A Verified Optimizer for Quantum Circuits

Kesha Hietala Robert Rand Shih-Han Hung
Xiaodi Wu Michael Hicks
University of Maryland, College Park, USA

Abstract

We present VOQC, the first fully verified compiler for quantum circuits, written using the Coq proof assistant. Quantum circuits are expressed as programs in a simple, low-level language called SQIR, 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. We evaluate VOQC's verified optimizations on a series of benchmarks, and it performs comparably to industrial-strength compilers. VOQC's optimizations reduce total gate counts on average by 17.7% on a benchmark of 29 circuit programs compared to a 10.7% reduction when using IBM's Qiskit compiler.

Keywords Formal Verification, Quantum Computing, Optimization, Certified Compilation, Programming Languages

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. For evidence of the former, consider that higher optimization levels of IBM's Qiskit compiler are known to have mistakes (as is evident from its issue tracker[1]). Kissinger and van de Wetering [24] discovered mistakes in the optimized outputs produced by the circuit compiler by Nam et al. [30]. And Nam et al. themselves found that the optimization library they compared against sometimes produced incorrect results. Making mistakes when optimizing by hand is also to be expected: as put well by Zamdzhiev [49], quantum computing can be frustratingly unintuitive.

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 may be 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. As an example, consider CompCert [26], which is a compiler for C programs that is written and proved correct using the Coq proof assistant [9]. 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"). While SQIR is designed to be a *simple quantum intermediate representation*, it is not very different from languages such as PyQuil [36], which are used to construct quantum source programs as circuits. VOQC applies a series of optimizations to SQIR programs, ultimately producing a result that is compatible with the specified quantum architecture. For added convenience, VOQC provides translators between SQIR and OpenQASM, the de facto standard format for quantum circuits [12]. (Section 2.)

Like CompCert, VOQC is implemented using the Coq proof assistant. SQIR program ASTs are represented in Coq via a deep embedding, and optimizations are implemented as Coq functions, which are then extracted to OCaml. We define two semantics for SQIR programs. The simplest denotes every quantum circuit as a *unitary matrix*. However this is only applicable to unitary circuits, i.e., circuits without measurement. For non-unitary circuits, we provide a denotation of *density matrices*. Properties of SQIR programs, or transformations of them, can be proved using whichever semantics is most convenient. As examples, we have proved correctness properties about several source programs written in SQIR, including *GHZ state preparation*, *quantum teleportation*, *superdense coding*, and the *Deutsch-Jozsa algorithm*. Generally speaking, SQIR is designed to make proofs as easy as possible. For example, we initially contemplated SQIR programs accessing qubits as Coq variables via higher order abstract syntax [32], but we found that proofs were far simpler when qubits were accessed as concrete indices into a global register. (Section 3.)

We have implemented and proved correct several transformations over SQIR programs. In particular, we have developed verified versions of many of the optimizations used in a state-of-the-art compiler developed by Nam et al. [30]. We have also verified a circuit mapping routine that transforms SQIR programs to satisfy constraints on how qubits may interact

[1] https://github.com/Qiskit/qiskit-terra/issues/2752

on a particular target architecture. These transformations were reasonably straightforward to prove correct thanks to SQIR's design. (Sections 4 and 5.)

We find that VOQC performs comparably to unverified, state-of-the-art compilers when run on a benchmark of 29 circuit programs developed by Amy et al. [5]. These programs range from 45 and 61,629 gates and use between 5 and 192 qubits. VOQC reduced total gate counts on average by 17.7% compared to 10.7% by IBM's Qiskit compiler [2]. There is still room for improvement: Nam et al. [30] produced reductions of 26.5% using additional optimizations we expect we could verify. (Section 6.)

VOQC is the first fully verified circuit optimizer for a realistic quantum circuit language. Amy et al. [6] developed a verified optimizing compiler from source Boolean expressions to reversible circuits, but did not handle general quantum programs. Rand et al. [34] developed a similar compiler for quantum circuits but without optimizations. Other low-level quantum languages [12, 42] have not been developed with verification in mind, and prior circuit-level optimizations [5, 20, 30] have not been formally verified. In concurrent work, Shi et al. [40] developed CertiQ, which uses symbolic execution and SMT solving to verify some circuit transformations in the Qiskit compiler. CertiQ is limited to verifying correct application of local equivalences, rather than more general circuit transformations, and sometimes verification fails in which case CertiQ must validate the optimized ("translated") circuits on-line. The PyZX compiler [24] likewise uses translation validation to check its rewrites of circuits using the equational theory of the ZX-Calculus [8]. (Section 7.)

Our work on VOQC and SQIR constitutes a step toward developing a full-scale verified compiler toolchain. Next steps include developing certified transformations from high-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/SQIRE.

2 Overview

We begin with a brief background on quantum programs, and then provide an overview of VOQC and SQIR.

2.1 Preliminaries

Quantum programs operate over *quantum states*, which consist of one or more *quantum bits* (aka, *qubits*). A single qubit is represented as a vector of complex numbers $\langle \alpha, \beta \rangle$ such that $|\alpha|^2 + |\beta|^2 = 1$. 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 $|\psi\rangle$ is called a *ket*, following Dirac's notation. We say a qubit is in a *superposition* of $|0\rangle$ and $|1\rangle$ when both α and β 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 $|\alpha|^2$ and 1 with probability $|\beta|^2$. 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 $\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. Applying H to state $|0\rangle$ yields $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle$, 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 $\langle \alpha, \beta, \gamma, \delta \rangle$ 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!

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 is $|0\rangle$ then both qubits are mapped to themselves, but if the first qubit is $|1\rangle$ then the second qubit is negated, e.g., $CNOT|00\rangle = |00\rangle$ while $CNOT|10\rangle = |11\rangle$.

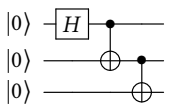
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

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 $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$) 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; an example is shown in Figure 1(a). In these circuits, each horizontal wire represents a *qubit* and boxes on these wires indicate unitary quantum operators, i.e., *gates*. For multiple-qubit gates, the inputs are often distinguished as being either



(a) Quantum Circuit

```
H 0
CNOT 0 1
CNOT 1 2
```

(b) Quil

```
def ghz_state(qubits):
  program = Program()
  program += H(qubits[0])
  for q1,q2 in zip(qubits, qubits[1:]):
    program += CNOT(q1, q2)
  return program
```

(c) PyQuil (arbitrary # qubits)

Figure 1. Example quantum program: GHZ state preparation

a *target* or a *control*. In software, these circuits are often represented using lists of instructions that describe the different gate applications. For example, Figure 1(b) is the Quil [42] representation of the circuit in Figure 1(a).

In the *QRAM model* [25] quantum computers are used as co-processors to classical computers. The classical computer generates descriptions of circuits to send to the quantum computer, and then processes the returned results. High-level quantum computing languages are designed to follow this model. For example, Figure 1(c) shows a program in PyQuil [36], a quantum language/framework embedded in Python. The `ghz_state` function takes an array `qubits` and constructs a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [19], which is an n-qubit entangled quantum state of the form

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

Calling `ghz_state([0,1,2])` would return the Quil program in Figure 1(b), which the quantum computer could subsequently execute. 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 VOQC: A Verified Optimizer for Quantum Circuits

The structure of VOQC is summarized in Figure 2.

Source programs are written in SQIR, whose syntax and semantics we give in Section 3. SQIR is a simple circuit-oriented language deeply embedded in Coq, similar in style to PyQuil. SQIR programs are given a formal semantics in Coq, which is the basis for proving properties about them. For example, we can prove that the `GHZ` program prepares the expected quantum state.

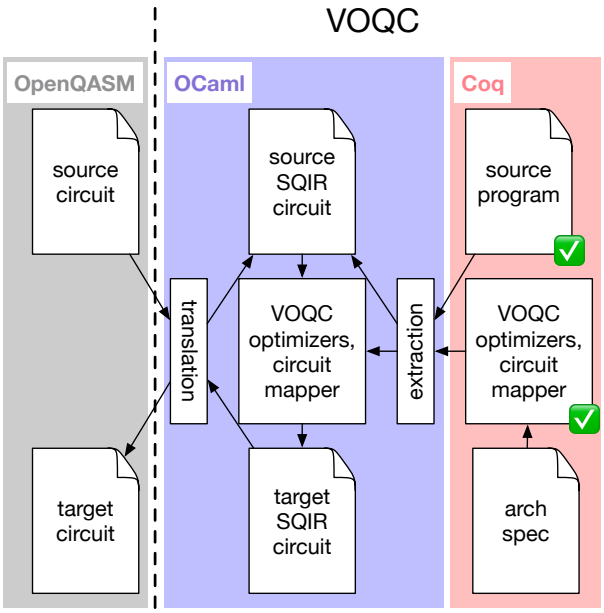


Figure 2. VOQC architecture

VOQC implements a series of transformations over SQIR programs, which we detail in Sections 4 and 5. Using the formal semantics for SQIR programs, we prove that each optimization is semantics preserving. 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.

Using Coq's standard code extraction mechanism, we can extract VOQC into a standalone OCaml program. We have implemented (unverified) conversion between OpenQASM [12] and SQIR, which we link against our extracted code. Since a number of quantum programming frameworks, including Qiskit [2], Project Q [44] and Cirq [46], 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 8000 lines of Coq, with just over 1500 for core SQIR and the rest for VOQC transformations. VOQC also uses some existing Coq libraries for quantum computing developed for the QWIRE language [35]. We use about 400 lines of standalone OCaml code for parsing OpenQASM programs and running VOQC on our benchmarks in Section 6. Our development additionally contains nearly 1000 lines of example SQIR programs and proofs about them.

3 SQIR: A Small Quantum Intermediate Representation

This section presents the syntax and semantics of SQIR programs. We begin with the core of SQIR, which describes unitary circuits. We then describe the expanded language, which allows measurement and initialization.

$$[\![U_1;\ U_2]\!]_d = [\![U_2]\!]_d \times [\![U_1]\!]_d$$

$$[\![G_1\ q]\!]_d = \begin{cases} apply_1(G_1,\ q,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}$$

$$[\![G_2\ q_1\ q_2]\!]_d = \begin{cases} apply_2(G_2,\ q_1,\ q_2,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}$$

Figure 3. Semantics of unitary sqir programs, assuming a global register of dimension d. The $apply_k$ 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. For example, $apply_1(X,\ q,\ d) = I_{2^q} \otimes \sigma_x \otimes I_{2^{(d-q-1)}}$ where σ_x is the matrix interpretation of the X gate.

3.1 Unitary Core

sqir is a language for describing quantum programs that is deeply embedded in the Coq proof assistant. In sqir, a qubit is referred to by a natural number that indices into a *global register* of quantum bits. Unitary sqir programs allow sequencing and unitary gate application to one or two qubits, drawing from a fixed set of gates.

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

Each sqir program is parameterized by a unitary gate set (from which G is drawn) and the dimension of the global register (i.e., the number of available qubits).

A unitary program U is *well-typed* if every gate application is valid. A gate application is valid if all of its arguments are in-bounds indices into the global register, and no index is repeated. This second requirement enforces linearity and thereby quantum mechanics' no-cloning theorem.

The semantics of unitary sqir programs is shown in Figure 3. If a program is not well-typed, its denotation is the zero matrix. The advantage of this definition is that it allows us to reference the denotation of a program without explicitly assuming or (re)proving that the program is well-typed, thus removing clutter from theorems and proofs.

When a program is well-typed, computing its denotation requires a matrix interpretation for every unitary gate G. In our development we define the semantics of sqir programs over the gate set $\{R_{\theta,\phi,\lambda},\ CNOT\}$ where $R_{\theta,\phi,\lambda}$ is a general single-qubit rotation parameterized by three real-valued rotation angles and $CNOT$ is the standard two-qubit controlled-not gate. We refer to this gate set as the *base set*. It is the same as the underlying set used by OpenQASM [12] and is *universal*, meaning that it can approximate any unitary operation to within arbitrary error. The matrix interpretation of the single-qubit $R_{\theta,\phi,\lambda}$ gate is

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

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

Figure 4. sqir density matrix semantics, assuming a global register of size d.

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

Common single-qubit gates can be defined in terms of $R_{\theta,\phi,\lambda}$. For example, the identity I is $R_{0,0,0}$; the Hadamard H gate is $R_{\pi/2,0,\pi}$; the Pauli X gate is $R_{\pi,0,\pi}$ 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, is a sequence of three $CNOT$ gates.

We say that two unitary programs are equivalent, written $U_1 \equiv U_2$, if their denotation is the same, i.e., $[\![U_1]\!]_d = [\![U_2]\!]_d$. For verifying equivalence of quantum programs, however, we will often want something more general since $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ (for $\theta \in \mathbb{R}$) represent the same physical state. We therefore say that two circuits are *equivalent up to a global phase*, written $U_1 \cong U_2$, when there exists a θ such that $[\![U_1]\!]_d = e^{i\theta}[\![U_2]\!]_d$.

3.2 Adding Measurement

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

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

The command $\texttt{meas}\ q\ P_1\ P_2$ (inspired by a similar construct in QPL [39]) 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:

$$\texttt{measure}\ q = \texttt{meas}\ q\ \texttt{skip}\ \texttt{skip}$$

$$\texttt{reset}\ q = \texttt{meas}\ q\ (X\ q)\ \texttt{skip}$$

Figure 4 defines the semantics of non-unitary programs in terms of density matrices, following the approach of several previous efforts [31, 48]. The density matrix semantics encodes different measurement outcomes as a probability distribution. We also provide a non-deterministic semantics in Appendix A.4, which is sometimes more convenient.

Since the density matrix semantics denotes programs as functions over matrices, we say that two programs P_1 and P_2 are equivalent if for every input ρ, $\{\!|P_1|\!\}_d(\rho) = \{\!|P_2|\!\}_d(\rho)$.

3.3 Example

Recall the GHZ preparation example from Section 2. The following Coq function `GHZ` recursively constructs an n-qubit

sqir program in the base set (i.e., a value of type `ucom base n`) that prepares the GHZ state. It is similar to the PyQuil program in Figure 1(c).

```
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.
```

When n is 0, the result is just the identity I on wire 0. When n is 1, the result is the Hadamard gate applied to wire 0. (Here, I and H are the notations for $R_{\theta,\phi,\lambda}$ gates presented in Section 3.1.) When n is greater than 1, it constructs the program $U_1; U_2$, where U_1 is the `GHZ` circuit on $n-1$ qubits, and U_2 is the appropriate *CNOT* gate. The result of `GHZ 3` is equivalent to the circuit shown in Figure 1(a).

3.4 Designing sqir

sqir was designed to be expressive while facilitating mechanically checked proofs of quantum programs. We conclude this section by discussing key features of sqir's design.

Expressiveness As a deeply embedded, domain-specific circuit language, sqir is similar to PyQuil [36] and Cirq [46], both in terms of programming experience and expressive power. As such, it makes a reasonable source programming language. However, unlike these languages, we can use sqir's host language, Coq, to prove properties about its programs. For example, we can prove that every circuit generated by `GHZ n`, above, produces the corresponding state $|\mathrm{GHZ}^n\rangle$ given in Section 2.2 when applied to $|0\ldots0\rangle$.

As a demonstration of sqir's expressiveness as a source language, Appendix A presents proof of this property along with other examples of sqir programs and corresponding formal properties about them. In particular, we present *quantum teleportation* and proof that it indeed transports the intended qubit; *superdense coding* and proof that it prepares the expected state; and the *Deutsch-Jozsa algorithm* with proof that it correctly distinguishes between a constant and balanced Boolean oracle.

Verification sqir's design has two key features that facilitate verification. First, unlike most quantum languages, 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 language.

Second, a sqir program uses *concrete (numeric) indices into a global register* to refer to wires. As such, the semantics can simply map concrete indices to rows and columns in the denoted matrix. In addition, wire disjointness in the sqir program is obvious—$G_1\ m$ operates on a different wire 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 lets us to reason about gates acting on different parts of the circuit in isolation).

The alternative, used by Quipper [17] and Qwire [31], is to use variables to refer to *abstract* wires, which are later allocated to concrete wires. These languages are embedded in Haskell and Coq, respectively, and take advantage of their host languages' variable bindings. This approach eases programmability—larger circuits can be built by composing smaller ones, connecting abstract outputs to abstract inputs. However, we find that this approach complicates formal proof. The semantics of a program that uses abstract wires must convert those wires into concrete indices into the denoted matrix. Reasoning about this conversion can be laborious, especially for recursive circuits and those that allocate and deallocate wires (entailing de Bruijn-style index shifting [33]). Moreover, notions like disjointness are no longer obvious—$G_1\ x$ and $G_2\ y$ for variables $x \neq y$ may not be disjoint if x and y could be allocated to the same concrete wire. Appendix B has more details.

4 Optimizing Unitary sqir Programs

The voqc compiler takes as input a sqir program and attempts to reduce its total gate count by applying a series of optimizations. This section describes optimizations on unitary sqir programs. The next section discusses how voqc optimizes full sqir programs and maps them to a connectivity-constrained architecture.

4.1 Overview

voqc's unitary optimizations are defined as Coq functions that map an input program to an optimized one. A program is represented as a list of gate applications. Sequences of gate applications are *flattened* so that a sqir program like $(G_1\ p; G_2\ q); G_3\ r$ is represented as the Coq list $[G_1\ p; G_2\ q; G_3\ r]$. This representation simplifies finding patterns of gates.

The optimization functions expect a program's gates G to be drawn from the set $\{H,\ X,\ Rz_{\pi/4},\ CNOT\}$ where $Rz_{\pi/4}(k)$ describes rotation about the z-axis by $k \cdot \pi/4$ for $k \in \mathbb{Z}$. Either the parser must produce input programs using this gate set, or voqc must convert the program to use it before optimizations can be applied. This gate set is universal, and is consistent with previous circuit optimizers, e.g., Amy et al. [5]. Rotations are not parameterized by arbitrary reals, which would make verification unsound if these were extracted to OCaml floating point numbers (which are used in the gate sets used by Nam et al. [30] and Qiskit [2]). It would be easy to support $Rz_{\pi/2^n}$ for higher n if finer-grained rotations are needed.

Programs in the list representation are deemed equivalent if their back-converted sqir programs are equivalent, per

the definition in Section 3.1. Conversion translates VOQC's gates H, X, and $Rz_{\pi/4}(k)$ into base gates $R_{\pi/2,0,\pi}$, $R_{\pi,0,\pi}$, and $R_{0,0,k\pi/4}$, respectively. (*CNOT* translates to itself.)

Most of VOQC's optimizations are inspired by the state-of-the-art circuit optimizer by Nam et al. [30]. There are two basic kinds of optimizations: *replacement* and *propagation and cancellation*. The former simply identifies a pattern of gates and replaces it by an equivalent pattern. The latter works by 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 each other out.

4.2 Proving Circuit Equivalences

All of VOQC's optimizations use circuit equivalences to justify local rewrites. Proof that an optimization is correct thus relies on proofs that the circuit equivalences it uses are correct. Many of our circuit equivalence proofs have a common form, which we illustrate by example.

Suppose we wish to prove the equivalence

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

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

$$\begin{aligned} &apply_1(X, n, d)\ \times\ apply_2(CNOT, m, n, d) = \\ &apply_2(CNOT, m, n, d)\ \times\ apply_1(X, n, d), \end{aligned} \tag{1}$$

per Figure 3. Suppose both sides of the equation are well typed ($m < d$ and $n < d$ and $m \neq n$), and consider the case $m < n$ (the $n < m$ case is similar). We expand $apply_1$ and $apply_2$ as follows with $p = n - m - 1$ and $q = d - n - 1$:

$$\begin{aligned} apply_1(X, n, d) &= I_{2^n} \otimes \sigma_x \otimes I_{2^q} \\ apply_2(CNOT, m, n, d) &= I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes \sigma_x \otimes I_{2^q} \\ &\quad + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q} \end{aligned}$$

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

Automation Matrix normalization and simplification are almost entirely automated in VOQC. We wrote 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 their "grid normal" form 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., $((..\times..)\otimes(..\times..))+((..\times..)\otimes(..\times..))$.

$$\begin{aligned} X\ q;\ H\ q &\equiv H\ q;\ Z\ q \\ Z\ q;\ H\ q &\equiv H\ q;\ X\ q \\ X\ q;\ Rz_{\pi/4}(k)\ q &\cong Rz_{\pi/4}(8-k)\ q;\ X\ q \\ X\ q_1;\ CNOT\ q_1\ q_2 &\equiv CNOT\ q_1\ q_2;\ X\ q_1;\ X\ q_2 \\ X\ q_2;\ CNOT\ q_1\ q_2 &\equiv CNOT\ q_1\ q_2;\ X\ q_2 \\ Z\ q;\ Rz_{\pi/4}(k)\ q &\equiv Rz_{\pi/4}(k)\ q;\ Z\ q \\ Z\ q_1;\ CNOT\ q_1\ q_2 &\equiv CNOT\ q_1\ q_2;\ Z\ q_1 \\ Z\ q_2;\ CNOT\ q_1\ q_2 &\equiv CNOT\ q_1\ q_2;\ Z\ q_1;\ Z\ q_2 \end{aligned}$$

Figure 5. Equivalences used in not propagation.

The `gridify` tactic rewrites an expression into this form by using the following rules of matrix arithmetic (where all the dimensions are appropriate):

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

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 $apply_2$ 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. $\sigma_x\sigma_x = I_2$).

In our example, after normalization and simplification by `gridify`, both sides of the equality in Equation (1) become

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

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 on up to three qubits. For equivalences over larger, non-concrete circuits like the one used in *rotation merging*, we do not use `gridify` directly, but still rely on our automation for matrix simplification.

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 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 5. These

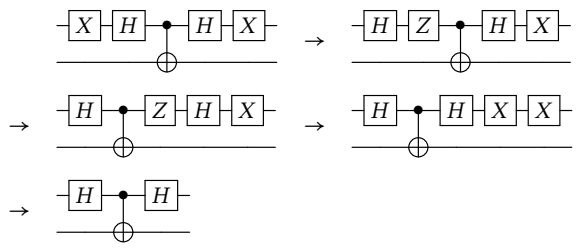


Figure 6. An example of not propagation. In the first step the leftmost X gate propagates through the leftmost H gate and becomes a Z gate. In the second step the Z gate propagates through the control of the $CNOT$ gate. In the third step the Z gate propagates through the rightmost H gate and becomes an X gate. In the final step the two X gates cancel.

rules switch between propagating X and Z (= $Rz_{\pi/4}(4)$) gates. In particular, an X gate can "propagate" past an H gate by becoming a Z, and likewise a Z can propagate past an H by becoming an X. An example application of the not propagation algorithm is shown in Figure 6.

This implementation may introduce extra X and Z gates at the end of a circuit. However, redundant Z gates will be removed by the single-qubit gate cancellation optimization, 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 more general than Nam et al.'s, which is specialized to a three-qubit *TOFF* gate. The *TOFF* gate can be decomposed into a $\{H, Rz_{\pi/4}, CNOT\}$ program, and Nam et al.'s propagation rules can be written in terms of the rules in Figure 5.

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 input (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 verified 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

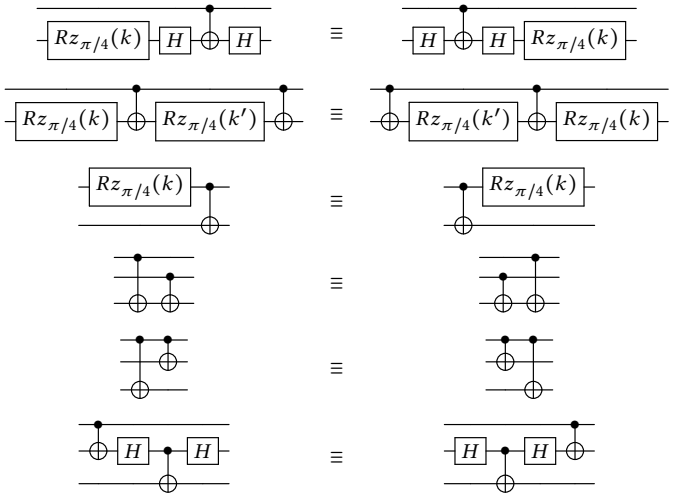


Figure 7. Commutation equivalences for single- and two-qubit gates adapted from Nam et al. [30, Figure 5].

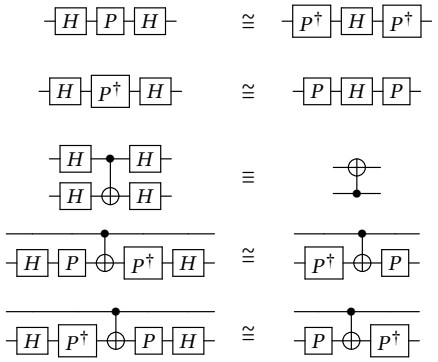


Figure 8. Equivalences for removing Hadamard gates adapted from Nam et al. [30, Figure 4]. P is the phase gate $Rz_{\pi/4}(2)$ and $P^\dagger$ is its inverse $Rz_{\pi/4}(6)$.

commutation rules, we use the rewrite rules shown Figure 7. For cancellation rules, we use the fact that H, X, and $CNOT$ are all self-cancelling and $Rz_{\pi/4}(k)$ and $Rz_{\pi/4}(k')$ combine to become $Rz_{\pi/4}(k + k')$.

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 8 to reduce the number of H gates in the program. Removing H gates is useful because H gates limit the size of the $\{X, Rz_{\pi/4}, CNOT\}$ sub-circuits used in the rotation merging optimization.

Rotation Merging The rotation merging optimization allows for combining $Rz_{\pi/4}$ 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 argument for the correctness of this optimization relies on the *phase polynomial* representation of a circuit. Let C be a circuit consisting of X gates, $CNOT$ gates, and rotations about the z-axis. Then on basis state $|x_1, ..., x_n\rangle$, C will produce the state

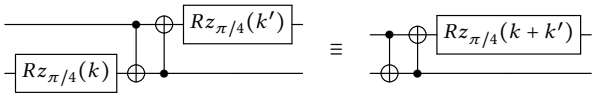
$$e^{ip(x_1,...,x_n)} |h(x_1, ..., x_n)\rangle$$

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

$$p(x_1, ..., x_n) = \sum_{i=1}^{l} (\theta_i \mod 2\pi)\ f_i(x_1, ..., x_n)$$

is a linear combination of affine boolean functions. $p(x_1, ..., x_n)$ 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 $f_i(x_1, ..., x_n) = f_j(x_1, ..., x_n)$ 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 $Rz_{\pi/4}(k)$ to the basis state $|x\rangle$ produces the state $e^{i(k\pi/4)x} |x\rangle$ and $CNOT |x, y\rangle$ produces the state $|x, x \oplus y\rangle$ where $\oplus$ is the xor operation. Evaluation of the left-hand circuit proceeds as follows:

$$\begin{aligned}
|x_1, x_2\rangle &\to e^{i(k\pi/4)x_2} |x_1, x_2\rangle \\
&\to e^{i(k\pi/4)x_2} |x_1, x_1 \oplus x_2\rangle \\
&\to e^{i(k\pi/4)x_2} |x_2, x_1 \oplus x_2\rangle \\
&\to e^{i(k\pi/4)x_2} e^{i(k'\pi/4)x_2} |x_2, x_1 \oplus x_2\rangle .
\end{aligned}$$

Whereas evaluation of the right-hand circuit produces

$$\begin{aligned}
|x_1, x_2\rangle &\to |x_1, x_1 \oplus x_2\rangle \\
&\to |x_2, x_1 \oplus x_2\rangle \\
&\to e^{i((k+k')\pi/4)x_2} |x_2, x_1 \oplus x_2\rangle .
\end{aligned}$$

The two resulting states are equal because $e^{i(k\pi/4)x_2} e^{i(k'\pi/4)x_2} = e^{i((k+k')\pi/4)x_2}$. 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 logic above for arbitrary $\{X, Rz_{\pi/4}, CNOT\}$ 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 rotations $Rz_{\pi/4}(k)$ when they are applied to qubits associated with the same Boolean function. To prove equivalence over $\{X, Rz_{\pi/4}, CNOT\}$ 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 is useful for proving equivalences that are not as direct as those listed in Figures 7 and 8.

Although our merge operation is identical to Nam et al.'s, we apply it to smaller circuits. For ease of verification, we only consider continuous $\{X, Rz_{\pi/4}, CNOT\}$ sub-circuits within the larger program. Nam et al. allow some intervening H gates, provided that those H gates do not impact computation of the phase polynomial. This is a restriction that we plan to relax.

4.5 Scheduling

VOQC applies all of these optimizations with its `optimize` function. It applies them one after the other, in the following order (due to Nam et al. [30]):

$$0, 1, 3, 2, 3, 1, 2, 4, 3, 2$$

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. The rationale for this ordering is that 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.

5 Other Verified Transformations

We have also implemented verified optimizations of non-unitary programs in VOQC (inspired by optimizations in IBM's Qiskit compiler [2]) and verified a transformation that maps a circuit to a connectivity-constrained architecture.

5.1 Non-unitary Optimizations

We have implemented two non-unitary optimizations: removing pre-measurement z rotations, and classical state propagation. For these optimizations, a non-unitary program P is represented as a list of *blocks*. A block is a binary tree, where a leaf is unitary program (in list form), and a node is a measurement `meas` q P_1 P_2 whose children P_1 and P_2 are lists of blocks. Equivalence is defined in terms of the density matrix semantics of the SQIR representation (per Section 3.2).

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. We have implemented and verified an optimization that locates $Rz_{\pi/4}$ gates before measurement operations and removes them. This

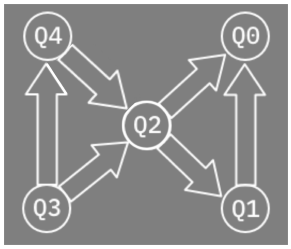


Figure 9. Two-qubit gate connections on IBM's Tenerife machine [21].

optimization was inspired by the RemoveDiagonalGatesBeforeMeasure pass implemented in Qiskit.

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 precomputation 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 analysis applies the following rules:

- $Rz_{\pi/4}(k)\ q$ 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 $X\ q'$.
- `meas` $q\ P_1\ P_0$ becomes P_i.
- $H\ q$ and $CNOT\ q'\ q$ destroy the classical state 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 original program. We express the requirement that qubit q be in classical state $i \in \{0, 1\}$ with the condition

$$|i\rangle_q \langle i| \times \rho \times |i\rangle_q \langle i| = \rho,$$

which says that projecting state ρ 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.

5.2 Circuit Mapping

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 [38, 50]. Circuit mapping algorithms take as input an arbitrary circuit and output a circuit that respects the connectivity constraints of some underlying architecture.

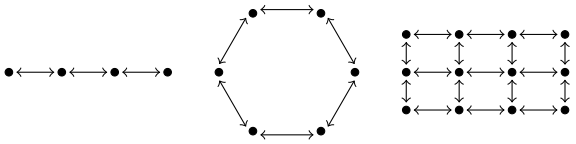


Figure 10. From left to right: LNN, LNN ring, and 2D grid architectures. Each architecture is shown with a fixed number vertices, but in our implementation the number of vertices in these architectures is a parameter. Double-ended arrows indicate that two-qubit gates are possible in both directions.

For example, consider the connectivity of IBMs's five-qubit Tenerife machine, shown in Figure 9. This is a representative example of a modern superconducting qubit system, where qubits are laid out in a 2-dimensional grid and possible interactions are described by directed edges between the qubits. The direction of the edge indicates which qubit can be the control of a two-qubit gate and which can be the target. Thus, no two-qubit gate is possible between physical qubits Q4 and Q1 on the Tenerife. A *CNOT* gate may be applied with Q4 as the control and Q2 as the target, but not the reverse.

We have implemented a simple circuit mapper for sqir programs and verified that it is sound and produces programs that satisfy the relevant hardware constraints. Our circuit mapper is parameterized by functions describing the connectivity of an architecture (in particular, one function that determines whether an edge is in the connectivity graph and another function that finds an undirected path between any two nodes). We map a program to this architecture by adding SWAP operations before and after every *CNOT* so that the target and control are adjacent when the *CNOT* is performed, and are returned to their original positions before the next operation. This algorithm inserts more SWAPs than the optimal solution, but our verification framework could be applied to optimized implementations as well. To handle directionality of edges in the connectivity graph, we make use of the equivalence `H a; H b; CNOT a b; H a; H b` ≡ `CNOT b a`.

We have implemented and verified mapping functions for the Tenerife architecture pictured in Figure 9 and the linear nearest neighbor (LNN), LNN ring, and 2D nearest neighbor architectures pictured in Figure 10.

6 Experimental Evaluation

We evaluate the performance of voqc's verified optimizations against IBM's Qiskit transpiler [2] and the optimizer presented in Nam et al. [30]. We find that voqc's performance is comparable to these state-of-the-art compilers.

Benchmarks We run on a set of benchmarks developed by Amy et al. [5] and evaluate performance by measuring the reduction in gate counts. The benchmarks consist of arithmetic circuits and implementations of multiple-control Toffoli gates. Each circuit contains between 45 and 61,629 gates and uses between 5 and 192 qubits. These benchmarks

only contain unitary circuits, so they only serve to evaluate our unitary circuit optimizations. Some of the benchmarks contain the three-qubit *TOFF* gate. Before applying optimizations we convert *TOFF* gates to VOQC's gate set using the following standard decomposition, where $\mathtt{R}_{\pi/4}(1)$ is the familiar T gate and $\mathtt{R}_{\pi/4}(7)$ is its inverse $T^\dagger$.

```
TOFF a b c :=
  [ H c ; CNOT b c ; R_π/4(7) c ; CNOT a c ;
    R_π/4(1) c ; CNOT b c ; R_π/4(7) c ; CNOT a c ;
    CNOT a b ; R_π/4(7) b ; CNOT a b ;
    R_π/4(1) a ; R_π/4(1) b ; R_π/4(1) c ; H c ].
```

Baseline We compare VOQC's performance with that of Nam et al. and Qiskit version 0.13.0 when run on the same programs. We do not include the results from Amy et al. because their optimization is aimed at reducing a particular type of gate, and often results in a higher total count.

As mentioned in Section 4, VOQC's verified optimizations are inspired by (unverified) ones implemented by Nam et al. [30], and these overlap with (also unverified) optimizations present in IBM's Qiskit compiler [2]. As such, our goal is not to beat the performance of these compilers, but rather to show that VOQC's suite of verified optimizations can achieve performance comparable to the state of the art.

Table 1 performs a direct comparison of functionality. 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 less powerful optimization (as discussed in Section 4.4). Conversely, VOQC's not propagation routine generalizes Nam et al.'s; and VOQC's one- and two-qubit cancellation routines generalize Qiskit's Optimize1qGates and CXCancellation when using 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.

In our experiment we evaluate two settings of Qiskit—*Qiskit A* and *Qiskit B*. The latter is Qiskit with all of its unitary optimizations enabled (i.e., up to optimization level 3). However, the circuits produced by *Qiskit B* are not guaranteed to be (and are most likely not) in VOQC's gate set, so the two are not entirely comparable. *Qiskit B* uses the gate set $\{u_1, u_2, u_3, CNOT\}$ where u_3 is $R_{\theta,\phi,\lambda}$ from VOQC's base set and u_1 and u_2 are u_3 with certain arguments fixed.

Qiskit A is more similar to VOQC in that it uses the gate set $\{H, X, u_1, CNOT\}$ where u_1 corresponds to rotation about the z-axis by an arbitrary angle (which ends up as a multiple of $\pi/4$ in our benchmark). *Qiskit A* includes all unitary program optimizations used up to optimization level 2; it does not include optimizations from level 3 because these optimizations produce circuits with gates outside VOQC's gate set.

Nam2018 et al.	
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)	
Qiskit 0.12.0	
CXCancellation (L_1)	✓*
Optimize1qGates (L_1, L_2, L_3)	✓*
CommutativeCancellation (L_2, L_3)	✓*
ConsolidateBlocks (L_3)	

Table 1. Summary of the unitary optimizations in VOQC as compared to Qiskit [2] and Nam et al. [30].

Results The results are shown in Table 2. In each row, we have marked in bold the gate count of the best-performing optimizer. The average gate count reduction for each optimizer is given in the last row, although performance varies substantially between benchmarks.

On average, Nam et al. [30] heavy optimization reduces the total gate count by 26.5%, *Qiskit A* reduces the total gate count by 4.6%, *Qiskit B* reduces the total gate count by 10.7%, and VOQC reduces the total gate count by 17.7%. VOQC outperforms *Qiskit A* on all benchmarks and outperforms or matches the performance of *Qiskit B* on all benchmarks except two. In 8 out of 29 cases VOQC outperforms Nam et al.'s heavy optimization.

The gap in performance between VOQC and Qiskit is primarily due to VOQC's rotation merging optimization, which has no analogue in Qiskit. The gap in performance between Nam et al. and VOQC is due in part to the fact that we have not yet implemented all their optimization passes (per Table 1). We see no fundamental difficulties in implementing these, but we expect the biggest performance boost will come from generalizing our rotation merging optimization to consider larger sub-circuits, which will require some additional verification effort.

These results are encouraging evidence that VOQC supports useful and interesting verified optimizations.

7 Related Work

Verified Quantum Programming We designed SQIR primarily as the intermediate language for VOQC's verified optimizations, but we find it adequate for verified source programming as well (per Section 3.4 and Appendix A).

Benchmark Name	Original	Nam (L)	Nam (H)	Qiskit A	Qiskit B	VOQC
adder_8	900	646	**606**	869	805	752
barenco_tof_10	450	294	**264**	427	394	380
barenco_tof_3	58	42	**40**	56	51	51
barenco_tof_4	114	78	**72**	109	100	98
barenco_tof_5	170	114	**104**	162	149	145
csla_mux_3	170	161	**155**	168	156	160
csum_mux_9	420	294	**266**	420	382	308
gf2^4_mult	225	**187**	**187**	213	206	192
gf2^5_mult	347	296	296	327	318	**291**
gf2^6_mult	495	**403**	**403**	465	454	410
gf2^7_mult	669	555	555	627	614	**549**
gf2^8_mult	883	712	712	819	804	**705**
gf2^9_mult	1095	891	891	1023	1006	**885**
gf2^10_mult	1347	**1070**	**1070**	1257	1238	1084
gf2^16_mult	3435	2707	2707	3179	3148	**2695**
gf2^32_mult	13562	10601	10601	12569	12506	**10577**
gf2^64_mult	61629	41563	N/A	49659	49532	**41515**
mod5_4	63	**51**	**51**	62	58	57
mod_mult_55	119	91	91	117	106	**90**
mod_red_21	278	184	**180**	261	227	214
qcla_adder_10	521	411	**399**	512	469	474
qcla_com_7	443	**284**	**284**	428	398	335
qcla_mod_7	884	636	**624**	853	793	764
rc_adder_6	200	142	**140**	195	170	167
tof_10	255	**175**	**175**	247	222	215
tof_3	45	**35**	**35**	44	40	40
tof_4	75	**55**	**55**	73	66	65
tof_5	105	**75**	**75**	102	92	90
vbe_adder_3	150	**89**	**89**	146	138	103
Average Reduction	–	25.2%	26.5%	4.6%	10.7%	17.7%

Table 2. Reduced gate counts on the Amy et al. [5] benchmarks.

Several lines of work have explored formally verifying aspects of a quantum computation. The earliest attempts to do so in a proof assistant were Green's Agda implementation of the Quantum IO Monad [18] and a small Coq quantum library by Boender et al. [7]. These were both proofs of concept, and neither developed beyond verifying basic protocols.

The high-level QWIRE programming language is, like SQIR, embedded in the Coq proof assistant, and has been used to verify a variety of simple programs [35], assertions regarding ancilla qubits [34], and its own metatheory [33]. VOQC and SQIR reuse parts of QWIRE's Coq development, and take inspiration and lessons from its design. However, as discussed in Section 3.4 and Appendix B, QWIRE's higher-level abstractions complicate verification. Moreover, such abstractions do not reflect the kind of quantum programming we can expect to do in the near future. For example, a key element of QWIRE is *dynamic lifting*, which permits measuring a qubit and using the result as a Boolean value in the host language to compute the remainder of a circuit [17]. Today's quantum computers cannot reliably exchange information between a (typically supercooled) quantum chip and a classical computer before qubits decohere. Thus, practically-minded languages like IBM's OpenQASM [12] only allow for a limited form of branching that is close to SQIR's.

Another line of work, pioneered by D'Hondt and Panangaden [15] and Ying [48], uses program logics to reason about quantum programs. These logics allow proof of a variety of program properties inside a formal deductive system. Liu et al. [27] implemented Ying's quantum Hoare logic inside the Isabelle proof assistant and used it to prove the correctness of Grover's algorithm, and Unruh [47] developed a *relational* quantum Hoare logic and built an Isabelle-based tool to prove the security of quantum cryptosystems. Implementing these kinds of logics in Coq and proving them correct with respect to SQIR's denotational semantics may prove useful (though we have proved several interesting SQIR programs correct directly).

Verified Quantum Compilation Quantum compilation is an active area. In addition to Qiskit and Nam et al. [30]

(discussed in Section 6), other recent compiler efforts include t|ket⟩ [10, 11], quilc [37], ScaffCC [22], and Project Q [44]. 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. [6] developed a certified optimizing compiler from source Boolean expressions to reversible circuits, but did not handle general quantum programs. Rand et al. [34] developed a similar compiler for quantum circuits but without optimizations (using the QWIRE language).

The problem of quantum program optimization verification has previously been considered in the context of the ZX-calculus [8], which is a formalism for describing quantum computation based on categorical quantum mechanics [1]. 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. The Quantomatic tool [16] automatically rewrites ZX diagrams according to this set of rules, but suffers from two limitations as a quantum compiler: Its rewriting procedures are not guaranteed to terminate and not every ZX diagram corresponds to a valid quantum circuit. PyZX [24] addresses both of these limitations, using the ZX-calculus as an intermediate representation for compiling quantum circuits, and generally achieving performance comparable to leading compilers. 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 performs translation validation on its compiled circuits, checking (where feasible) that the compiled circuit is equivalent to the original.

A recent paper from Smith and Thornton [41] presents a compiler with built-in translation validation via QMDD equivalence checking [29]. However the optimizations they consider are much simpler than ours 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. [40] 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 can 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.

8 Conclusions and Future Work

This paper has presented VOQC, the first verified optimizer for quantum circuits implemented within a proof assistant. A key component of VOQC is SQIR, a simple, low-level quantum language deeply embedded in the the Coq proof assistant. Compiler passes are expressed as Coq functions which are proved to preserve the semantics of their input SQIR programs. VOQC'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 VOQC are proved correct. When applied to a benchmark suite of 29 circuit programs, we found VOQC performed comparably to state of the art compilers, reducing gate counts on average by 17.7% compared to 10.7% for IBM's Qiskit compiler, and 26.5% for the cutting-edge research compiler of Nam et al. [30].

Moving forward, we plan to incorporate VOQC into a full-featured verified compilation stack for quantum programs, following the vision of a recent Computing Community Consortium report [28]. We can implement validated parsers [23] for languages like OpenQASM and verify their translation to SQIR (e.g., using metaQASM's semantics [4]). 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 [43]. We leave it as future work to incorporate optimizations and mapping algorithms from additional compilers into VOQC. Our experience so far makes us optimistic about the prospects for doing so successfully.

Acknowledgments

We thank Leonidas Lampropoulos and Kartik Singhal for 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

[1] Samson Abramsky and Bob Coecke. 2009. Categorical quantum mechanics. *Handbook of quantum logic and quantum structures* 2 (2009), 261–325.

[2] 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, Łukasz 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 Martín-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 Prutyanov, 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

[3] Thorsten Altenkirch and Alexander S Green. 2010. The quantum IO monad. *Semantic Techniques in Quantum Computation* (2010), 173–205.

[4] 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.

[5] 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

[6] 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/

[7] 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)*, Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.), Vol. 195. Open Publishing Association, 71–83. https://doi.org/10.4204/EPTCS.195.6

[8] Bob Coecke and Ross Duncan. 2011. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics* 13, 4 (2011), 043016.

[9] The Coq Development Team. 2019. The Coq Proof Assistant, version 8.10.0. https://doi.org/10.5281/zenodo.3476303

[10] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. 2019. On the qubit routing problem. (Feb 2019). arXiv:quant-ph/1902.08091

[11] Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. 2019. Phase Gadget Synthesis for Shallow Circuits. (Jun 2019). arXiv:quant-ph/1906.01734

[12] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. *arXiv e-prints* (Jul 2017). arXiv:quant-ph/1707.03429

[13] 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.

[14] David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439, 1907 (1992), 553–558.

[15] Ellie D'Hondt and Prakash Panangaden. 2006. Quantum weakest preconditions. *Mathematical Structures in Computer Science* 16, 03 (2006), 429–451.

[16] 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*.

[17] 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.

[18] Alexander S Green. 2010. *Towards a formally verified functional quantum programming language*. Ph.D. Dissertation. University of Nottingham.

[19] 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

[20] Luke Heyfron and Earl T. Campbell. 2017. An Efficient Quantum Compiler that reduces T count. *Quantum Science and Technology* 4 (12 2017). https://doi.org/10.1088/2058-9565/aad604

[21] IBM. [n. d.]. IBM Q 5 Tenerife V1.x.x Version Log. https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md

[22] Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, 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 (CF '14)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2597917.2597939

[23] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP)*.

[24] Aleks Kissinger and John van de Wetering. 2019. PyZX: Large Scale Automated Diagrammatic Reasoning. In *Proceedings of the 16th International Conference on Quantum Physics and Logic, QPL 2019*.

[25] Emmanuel Knill. 1996. *Conventions for quantum pseudocode*. Technical Report. Los Alamos National Lab., NM (United States).

[26] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10/c9sb7q

[27] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Computer Aided Verification (CAV)*. Springer International Publishing.

[28] Margaret Martonosi and Martin Roetteler. 2019. Next Steps in Quantum Computing: Computer Science's Role. arXiv:cs.ET/1903.10541

[29] 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] 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

[31] 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 (POPL 2017)*. ACM, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

[32] Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

[33] Robert Rand. 2018. *Formally Verified Quantum Programming*. Ph.D. Dissertation. University of Pennsylvania.

[34] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2018. 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*.

[35] 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

[36] Rigetti Computing. 2019. Pyquil Documentation. http://pyquil.readthedocs.io/en/latest/

[37] Rigetti Computing. 2019. The @rigetti optimizing Quil compiler. https://github.com/rigetti/quilc

[38] 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

[39] Peter Selinger. 2004. Towards a Quantum Programming Language. *Mathematical Structures in Computer Science* 14, 4 (Aug. 2004), 527–586.

[40] 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:quant-ph/1908.08963

[41] 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)*.

[42] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A Practical Quantum Instruction Set Architecture. *arXiv e-prints* (Aug 2016). arXiv:quant-ph/1608.03355

[43] 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/

[44] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (2018), 49.

[45] 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.

[46] The Cirq Developers. 2019. Cirq: A python library for NISQ circuits. https://cirq.readthedocs.io/en/stable/

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

[48] Mingsheng Ying. 2011. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 6 (2011), 19.

[49] Vladimir Zamdzhiev. 2016. Quantum Computing: The Good, the Bad, and the (not so) ugly! Invited talk, Tulane University.

[50] 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:quant-ph/1712.04722

```
Definition a : ℕ := 0.
Definition b : ℕ := 1.

Definition bell00 := H a; CNOT a b.

Definition encode (b1 b2 : 𝔹) :=
    (if b2 then X a else I a);
    (if b1 then Z a else I a).

Definition decode := CNOT a b; H a.

Definition superdense (b1 b2 : 𝔹) :=
    bell00 ; encode b1 b2; decode.
```

Figure 12. SQIR program for the unitary portion of the superdense coding algorithm. We have removed type annotations for clarity, but each SQIR program has type `ucom base 2`, which describes a unitary circuits that uses the base gate set and has a global register of dimension two.

A SQIR for General Verification

SQIR's simple structure and semantics allow us to easily verify general properties of quantum programs. In this section we discuss correctness properties we have proved of four simple quantum programs written in SQIR.

A.1 Superdense Coding

Superdense coding is a protocol that allows a sender to transmit two classical bits, b_1 and b_2, to a receiver using a single quantum bit. The circuit for superdense coding is shown in Figure 11. The SQIR program corresponding to the unitary part of this circuit is shown in Figure 12. Note that in the SQIR program, encode is a Coq function that takes two Boolean values and returns a circuit.

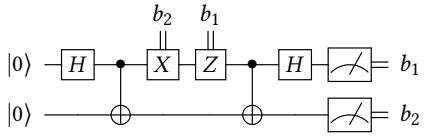


Figure 11. Circuit for superdense coding.

We can prove that the result of evaluating the program `superdense b1 b2` on an input state consisting of two qubits initialized to zero is the state $|b_1, b_2\rangle$.

```
Lemma superdense_correct : ∀ b1 b2,
  ⟦superdense b1 b2⟧₂ × | 0,0 ⟩ = | b1,b2 ⟩.
```

A.2 GHZ State Preparation

The Greenberger-Horne-Zeilinger (GHZ) state [19] is an n-qubit entangled quantum state of the form

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

This vector can be defined in Coq as follows:

```
Definition ghz (n : ℕ) : Matrix (2 ^ n) 1 :=
  match n with
  | 0 ⇒ I 1
  | S n' ⇒ 1/√2 * nket n |0⟩ + 1/√2 * nket n |1⟩
  end.
```

Above, `nket n |i⟩` is the tensor product of n copies of the basis vector $|i\rangle$. The GHZ state can be prepared by a circuit that begins with all qubits initialized to the $|0\rangle$ state, applies an H to the first qubit (yielding a $|+\rangle$), and then sequentially applies a *CNOT* from each qubit to the next. A circuit that prepares the 3-qubit GHZ state is shown in Figure 1(a) and the SQIR description of this circuit can be produced by the recursive function in Figure 1(d).

The function `GHZ` describes a *family* of SQIR circuits: For every n, `GHZ n` is a valid SQIR program and quantum circuit.[2] We aim to show via an inductive proof that every circuit generated by `GHZ n` produces the corresponding `ghz n` vector when applied to $|0\ldots0\rangle$. We prove the following theorem:

[2] For the sake of readability, Figure 1(d) elides a coercion from `ucom base n'` to `ucom base n` in the recursive case.

```
Theorem ghz_correct : ∀ n : ℕ,
  ⟦GHZ n⟧ᵤ × nket n |0⟩ = ghz n.
```

The proof applies induction on n. For the base case, we show H applied to $|0\rangle$ produces the $|+\rangle$ state. For the inductive step, the induction hypothesis says that the result of applying `GHZ n'` to the input state `nket n' |0⟩` produces the state

`(1/√2 * nket n' |0⟩ + 1/√2 * nket n' |1⟩) ⊗ |0⟩`.

By applying `CNOT (n'-1) n'` to this state, we can show that `GHZ (n'+1)` produces $|\mathrm{GHZ}^{n'+1}\rangle$.

A.3 Teleportation

The circuit for quantum teleportation is given in Figure 13. In the quantum teleportation protocol, Alice communicates state $|\psi\rangle$ from wire 0 (on the top) to Bob on wire 2 (on the bottom). The protocol begins by constructing a bell pair on wires 1 and 2, with the first element of the pair given to Alice and the second given to Bob. Alice then entangles $|\psi\rangle$ with wire 1 and measures both wires, outputting a pair of bits. Bob uses these measurement results to transform his qubit on wire 2 into $|\psi\rangle$.

The circuit in Figure 13 corresponds to the following SQIR program:

```
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.
```

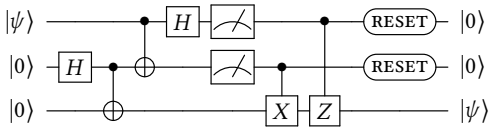


Figure 13. Circuit for quantum teleportation.

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 wires 0 and 1 and then resets these wires 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 $|0\rangle\langle 0| \otimes |0\rangle\langle 0| \otimes \rho$. Formally,

```
Lemma teleport_correct : ∀ (ρ : Density 2),
 WF_Matrix ρ →
 ⟦teleport⟧_d (ρ ⊗ |0⟩⟨0| ⊗ |0⟩⟨0|) = |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ ρ.
```

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

A.4 Nondeterministic Semantics

The proof of the correctness of `teleport` for the density matrix semantics is simple but not useful for understanding *why* the protocol is correct. A more illuminating proof can be carried out with an alternative *nondeterministic semantics* in which evaluation is given as a relation. Given a state ψ, unitary program u will (deterministically) evaluate to $[\![u]\!]_d \times \psi$. However, `meas q p1 p2` may evaluate to either `p1` applied to $|1\rangle\langle 1| \times \psi$ or `p2` applied to $|0\rangle\langle 0| \times \psi$. We have found the nondeterministic semantics simpler to work with for certain types of proofs.

However, because we do not rescale the output of measurement (to avoid reasoning about matrix norms in Coq), the nondeterministic semantics is only useful for proving properties for which measurement outcome does not matter. The correctness of the teleport protocol above is an example of such a property, because the values measured by Alice do not impact the final output state. The nondeterministic semantics cannot be used for verifying soundness of non-unitary transformations in VOQC because equivalence between programs requires equality between output probability distributions.

For the non-deterministic semantics the proof of teleport is more involved, but also more illustrative of the inner workings of the algorithm. Under the non-deterministic semantics, we aim to prove the following:

```
Lemma teleport_correct :
  ∀ (ψ : Vector (2^1)) (ψ' : Vector (2^3)),
  WF_Matrix ψ →
  teleport / (ψ  ⊗ |0,0⟩) ⇓ ψ' →
  ψ' ∝ |0,0⟩ ⊗ ψ.
```

This says that on input $|\psi\rangle \otimes |0,0\rangle$, `teleport` will produce a state that is proportional to ($\propto$) $|0,0\rangle \otimes |\psi\rangle$. Note that this statement is quantified over every outcome ψ' and hence all possible paths to ψ'. If instead we simply claimed that

```
teleport / (ψ  ⊗ |0,0⟩) ⇓ 1/2 * (|0,0⟩ ⊗ ψ),
```

where the 1/2 factor reflects the probability of each measurement outcome ($(1/2)^2 = 1/4$), we would only be stating that some such path exists.

The first half of the circuit is unitary, so we can simply compute the effect of applying a H gate, two $CNOT$ gates and another H gate to the input state. We can then take both measurement steps, leaving us with four different cases to prove correct. In each of the four cases, we can use the outcomes of measurement to correct the final qubit, putting it into the state $|\psi\rangle$. Finally, resetting the already-measured qubits is deterministic, and leaves us in the desired state.

A.5 The Deutsch-Jozsa Algorithm

In the quantum query model, we are given access to a Boolean function $f : \{0,1\}^n \to \{0,1\}$ through an oracle defined by the map $U_f : |y,x\rangle \mapsto |y \oplus f(x), x\rangle$. For a function f on n bits, the unitary matrix U_f is a linear operator over a 2^{n+1} dimensional Hilbert space. In order to describe the Deutsch-Jozsa algorithm in SQIR, we must first give a SQIR definition of oracles.

To begin, note that any n-bit Boolean function f can be written as

$$f(x_1,\ldots,x_n) = \begin{cases} f_0(x_1,\ldots,x_{n-1}) & \text{if } x_n = 0 \\ f_1(x_1,\ldots,x_{n-1}) & \text{if } x_n = 1 \end{cases}$$

where $f_b(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},b)$ is a Boolean function on $(n-1)$ bits for $b \in \{0,1\}$. Similarly, an oracle can be written as $U_f = U_{f_0} \otimes |0\rangle\langle 0| + U_{f_1} \otimes |1\rangle\langle 1|$ for $U_f\,|y,x_1,\ldots,x_{n-1},b\rangle = U_{f_b}\,|y,x_1,\ldots,x_{n-1}\rangle\,|b\rangle$. In the base case ($n = 0$), a Boolean function is a constant function of the form $f(\perp) = 0$ or $f(\perp) = 1$ and an oracle is either the identity matrix, i.e., $|y\rangle \mapsto |y\rangle$, or a Pauli-$X$ matrix, i.e., $|y\rangle \mapsto |y \oplus 1\rangle$. As a concrete example, consider the following correspondences between the 1-bit Boolean functions and 4×4 unitary matrices:

$$\begin{aligned} f_{00}(x) &= 0 & U_{f_{00}} &= I \otimes |0\rangle\langle 0| + I \otimes |1\rangle\langle 1|, \\ f_{01}(x) &= 1 - x & U_{f_{01}} &= X \otimes |0\rangle\langle 0| + I \otimes |1\rangle\langle 1|, \\ f_{10}(x) &= x & U_{f_{10}} &= I \otimes |0\rangle\langle 0| + X \otimes |1\rangle\langle 1|, \\ f_{11}(x) &= 1 & U_{f_{11}} &= X \otimes |0\rangle\langle 0| + X \otimes |1\rangle\langle 1|. \end{aligned}$$

The observation above enables the following inductive definition of an oracle.

```
Inductive boolean : ∀ dim, ucom dim → Set :=
  | boolean_I : ∀ u, u ≡ I 0 → boolean 1 u
  | boolean_X : ∀ u, u ≡ X 0 → boolean 1 u
  | boolean_U : ∀ dim u u1 u2,
                boolean dim u1 →
```

```
Fixpoint cpar n (u : ℕ → ucom base n) :=
  match n with
  | 0 ⇒ uskip
  | S n' ⇒ cpar n' u ; u n'
  end.
Definition deutsch_jozsa n (U : ucom base n) :=
  X 0 ; cpar n H ; U ; cpar n H.
```

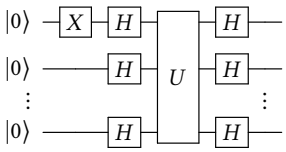


Figure 14. The Deutsch-Jozsa algorithm in SQIR and as a circuit.

$$\begin{aligned}&\texttt{boolean dim u2} \to\\ &[\![\texttt{u}]\!]_{1+dim} = [\![\texttt{u1}]\!]_{dim} \otimes |0\rangle\langle 0| \ .+\\ &\qquad\qquad [\![\texttt{u2}]\!]_{dim} \otimes |1\rangle\langle 1| \to\\ &\texttt{boolean (1 + dim) u.}\end{aligned}$$

`boolean dim U` describes an oracle for a `dim`–1-bit Boolean function whose denotation is a $2^{dim} \times 2^{dim}$ unitary matrix.

A Boolean function is balanced if the number of inputs that evaluate to 1 is exactly half of the domain size. A Boolean function is constant if for all inputs, the function evaluates to the same output, i.e., $\forall x.\ f(x) = 0$ or $\forall x.\ f(x) = 1$. Given an oracle, we can determine whether it describes a balanced or constant function by counting the number of inputs that evaluate to 1.

```
Fixpoint count {dim : ℕ} {U : ucom base dim} (P :
    boolean dim U) : ℕ :=
  match P with
  | boolean_I _ _ ⇒ 0
  | boolean_X _ _ ⇒ 1
  | boolean_U _ _ _ _ P1 P2 _ ⇒ count P1 + count P2
  end.
```

We define balanced and constant oracles in SQIR as follows.

```
Definition balanced {dim : ℕ} {U : ucom base dim}
  (P : boolean dim U) : ℙ :=
  count P = 2 ^ (dim - 2).
Definition constant {dim : ℕ} {U : ucom base dim}
  (P : boolean dim U) : ℙ :=
  count P = 0 ∨ count P = 2 ^ (dim - 1).
```

In the Deutsch-Jozsa [1992] problem, we are promised that the function f is either balanced or constant, and the goal is to decide which is the case by querying the oracle. The Deutsch-Jozsa algorithm begins with an all $|0\rangle$ state, and prepares the input state $|-\rangle \otimes$ `nket dim` $|+\rangle$. This state is prepared by applying an X gate on the first qubit, and then applying a H gate to every qubit in the program. Next, the oracle U is queried, and a H gate is again applied to every qubit in the program. Finally, all qubits except the first are measured in the standard basis. This algorithm is shown as a circuit and in SQIR in Figure 14. Note the use of Coq function `cpar`, which constructs a SQIR program that applies the same operation to every qubit in the program.

If measuring all the qubits after the first yields an all-zero string, then the algorithm outputs "accept," which indicates that the function is constant. Otherwise the algorithm outputs "reject". Instead of manually doing to the measurement, we will mathematically describe the output. Formally, the algorithm will output "accept" when the output state is supported on $\Pi = I \otimes |0\rangle\langle 0|^{\otimes dim}$ and output "reject" when the output state is orthogonal to Π. We can express this in Coq as follows:

```
Definition accept {dim : ℕ} {U : ucom base dim}
  (P : boolean dim U) : ℙ :=
  ∃ (ψ : Matrix 2 1),
  ((ψ ⊗ nket (dim-1) |0⟩)† × ⟦deutsch_jozsa dim U⟧u ×
     (nket dim |0⟩)) 0 0 = 1.
Definition reject {dim : ℕ} {U : ucom base}
  (P : boolean dim U) : ℙ :=
  ∀ (ψ : Matrix 2 1), WF_Matrix ψ →
 ((ψ ⊗ nket (dim-1) |0⟩)† × ⟦deutsch_jozsa dim U⟧u ×
    (nket dim |0⟩)) 0 0 = 0.
```

We now prove the following theorems.

```
Theorem deutsch_jozsa_constant_correct :
 ∀ (dim : ℕ) (U : ucom base dim) (P : boolean dim U),
     constant P → accept P.
Theorem deutsch_jozsa_balanced_correct :
 ∀ (dim : ℕ) (U : ucom base dim) (P : boolean dim U),
     balanced P → reject P.
```

The key lemma in our proof states that the probability of outputting "accept" depends on the number of inputs that evaluate to 1, i.e., `count P`.

```
Lemma deutsch_jozsa_success_probability :
 ∀ {dim : ℕ} {U : ucom base dim} (P : boolean dim U)
    (ψ : Matrix 2 1) (WF : WF_Matrix ψ),
    (ψ ⊗ nket (dim-1) |0⟩)† × ⟦deutsch_jozsa dim U⟧u
    × (nket dim |0⟩))
    = (1 - 4 * count P * /2 ^ dim) .* (ψ† × |1⟩).
```

This lemma is proved by induction on `P`, which is the proof that `U` is a Boolean oracle. We sketch the structure of the proof below, using mathematical notation for ease of presentation.

In the base case, either `U` ≡ `I 0` or `U` ≡ `X 0`, the former of which is constant, and the latter is balanced. The lemma holds for the base case since $\langle\psi|HIH|1\rangle = \langle\psi|1\rangle$ for `U` ≡ `I 0` and $\langle\psi|HXH|1\rangle = -\langle\psi|1\rangle$ for `U` ≡ `X 0`. Thus the factor can be written as `1 − 2 * count P`.

For the inductive step, the inductive hypothesis says that, for any Boolean function of `dim − 1` bits, the factor is `1 − 4 * count P / 2 ^ dim`. We observe that with probability 1/2, the bit x_{dim} input to the oracle of a `dim`-bit Boolean function f

is either 0 or 1. Conditioned on the first bit being 0 (resp. 1), the function is f_0 (resp. f_1). Let s_b the number of inputs evaluating to 1 when $x_{dim} = b \in \{0, 1\}$. Thus the sign can be calculated as

$$\frac{1}{2}\left(1 - \frac{s_0}{2^{dim-2}}\right) + \frac{1}{2}\left(1 - \frac{s_1}{2^{dim-2}}\right) = 1 - \frac{s_0 + s_1}{2^{dim-1}}. \quad (2)$$

Since the number of inputs evaluating to 1 is $s_0 + s_1$ for f, we conclude the lemma.

For a balanced function, `count P` is equal to `2 ^ (dim - 2)`. For a constant function, `count P` is either `0` or `2 ^ (dim - 1)`. Thus the factor `1 - 4 * count P / 2 ^ dim` is 0 for a balanced function and ±1 for a constant function. For a balanced function, no input state ψ can succeed with nonzero probability, and thus `reject P` is true, whereas for a constant function, one can present a state to maximize the probability to 1, and thus `accept P` can be proved.

B QWIRE vs. SQIR

When we first set out to build VOQC, we thought to do it using QWIRE [31], 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?

As discussed in Section 3.4, the fundamental difference between SQIR and QWIRE is that SQIR relies on a global register of qubits. Every operation is applied to an explicit set of qubits within the global register. By contrast, QWIRE uses Higher Order Abstract Syntax [32] to take advantage of Coq variable binding and function composition. This difference is most noticeable in how the two languages 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 and have a corresponding wire type `w`, corresponding to some collection of bits and qubits. The definition of `gate` takes in a parameterized `Gate`, an appropriate input pattern, and a *continuation* of the form `Pat w2 → Circuit w`, which is a placeholder for the next gate to connect to. This is evident in the definition of the composition function:

```
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 the `gate` case, the continuation is applied directly to the output of the first circuit.

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.
```

This representation allows for easy composition: Any two circuits with matching input and output types can easily be combined using standard function application. For example, consider the following convenient functions for sequential and parallel composition of closed terms:

```
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. Since the denotation of a circuit must be (in the unitary case) a square matrix of size 2^n for some n, we need to map all of our variables to 0 through $n - 1$, ensuring that the mapping function has no gaps even when we initialize or discard qubits. We maintain this invariant through compiling to a de Bruijn-style variable representation [13]. Reasoning about the denotation of our circuits, then, involves reasoning about this compilation procedure. In the case of open circuits (our 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. For example, consider the following code, which composes two SQIR programs in parallel.

```
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.
```

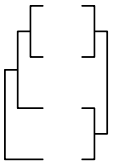


Figure 15. The patterns for the output of GHZ state preparation ghz on a list of 4 qubits and the controlled swap on a tree of 4 qubits fredkin_seq. The left part describes the output pattern (((q1,q2),q3),q4) of ghz, and the right part describes the pattern ((q1,q2),(q3,q4)) of fredkin_seq. In composition, the mismatching patterns require an extra gadget to transform the former into the latter.

```
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.

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 →
  ⟦inPar c1 c2 d1⟧_{d1+d2} = ⟦c1⟧_{d1} ⊗ ⟦c2⟧_{d2}.
```

The inPar function is relatively simple, but more involved than the corresponding QWIRE definition because it requires relabeling the qubits in program c_2.

General composition in SQIR requires even more involved relabeling functions that are less straightforward to describe. For example, consider the composition expressed in the following QWIRE 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 QWIRE, 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 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.

Quantum Data Structures SQIR also lacks some other useful features present in higher-level languages. For example, in QIO [3] and Quipper [17] 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 15, the GHZ program written in QWIRE emits a list of qubits while the fredkin_seq circuit takes in a tree of qubits. Connecting the wires from a GHZ to 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. [17]. 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. [31] 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 `measure` 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, though in practice we will often reason about families of circuits described in Coq.

Other Differences Another important difference between $\mathcal{Q}$WIRE and sQIR is that $\mathcal{Q}$WIRE circuits cannot be easily decomposed into smaller circuits because output variables are bound in different places in the circuit. By contrast, a sQIR program is an arbitrary nesting of smaller programs, and `c1;((c2;(c3;c4));c5)` is equivalent to `c1;c2;c3;c4;c5` under all semantics, whereas every $\mathcal{Q}$WIRE circuit (only) associates to the right. As such, rewriting using sQIR identities is substantially easier.

The differences between these tools stem from the fact that $\mathcal{Q}$WIRE was developed as a programming language for quantum computers [31], and was later used as a verification tool [34, 35]. By contrast, sQIR is mainly a tool for verifying quantum programs, ideally compiled from another language such as Q# [45], Quipper [17] or even $\mathcal{Q}$WIRE itself.