectures; the next section discusses how `voqc` optimizes non-unitary `sqir` programs.

4.1 Program Equivalence

The `voqc` optimizer takes as input a `sqir` program and attempts to reduce its total gate count by applying a series of optimizations. For each optimization, we verify that it is *semantics preserving* (or *sound*), meaning that the output program is guaranteed to be equivalent to the input program.

We say that two unitary programs of dimension d are equivalent, written $U_1 \equiv U_2$, if their denotation is the same, i.e., $\llbracket U_1 \rrbracket_{0d} = \llbracket U_2 \rrbracket_{0d}$. We also support a more general version of equivalence: We say that two circuits are *equivalent up to a global phase*, written $U_1 \cong U_2$, when there exists a θ such that $\llbracket U_1 \rrbracket_{0d} = e^{i\theta} \llbracket U_2 \rrbracket_{0d}$. This is useful in the quantum setting because $|\psi\rangle$ and $e^{i\theta} |\psi\rangle$ (for $\theta \in \mathbb{R}$) represent the same physical state. Note that the latter notion of equivalence matches the former when $\theta = 0$.

Given this definition of equivalence we can write our soundness condition for optimization function `optimize` as follows.

Definition `sound` $\{G\}$ (`optimize` : $\forall \{d : \mathbb{N}\}, \text{ucom } G \ d \rightarrow \text{ucom } G \ d$) :=
 $\forall (d : \mathbb{N}) (u : \text{ucom } G \ d), \llbracket \text{optimize } u \rrbracket_{0d} \cong \llbracket u \rrbracket_{0d}$.

This property is quantified over G , d , and u , meaning that the property holds for *any program that uses any set of gates and any number of qubits*. The optimizations in our development are defined over a particular gate set, defined below, but still apply to programs that use any number of qubits. Our statements of soundness also occasionally have an additional precondition that requires program u to be well typed.

4.2 voqc Optimization Overview

`voqc` implements two basic kinds of optimizations: *replacement* and *propagate-cancel*. The former simply identifies a pattern of gates and replaces it with an equivalent pattern. The latter works by

¹Laurent Théry helpfully pointed us to Interval and provided proofs of our `sin_sublinear` and `sin_PiX_ge_2x` lemmas.

$$\begin{aligned}
X q; H q &\equiv H q; Z q \\
X q; Rz(k) q &\cong Rz(2-k) q; X q \\
X q_1; CNOT q_1 q_2 &\equiv CNOT q_1 q_2; X q_1; X q_2 \\
X q_2; CNOT q_1 q_2 &\equiv CNOT q_1 q_2; X q_2
\end{aligned}$$

Fig. 4. Equivalences used in not propagation.

commuting sets of gates when doing so produces an equivalent quantum program—often with the effect of “propagating” a particular gate rightward in the program—until two adjacent gates can be removed because they cancel out.

To ease the implementation of and proofs about these optimizations, we developed a framework of supporting library functions that operate on `sqir` programs as lists of gate applications, rather than on the native `sqir` representation. The conversion code takes a sequence of gate applications in the original `sqir` program and *flattens* it so that a program like $(G_1 p; G_2 q); G_3 r$ is represented as the Coq list $[G_1 p; G_2 q; G_3 r]$. The denotation of the list representation is the denotation of its corresponding `sqir` program. Examples of the list operations our framework provides include:

- Finding the next gate acting on a qubit that satisfies some predicate f .
- Propagating a gate using a set of cancellation and commutation rules (see Section 4.3).
- Replacing a sub-program with an equivalent program (see Section 4.4).
- Computing the maximal matching prefix of two programs.

We verify that these functions have the intended behavior (e.g., in the last example, that the returned sub-program is indeed a prefix of both input programs).

Our framework supports arbitrary gate sets (for example, the functions listed above are all parameterized by choice of gate set). However, in the optimizations described below we use a specific, universal gate set $\{H; X; Rz; CNOT\}$ where $Rz(k)$ describes rotation about the z-axis by $k \cdot \pi$ for $k \in \mathbb{Q}$. This gate set is more convenient than `sqir`’s base gate set for two reasons. First, using a discrete gate set makes it possible to define optimizations using Coq’s built-in pattern matching (with occasional equality checks between rational values). Second, using rational parameters instead of real parameters allows us to extract to OCaml rational numbers rather than floating point numbers, which would render verification unsound. Most existing tools (e.g., Qiskit [Aleksandrowicz et al. 2019] and Nam et al. [2018]) allow gates parameterized by floats, which invites rounding error and can lead to unsound optimization.

To compute a program’s denotation, `voqc`’s gates H , X , and $Rz(k)$ are translated into $R_{\pi/2;0}$, $R_{\pi;0}$, and $R_{0;0;k}$ in `sqir`’s base gate set. ($CNOT$ translates to itself.) `voqc`’s gate set is identical to Nam et al.’s, with the exception of the z-axis rotation parameter type (rational, not float).

4.3 Optimization by Propagation and Cancellation

Our *propagate-cancel* optimizations have two steps. First we localize a set of gates by repeatedly applying commutation rules. Then we apply a circuit equivalence to replace that set of gates. In `voqc`, most optimizations of this form use a library of code patterns, but one—*not propagation*—is slightly different, so we discuss it first.

Not Propagation. The goal of not propagation is to remove cancelling X (“not”) gates. Two X gates cancel when they are adjacent or they are separated by a circuit that commutes with X . We find X gates separated by commuting circuits by repeatedly applying the propagation rules in Figure 4. An example application of the not propagation algorithm is shown in Figure 5.

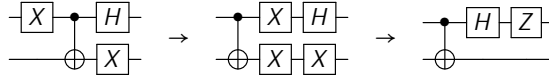


Fig. 5. An example of not propagation. In the first step the leftmost X gate propagates through the $CNOT$ gate to become two X gates. In the second step the upper X gate propagates through the H gate and the lower X gates cancel.

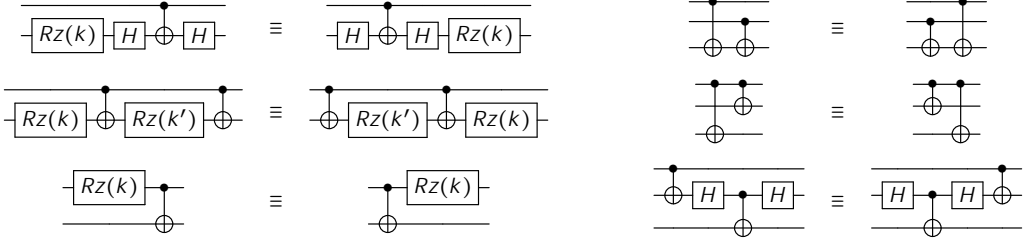


Fig. 6. Commutation equivalences for single- and two-qubit gates adapted from Nam et al. [Nam et al. 2018, Figure 5]. We use the second and third rules for propagating both single- and two-qubit gates.

This implementation may introduce extra X gates at the end of a circuit or extra Z gates in the interior of the circuit. Extra Z gates are likely to be cancelled by the gate cancellation and rotation merging passes that follow, and moving X gates to the end of a circuit makes the rotation merging optimization more likely to succeed.

We note that our version of this optimization is a simplification of Nam et al.’s, which is specialized to a three-qubit $TOFF$ gate; this gate can be decomposed into a $\{H; Rz; CNOT\}$ program. In our experiments, we did not observe any difference in performance between voqc and Nam et al. due to this simplification.

Gate Cancellation. The single- and two-qubit gate cancellation optimizations rely on the same propagate-cancel pattern used in not propagation, except that gates are returned to their original location if they fail to cancel. To support this pattern, we provide a general propagate function in voqc. This function takes as inputs (i) an instruction list, (ii) a gate to propagate, and (iii) a set of rules for commuting and cancelling that gate. At each iteration, propagate performs the following actions:

- (1) Check if a cancellation rule applies. If so, apply that rule and return the modified list.
- (2) Check if a commutation rule applies. If so, commute the gate and recursively call propagate on the remainder of the list.
- (3) Otherwise, return the gate to its original position.

We have proved that our propagate function is sound when provided with valid commutation and cancellation rules.

Each commutation or cancellation rule is implemented as a partial Coq function from an input circuit to an output circuit. A common pattern in these rules is to identify one gate (e.g., an X gate), and then to look for an adjacent gate it might commute with (e.g., $CNOT$) or cancel with (e.g., X). For commutation rules, we use the rewrite rules shown Figure 6. For cancellation rules, we use the fact that H , X , and $CNOT$ are all self-cancelling and $Rz(k)$ and $Rz(k')$ combine to become $Rz(k + k')$.

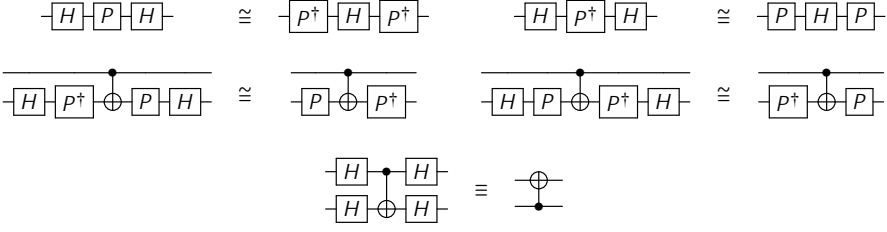


Fig. 7. Equivalences for removing Hadamard gates adapted from Nam et al. [2018, Figure 4]. P is the phase gate $Rz(1/2)$ and P^\dagger is its inverse $Rz(3/2)$.

4.4 Circuit Replacement

We have implemented two optimizations—Hadamard reduction and rotation merging—that work by replacing one pattern of gates with an equivalent one; no preliminary propagation is necessary. These aim either to reduce the gate count directly, or to set the stage for additional optimizations.

Hadamard Reduction. The Hadamard reduction routine employs the equivalences shown in Figure 7 to reduce the number of H gates in the program. Removing H gates is useful because H gates limit the size of the $\{Rz, CNOT\}$ subcircuits used in the rotation merging optimization.

Rotation Merging. The rotation merging optimization allows for combining Rz gates that are not physically adjacent in the circuit. This optimization is more sophisticated than the previous optimizations because it does not rely on small structural patterns (e.g., that adjacent X gates cancel), but rather on more general (and non-local) circuit behavior. The basic idea behind rotation merging is to (i) identify subcircuits consisting of only $CNOT$ and Rz gates and (ii) merge Rz gates within those subcircuits that are applied to qubits in the same logical state.

The argument for the correctness of this optimization relies on the *phase polynomial* representation of a circuit. Let C be a circuit consisting of $CNOT$ gates and rotations about the Z -axis. Then on basis state $|x_1; \dots; x_n\rangle$, C will produce the state

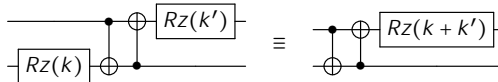
$$e^{ip(x_1; \dots; x_n)} |h(x_1; \dots; x_n)\rangle$$

where $h: \{0; 1\}^n \rightarrow \{0; 1\}^n$ is an affine reversible function and

$$p(x_1; \dots; x_n) = \sum_{i=1}^l \binom{i \bmod 2}{i} f_i(x_1; \dots; x_n)$$

is a linear combination of affine boolean functions. $p(x_1; \dots; x_n)$ is called the phase polynomial of circuit C . Each rotation gate in the circuit is associated with one term of the sum and if two terms of the phase polynomial satisfy $f_i(x_1; \dots; x_n) = f_j(x_1; \dots; x_n)$ for some $i \neq j$, then the corresponding i and j rotations can be merged.

As an example, consider the two circuits shown below.



To prove that these circuits are equivalent, we can consider their behavior on basis state $|x_1; x_2\rangle$. Recall that applying $Rz(k)$ to the basis state $|x\rangle$ produces the state $e^{ik} |x\rangle$ and $CNOT |x; \rangle$ produces the state $|x; x \oplus \rangle$ where \oplus is the xor operation. Evaluation of the left-hand circuit proceeds as follows:

$$|x_1; x_2\rangle \rightarrow e^{ik} |x_2\rangle |x_1; x_2\rangle \rightarrow e^{ik} |x_2\rangle |x_1; x_1 \oplus x_2\rangle \rightarrow e^{ik} |x_2\rangle |x_2; x_1 \oplus x_2\rangle \rightarrow e^{ik} |x_2\rangle e^{ik'} |x_2\rangle |x_2; x_1 \oplus x_2\rangle:$$

Whereas evaluation of the right-hand circuit produces

$$|x_1; x_2\rangle \rightarrow |x_1; x_1 \oplus x_2\rangle \rightarrow |x_2; x_1 \oplus x_2\rangle \rightarrow e^{i(k+k') x_2} |x_2; x_1 \oplus x_2\rangle :$$

The two resulting states are equal because $e^{ik x_2} e^{ik' x_2} = e^{i(k+k') x_2}$. This implies that the unitary matrices corresponding to the two circuits are the same. We can therefore replace the circuit on the left with the one on the right, removing one gate from the circuit.

Our rotation merging optimization follows the reasoning above for arbitrary $\{RZ; CNOT\}$ circuits. For every gate in the program, it tracks the Boolean function associated with every qubit (the Boolean functions above are $x_1, x_2, x_1 \oplus x_2$), and merges RZ rotations when they are applied to qubits associated with the same Boolean function. To prove equivalence over $\{RZ; CNOT\}$ circuits, we show that the original and optimized circuits produce the same output on every basis state. We have found evaluating behavior on basis states to be useful for proving equivalences that are not as direct as those listed in Figures 6 and 7.

Although our merge operation is identical to Nam et al.'s, our approach to constructing $\{RZ; CNOT\}$ subcircuits differs. We construct a $\{RZ; CNOT\}$ subcircuit beginning from a RZ gate whereas Nam et al. begin from a $CNOT$ gate. The result of this simplification is that we may miss some opportunities for merging. However, in our experiments (Section 6) we found that this choice impacted only one benchmark.

4.5 Proving Low-Level Circuit Equivalences

voQC optimizations make heavy use of circuit equivalences such as those shown in Figures 4, 6 and 7. To prove that voQC optimizations are sound, we must formally verify these equivalences are correct. Such proofs require showing equality between two matrix expressions, which can be tedious in the case where the matrix size is left symbolic. For example, consider the following equivalence used in *not propagation*:

$$X n; CNOT m n \equiv CNOT m n; X n$$

for arbitrary $n; m$ and dimension d . Applying our definition of equivalence, this amounts to proving

$$appl_1(X; n; d) \times appl_2(CNOT; m; n; d) = appl_2(CNOT; m; n; d) \times appl_1(X; n; d); \quad (1)$$

per the semantics in Figure 3. Suppose both sides of the equation are well typed ($m < d$ and $n < d$ and $m \neq n$), and consider the case where $m < n$ (the $n < m$ case is similar). We expand $appl_1$ and $appl_2$ as follows with $p = n - m - 1$ and $q = d - n - 1$:

$$\begin{aligned} appl_1(X; n; d) &= I_{2^n} \otimes_x I_{2^q} \\ appl_2(CNOT; m; n; d) &= I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes_x I_{2^q} + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q} \end{aligned}$$

Here, x is the matrix interpretation of the X gate and $|1\rangle\langle 1| \otimes_x + |0\rangle\langle 0| \otimes I_2$ is the matrix interpretation of the $CNOT$ gate (in Dirac notation). We can complete the proof of equivalence by normalizing and simplifying each side of Equation (1), showing both sides to be the same.

Automation. We address the tedium of such proofs in voQC by almost entirely automating the matrix normalization and simplification steps. We provide a Coq tactic called `gridify` for proving general equivalences correct. Rather than assuming $m < n < d$ as above, the `gridify` tactic does case analysis, immediately solving all cases where the circuit is ill-typed (e.g., $m = n$ or $d \leq m$) and thus has the zero matrix as its denotation. In the remaining cases ($m < n$ and $n < m$ above), it puts the expressions into a form we call *grid normal* and applies a set of matrix identities.

In grid normal form, each arithmetic expression has addition on the outside, followed by tensor product, with multiplication on the inside, i.e., $((:: \times ::) \otimes (:: \times ::)) + ((:: \times ::) \otimes (:: \times ::))$. The *gridify* tactic rewrites an expression into this form by using the following rules of matrix arithmetic:

- $I_{mn} = I_m \otimes I_n$
- $A \times (B + C) = A \times B + A \times C$
- $(A + B) \times C = A \times C + B \times C$
- $A \otimes (B + C) = A \otimes B + A \otimes C$
- $(A + B) \otimes C = A \otimes C + B \otimes C$
- $(A \otimes B) \times (C \otimes D) = (A \times C) \otimes (B \times D)$

The first rule is applied to facilitate application of the other rules. (For instance, in the example above, I_{2^n} would be replaced by $I_{2^m} \otimes I_2 \otimes I_{2^p}$ to match the structure of the *appl*₂ term.) After expressions are in grid normal form, *gridify* simplifies them by removing multiplication by the identity matrix and rewriting simple matrix products (e.g. $x \times x = I_2$).

In our example, normalization and simplification by *gridify* rewrites each side of the equality in Equation (1) to be the following

$$I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q} + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes x \otimes I_{2^q};$$

thus proving that the two expressions are equal.

We use *gridify* to verify most of the equivalences used in the optimizations given in Sections 4.3 and 4.4. The tactic is most effective when equivalences are small: The equivalences used in *gate cancellation* and *Hadamard reduction* apply to patterns of at most five gates applied to up to three qubits within an arbitrary circuit. For equivalences over large sets of qubits, like the one used in *rotation merging*, we do not use *gridify* directly, but still rely on our automation for matrix simplification.

4.6 Scheduling

The *voqc optimize* function applies each of the optimizations we have discussed one after the other, in the following order (due to Nam et al.):

$$0; 1; 3; 2; 3; 1; 2; 4; 3; 2$$

where 0 is not propagation, 1 is Hadamard reduction, 2 is single-qubit gate cancellation, 3 is two-qubit gate cancellation, and 4 is rotation merging. Nam et al. justify this ordering at length, though they do not prove that it is optimal. In brief, removing X and H gates (0,1) allows for more effective application of the gate cancellation (2,3) and rotation merging (4) optimizations. In our experiments (Section 6), we observed that single-qubit gate cancellation and rotation merging were the most effective at reducing gate count.

4.7 Circuit Mapping

We have also implemented and verified a transformation that maps a circuit to a connectivity-constrained architecture. Similar to how optimization aims to reduce qubit and gate usage to make programs more feasible to run on near-term machines, *circuit mapping* aims to address the connectivity constraints of near-term machines [Saeedi et al. 2011; Zulehner et al. 2017]. Circuit mapping algorithms take as input an arbitrary circuit and output a circuit that respects the connectivity constraints of some underlying architecture.

For example, consider the connectivity of IBMs's five-qubit Tenerife machine [IBM [n.d.]] shown in Figure 8(a). This is a representative example of a superconducting qubit system, where qubits are laid out in a 2-dimensional grid and possible interactions are described by directed edges between them. The direction of the edge indicates which qubit can be the control of a two-qubit gate and

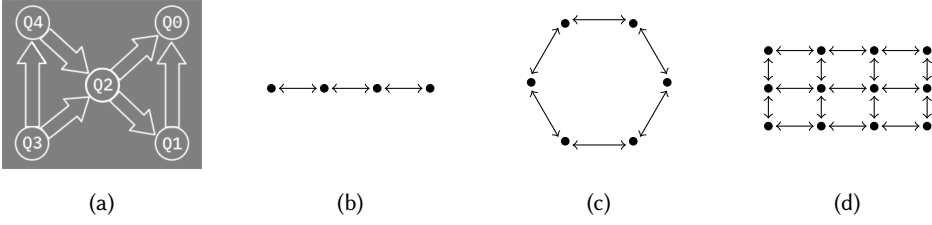


Fig. 8. Examples of two-qubit gate connections on near-term quantum machines. From left to right: IBM’s Tenerife machine [IBM [n.d.]], LNN, LNN ring, and 2D grid. The last three architectures are shown with a fixed number vertices, but in our implementation the number of vertices is a parameter. Double-ended arrows indicate that two-qubit gates are possible in both directions.

which can be the target. For instance, a *CNOT* gate may be applied with Q4 as the control and Q2 as the target, but not the reverse. No two-qubit gate is possible between physical qubits Q4 and Q1.

We have implemented a simple circuit mapper for unitary SQIR programs and verified that it is sound and produces programs that satisfy the relevant hardware constraints. Our circuit mapper is parameterized by two functions that describe the connectivity of an architecture: one function determines whether an edge is in the connectivity graph and another function finds an undirected path between any two nodes. Our mapping algorithm takes as input (i) these functions, (ii) a program referencing *logical* qubits, and (iii) a map expressing the initial correspondence between the program’s logical qubits and the *physical* qubits available on the machine. The algorithm produces a program referencing physical qubits as well as an updated correspondence. Every time a *CNOT* occurs between two logical qubits whose corresponding physical qubits are not adjacent in the underlying architecture, we insert *SWAP* operations to move the target and control into adjacent positions and update the physical-logical qubit correspondence accordingly. To apply a *CNOT* when an edge points in the wrong direction, we make use of the equivalence $\text{CNOT } b \text{ a} \equiv \text{H a}; \text{H b}; \text{CNOT a b}; \text{H a}; \text{H b}$. For soundness, we prove that the mapped circuit is equivalent to the original up to a permutation of the qubits.

Although our mapping algorithm is simple, it allows for some flexibility in design because we do not specify the method for choosing the initial physical-logical qubit correspondence (called “placement”) or the implementation of the function that finds paths in the connectivity graph (“routing”). This allows, for example, placement and routing strategies that take into account error characteristics of the machine [Tannu and Qureshi 2019]. We expect that our verification framework can be applied to more sophisticated mapping algorithms such as those that partition the circuit into layers and insert *SWAPs* between layers rather than naïvely inserting *SWAPs* before *CNOT* gates [Zulehner et al. 2017]. We have used our framework to implement and verify mapping functions for the Tenerife architecture pictured in Figure 8(a) as well as the linear nearest neighbor (LNN), LNN ring, and 2D nearest neighbor architectures pictured in Figure 8(b-d).

5 FULL SQIR: ADDING MEASUREMENT

While the bulk of voqc proofs only use the unitary core of SQIR, we also support programs with measurement. Measurement plays a key role in protocols from quantum teleportation and quantum key distribution [Bennett and Brassard 2020] to repeat-until-success loops [Paetznick and Svore 2014] and error-correcting codes [Gottesman 2010]. It also enables a number of interesting optimizations, which we discuss in Section 5.3. We begin with the syntax and semantics of full SQIR, and a proof that makes use of the non-unitary semantics.

$$\begin{aligned}
\{\text{skip}\}_d(\rho) &= \rho \\
\{P_1; P_2\}_d(\rho) &= (\{P_2\}_d \circ \{P_1\}_d)(\rho) \\
\{U\}_d(\rho) &= nU_{0_d} \times \rho \times nU_{0_d}^\dagger \\
\{\text{meas } q \ P_1 \ P_2\}_d(\rho) &= \{P_2\}_d(|0\rangle_q\langle 0| \times \rho \times |0\rangle_q\langle 0|) \\
&\quad + \{P_1\}_d(|1\rangle_q\langle 1| \times \rho \times |1\rangle_q\langle 1|)
\end{aligned}$$

Fig. 9. sqIR density matrix semantics, assuming a global register of size d .

5.1 Syntax and Semantics

To describe general quantum programs P , we extend unitary sqIR with a *branching measurement* operation.

$$P ::= \text{skip} \mid P_1; P_2 \mid U \mid \text{meas } q \ P_1 \ P_2$$

The command $\text{meas } q \ P_1 \ P_2$ (inspired by a similar construct in QPL [Selinger 2004]) measures the qubit q and either performs program P_1 or P_2 depending on the result. We define non-branching measurement and resetting a qubit to $|0\rangle$ in terms of branching measurement:

$$\begin{aligned}
\text{measure } q &= \text{meas } q \ \text{skip} \ \text{skip} \\
\text{reset } q &= \text{meas } q \ (X \ q) \ \text{skip}
\end{aligned}$$

Figure 9 defines the semantics of a non-unitary program as a function from *density matrices* to density matrices, following the approach of several previous efforts [Paykin et al. 2017; Ying 2011]. Density matrices provide a way to describe arbitrary quantum states, including *mixed states* which are probability distributions over *quantum pure states* and arise in the analysis of general quantum programs. For example, $\frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ represents a 50% chance of $|0\rangle$ and a 50% chance of $|1\rangle$.

5.2 Example: Quantum Teleportation

The goal of quantum teleportation is to transmit a state $|\psi\rangle$ from one party (Alice) to another (Bob) using a shared entangled state. The circuit for quantum teleportation is shown in Figure 10 and the corresponding sqIR program is given below.

Definition `bell` : `ucom base 3 := H 1; CNOT 1 2.`

Definition `alice` : `com base 3 := CNOT 0 1; H 0; measure 0; measure 1.`

Definition `bob` : `com base 3 := CNOT 1 2; CZ 0 2; reset 0; reset 1.`

Definition `teleport` : `com base 3 := bell; alice; bob.`

The `bell` circuit prepares a Bell pair on qubits 1 and 2, which are respectively sent to Alice and Bob. Alice applies CNOT from qubit 0 to qubit 1 and then measures both qubits and (implicitly) sends them to Bob. Finally, Bob performs operations controlled by the (now classical) values on qubits 0 and 1 and then resets them to the zero state.

The correctness property for this program says that for any (well-formed) density matrix ρ , `teleport` takes the state $\rho \otimes |0\rangle\langle 0| \otimes |0\rangle\langle 0|$ to the state $|\psi\rangle\langle \psi| \otimes |0\rangle\langle 0|$.

Lemma `teleport_correct` : $\forall (\rho : \text{Density } 2),$

$$\text{WF_Matrix } \rho \rightarrow \{\text{teleport}\}_3(\rho \otimes |0\rangle\langle 0| \otimes |0\rangle\langle 0|) = |\psi\rangle\langle \psi| \otimes |0\rangle\langle 0|$$

The proof is simple: We perform (automated) arithmetic to show that the output matrix has the desired form.

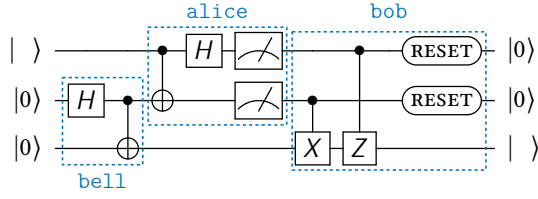


Fig. 10. Circuit for quantum teleportation. In the standard presentation Bob only acts on the last qubit, given two classical bits as input. In our presentation, Bob (equivalently) performs operations controlled by the first two qubits, which are in a post-measurement classical state. We include the reset operations to simplify our statement of correctness.

Quantum teleportation is a rare case in which we prove something about a fixed-size program (i.e., n is fixed to 3 qubits) and not one of arbitrary dimension. Using the density matrix semantics to prove properties about programs of arbitrary dimension n is more involved. Stating the property requires introducing a symbolic density matrix, which will be multiplied on the left and right by $2^n \times 2^n$ matrices in the denotation. In our experience this results in complicated proof terms that are tedious to manipulate, even with automation. By contrast, when reasoning about the equivalence of two unitary programs we can simply compare their unitary matrices, without carrying around a symbolic (or input vector).

5.3 Non-unitary Optimizations

We have implemented two verified optimizations of non-unitary programs in voqc, inspired by optimizations in IBM’s Qiskit compiler [Aleksandrowicz et al. 2019]: removing pre-measurement Z-rotations, and classical state propagation. For these optimizations, we represent a non-unitary program P as a list of *blocks*. A block is a binary tree whose leaves are unitary programs (in list form) and nodes are measurements $\text{meas } q P_1 P_2$ whose children P_1 and P_2 are lists of blocks. Since the density matrix semantics denotes programs as functions over matrices, we say that programs P_1 and P_2 of dimension d are equivalent if for every input ρ , $\{P_1\}_d(\rho) = \{P_2\}_d(\rho)$.

Z-rotations Before Measurement. Z-axis rotations (or, more generally, diagonal unitary operations) before a measurement will have no effect on the measurement outcome, so they can safely be removed from the program. This optimization locates RZ gates before measurement operations and removes them. It was inspired by Qiskit’s `RemoveDiagonalGatesBeforeMeasure` pass.

Classical State Propagation. Once a qubit has been measured, the subsequent branch taken provides information about the qubit’s (now classical) state, which may allow pre-computation of some values. For example, in the branch where qubit q has been measured to be in the $|0\rangle$ state, any $CNOT$ with q as the control will be a no-op and any subsequent measurements of q will still produce zero.

In detail, given a qubit q in classical state $|i\rangle$, our optimization applies these propagation rules:

- $Rz(k)$ q preserves the classical state of q .
- X q flips the classical state of q .
- If $i = 0$ then $CNOT$ q q' is removed, and if $i = 1$ then $CNOT$ q q' becomes X q' .
- $\text{meas } q P_1 P_0$ becomes P_i .
- H q and $CNOT$ q' q make q non-classical and terminate analysis.

Our statement of correctness for one round of propagation says that if qubit q is in a classical state in the input, then the optimized program will have the same denotation as the unoptimized original. We express the requirement that qubit q be in classical state $i \in \{0; 1\}$ with the condition

$|i\rangle_q \langle i| \times \dots \times |i\rangle_q \langle i| = \dots$; which says that projecting state onto the subspace where q is in state $|i\rangle$ results in no loss of information.

This optimization is not implemented directly in Qiskit, but Qiskit contains passes that have a similar effect. For example, the `RemoveResetInZeroState` pass removes adjacent reset gates, as the second has no effect.

6 EXPERIMENTAL EVALUATION

The value of voqc (and sqir) is determined by the quality of the verified optimizations we can write with it. We can judge optimization quality empirically. In particular, we can run voqc on a benchmark of circuit programs and see how well it optimizes those programs, compared to (non-verified) state-of-the-art compilers.

To this end, we compared the performance of voqc’s verified (unitary) optimizations against IBM’s Qiskit compiler [Aleksandrowicz et al. 2019], CQC’s t|ket) [Cambridge Quantum Computing Ltd 2019; Sivarajah et al. 2020], PyZX [Kissinger and van de Wetering 2020], and the optimizers presented by Nam et al. [2018] and Amy et al. [2013] on a set of benchmarks developed by Amy et al. We find that voqc has comparable performance to all of these: it generally beats all but Nam et al. in terms of both total gate count reduction and T -gate reduction, and often matches Nam et al. However, our aim is not to claim superiority over these tools (after all, we have implemented a subset of the unitary optimizations available in Nam et al., and Qiskit and t|ket) contain many features that Nam et al. does not), but to demonstrate that the optimizations we have implemented are on par with existing *unverified* tools.

Benchmarks. We evaluated performance by applying voqc and the other compilers to the benchmark of Amy et al. [2013], which consists of programs written in the “Clifford+T” gate set ($CNOT$, H , S and T , where S and T are Z -axis rotations by $\pi/2$ and $\pi/4$, respectively). The benchmark programs contain arithmetic circuits, implementations of multiple-control X gates, and Galois field multiplier circuits, ranging from 45 to 13,593 gates and 5 to 96 qubits. All of these circuits are unitary, so they only serve to evaluate our unitary circuit optimizations. We do not evaluate circuit mapping transformations.

We measure the reduction in total gate count and T -gate count. Total gate count is a useful metric for near-term quantum computing, where the length of the computation must be minimized to reduce error. T -gate count is relevant in the *fault-tolerant regime* where qubits are encoded using quantum error correcting codes and operations are performed fault-tolerantly. In this regime the standard method for making Clifford+T circuits fault tolerant produces particularly expensive translations for T gates, so reducing T -count is a common optimization goal. The Clifford+T set is a subset of voqc’s gate set where each Z -axis rotation is restricted to be a multiple of $\pi/4$ (an odd multiple of $\pi/4$ corresponds to one T gate).

Some of the benchmarks contain doubly-controlled Z , or CCZ , gates. Before applying optimizations we convert CCZ gates to voqc’s gate set using the following standard decomposition, where T is the $Rz(1/4)$ gate and T^\dagger is its inverse $Rz(7/4)$.

$$CCZ \text{ a b c } := [CNOT \text{ b c } ; T^\dagger \text{ c } ; CNOT \text{ a c } ; T \text{ c } ; CNOT \text{ b c } ; T^\dagger \text{ c } ; CNOT \text{ a c } ; CNOT \text{ a b } ; T^\dagger \text{ b } ; CNOT \text{ a b } ; T \text{ a } ; T \text{ b } ; T \text{ c }].$$

We also evaluated the performance of voqc on the benchmark used by Nam et al., of which Amy et al. is a subset. Nam et al.’s benchmark includes additional unitary circuits for simulating Hamiltonian dynamics as well as quantum Fourier transform and adder circuits, which are subroutines in Shor’s integer factoring algorithm [Shor 1994]. voqc’s results on Amy et al.’s benchmark are representative of voqc’s behavior on the full set; details are given in Appendix A.

Table 1. Summary of unitary optimizations for reducing total gate count.

<u>Nam et al.</u>	
Not propagation (P)	✓*
Hadamard gate reduction (L, H)	✓
Single-qubit gate cancellation (L, H)	✓
Two-qubit gate cancellation (L, H)	✓
Rotation merging using phase polynomials (L)	✓*
Floating R_z gates (H)	
Special-purpose optimizations (L, H)	
• LCR optimizer	✓
• Toffoli decomposition	
<hr/>	
<u>Qiskit Terra 0.15.2</u>	
CXCancellation (L_1)	✓*
Optimize1qGates (L_1, L_2, L_3)	✓*
CommutativeCancellation (L_2, L_3)	✓*
ConsolidateBlocks (L_3)	
<hr/>	
<u>t ket) 0.6.0</u>	
FullPeepholeOptimise	✓*

Baseline: Total Gate Count. To evaluate reduction in total gate count, we compare voqc’s performance with that of Nam et al., Qiskit Terra version 0.15.2 (release date September 8, 2020), and t|ket) version 0.6.0 (September 18, 2020). We do not include the results from Amy et al. or PyZX because their optimizations are aimed at reducing T -count, and often result in a higher total gate count. Table 1 performs a direct comparison of functionality provided by voqc versus Nam et al., Qiskit, and t|ket). For the Qiskit optimizations, L_i indicates that a routine is used by optimization level i . For Nam et al., P stands for “preprocessing” and L and H indicate whether the routine is in the “light” or “heavy” versions of the optimizer. voqc provides the complete and verified functionality of the routines marked with ✓; we write ✓* to indicate that voqc contains a verified optimization with similar, although not identical, behavior.

Compared to Nam et al.’s rotation merging, voqc performs a slightly less powerful optimization (as discussed in Section 4.4). Conversely, voqc’s one- and two-qubit gate cancellation routines generalize Qiskit’s Optimize1qGates and CXCancellation when restricted to voqc’s gate set. For CommutativeCancellation, Qiskit’s routine follows the same pattern as our gate cancellation routines, but uses matrix multiplication to determine whether gates commute while we use a rule-based approach; neither is strictly more effective than the other. t|ket)’s FullPeepholeOptimise performs local rewrites similar to those applied by Qiskit.

When evaluating Qiskit, we include all unitary optimizations up to level 3. In our evaluation, both Qiskit and t|ket) use the gate set $\{u_1; u_2; u_3; CNOT\}$ where u_3 is R_z ; from voqc’s base set and u_1 and u_2 are u_3 with certain arguments fixed. This gate set gives these industrial compilers an advantage over voqc because they can, for example, represent X followed by H with a single gate.

Baseline: T -Gate Count. To evaluate reduction in T -gate count, we compare voqc against Nam et al., Amy et al., and PyZX version 0.6.0 (release date June 16, 2020). We do not include results from Qiskit or t|ket) because these compilers produce circuits that do not use the Clifford+T set. When evaluating PyZX, we use the full_reduce method, which applies an optimization similar in intent to rotation merging, but implemented in terms of the ZX-calculus.

Table 2. Reduced total gate counts on Amy et al. [2013] benchmarks. Red cells indicate programs optimized incorrectly. Bold results mark the best performing optimizer.

Name	Total Gate Count					
	Original	Qiskit	t ket)	Nam (L)	Nam (H)	VOQC
adder_8	900	805	775	646	606	682
barenco_tof_3	58	51	51	42	40	50
barenco_tof_4	114	100	100	78	72	95
barenco_tof_5	170	149	149	114	104	140
barenco_tof_10	450	394	394	294	264	365
csla_mux_3	170	156	155	161	155	158
csum_mux_9	420	382	361	294	266	308
gf2^4_mult	225	206	206	187	187	192
gf2^5_mult	347	318	319	296	296	291
gf2^6_mult	495	454	454	403	403	410
gf2^7_mult	669	614	614	555	555	549
gf2^8_mult	883	804	806	712	712	705
gf2^9_mult	1095	1006	1009	891	891	885
gf2^10_mult	1347	1238	1240	1070	1070	1084
gf2^16_mult	3435	3148	3150	2707	2707	2695
gf2^32_mult	13593	12506	12507	10601	10601	10577
mod5_4	63	58	58	51	51	56
mod_mult_55	119	106	102	91	91	90
mod_red_21	278	227	224	184	180	214
qcla_adder_10	521	469	460	411	399	438
qcla_com_7	443	398	392	284	284	314
qcla_mod_7	884	793	780	636	624	723
rc_adder_6	200	170	172	142	140	157
tof_3	45	40	40	35	35	40
tof_4	75	66	66	55	55	65
tof_5	105	92	92	75	75	90
tof_10	255	222	222	175	175	215
vbe_adder_3	150	138	139	89	89	101
Geo. Mean Reduction	–	10.1%	10.6%	23.3%	24.8%	17.8%

Results. The results are shown in Table 2 and Table 3. In each row, we have marked in bold the gate count of the best-performing optimizer. The geometric mean of the reduction in each benchmarks is given in the last row. Shaded cells mark that the resulting optimized circuit has been found to be inequivalent to the original circuit, indicating a bug in the relevant optimizer.² Results for incorrectly-optimized circuits are not included in the averages on the last line. We do not re-run Nam et al. (which is proprietary software) or Amy et al. We report results from Nam et al. [2018].

On average, Qiskit reduces the total gate count by 10.1%, t|ket) by 10.6%, Nam et al. by 23.3% (light) and 24.8% (heavy), and voqc by 17.8%. voqc outperforms or matches the performance of Qiskit and t|ket) on all benchmarks but one. In 8 out of 28 cases voqc outperforms Nam et al. The gap in performance between voqc and the industrial compilers is due to voqc’s rotation merging

²The bug in csla_mux_3 was found by Nam et al. [2018] and the bug in qcla_com_7 was found by Kissinger and van de Wetering [2019]; both were discovered using translation validation.

Table 3. Reduced T -gate counts on the Amy et al. [2013] benchmarks. Red cells indicate programs optimized incorrectly. Bold results mark the best performing optimizer.

Name	T -Gate Count					
	Original	Amy	PyZX	Nam (L)	Nam (H)	VOQC
adder_8	399	215	173	215	215	215
barenco_tof_3	28	16	16	16	16	16
barenco_tof_4	56	28	28	28	28	28
barenco_tof_5	84	40	40	40	40	40
barenco_tof_10	224	100	100	100	100	100
csla_mux_3	70	62	62	64	64	64
csum_mux_9	196	112	84	84	84	84
gf2^4_mult	112	68	68	68	68	68
gf2^5_mult	175	111	115	115	115	115
gf2^6_mult	252	150	150	150	150	150
gf2^7_mult	343	217	217	217	217	217
gf2^8_mult	448	264	264	264	264	264
gf2^9_mult	567	351	351	351	351	351
gf2^10_mult	700	410	410	410	410	410
gf2^16_mult	1792	1040	1040	1040	1040	1040
gf2^32_mult	7168	4128	4128	4128	4128	4128
mod5_4	28	16	8	16	16	16
mod_mult_55	49	37	35	35	35	35
mod_red_21	119	73	73	73	73	73
qcla_adder_10	238	162	162	162	162	164
qcla_com_7	203	95	95	95	95	95
qcla_mod_7	413	249	237	237	235	249
rc_adder_6	77	63	47	47	47	47
tof_3	21	15	15	15	15	15
tof_4	35	23	23	23	23	23
tof_5	49	31	31	31	31	31
tof_10	119	71	71	71	71	71
vbe_adder_3	70	24	24	24	24	24
Geo. Mean Reduction	–	39.7%	42.6%	41.4%	41.4%	41.4%

optimization, which has no analogue in Qiskit or t|ket>). The gap in performance between Nam et al. and voqc is due to the fact that we have not yet implemented all their optimization passes (per Table 1). In particular, Nam et al.’s “special-purpose Toffoli decomposition” (which affects how CCZ gates are decomposed) enables rotation merging and single-qubit gate cancellation to cancel two gates (e.g. cancel T and T^\dagger) where we instead combine two gates into one (e.g. T and T becomes P). Interestingly, the cases where voqc outperforms Nam et al. can also be attributed to their Toffoli decomposition heuristics, which sometimes result in fewer cancellations than the naïve decomposition that we use. We do not expect adding and verifying this form of Toffoli decomposition to pose a challenge in voqc.

voqc’s performance is closer to Nam et al.’s when considering T -count. On average, Amy et al. reduce the T -gate count by 39.7%, PyZX by 42.6%, and Nam et al. and voqc by 41.4%. voqc matches Nam et al. on all benchmarks but two. The first case (qcla_adder_10) is due to our simplification in rotation merging. In the second case (qcla_mod_7), Nam et al.’s optimized circuit was later found to

be inequivalent to the original circuit [Kissinger and van de Wetering 2019], so the lower T -count is spurious. On 16 benchmarks, all optimizers produce the same T -count. This is somewhat surprising since, although all these optimizers rely on some form of rotation merging, their implementations differ substantially. Kissinger and van de Wetering [2019] posit that these results indicate a local optimum in the ancilla-free case for some of the benchmarks (in particular the tof benchmarks, whose T -count is not reduced by applying additional techniques [Heyfron and T. Campbell 2018]).

To compare the running times of the different tools, we ran 11 trials of voqc, Qiskit, t|ket), and PyZX (taking the median time for each benchmark) on a standard laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Catalina. We consider the timings for Amy et al. and Nam et al. given in Nam et al. [2018, Table 4], which were measured on a similar machine with 8 GB RAM running OS X El Capitan. We show the geometric mean running times over all 28 benchmarks below.

voqc	Nam (L)	Nam (H)	Qiskit	t ket)	Amy	PyZX
0.013s	0.002s	0.018s	0.812s	0.129s	0.007s	0.384s

All the tools are fast; Nam et al. light optimization tends to be the fastest and Qiskit tends to be the slowest. However, these means are not the entire story: the tools' performances scale differently with increasing qubit and gate count. For example, on `gf2^32_mult` (the largest benchmark) Qiskit and voqc are comparable with running times of 31.6s and 27.4s respectively; Nam et al. light optimization and t|ket) are very fast with running times of 1.8s and 7.0s; and Nam et al. heavy optimization, Amy et al., and PyZX are fairly slow with running times of 275.7s, 602.6s, and 577.1s. For more details on voqc's performance, see Appendix A.

These results are encouraging evidence that voqc supports useful and interesting verified optimizations, and that we have faithfully implemented Nam et al.'s optimizations. Furthermore, despite having been written with verification in mind, voqc's running times are not significantly worse than (and sometimes better than) that of current tools.

Trusted Code. For performance, voqc uses OCaml primitives for describing rational numbers, maps and sets, rather than the code extracted from Coq. Thus we implicitly trust that the OCaml implementation of these data types is consistent with Coq's; we believe that this is a reasonable assumption. Furthermore, our translation from OpenQASM to sqir and extraction from Coq to OCaml are not formally verified.

7 RELATED WORK

Our work on voqc and sqir is primarily related to work on quantum program compilation, especially work aiming to add assurance to the compilation process. It is also related to work on quantum source-program verification.

Verified Quantum Compilation. Quantum compilation is an active area. In addition to Qiskit, t|ket), and Nam et al. [2018] (discussed in Section 6), other recent compiler efforts include quilc [Rigetti Computing 2019b], Scaffold [Javadi-Abhari et al. 2014], and Project Q [Steiger et al. 2018]. Due to resource limits on near-term quantum computers, most compilers for quantum programs contain some degree of optimization, and nearly all place an emphasis on satisfying architectural requirements, like mapping to a particular gate set or qubit topology. None of the optimization or mapping code in these compilers is formally verified.

However, voqc is not the only quantum compiler to which automated reasoning or formal verification has been applied. Amy et al. [2017] developed a certified optimizing compiler from source Boolean expressions to reversible circuits, but did not handle general quantum programs.

Rand et al. [2018b] developed a similar compiler for quantum circuits but without optimizations (using the *QWIRE* language).

The problem of optimization verification has also been considered in the context of the ZX-calculus [Coecke and Duncan 2009], which is a formalism for describing quantum tensor networks (which generalize quantum circuits) based on categorical quantum mechanics [Abramsky and Coecke 2009]. The ZX-calculus is characterized by a small set of rewrite rules that allow translation of a diagram to any other diagram representing the same computation [Jeandel et al. 2018]. Fagan and Duncan [2018] verified an optimizer for ZX diagrams representing Clifford circuits (which use the non-universal gate set $\{CNOT; H; S\}$) in the Quantomatic graphical proof assistant [Kissinger and Zamdzhiev 2015]. PyZX [Kissinger and van de Wetering 2020] uses ZX diagrams as an intermediate representation for compiling quantum circuits, and generally achieves performance comparable to leading compilers [Kissinger and van de Wetering 2019]. While PyZX is not verified in a proof assistant like Coq (the “Py” stands for Python), it does rely on a small, well-studied equational theory. Additionally, PyZX can perform translation validation to check if a compiled circuit is equivalent to the original. However, PyZX’s translation validator is not guaranteed to succeed for any two equivalent circuits.

A recent paper from Smith and Thornton [2019] presents a compiler with built-in translation validation via QMDD equivalence checking [Miller and Thornton 2006]. However the optimizations they consider are much simpler than voqc’s and the QMDD approach scales poorly with increasing number of qubits. Our optimizations are all verified for arbitrary dimension.

Concurrently with our work, Shi et al. [2019] developed CertiQ, an approach to verifying properties of circuit transformations in the Qiskit compiler, which is implemented in Python. Their approach has two steps. First, it uses matrix multiplication to check that the unitary semantics of two concrete gate patterns are equivalent. Second, it uses symbolic execution to generate verification conditions for parts of Qiskit that manipulate circuits. These are given to an SMT solver to verify that pattern equivalences are applied correctly according to programmer-provided function specifications and invariants. That CertiQ can analyze Python code directly in a mostly automated fashion is appealing. However, it is limited in the optimizations it can verify. For example, equivalences that range over arbitrary indices, like $CNOT\ m\ x; CNOT\ n\ x \equiv CNOT\ n\ x; CNOT\ m\ x$ cannot be verified by matrix multiplication; CertiQ checks a concrete instance of this pattern and then applies it to more general circuits. More complex optimizations like rotation merging (the most powerful optimization in our experiments) cannot be generalized from simple, concrete circuits. CertiQ may also fail to prove an optimization correct, e.g., because of complicated control code; in this case it falls back to translation validation, which adds extra cost and the possibility of failure at run-time. By contrast, every optimization in voqc has been proved correct. Finally, CertiQ does not directly represent the semantics of quantum programs, so it cannot be used as a tool for verifying general properties of a program’s semantics (as we do in Section 3).

Verified Quantum Programming. We designed sqir primarily as the intermediate language for voqc’s verified optimizations, but it can be used for verified source programming as well, per Section 3 and ongoing work [Hietala et al. 2020]. Early attempts to formally verify aspects of a quantum computation in a proof assistant were an Agda implementation of the Quantum IO Monad [Green 2010] and a small Coq quantum library by Boender et al. [2015]. Later, Rand et al. [2017] embedded the higher-level *QWIRE* programming language in the Coq proof assistant, and used it to verify a variety of simple programs [Rand et al. 2017], assertions regarding ancilla qubits [Rand et al. 2018b], and its own metatheory [Rand 2018]. voqc and sqir reuse parts of *QWIRE*’s Coq development, and take inspiration and lessons from its design. However, as discussed in Section 3.3

and Appendix B, \mathcal{Q} WIRE’s higher-level abstractions (notably, its representation of variables using higher-order abstract syntax) complicate verification.

Concurrently with this work, Chareton et al. [2020] introduced \mathcal{Q} BRICKS, a tool implemented in Why3 [Filliâtre and Paskevich 2013] whose aim is to support mostly-automated verification of complex quantum algorithms. Their design in many ways mirrors \mathcal{Q} SQR’s: both tools provide special support for reasoning about quantum programs and the languages are simplified so that programs have a straightforward translation to their semantics. A \mathcal{Q} BRICKS program specifies a circuit without variables, instead using operators for parallel and sequential composition. \mathcal{Q} BRICKS defines the meaning of its programs using a “higher order” path-sum semantics [Amy 2018], which limits it to unitary programs, but substantially enhances automation. \mathcal{Q} SQR and \mathcal{Q} BRICKS have been used to verify similar algorithms (e.g., Grover’s algorithm and quantum phase estimation [Hietala et al. 2020]), but \mathcal{Q} BRICKS’ approach reduces manual effort.

8 CONCLUSIONS AND FUTURE WORK

This paper has presented \mathcal{V} OQC, the first fully verified optimizer for quantum circuits. A key component of \mathcal{V} OQC is \mathcal{Q} SQR, a simple, low-level quantum language deeply embedded in the the Coq proof assistant, which gives a semantics to quantum programs that is amenable to proof. Optimization passes are expressed as Coq functions which are proved to preserve the semantics of their input \mathcal{Q} SQR programs. \mathcal{V} OQC’s optimizations are mostly based on local circuit equivalences, implemented by replacing one pattern of gates with another, or commuting a gate rightward until it can be cancelled. Others, like rotation merging, are more complex. These were inspired by, and in some cases generalize, optimizations in industrial compilers, but in \mathcal{V} OQC are proved correct. When applied to a benchmark suite of 28 circuit programs, we found \mathcal{V} OQC performed comparably to state-of-the-art compilers, reducing gate count on average by 17.8% compared to 10.1% for IBM’s Qiskit compiler, 10.6% for CQC’s t|ket), and 24.8% for the cutting-edge research optimizer by Nam et al. [2018]. Furthermore, \mathcal{V} OQC reduced T -gate count on average by 41.4% compared to 39.7% by Amy et al. [2013], 41.4% by Nam et al., and 42.6% by the PyZX optimizer.

Moving forward, we plan to incorporate \mathcal{V} OQC into a full-featured verified compilation stack for quantum programs, following the vision of a recent Computing Community Consortium report [Martonosi and Roetteler 2019]. We can verify compilation from high-level languages with formal semantics like Silq [Bichsel et al. 2020] to \mathcal{Q} SQR circuits. We can implement validated parsers [Jourdan et al. 2012] for languages like OpenQASM and verify their translation to \mathcal{Q} SQR (e.g., using metaQASM’s semantics [Amy 2019]); this work is already in progress [Singhal et al. 2020]. We can also add support for hardware-specific transformations that compile to a particular gate set. Indeed, most of the sophisticated code in Qiskit is devoted to efficiently mapping programs to IBM’s architecture, and IBM’s 2018 Developer Challenge centered around designing new circuit mapping algorithms [IBM Research Editorial Staff 2018]. We leave it as future work to incorporate optimizations and mapping algorithms from additional compilers into \mathcal{V} OQC. Our experience so far makes us optimistic about the prospects for doing so successfully.

ACKNOWLEDGMENTS

We thank Leonidas Lampropoulos, Kartik Singhal, and anonymous reviewers for their helpful comments on drafts of this paper. This material is based upon work supported 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.

REFERENCES

- Samson Abramsky and Bob Coecke. 2009. Categorical quantum mechanics. *Handbook of quantum logic and quantum structures 2* (2009), 261–325.
- Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Lukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martin-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O’Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyranov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An open-source framework for quantum computing. <https://doi.org/10.5281/zenodo.2562110>
- Thorsten Altenkirch and Alexander S Green. 2010. The quantum IO monad. *Semantic Techniques in Quantum Computation* (2010), 173–205.
- Matthew Amy. 2018. Towards large-scale functional verification of universal quantum circuits. https://www.mathstat.dal.ca/qpl2018/papers/QPL_2018_paper_30.pdf. Presented at QPL 2018.
- Matthew Amy. 2019. Sized types for low-level quantum metaprogramming. In *Reversible Computation*, Michael Kirkedal Thomsen and Mathias Soeken (Eds.). Springer International Publishing, Cham, 87–107. https://doi.org/10.1007/978-3-030-21500-2_6
- Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. 2018. On the controlled-NOT complexity of controlled-NOT-phase circuits. *Quantum Science and Technology* 4, 1 (2018).
- Matthew Amy, Dmitri Maslov, and Michele Mosca. 2013. Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33 (03 2013). <https://doi.org/10.1109/TCAD.2014.2341953>
- Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified compilation of space-efficient reversible circuits. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer. <https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/>
- Charles H Bennett and Gilles Brassard. 2020. Quantum cryptography: Public key distribution and coin tossing. *arXiv preprint arXiv:2003.06557* (2020).
- Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3385412.3386007>
- Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. 2015. Formalization of quantum protocols using Coq. In *Proceedings of the 12th International Workshop on Quantum Physics and Logic, Oxford, U.K., July 15-17, 2015 (Electronic Proceedings in Theoretical Computer Science, Vol. 195)*, Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.). Open Publishing Association, 71–83. <https://doi.org/10.4204/EPTCS.195.6>
- Cambridge Quantum Computing Ltd. 2019. pytket. <https://cqcl.github.io/pytket/build/html/index.html>
- Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. 2020. Toward certified quantum programming. *arXiv e-prints* (2020). arXiv:2003.05841 [cs.PL]
- Bob Coecke and Ross Duncan. 2009. Interacting quantum observables: Categorical algebra and diagrammatics. *New Journal of Physics* 13 (06 2009). <https://doi.org/10.1088/1367-2630/13/4/043016>
- The Coq Development Team. 2019. The Coq proof assistant, version 8.10.0. <https://doi.org/10.5281/zenodo.3476303>
- Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. *arXiv e-prints* (Jul 2017). arXiv:1707.03429 [quant-ph]
- Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392. [https://doi.org/10.1016/1385-7258\(72\)90034-0](https://doi.org/10.1016/1385-7258(72)90034-0)
- Andrew Fagan and Ross Duncan. 2018. 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*. <https://doi.org/10.4204/EPTCS.287.5>

- Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming (Lecture Notes in Computer Science)*.
- Daniel Gottesman. 2010. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, Vol. 68. 13–58.
- Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 333–342. <https://doi.org/10.1145/2491956.2462177>
- Alexander S Green. 2010. *Towards a formally verified functional quantum programming language*. Ph.D. Dissertation. University of Nottingham.
- Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going beyond Bell's Theorem*. Springer Netherlands, Dordrecht, 69–72. https://doi.org/10.1007/978-94-017-0849-4_10
- Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*. 212–219.
- Luke Heyfron and Earl T. Campbell. 2018. An efficient quantum compiler that reduces T count. *Quantum Science and Technology* 4 (2018). <https://doi.org/10.1088/2058-9565/aad604>
- Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2020. Proving Quantum Programs Correct. arXiv:2010.01240 [cs.PL]
- IBM. [n.d.]. IBM Q5 Tenerife V1.x.x version log. https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md
- IBM Research Editorial Staff. 2018. We have winners! ... of the IBM Qiskit developer challenge. <https://www.ibm.com/blogs/research/2018/08/winners-qiskit-developer-challenge/>
- Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Hecke, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (Cagliari, Italy) (CF '14)*. ACM, New York, NY, USA, Article 1, 10 pages. <https://doi.org/10.1145/2597917.2597939>
- Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. 2018. A complete axiomatisation of the ZX-calculus for Clifford+ T quantum mechanics. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 559–568. <https://doi.org/10.1145/3209108.3209131>
- Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) parsers. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416.
- Aleks Kissinger and John van de Wetering. 2019. Reducing T-count with the ZX-calculus. *arXiv e-prints* (2019). arXiv:1903.10477 [quant-ph]
- Aleks Kissinger and John van de Wetering. 2020. PyZX: Large scale automated diagrammatic reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (04 2020), 230–242. <https://doi.org/10.4204/EPTCS.318.14>
- Aleks Kissinger and Vladimir Zamdzhiev. 2015. Quantomatic: A proof assistant for diagrammatic reasoning. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 326–336.
- Emmanuel Knill. 1996. *Conventions for quantum pseudocode*. Technical Report. Los Alamos National Lab., NM (United States).
- Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. <https://doi.org/10/c9sb7q>
- Margaret Martonosi and Martin Roetteler. 2019. Next steps in quantum computing: computer science's role. arXiv:1903.10541 [cs.ET]
- Guillaume Melquiond. 2020. Interval package for Coq. <https://gitlab.inria.fr/coqinterval/interval>
- D. M. Miller and M. A. Thornton. 2006. QMDD: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*. 30–30. <https://doi.org/10.1109/ISMVL.2006.35>
- Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (2018), 23. <https://doi.org/10.1038/s41534-018-0072-4>
- Adam Paetznick and Krysta M Svore. 2014. Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. *Quantum Information & Computation* 14, 15-16 (2014), 1277–1301.
- Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (Paris, France) (POPL 2017)*. ACM, New York, NY, USA, 846–858. <https://doi.org/10.1145/3009837.3009894>
- Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia, USA) (PLDI '88)*. ACM, New York, NY, USA, 199–208. <https://doi.org/10.1145/53990.54010>
- John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. <https://doi.org/10.22331/q-2018-08-06-79>

- Robert Rand. 2018. *Formally verified quantum programming*. Ph.D. Dissertation. University of Pennsylvania.
- Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2018b. 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*. <https://doi.org/10.4204/EPTCS.287.17>
- Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. 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*. 119–132. <https://doi.org/10.4204/EPTCS.266.8>
- Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018a. Phantom types for quantum programs. The Fourth International Workshop on Coq for Programming Languages.
- Rigetti Computing. 2019a. Pyquil documentation. <http://pyquil.readthedocs.io/en/latest/>
- Rigetti Computing. 2019b. The @rigetti optimizing Quil compiler. <https://github.com/rigetti/quilc>
- Mehdi Saeedi, Robert Wille, and Rolf Drechsler. 2011. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing* 10, 3 (01 Jun 2011), 355–377. <https://doi.org/10.1007/s11128-010-0201-2>
- Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (Aug. 2004), 527–586. <https://doi.org/10.1017/S0960129504004256>
- Yunong Shi, Xupeng Li, Runzhou Tao, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2019. Contract-based verification of a realistic quantum compiler. *arXiv e-prints* (Aug 2019). arXiv:1908.08963 [quant-ph]
- P. W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*.
- DR Simon. 1994. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. 116–123.
- Kartik Singhal, Robert Rand, and Michael Hicks. 2020. Verified translation between low-level quantum languages. The First International Workshop on Programming Languages for Quantum Computing.
- Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket> : A retargetable compiler for NISQ Devices. *arXiv e-prints* (2020). arXiv:2003.10611 [quant-ph]
- Kaitlin N. Smith and Mitchell A. Thornton. 2019. A quantum computational compiler and design tool for technology-specific targets. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. <https://doi.org/10.1145/3307650.3322262>
- Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A practical quantum instruction set architecture. *arXiv e-prints* (Aug 2016). arXiv:1608.03355 [quant-ph]
- Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *Quantum* 2 (2018), 49. <https://doi.org/10.22331/q-2018-01-31-49>
- Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 7. <https://doi.org/10.1145/3183895.3183901>
- Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. <https://doi.org/10.1145/3297858.3304007>
- The Cirq Developers. 2019. Cirq: A python library for NISQ circuits. <https://cirq.readthedocs.io/en/stable/>
- Mingsheng Ying. 2011. Floyd-hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2011), 19. <https://doi.org/10.1145/2049706.2049708>
- Vladimir Zamdzhiev. 2016. Quantum computing: The good, the bad, and the (not so) ugly! Invited talk, Tulane University.
- Alwin Zulehner, Alexandru Paler, and Robert Wille. 2017. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *arXiv e-prints* (Dec 2017). arXiv:1712.04722 [quant-ph]

A ADDITIONAL BENCHMARK RESULTS

In this section, we evaluate voqc’s performance on all 99 benchmark programs considered by Nam et al. [2018], confirming our conclusion from Section 6 that voqc is a faithful implementation of a subset of the optimizations present in Nam et al. (along with being proved correct!). The benchmarks are divided into three categories, as described below. Our versions of the benchmarks are available online.³ All results were obtained using a laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Catalina. For timings, we take the median of three trials. Nam et al.’s results are from a similar machine with 8 GB RAM running OS X El Capitan. Their implementation is written in Fortran.

Overall, the results are consistent with those presented in Section 6. In cases where Toffoli decomposition and heavy optimization are not used (the QFT, QFT-based adder, and product formula circuits), voqc’s results are identical to Nam et al.’s. In the other cases, voqc is slightly less effective than Nam et al. for the reasons discussed in Section 6. In the worst case, voqc’s run time is four orders of magnitude worse than Nam et al.’s. However, voqc’s run time is often less than a second. We view this performance as acceptable, given that benchmarks with more than 1000 two-qubit gates (the only programs for which voqc optimization takes longer than one second) are well out of reach of current quantum hardware [Preskill 2018]. We are optimistic that voqc’s performance can be improved through more careful engineering.

Arithmetic and Toffoli. These benchmarks are a superset of the arithmetic and Toffoli circuits discussed in Section 6. They range from 45 to 346,533 gates and 5 to 489 qubits. Results on all 32 benchmarks are given in Table 4. As discussed in Section 6, voqc’s performance does not match Nam et al.’s because we have not yet implemented all of their transformations (in particular, we are missing “Toffoli decomposition” and “Floating R_z gates”).

QFT and Adders. These benchmarks consist of components of Shor’s integer factoring algorithm, in particular the quantum Fourier transform (QFT) and integer adders. Two types of adders are considered: an in-place modulo $2q$ adder implemented in the Quipper library and an in-place adder based on the QFT. These benchmarks range from 148 to 381,806 gates and 8 to 4096 qubits. Results on all 27 benchmarks are given in Table 5, Table 6, and Table 7. The Quipper adder programs use similar gates to the arithmetic and Toffoli circuits, so the results are similar—voqc is close to Nam et al., but under-performs due to our simplified Toffoli decomposition. The QFT circuits use rotations parameterized by $\pi/2^n$ for varying $n \in \mathbb{N}$ (and no Toffoli gates) so voqc’s results are identical to Nam et al.’s. For consistency with Nam et al., on the QFT and QFT-based adder circuits we run a simplified version of our optimizer that does not include rotation merging.

Product Formula. These benchmarks implement product formula algorithms for simulating Hamiltonian dynamics. The benchmarks range from 260 to 127,500 gates and 10 to 100 qubits; they use rotations parameterized by floating point numbers, which we convert to OCaml rationals at parse time. The product formula circuits are intended to be repeated for a fixed number of iterations, and our resource estimates account for this. voqc applies Nam et al.’s “LCR” optimization routine to optimize programs across loop iterations. On all 40 product formula benchmarks, our results are the same as those reported by Nam et al. [2018, Table 3]. H gate reductions range from 62.5% to 75%. Reductions in Clifford Z -axis rotations (i.e. rotations by multiples of $\pi/2$) range from 75% to 87.5% while reductions in non-Clifford Z -axis rotations range from 0% to 28.6%. $CNOT$ gate reductions range from 0% to 33%. Runtimes range from 0.01s for parsing and optimizing to 610.46s for parsing and 406.93s for optimizing. By comparison, Nam et al.’s runtimes range from 0.004s to 0.137s.

³<https://github.com/inQWIRE/VOQC-benchmarks>

Table 4. Total gate count reduction on the “Arithmetic and Toffoli” circuits. Includes all programs listed in Table 2 and Table 3 as well as four gf programs omitted from that table. The reported voqc time only includes optimization time. voqc’s parse time was less than 0.001s for all benchmarks except the larger gf programs; the largest, gf2^163_mult, required 427s (7.1min) to parse. Nam (H) results were not available for the large benchmarks.

Name	Orig. Total	Nam (L)		Nam (H)		voqc	
		Total	t(s)	Total	t(s)	Total	t(s)
adder_8	900	646	0.004	606	0.101	682	0.048
barenco_tof_3	58	42	<0.001	40	0.001	50	0.001
barenco_tof_4	114	78	<0.001	72	0.001	95	0.002
barenco_tof_5	170	114	<0.001	104	0.003	140	0.003
barenco_tof_10	450	294	0.001	264	0.012	365	0.019
csla_mux_3	170	161	<0.001	155	0.009	158	0.003
csum_mux_9	420	294	<0.001	266	0.009	308	0.006
gf2^4_mult	225	187	0.001	187	0.009	192	0.006
gf2^5_mult	347	296	0.001	296	0.020	291	0.012
gf2^6_mult	495	403	0.003	403	0.047	410	0.025
gf2^7_mult	669	555	0.004	555	0.105	549	0.045
gf2^8_mult	883	712	0.006	712	0.192	705	0.070
gf2^9_mult	1095	891	0.010	891	0.347	885	0.119
gf2^10_mult	1347	1070	0.009	1070	0.429	1084	0.183
gf2^16_mult	3435	2707	0.065	2707	5.566	2695	1.347
gf2^32_mult	13593	10601	1.834	10601	275.698	10577	26.808
gf2^64_mult	53691	41563	58.341	–	–	41515	546.887
gf2^128_mult	213883	165051	1744.746	–	–	164955	9841.797
gf2^131_mult	224265	173370	1953.353	–	–	173273	10877.112
gf2^163_mult	346533	267558	4955.927	–	–	267437	27612.565
mod5_4	63	51	<0.001	51	0.001	56	<0.001
mod_mult_55	119	91	<0.001	91	0.002	90	0.002
mod_red_21	278	184	<0.001	180	0.008	214	0.005
qcla_adder_10	521	411	0.002	399	0.044	438	0.018
qcla_com_7	443	284	0.001	284	0.016	314	0.013
qcla_mod_7	884	636	0.004	624	0.077	723	0.058
rc_adder_6	200	142	<0.001	140	0.004	157	0.003
tof_3	45	35	<0.001	35	<0.001	40	<0.001
tof_4	75	55	<0.001	55	<0.001	65	0.001
tof_5	105	75	<0.001	75	0.001	90	0.002
tof_10	255	175	<0.001	175	0.004	215	0.006
vbe_adder_3	150	89	<0.001	89	0.001	101	0.002
Avg. Red.		24.6%		26.4%		19.2%	

Table 5. Total gate count reduction on Quipper adder circuits. voqc's H and T counts are identical to Nam (L) and (H), but the total R_z and $CNOT$ counts are higher due to Nam et al.'s specialized Toffoli decomposition. The difference between Nam (L) and Nam (H) is entirely due to $CNOT$ count. Our initial gate counts are higher than those reported by Nam et al. because we do not have special handling for \pm control Toffoli gates; we simply consider the standard Toffoli gate conjugated by additional X gates.

n	Original	Nam (L)		Nam (H)		voqc		
	Total	Total	t(s)	Total	t(s)	Total	Parse t(s)	Opt. t(s)
8	585	239	0.001	190	0.006	352	<0.01	0.02
16	1321	527	0.003	414	0.018	784	<0.01	0.12
32	2793	1103	0.014	862	0.066	1648	0.01	0.63
64	5737	2255	0.057	1758	0.598	3376	0.03	3.30
128	11625	4559	0.244	3550	4.697	6832	0.16	16.37
256	23401	9167	1.099	7134	34.431	13744	1.06	79.74
512	46953	18383	5.292	14302	307.141	27568	7.50	394.74
1024	94057	36815	25.987	28638	2446.336	55216	45.76	1894.41
2048	188265	73679	145.972	57310	23886.841	110512	252.48	9307.36
Avg. Red.		63.7%		71.6%		45.7%		

Table 6. Results on QFT circuits. Exact timings and gate counts are not available for Nam (L) or Nam (H), but our results are consistent with those reported in Nam et al. [2018, Figure 1].

n	Original			voqc				
	$CNOT$	R_z	H	$CNOT$	R_z	H	Parse t(s)	Opt. t(s)
8	56	84	8	56	42	8	<0.01	<0.01
16	228	342	16	228	144	16	<0.01	<0.01
32	612	918	32	612	368	32	0.01	0.01
64	1380	2070	64	1380	816	64	0.05	0.07
128	2916	4374	128	2916	1712	128	0.29	0.39
256	5988	8982	256	5988	3504	256	1.99	2.34
512	12132	18198	512	12132	7088	512	13.57	15.69
1024	24420	36630	1024	24420	14256	1024	93.45	106.71
2048	48996	73494	2048	48996	28592	2048	562.18	674.11
Avg. Red.				0%	59.3%	0%		

Table 7. Results on QFT-based adder circuits. Final gate counts are identical for voqc and Nam (L).

n	Original			voqc					Nam (L)
	$CNOT$	R_z	H	$CNOT$	R_z	H	Parse t(s)	Opt. t(s)	t(s)
8	184	276	16	184	122	16	<0.01	<0.01	<0.001
16	716	1074	32	716	420	32	0.01	0.02	0.001
32	1900	2850	64	1900	1076	64	0.10	0.13	0.002
64	4268	6402	128	4268	2388	128	0.76	0.90	0.004
128	9004	13506	256	9004	5012	256	5.80	5.52	0.08
256	18476	27714	512	18476	10260	512	43.73	36.80	0.018
512	37420	56130	1024	37420	20756	1024	293.19	255.20	0.045
1024	75308	112962	2048	75308	41748	2048	1516.76	1695.65	0.115
2048	151084	226626	4096	151084	83732	4096	7488.03	8481.66	0.215
Avg. Red.				0%	61.8%	0%			

B SQIR VS. QWIRE

As discussed in Section 3.3, a key difference between SQIR and QWIRE is how they refer to qubits: SQIR uses concrete indices into a global register, while QWIRE uses abstract variables, implemented using higher-order abstract syntax [Pfenning and Elliott 1988]. The tradeoff between the two approaches is most evident in how they support composition.

Composition in QWIRE. QWIRE circuits have the following form:

```
Inductive Circuit (w : WType) : Set :=
| output : Pat w → Circuit w
| gate   : ∀ {w1 w2}, Gate w1 w2 → Pat w1 → (Pat w2 → Circuit w) → Circuit w
| lift   : Pat Bit → (B → Circuit w) → Circuit w.
```

Patterns Pat type the variables in QWIRE circuits; their type index w corresponds to some collection of bits and qubits. The circuit output p is simply a wire with the wire type associated with p. Note that this, like all Circuits, is an open term. The definition of gate takes in a Gate parameterized by an input type w1 and an output w2, an appropriately typed input pattern, and a *continuation* of the form Pat w2 → Circuit w, which describes how the gate's output is used in the rest of the circuit. Finally, lift takes a single Bit (or classical wire) and a continuation that constructs a circuit based on that bit's interpretation as a Boolean value.

This use of continuations makes composition easy to define:

```
Fixpoint compose {w1 w2} (c : Circuit w1) (f : Pat w1 → Circuit w2) : Circuit w2 :=
  match c with
  | output p   ⇒ f p
  | gate g p c' ⇒ gate g p (fun p' ⇒ compose (c' p') f)
  | lift p c'   ⇒ lift p (fun bs ⇒ compose (c' bs) f)
  end.
```

In each case, the continuation is applied directly to the output of the first circuit.

While circuits correspond to open terms, closed terms are represented by *boxed* circuits:

```
Inductive Box w1 w2 : Set := box : (Pat w1 → Circuit w2) → Box w1 w2.
```

For convenience, box (fun w ⇒ c) is written as simply box w ⇒ c and let p ← c1 ; c2 is similarly defined as notation for compose c1 (fun p ⇒ c2). One can unbox a boxed circuit Box w1 w2 simply by providing a valid Pat w1, obtaining a Circuit w2.

This representation allows for easy sequential and parallel composition of closed circuits. Running two circuits in sequence involves connecting the output of the first circuit to the input of the second circuit (where the types will guarantee compatibility) and running them in parallel gives the circuits disjoint inputs and outputs their results (leading to a tensor type).

```
Definition inSeq {w1 w2 w3} (c1 : Box w1 w2) (c2 : Box w2 w3) : Box w1 w3 :=
  box p1 ⇒
    let p2 ← unbox c1 p1;
    unbox c2 p2.
```

```
Definition inPar {w1 w2 w1' w2'} (c1 : Box w1 w2) (c2 : Box w1' w2')
  : Box (w1 ⊗ w1') (w2 ⊗ w2') :=
  box (p1,p2) ⇒
    let p1' ← unbox c1 p1;
    let p2' ← unbox c2 p2;
    (p1',p2').
```

Unfortunately, proving useful specifications for these functions is quite difficult. The denotation of a circuit is (in the unitary case) a square matrix of size 2^n for some n . To construct such a matrix we need to map all of a circuit's (abstract) variables to 0 through $n - 1$, ensuring that the mapping function has no gaps even when we initialize or discard qubits. \mathcal{Q} WIRE maintains this invariant through compiling to a de Bruijn-style variable representation [de Bruijn 1972]. Reasoning about the denotation of circuits, then, involves reasoning about this compilation procedure. In the case of open circuits (the most basic circuit type), we must also reason about the contexts that type the available variables, which change upon every gate application.

Composition in SQIR. Composing two SQIR programs requires manually defining a mapping from the global registers of both programs to a new, combined global register. To do this, we provide two helper functions, which respectively renumber a unitary program's concrete indices according to a mapping f , and change the program's global register size.

```
Fixpoint map_qubits {U dim} (f : N → N) (c : ucom U dim) : ucom U dim :=
  match c with
  | c1; c2 ⇒ map_qubits f c1; map_qubits f c2
  | uapp1 u n ⇒ uapp1 u (f n)
  | uapp2 u m n ⇒ uapp2 u (f m) (f n)
  end.
```

```
Fixpoint cast {U dim} (c : ucom U dim) dim' : ucom U dim' :=
  match c with
  | c1; c2 ⇒ cast c1 dim' ; cast c2 dim'
  | uapp1 u n ⇒ uapp1 u n
  | uapp2 u m n ⇒ uapp2 u m n
  end.
```

With these, we can define parallel composition in SQIR:

```
Definition inPar {U dim1 dim2} (c1 : ucom U dim1) (c2 : ucom U dim2) :=
  cast c1 (dim1 + dim2);
  cast (map_qubits (fun q ⇒ dim1 + q) c2) (dim1 + dim2).
```

The correctness property for `inPar` says that the denotation of `inPar c1 c2` can be constructed from the denotations of `c1` and `c2`.

```
Lemma inPar_correct : ∀ c1 c2 d1 d2,
  uc_well_typed d1 c1 → ninPar c1 c2 d1 d1+d2 = nc10d1 ⊗ nc20d2.
```

The `inPar` function is relatively simple, but more involved than the corresponding \mathcal{Q} WIRE definition because it requires relabeling the qubits in program c_2 .

General composition in SQIR (including sequential composition) requires even more involved relabeling functions that are less straightforward to describe. For example, consider the composition expressed in the following \mathcal{Q} WIRE program:

```
box (ps, q) ⇒
  let (x, y, z) ← unbox c1 ps;
  let (q, z) ← unbox c2 (q, z);
  (x, y, z, q).
```

This program connects the last output of program c_1 to the second input of program c_2 . This operation is natural in \mathcal{Q} WIRE, but describing this type of composition in SQIR requires some effort. In particular, the programmer must determine the required size of the new global register (in this

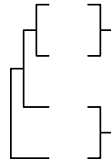


Fig. 11. The patterns for the output of `ghz` (left) and the input to `fredkin_seq` (right) over four qubits. `ghz` produces a list of qubits $((q_1, q_2), q_3), q_4$ whereas `fredkin_seq` expects a tree $((q_1, q_2), (q_3, q_4))$. In composition, the mismatched patterns require an extra gadget to transform the former into the latter.

case, 4) and explicitly provide a mapping from qubits in c_1 and c_2 to indices in the new register (for example, the first qubit in c_2 might be mapped to the fourth qubit in the new global register). When `SQIR` programs are written directly, this puts extra burden on the programmer. When `SQIR` is used as an intermediate representation, however, these mapping functions should be produced automatically by the compiler. The issue remains, though, that any proofs we write about the result of composing c_1 and c_2 will need to reason about the mapping function used (whether produced manually or automatically).

As an informal comparison of the impact of `QWIRE`'s and `SQIR`'s representations on proof, we note that while proving the correctness of the `inPar` function in `SQIR` took a matter of hours, there is no correctness proof for the corresponding function in `QWIRE`, despite many months of trying. Of course, this comparison is not entirely fair: `QWIRE`'s `inPar` is more powerful than `SQIR`'s equivalent. `SQIR`'s `inPar` function does not require every qubit within the global register to be used – any gaps will be filled by identity matrices. Also, `SQIR` does not allow introducing or discarding qubits, which we suspect will make ancilla management difficult to reason about.

Quantum Data Structures. `SQIR` also lacks some other useful features present in higher-level languages. For example, in `QIO` [Altenkirch and Green 2010] and `Quipper` [Green et al. 2013] one can construct circuits that compute on quantum data structures, like lists and trees of qubits. In `QWIRE`, this concept is refined to use more precise dependent types to characterize the structures; e.g., the type for the `GHZ` program indicates it takes a list of n qubits to a list of n qubits. More interesting dependently-typed programs, like the quantum Fourier transform, use the parameter n as an argument to rotation gates within the program.

Regrettably, these structures can make reasoning about programs difficult. For instance, as shown in Figure 11, the `GHZ` program written in `QWIRE` emits a list of qubits while the `fredkin_seq` circuit takes in a tree of qubits. Connecting the qubits from a `GHZ` to a `fredkin_seq` circuit with the same arity requires an intermediate gadget. And if we want to verify a property of this composition, we need to prove that this gadget is an identity. In `SQIR`, which has neither quantum data structures nor typed circuits, this issue does not present itself.

Dynamic Lifting. `SQIR` also does not support *dynamic lifting*, which refers to a language feature that permits measuring a qubit and using the result as a Boolean value in the host language to compute the remainder of a circuit [Green et al. 2013]. Dynamic lifting is used extensively in `Quipper` and `QWIRE`. Unfortunately, its presence complicates the denotational semantics, as the semantics of any `Quipper` or `QWIRE` program depends on the semantics of `Coq` or `Haskell`, respectively. In giving a denotational semantics to `QWIRE`, Paykin et al. [2017] assume an operational semantics for an arbitrary host language, and give a denotation for a lifted circuit only when both of its branches reduce to valid `QWIRE` circuits.

Although `SQIR` does not support dynamic lifting, its `meas` construct is a simpler alternative. Since the outcome of the measurement is not used to compute a new circuit, `SQIR` does not need a classical

host language to do computation: It is an entirely self-contained, deeply embedded language. As a result, we can reason about SQIR circuits in isolation, and also easily reason about families of SQIR circuits described in Coq.

Other Differences. Another important difference between QWIRE and SQIR is that QWIRE circuits cannot be easily decomposed into subcircuits because output variables are bound in different places throughout the circuit. By contrast, a SQIR program is an arbitrary nesting of smaller programs. This means that for SQIR, the program $c1;((c2;(c3;c4));c5)$ is equivalent to $c1;c2;c3;c4;c5$ under all semantics, whereas every QWIRE circuit (only) associates to the right. This allows us to arbitrarily flatten SQIR programs into a convenient list representation, as is done in VOQC (Section 4), and makes it easy to rewrite using SQIR identities.

Also, unlike most quantum languages and as already discussed in the main body of the paper, SQIR features a distinct core language of unitary operators; the full language adds measurement to this core. The semantics of a unitary program is expressed directly as a matrix, which means that proofs of correctness of unitary optimizations (the bulk of VOQC) involve reasoning directly about matrices. Doing so is far simpler than reasoning about functions over density matrices, as is required for the full SQIR language or any program in QWIRE.

Concluding thoughts. Upon reflection, we can see that the differences between QWIRE and SQIR ultimately stem from their design goals. QWIRE was developed as a general-purpose programming language for quantum computers [Paykin et al. 2017], with ease of programmability as a key concern; only later was it extended as a tool for verification [Rand et al. 2018b, 2017]. By contrast, SQIR was designed from the start with verification in mind, with by-hand programmability a secondary consideration; we expected SQIR would be compiled from another language such as Q# [Svore et al. 2018], Quipper [Green et al. 2013] or even QWIRE itself. That said, for near-term quantum programs SQIR's lower-level abstractions have not proved difficult to use, even for source programming [Hietala et al. 2020]. As programs scale up, finding the right way to extend SQIR (or a language like it) with higher level abstractions without overly complicating verification will be an important goal.