A Verified Optimizer for Quantum Circuits

KESHA HIETALA, University of Maryland
ROBERT RAND, University of Maryland
SHIH-HAN HUNG, University of Maryland
XIAODI WU, University of Maryland
MICHAEL HICKS, University of Maryland

We present VOQC, the first 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. SQIR is also sufficiently expressive write source programs and prove them correct. 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 8.4% (and rotation gates on average by 16.1%) on a benchmark of 29 circuit programs compared to a 4.7% reduction (resp, a 8.3% reduction) when using IBM's Qiskit compiler at optimization level 2.

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

1 INTRODUCTION

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

Of course, both compiler and by-hand optimizations will inevitably lead to mistakes. For example, higher optimization levels of IBM's Qiskit compiler are known to have bugs (as is evident from its issue tracker[1]). Kissinger and van de Wetering [2019] discovered mistakes in the optimized outputs produced by the circuit compiler of Nam et al. [2018]. And Nam et al. themselves found that the optimization library they compared against sometimes produced incorrect results. This is not surprising; quantum computing is regarded as unintuitive (as put well by Zamdzhiev [2016]).

Unfortunately, we cannot apply standard software assurance techniques to find and fix bugs. Unit testing and debugging are infeasible due to both the indeterminacy of quantum algorithms and the substantial expense involved in executing or simulating them. Indeed, it may be impossible to test an optimizer by comparing runs of the source program to its optimized version because resources are 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 a optimization or algorithm always does what it is intended to do. CompCert [Leroy et al. 2004] is a *certified* compiler for C programs that is written and *proved correct* using the Coq proof assistant [Coq Development Team 2019]. CompCert includes sophisticated optimizations whose proofs of correctness are verified to be valid by Coq's type checker. Can we do the same for quantum programs? This paper takes a step toward answering that question in the affirmative.

In this paper, 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

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

was designed to be a *simple quantum intermediate representation*, it is not very different from languages such as PyQuil [Rigetti Computing 2019] or frameworks like Qiskit [Aleksandrowicz et al. 2019], 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, a standard format for quantum circuits [Cross et al. 2017]. (Section 2.)

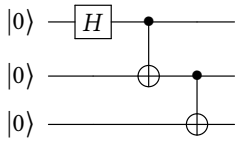
Like CompCert, VOQC is implemented using the Coq proof assistant, with SQIR implemented as a deeply embedded language and optimizations implemented as Coq functions that are then extracted to OCaml. We define several semantics for SQIR programs. The simplest denotes every quantum circuit as a *unitary matrix*, a particular kind of square matrix. However this is only applicable to unitary circuits, i.e. circuits without measurement. For non-unitary circuits, we provide a (nondeterministic) relation among quantum states as well as a denotation of *density matrices*, which encode nondeterminism in the matrices themselves. Properties of SQIR programs, or transformations of them, can be proved using whichever semantics is most convenient. 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 [Pfenning and Elliott 1988], 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 a handful of transformations over SQIR programs. In particular, we have developed several peephole optimizations that apply small circuit identities to reduce overall gate count of the circuit. These were inspired by, and generalize, single- and two-qubit gate cancellation routines in a compiler developed by Nam et al. [2018] and IBM's Qiskit compiler [Aleksandrowicz et al. 2019]. We also developed a circuit mapping routine that transforms SQIR programs to satisfy constraints on how qubits may interact. These transformations were reasonably straightforward to prove correct thanks to SQIR's design. We find that VOQC performs comparably to unverified compilers when run on a benchmark of 29 circuit programs developed by Nam et al. [2018] and Amy et al. [2013]. These programs range from 45 and 61,629 gates and use between 5 and 192 qubits. VOQC reduced total gate counts on average by 8.4% compared to 4.7% by Qiskit at optimization level 2. There is room for improvement: Nam et al. [2018] produced reductions of 25.2% using additional optimizations we expect we could verify. (Section 4.)

As mentioned above, SQIR is not only useful as the target of compilation and optimization. Its simple structure and semantics assists in proving correctness properties about programs written in SQIR directly. In particular, we have proved that the SQIR program to prepare a GHZ state indeed produces the correct state, and showed the correctness of quantum teleportation and the Deutsch-Jozsa algorithm. (Section 5.)

As far as we are aware, our SQIR-based transformations are the first certified-correct optimizations applied to a realistic quantum circuit language. Amy et al. [2017] developed a verified optimizing compiler from source Boolean expressions to reversible circuits, but did not handle general quantum programs. Rand et al. [2018] developed a similar compiler for quantum circuits but without optimizations. Other low-level quantum languages [Cross et al. 2017; Smith et al. 2016] have not been developed with verification in mind, and prior circuit-level optimizations [Amy et al. 2013; Heyfron and T. Campbell 2017; Nam et al. 2018] have not been formally verified. (Related work is covered in detail in section 6.)

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 additional interesting program transformations. All code we reference in this paper can be found on-line (URL anonymized for blind review).



(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)

Fig. 1. Example quantum program: GHZ state preparation

2 OVERVIEW

We start with some brief background about 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 $\langle \alpha, \beta \rangle$ 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 both $|0\rangle$ 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*, in which case 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 $|1\rangle$. This way, all subsequent measurements return the same answer.

Operators on quantum states are linear mappings. Operators can be expressed as matrices, and their application to a state is expressed as matrix multiplication. For example, the famous *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 $|+\rangle$. A quantum operator is not only linear, it must also be *unitary*—the conjugate transpose (or adjoint) of its matrix must be its inverse. This ensures that multiplying a qubit preserves the sum of its squares. 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 quantum computer of around 50 qubits using the most powerful classical computer!

N-qubit evolution operators are represented as $2^N \times 2^N$ (unitary) 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.

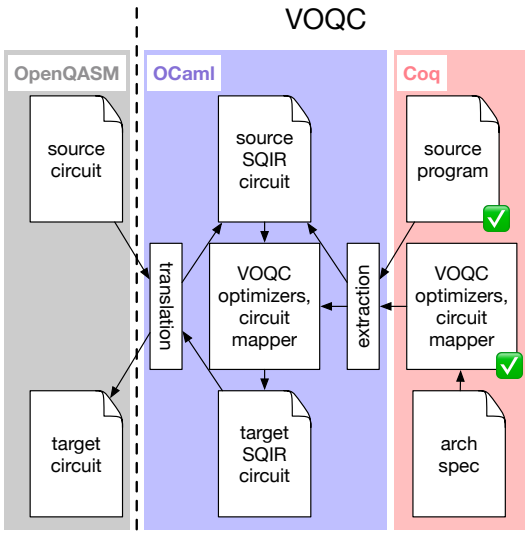


Fig. 2. voqc architecture

2.2 Quantum Circuits

Quantum programs are typically expressed as circuits; an example is shown in fig. 1(a). In these circuits, each horizontal wire represents a *qubit* and boxes on these wires indicate *quantum gates*. For multiple-qubit gates, the inputs are often distinguished as being either a *target* or a *control*. In software, these circuits are often represented using lists of instructions that describe the different gate applications. For example, fig. 1(b) is the QUIL [Smith et al. 2016] representation of the circuit in fig. 1(a).

In the typical programming model, called the *QRAM model* [Knill 1996], quantum computers are used as co-processors to classical (standard) 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 match. For example, fig. 1(c) gives a program in PyQuil [Rigetti Computing 2019], a quantum language/framework embedded in Python. The `ghz_state` function takes an array `qubits` and constructs a circuit that prepares the GHZ state. Calling `ghz_state([0,1,2])` would return the QUIL program in fig. 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 given in fig. 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. For example, the sqir program to the right constructs a GHZ-preparation circuit in a manner similar to the PyQuil program in fig. 1(c).

$$
\begin{array}{ll}
P & \to skip \\
& \mid P_1;\ P_2 \\
& \mid U\ q \\
& \mid U\ q_1\ q_2 \\
\\
U & \to H \mid X \mid Y \mid Z \mid R_\phi \mid CNOT
\end{array}
\qquad
\begin{array}{rl}
[\![skip]\!]_u &= I_{2^{dim}} \\
[\![P_1;\ P_2]\!]_u &= [\![P_2]\!]_u \times [\![P_1]\!]_u \\
[\![U\ q]\!]_u &= \begin{cases} pad_1(U,\ q) & \text{well-typed} \\ 0_{2^{dim}} & \text{otherwise} \end{cases} \\
[\![U\ q_1\ q_2]\!]_u &= \begin{cases} pad_2(U,\ q_1,\ q_2) & \text{well-typed} \\ 0_{2^{dim}} & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 3. Abstract syntax and semantics of unitary fragment of SQIR, assuming a global register of size *dim*. We write $[\![P]\!]_u$ to describe the semantics of unitary program P. The padding functions extend the intended operation to the correct dimension by applying an identity operation on every other qubit in the system. For example, $pad_1(H,\ q) = I_{2^q} \otimes H \otimes I_{2^{dim-q-1}}$.

SQIR programs are given a formal semantics in Coq, which is the basis for proving properties about them. For example, we can prove that a SQIR program computes the same result as a classical version of the program. Or we could prove that it prepares the expected quantum state. We give some examples of this in section 5.

```
Fixpoint GHZ (n : ℕ) : ucom n :=
  match n with
  | 0 ⇒ uskip
  | 1 ⇒ H 0
  | S n' ⇒ GHZ n'; CNOT (n'-1) n'
  end.
```

VOQC implements a series of optimizations over SQIR programs: each one is a function from SQIR programs to SQIR programs. We detail these in section 4. 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 quantum 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) conversions between OpenQASM [Cross et al. 2017] and SQIR, which we link against our extracted code. Since a number of quantum programming frameworks, including Qiskit [Aleksandrowicz et al. 2019], Project Q [Steiger et al. 2018] and Cirq [The Cirq Developers 2019], output OpenQASM, this allows us to run VOQC on a variety of generated circuits, without requiring the user to program in Coq.

3 SQIR: A SMALL QUANTUM INTERMEDIATE REPRESENTATION

This section presents the syntax and semantics of SQIR programs. To begin, we restrict our attention to the fragment of SQIR that describes unitary circuits. We then describe the full language, which allows control, measurement, and initialization.

3.1 Unitary SQIR

SQIR is a language for describing quantum programs as circuits, implemented as a deep embedding in Gallina, the language of the Coq proof assistant. Its development borrows heavily from Coq libraries for quantum computing developed for the QWIRE language [Rand et al. 2017] (but the languages are very different, as described in section 3.3).

In SQIR, a qubit is referred to by a natural number that indexes into a global register. Unitary SQIR programs allow four operations, as shown on the left of fig. 3: skip, sequencing, and unitary gate application to either one or two qubits, drawing from a fixed set of standard gates.

```
Inductive ucom (dim : ℕ): Set :=
| uskip : ucom dim
| useq : ucom dim → ucom dim → ucom dim
| uapp1 : Unitary 1 → ℕ → ucom dim
| uapp2 : Unitary 2 → ℕ → ℕ → ucom dim.

Inductive uc_well_typed {dim} : ucom dim → ℙ :=
| WT_uskip : uc_well_typed uskip
| WT_seq : ∀ c1 c2, uc_well_typed c1 → uc_well_typed c2 → uc_well_typed (c1; c2)
| WT_app1 : ∀ u n, n < dim → uc_well_typed (uapp1 u n)
| WT_app2 : ∀ u m n, m < dim → n < dim → m ≠ n → uc_well_typed (uapp2 u m n).
```

Fig. 4. Coq definitions of unitary programs and well-typedness.

A unitary program is well-typed if every unitary application is valid. A single-qubit unitary application is valid if its argument is a valid index into the global register. A two-qubit uitary application is valid if its arguments are both valid indices into the global register and the two arguments are not equal. This final requirement enforces linearity and thereby quantum mechanics' no-cloning theorem. The Coq definitions of unitary sQIR programs and well-typedness are shown in fig. 4. Note that a unitary sQIR program is parameterized by the size of the global register, `dim`.

The semantics for unitary sQIR programs is shown on the right of fig. 3. If a program is well-typed, then we can compute its denotation in the expected way. If a program is not well-typed, we ensure that its denotation is the zero matrix by returning zero whenever a unitary is applied to invalid arguments. The advantage of this definition is that it allows us to talk about the denotation of a program without explicitly proving that the program is well-typed. While doing so is not hard, it would result in proofs becoming cluttered with extra reasoning (i.e., assumption and preservation of well typing).

sQIR supports a fixed (universal) set of gates: H, X, Y, Z, R_ϕ, and $CNOT$. R_ϕ represents a phase shift (also called a z-axis rotation) by an arbitrary real number ϕ. In an effort to simplify the denotation function, the only multi-qubit gate we support is $CNOT$. sQIR can be easily extended with other built-in gates, or new gates can be defined in terms of existing gates. For instance, we define SWAP and controlled-Z as follows:

```
Definition SWAP {dim} (a b : ℕ) : ucom dim := CNOT a b; CNOT b a; CNOT a b.
Definition CZ {dim} (a b : ℕ) : com dim := H b ; CNOT a b ; H b.
```

Note that in these examples, `H` and `CNOT` stand for `uapp1 hadamard` and `uapp2 CNOT`, respectively, and the semicolon is an infix notation for `useq`. The dimensions are declared implicit. As a result, our derived gates and built-ins look the same to the programmer. We can also state and prove properties about the semantics of defined operations. In the first example above, we prove that the `SWAP` program swaps its arguments, as intended.

3.2 General sQIR: Adding Measurement

To describe general quantum programs, we extend unitary sQIR with an operation for performing measurement. The command `meas q c1 c2` (inspired by a similar construct in QPL [Selinger 2004]) measures the qubit `q` and either performs program `c1` or `c2` depending on the result. We can define non-branching measurement and resetting a qubit to $|0\rangle$ in terms of branching measurement:

```
Notation "'measure' n" := (meas n skip skip).
```

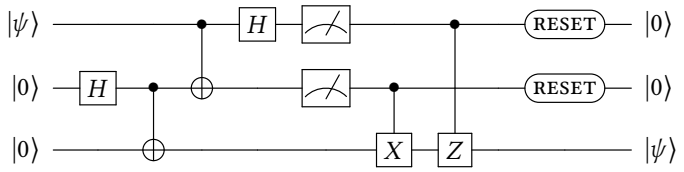


Fig. 5. Circuit for quantum teleportation.

```
Notation "'reset' n" := (meas n (X n) skip).
```

(It is easy to prove that `measure` and `reset` have the expected semantics.)

Example: Quantum Teleportation. The circuit for quantum teleportation is given in fig. 5. In the quantum teleportation protocol, Alice wishes to communicate a qubit $|\psi\rangle$ 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 then uses those classical bits to transform his qubit on wire 2 into $|\psi\rangle$.

This circuit corresponds to the following (general) SQIR program. The three components are arrayed in sequence by `teleport`. First, `bell` prepares a bell pair on qubits 1 and 2, the former of which is sent to `alice`'s `CNOT` along with qubit 0. After `alice` performs the measurements of the qubits, `bob` performs operations controlled by the (now classical) values of wire 0 and 1. Finally, `bob` resets the first two wires to the zero state.

```
Definition bell : com 3 := H 1 ; CNOT 1 2.
Definition alice : com 3 := CNOT 0 1 ; H 0 ; measure 0 ; measure 1.
Definition bob : com 3 := CNOT 1 2; CZ 0 2; reset 0; reset 1.
Definition teleport : com 3 := bell; alice; bob.
```

3.2.1 Nondeterministic Semantics. We define a *nondeterministic semantics* for general SQIR, which extends the semantics from fig. 3. Evaluation is given as a relation, where measurement can yield multiple outcomes. An illustrative fragment of this semantics is shown below.

```
Inductive nd_eval {dim} : com dim → Vector (2^dim) → Vector (2^dim) → ℙ :=
  | nd_app1 : ∀ (u : Unitary 1) (q : ℕ) (ψ : Vector (2^dim)),
      app1 u q / ψ ⇓ ((pad u q) × ψ)
  | nd_meas_t : ∀ q c1 c2 (ψ : Vector (2^dim)),
      let ψ' := pad |1⟩⟨1| q × ψ in
      norm ψ' ≠ 0 →
      c1 / ψ' ⇓ ψ'' →
      meas q c1 c2 / ψ ⇓ ψ''
  | nd_meas_f : ∀ q c1 c2 (ψ : Vector (2^dim)),
      let ψ' := pad |0⟩⟨0| q × ψ in
      norm ψ' ≠ 0 →
      c2 / ψ' ⇓ ψ'' →
      meas q c1 c2 / ψ ⇓ ψ''
where "c '/' ψ '⇓' ψ'" := (nd_eval c ψ ψ').
```

The `nd_app` rule says that, given state ψ, `app1 u q` evaluates to `(pad u q)` $\times\psi$, as expected. The `nd_meas_t` rule says that, if the result of projecting the q^{th} qubit onto the $|1\rangle\langle 1|$ subspace is not the zero matrix (meaning that it is possible to project onto this subspace), measuring q yields the

$$
\begin{aligned}
[\![skip]\!]_d(\rho) &= I_{2^{dim}} \times \rho \times I_{2^{dim}}^\dagger \\
[\![P_1;\ P_2]\!]_d(\rho) &= ([\![P_2]\!]_d \circ [\![P_1]\!]_d)(\rho) \\
[\![U\ q]\!]_d(\rho) &= \begin{cases} pad_1(U,\ q) \times \rho \times pad_1(U,\ q)^\dagger & \text{well-typed} \\ 0_{2^{dim}} & \text{otherwise} \end{cases} \\
[\![U\ q_1\ q_2]\!]_d(\rho) &= \begin{cases} pad_2(U,\ q_1,\ q_2) \times \rho \times pad_2(U,\ q_1,\ q_2)^\dagger & \text{well-typed} \\ 0_{2^{dim}} & \text{otherwise} \end{cases} \\
[\![meas\ q\ P_1\ P_2]\!]_d(\rho) &= [\![P_2]\!]_d(|0\rangle_q\langle 0|\rho|0\rangle_q\langle 0|) + [\![P_1]\!]_d(|1\rangle_q\langle 1|\rho|1\rangle_q\langle 1|)
\end{aligned}
$$

Fig. 6. SQIR density matrix semantics, assuming a global register of size *dim*. We use the notation $[\![P]\!]_d$ to describe the semantics of program P. We use $|i\rangle_q\langle j|$ as shorthand for $I_{2^q} \otimes |i\rangle\langle j| \otimes I_{2^{dim-q-1}}$, which applies the projector to the relevant qubit and an identity operation to every other qubit in the system.

program c_1 applied to this projection. Most quantum states can step via both the `nd_meas_t` and `nd_meas_f` rules; hence the semantics is nondeterministic. To simplify the reasoning process, we do not rescale the output of measurement. Thus the nondeterministic semantics is primarily useful for proving properties that are invariant to scaling. The correctness of the teleport protocol above is one example of a property that is invariant to scaling, because the qubit is always correctly transmitted.

We show that the nondeterministic semantics of the unitary fragment of SQIR is identical to the (full) unitary semantics, albeit in relational form:

```
Lemma nd_eval_ucom : ∀ {dim} (c : ucom dim) (ψ ψ' : Vector (2^dim)),
    WF_Matrix ψ → (c / ψ ⇓ ψ' ↔  ⟦c⟧u× ψ = ψ').
```

The `WF_Matrix` predicate here ensures that ψ is a valid input to the circuit.

3.2.2 Density Matrix Semantics. We also define an alternative semantics for full SQIR based on density matrices, following the approach of several previous efforts [Paykin et al. 2017; Ying 2011]. While the nondeterministic semantics simply considers the possible resulting states following measurement, the density matrix semantics encodes the *probability distribution* over resulting states. As such it is more useful for proofs that directly consider the likelihood of each outcome. The density matrix semantics is given in fig. 6.

We prove the following correspondence between the density matrix semantics (with subscript d) and the unitary semantics (subscript u) of fig. 3:

```
Lemma c_eval_ucom : ∀ {dim} (c : ucom dim),
    ⟦c⟧d = fun ρ ⇒ ⟦c⟧u × ρ × ⟦c⟧u† .
```

That is, the density matrix denotation of a unitary program simply multiplies the input state on both sides by the unitary denotation of the same program.

3.3 Discussion: Programmability vs. Verification

SQIR is a simple, low-level language, which makes programming more tedious, but also makes formal proofs substantially easier.

Concrete wires vs. abstract variables. A key design choice in SQIR is to use a global register of concrete indexes for wires. An alternative is to use variables as abstract wires (which would later be

allocated to concrete wires). This approach is taken by QWIRE [Paykin et al. 2017], a higher-level language also embedded in Coq, employing Higher Order Abstract Syntax [Pfenning and Elliott 1988] to take advantage of Coq's variable binding and function composition facilities.

From a programming perspective, abstract wires are better. You write your algorithm without worrying about the particular wires you are using and whether or not you can reuse some of them for temporary storage (ancillae). You can build larger circuits by composing smaller ones, connecting abstract outputs to abstract inputs.

On the other hand, using concrete indexes and a global register significantly simplifies verification. As we saw earlier, the semantics of a quantum program is in terms of vectors or matrices, which are a highly structured way of representing data. To give semantics to a program that uses abstract wires, you have to convert those wires to concrete indexes in a vector or matrix. In QWIRE, programs can arbitrarily allocate and deallocate qubits, which entails de Bruijn-style index shifting (hence Rand [2018] refers to these as "de Bruijn circuits"). Reasoning about this conversion process can be laborious especially for recursive circuits and the corresponding inductive proofs. By contrast, SQIR's static global register of qubits obviates the need to worry about discarding qubits and shifting the ones that remain.

We discuss the advantages and disadvantages of SQIR's use of concrete indexes vs. QWIRE's higher-order abstract syntax in greater detail in appendix A.

Single wires vs. data structures. SQIR also lacks some other useful features present in higher-level languages. For example, in QIO [Altenkirch and Green 2010] and Quipper [Green et al. 2013] one can construct circuits that compute on structures over quantum data, rather than single qubits. In QWIRE, this concept is refined to express relationships using dependent types; e.g., the type for the quantum fourier transform indicates it takes a list of n qubits to a list of n qubits, and moreover makes use of n in the program itself.

Regrettably, these structures can make reasoning about programs difficult. It is trivially easier to reason about a qubit (which is represented by a length-2 vector or a 2×2 density matrix) than about an indexed-list of qubits (represented by a $2^n \times 2^n$ matrix for an arbitrary n). Moreover, while a tree of n qubits and a list of 2^n qubits are trivially isomorphic to one another, proving the correctness of a given circuit will often be easier in one of the two representations. Unfortunately, if we want to compose a circuit that produces a list with one that takes in a tree, we will require a gadget to connect the two, and then prove that this gadget is semantically equal to the identity. In SQIR, which has neither quantum datatypes nor typed circuits, this issue simply doesn't present itself.

Single measurements vs. dynamic lifting. The term *dynamic lifting* was coined by Green et al. [2013]. It 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. Dynamic lifting is used extensively in Quipper and QWIRE. Unfortunately, its presence complicates the denotational semantics, as the semantics of any Quipper or QWIRE program depends on the semantics of Coq or Haskell, respectively. In giving a denotational semantics to QWIRE, Paykin et al. [2017] assume an operational semantics for an arbitrary host language, and give a denotation for a lifted circuit only when both of its branches reduce to valid QWIRE circuits.

While 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.

Near-term programming. While the abstractions that languages like QWIRE and Quipper provide are convenient, they do not reflect the kind of quantum programming we can expect to do in the near future. QWIRE allows for the arbitrary allocation and deallocation of qubits, but modern hardware does not: As we will see in section 4, today's quantum computers are severely limited by qubit count and position. Dynamic lifting requires exchanging information between a (typically supercooled) quantum chip and a classical computer before qubits decohere, which is unrealistic for current technology. As a result, IBM's practically-minded OpenQASM language [Cross et al. 2017] only allows for a limited form of branching that is close to SQIR's. In the near term, then, it it reasonable to both write *and* verify programs in SQIR, as well as prove transformations of them correct.

4 VERIFYING PROGRAM TRANSFORMATIONS

Because near-term quantum machines will only be able to perform small computations before decoherence takes effect, compilers for quantum programs must apply optimizations to reduce resource usage. These optimizations are vulnerable to programmer error. Testing their correctness will be difficult on near-term quantum devices, due to noise and limited scale, and simulating them on classical machines is limited by state space explosion. Thus, a promising way forward is to *formally verify* that the implementations of circuit optimizations are correct. SQIR was designed to facilitate such verification.

We have verified several optimizations of SQIR programs, inspired by existing circuit optimizers. We have found that the performance of our verified implementation is comparable to existing industrial optimizer implementations. We have also verified a simple circuit mapping algorithm.

4.1 Equivalence of SQIR Programs

In general, we are interested in proving that a transformation is *semantics preserving*, meaning that the transformation does not change the denotation of the program. When a transformation is semantics preserving, we say that it is *sound*. We will express soundness by requiring equivalence between the input and output of the transformation function. Equivalence over (unitary) SQIR programs is defined as follows:

```
Definition uc_equiv {dim} (c1 c2 : ucom dim) := ⟦c1⟧u = ⟦c2⟧u.
```

The curly braces above indicate that the `dim` argument is implicit (and can be inferred from `c1` and `c2`) allowing us to use the following notation:

```
Notation "c1 ≡ c2" := (uc_equiv c1 c2).
```

4.1.1 Skip Elimination. As a simple example of a transformation we can prove sound, consider the recursive *skip removal* function, shown below.

```
Fixpoint rm_uskips {dim} (c : ucom dim) : ucom dim :=
  match c with
  | c1 ; c2 ⇒ match rm_uskips c1, rm_uskips c2 with
              | uskip, c2' ⇒ c2'
              | c1', uskip ⇒ c1'
              | c1', c2'   ⇒ c1'; c2'
              end
  | c'      ⇒ c'
end.
```

We prove this transformation is sound in the following lemma.

```
Lemma rm_uskips_sound : ∀ {dim} (c : ucom dim), c ≡ (rm_uskips c).
```

The proof is straightforward and relies on the identities `uskip; c ≡ c` and `c; uskip ≡ c` (which are also easily proven in our development).

We can also prove other useful structural properties about `rm_uskips`. First, we inductively define a predicate on circuits that says it has no skips.

```
Inductive skip_free {dim} : ucom dim → ℙ :=
  | SF_seq : ∀ c1 c2, skip_free c1 → skip_free c2 → skip_free (c1; c2)
  | SF_app1 : ∀ n a (u : Unitary 1), skip_free (uapp1 u a)
  | SF_app2 : ∀ n a b (u : Unitary 2), skip_free (uapp2 u a b).
```

Then we can prove that the output of `rm_uskips` is either a single skip operation, or satisfies the `skip_free` predicate.

```
Lemma rm_uskips_correct : ∀ {dim} (c : ucom dim),
    (rm_uskips c) = uskip ∨  skip_free (rm_uskips c).
```

We can also prove that the output of `rm_uskips` contains no more skip operations or unitary applications that the original input program.

```
Fixpoint count_ops {dim} (c : ucom dim) : ℕ :=
  match c with
  | c1; c2 ⇒ (count_ops c1) + (count_ops c2)
  | _ ⇒ 1
  end.

Lemma rm_uskips_reduces_count : ∀ {dim} (c : ucom dim),
  count_ops (rm_uskips c) ≤ count_ops c.
```

4.2 Representation of Programs in voqc

voqc's optimizations generally employ a list representation of programs rather than sqir's general sequence construct. The reason for this is that it is easier to search for patterns of gates in a list representation than in arbitrarily nested tuples. Converting from a `ucom` to a list flattens all sequences and removes all skip operations.

```
Inductive gate_app (dim : ℕ): Set :=
| App1 : Unitary 1 → ℕ → gate_app dim
| App2 : Unitary 2 → ℕ → ℕ → gate_app dim.

Definition gate_list (dim : ℕ) := list (gate_app dim).

Fixpoint ucom_to_list {dim} (c: ucom dim) : gate_list dim :=
  match c with
  | c1; c2 ⇒ (ucom_to_list c1) ++ (ucom_to_list c2)
  | uapp1 u n ⇒ [App1 u n]
  | uapp2 u m n ⇒ [App2 u m n]
  | uskip ⇒ []
  end.
```

We also define a conversion from gate lists to sqir programs, along with a notion of equivalence defined in terms of equivalence of sqir programs.

Table 1. Summary of the optimization routines in Qiskit and the Nam et al. [2018] optimizer. VOQC provides the complete and verified functionality of the routines in bold over the gate set $\{H, X, R_{\pi/4}, CNOT\}$. The italicized optimizations are implemented but not (yet) fully verified. In the Qiskit column, LX indicates that a routine is used by optimization level X. Under the Nam et al. column, P stands for "preprocessing" and L and H indicate whether the routine is in the "light" or "heavy" versions of the optimizer.

Qiskit	Nam et al. [2018]
Unitary optimizations • **CXCancellation** (L1) • **Optimize1qGates** (L1, L2, L3) • **CommutativeCancellation** (L2, L3) • ConsolidateBlocks (L3) Non-unitary optimizations • RemoveResetInZeroState (L1, L2, L3) • OptimizeSwapBeforeMeasure (L3) • **RemoveDiagonalGatesBeforeMeasure** (L3)	Unitary optimizations • **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)

```
Fixpoint list_to_ucom {dim} (l : gate_list dim) : ucom dim :=
  match l with
  | [] ⇒ uskip
  | (App1 u n)::t ⇒ (uapp1 u n) ; (list_to_ucom t)
  | (App2 u m n)::t ⇒ (uapp2 u m n) ; (list_to_ucom t)
  end.

Lemma ucom_list_equiv : ∀ {dim} (c : ucom dim) l, c ≡ list_to_ucom (ucom_to_list c).
```

When converting to the list representation, we limit ourselves to the gate set $\{H, X, R_{\pi/4}, CNOT\}$. This is to be consistent with existing optimizers against which we compare, below. The $R_{\pi/4}$ gate allows for rotation by an integer multiple of $\pi/4$, and allows us to express common gates such as Z, P, and T.

4.3 Optimizations of Unitary Programs

We have implemented several optimizations over unitary programs inspired by existing quantum circuit optimizers, notably those in IBM's Qiskit compiler [Aleksandrowicz et al. 2019] and the compiler by Nam et al. [2018]. VOQC's optimizations are not one-to-one with these prior optimizers. In table 1, we boldface those optimizations in Qiskit and Nam et al. that are covered by fully verified[2] qcomp optimizations. Those in italics are partly covered by verified VOQC optimizations.

VOQC only considers "peephole" optimizations that apply small circuit identities to try to reduce overall gate count of the circuit. In developing these optimizations, we have verified many small circuit equivalences as well as utility functions (e.g., a transformation that replaces an arbitrary one-qubit circuit with a different, equivalent circuit).

Our most sophisticated optimization involves cancelling redundant gates. The idea of the optimization is to propagate each gate in the circuit as far right as possible, using a predefined set of commutation rules, until a cancelling gate is encountered. One gate cancels another if the

[2]The proofs that are "fully verified" rely on one admitted lemma that says that $CNOT$ gates that act on disjoint qubits commute. This is certainly true, although requires a tedious proof with our current linear algebra utilities.

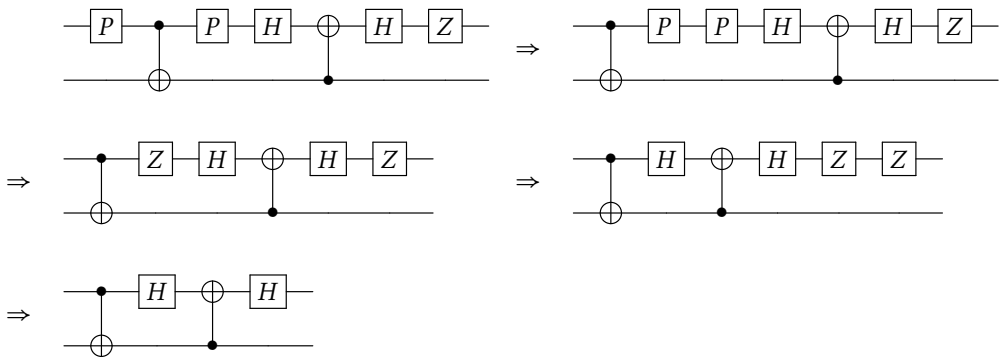


Fig. 7. An example of cancellation with commutation. Note that $P = R_{\pi/4}(\pi/2)$ and $Z = R_{\pi/4}(\pi)$. In the first step, the leftmost P gate is propagated through the control of the leftmost $CNOT$ gate. In the second step, the two P gates are combined into a single Z gate. In the third step, the leftmost Z gate is propagated through a sub-circuit consisting of two H gates and a $CNOT$ gate. In the final step, the two Z gates cancel.

unitaries corresponding to the two gates are inverses. For example, the $CNOT$, X, and H gates are all self-cancelling (section 2.1 illustrates the last of these). If a cancelling gate is encountered, then both gates can be removed from the circuit. If two z-axis rotation gates are adjacent, then they can be merged by adding their rotation angles, keeping in mind that a rotation by zero is a no-op. An example of applying this optimization on a particular circuit is shown in fig. 7.

This "cancellation with commutation" optimization is based on the single- and two-qubit gate cancellation routines from Nam et al. [2018] and a similar routine in IBM's Qiskit compiler [Aleksandrowicz et al. 2019]. We use a subset of the commutation rules presented in Nam et al. [2018].

We have also implemented, but only partly verified, an optimization by Nam et al. [2018] that performs circuit rewrites to reduce the number of H gates in a circuit. (It is italicized in the table.)

4.4 Optimizations of Non-unitary Programs

Although our focus has been on optimizing unitary SQIR programs, we have also verified some optimizations of non-unitary programs. For example, we have shown that performing a measure or reset after a reset is equivalent to simply performing the reset.

```
Lemma meas_reset : ∀ dim q,
  measure q ; reset q ≡ reset q.

Lemma reset_reset : ∀ dim q,
  reset q ; reset q ≡ reset q.
```

We have also shown that gates that perform z-axis rotations can be removed before measurement without affecting the measurement result.

```
Lemma R_mif : ∀ dim θ n c1 c2,
  (R θ) n ; mif n then c1 else c2 ≡ mif n then c1 else c2.
```

We have used this last lemma to verify a lightweight optimization that removes R or Z gates before measurement (including `measure` and `reset`). Qiskit also employs this technique. Other optimizations that Qiskit performs on non-unitary programs include removing resets applied to qubits in an initial zero state (using reasoning similar to our `reset_reset` lemma), and removing SWAP operations before measurement.

Table 2. Average gate count reduction on random circuits.

	H	X	$u1$	$CNOT$
Qiskit	21.4%	25.3%	67.6%	2.3%
VOQC	19.2%	22.3%	65.8%	1.1%

4.5 Experimental Evaluation

We compare VOQC's unitary optimization routines against IBM's Qiskit transpiler and the results presented in Nam et al's [2018] paper.

We do this by running VOQC and Qiskit on a subset of the benchmarks used by Nam et al. [2018] and Amy et al. [2013]. The benchmarks consist of arithmetic circuits and implementations of multiple-control Toffoli gates (all of which are unitary). The benchmarks contain between 45 and 61,629 gates and use between 5 and 192 qubits. For VOQC we only enable our fully verified optimizations. Because the benchmarks are all unitary programs, we do not evaluate our non-unitary optimizations.

We run Qiskit's transpiler framework with the basis set $\{H,\ X,\ u1,\ CNOT\}$ where $u1$ is Qiskit's single-qubit z-axis rotation gate. We include all of Qiskit's level-1 and level-2 unitary optimizations and iterate until the depth of the circuit does not change. We do not include level-3 optimizations because our aim is to compare VOQC against implementations of similar optimizations. Similarly, we do not report the results of the heavy level of optimizations of the Nam et al. compiler. We expect that the level-3 optimizations used in Qiskit and the heavy optimizations used by Nam et al. are verifiable in our framework, and we are actively working on increasing the scope of the optimizations that we have verified.

We did not re-run Nam et al. [2018] but simply report their results as their compiler is (now) proprietary.

The results are shown in fig. 8 and fig. 9. On all benchmarks, VOQC and Qiskit are most effective at reducing $R_{\pi/4}$ gates (as opposed to H or X gates). This is expected since most of our equivalences target $\mathbb{R}_{\pi/4}$ gates. As can be seen in fig. 8, VOQC is always slightly more effective than Qiskit at reducing the number of rotation gates. On average, VOQC reduces rotation gates by 16.1%, compared to Qiskit at 8.3%.

fig. 9 compares the total gate reduction by VOQC and Qiskit to the results reported by Nam et al. for the light level of optimization ((L) in table 1). Here, VOQC outperforms Qiskit again, seeing average reductions of 8.4% vs. 4.7%. Unsurprisingly, the Nam et al. optimizer performs better—gate counts are reduced, on average, by 25.2%—because it contains more optimizations that VOQC. However, the optimizations performed by VOQC are all verified in Coq.

As a further evaluation of Qiskit and VOQC, we ran both optimizers on randomly-generated circuits. table 2 shows the results of running VOQC and Qiskit on 20 random circuits, each with 15 qubits and 5000 gates drawn uniformly from the set

$$\{H,\ X,\ R_{\pi/4}(\pi/4),\ R_{\pi/4}(-\pi/4),\ R_{\pi/4}(\pi/2),\ R_{\pi/4}(-\pi/2),\ CNOT\}.$$

Qiskit consistently slightly outperforms VOQC on these random benchmarks, even though it fared worse on the real benchmarks. The reason for this is the difference in how VOQC and Qiskit determine commutativity. VOQC uses a fixed set of rules designed by Nam et al. [2018] to detect commutativity of subcircuits in program like the ones used as benchmarks. Qiskit performs matrix multiplication to determine whether two gates commute, which allows it to find more opportunities for commutation in the random benchmarks, but does not allow it to find commuting subcircuits in the real benchmarks.

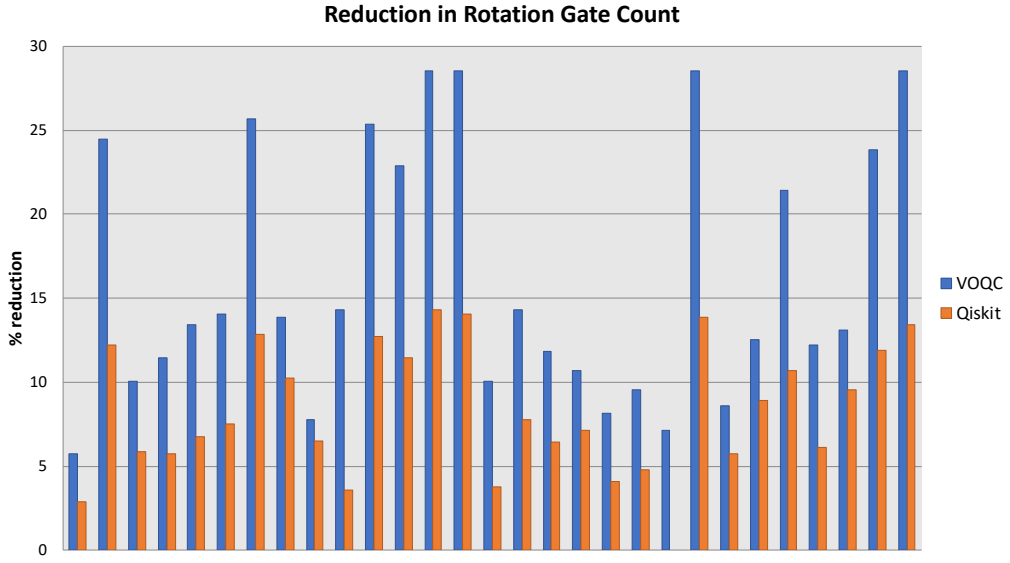


Fig. 8. Comparison of the performance of voqc and Qiskit on 29 benchmark programs. The y-axis is the percentage $R_{\pi/4}$ gates removed from the benchmark.

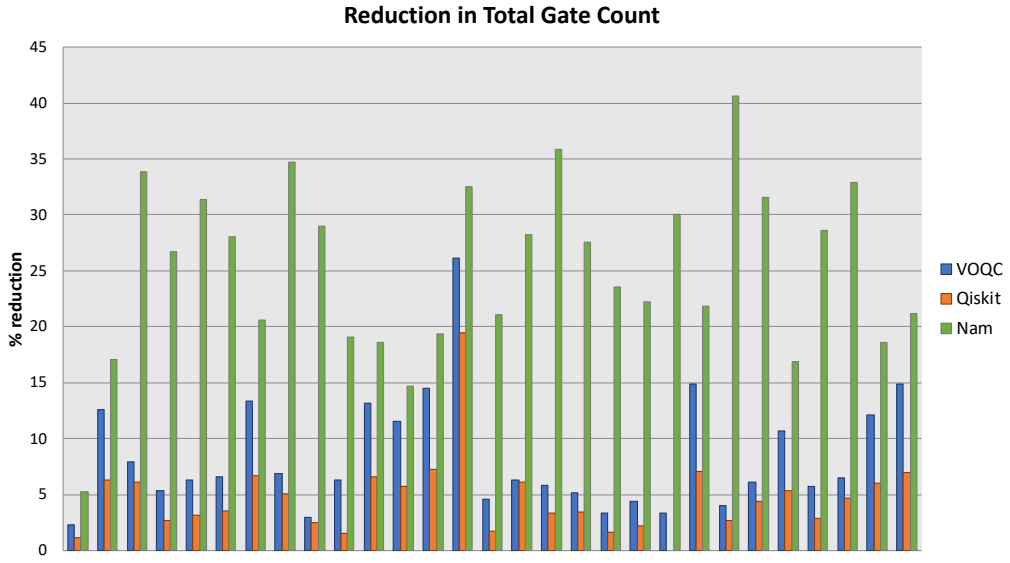


Fig. 9. Comparison of the performance of voqc, Qiskit, and the Nam et al. [2018] optimizer on 29 benchmark programs. The y-axis is the total percentage gates removed from the benchmark.

4.6 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 [Saeedi et al. 2011; Zulehner et al. 2017]. Circuit mapping algorithms take as input an arbitrary circuit and output a circuit that respects the connectivity constraints of some underlying architecture.

For example, consider IBMs's five-qubit Tenerife machine, whose mapping is visualized in fig. 10. 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 and which can be the target. For example, on the Tenerife machine, a *CNOT* gate may be applied with physical qubit Q4 as the control and physical qubit Q2 as the target. However, no two-qubit gate is possible between physical qubits Q4 and Q1.

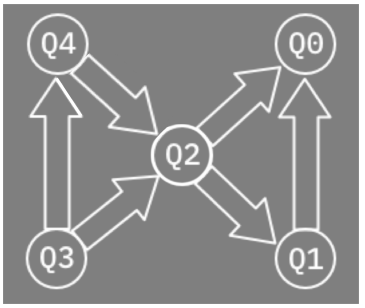


Fig. 10. Two-qubit gate connections on IBM's Tenerife machine. From https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md

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. In particular, we consider a linear nearest neighbor (LNN) architecture where qubits are connected to adjacent qubits in the global register (so qubit i is connected to qubits $i-1$ and $i+1$, but qubit 0 and qubit $dim-1$ are not connected). A program will be able to run on our LNN architecture if all *CNOT* operations occur between connected qubits. We can represent this constraint as follows.

```
Inductive respects_LNN {dim} : ucom dim → ℙ :=
  | LNN_skip : respects_LNN uskip
  | LNN_seq : ∀ c1 c2,
      respects_LNN c1 → respects_LNN c2 → respects_LNN (c1; c2)
  | LNN_app_u : ∀ (u : Unitary 1) n, respects_LNN (uapp1 u n)
  | LNN_app_cnot_left : ∀ n, respects_LNN (CNOT n (n+1))
  | LNN_app_cnot_right : ∀ n, respects_LNN (CNOT (n+1) n).
```

This definition says that `uskip` and single-qubit unitary operations always satisfy the LNN constraint, a sequence construct satisfies the LNN constraint if both of its components do, and a *CNOT* satisfies the LNN constraint if its arguments are adjacent in the global register.

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.

We have verified that our transformation is semantics preserving, and that the output program satisfies the LNN constraint.

```
Lemma map_to_lnn_sound : ∀ {dim} (c : ucom dim), c ≡ map_to_lnn c.

Lemma map_to_lnn_correct : ∀ {dim} (c : ucom dim),
  uc_well_typed c → respects_LNN (map_to_lnn c).
```

We have also built a mapper to the Tenerife architecture, which is similar to our mapper for the LNN architecture. To handle the directionality of the edges in the connectivity graph, we make

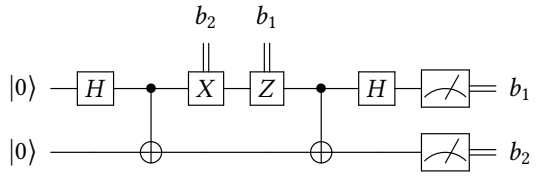


Fig. 11. Circuit for the superdense coding algorithm.

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

Definition bell00 : ucom 2 := H a; CNOT a b.

Definition encode (b1 b2 : 𝔹): ucom 2 :=
    (if b2 then X a else uskip);
    (if b1 then Z a else uskip).

Definition decode : ucom 2 := CNOT a b; H a.

Definition superdense (b1 b2 : 𝔹) : ucom 2 :=
    bell00 ; encode b1 b2; decode.
```

Fig. 12. SQIR program for the unitary portion of the superdense coding algorithm. Note that `U q` is syntactic sugar for applying unitary U to qubit q.

use of the identity `H a; H b; CNOT a b; H a; H b ≡ CNOT b a`. However, we have not yet fully verified this mapper.

5 SQIR FOR GENERAL VERIFICATION

SQIR's simple structure and semantics allow us to easily verify general properties of quantum programs, as discussed in section 3.3. In this section we discuss correctness properties we have proved of four simple quantum programs written in SQIR.

5.1 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 fig. 11. The SQIR program corresponding to the unitary part of this circuit is shown in fig. 12. In the SQIR program, note that `encode` is a Coq function that takes two Boolean values and returns a circuit.

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⟧u × | 0,0 ⟩ = | b1,b2 ⟩.
```

Note that we are applying the denotation of `superdense` to a vector rather than a density matrix, and that we use Dirac (bra-ket) notation to represent this vector. In our experience, treating states as vectors and performing rewriting over bra-ket expressions simplifies reasoning.

5.2 GHZ State Preparation

The Greenberger-Horne-Zeilinger (GHZ) state [Greenberger et al. 1989] is an n-qubit entangled quantum state of the form

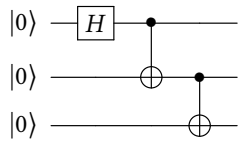
$$|\text{GHZ}\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}).$$

This vector representing this state 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 below, on the right. The sqir description of (the unitary portion of) this circuit can be produced by the recursive function below, on the left.

```
Fixpoint GHZ (n : ℕ) : ucom n :=
  match n with
  | 0 ⇒ uskip
  | 1 ⇒ H 0
  | S n' ⇒ GHZ n'; CNOT (n'-1) n'
  end.
```



The function `GHZ` describes a *family* of sqir circuits: For every n, `GHZ n` is a valid sqir program and quantum circuit.[3] 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:

```
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, given an n', 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 considering the effect of applying `CNOT (n'−1) n'` to this state, we can show that `GHZ (n'+1)` produces the $|GHZ\rangle$ state on $n' + 1$ qubits.

5.3 Teleportation

Quantum teleportation was introduced in section 3.2. Here we prove a correctness property about it: the input qubit is the same as the output qubit. Since sqir does not permit us to discard the two measured qubits, we instead reset them to $|0\rangle$. Under the density matrix semantics, we aim to prove the following:

```
Lemma teleport_correct : ∀ (ρ : Density (2^1)),
  WF_Matrix ρ →
  ⟦teleport⟧d (ρ ⊗ |0⟩⟨0| ⊗ |0⟩⟨0|) = |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ ρ.
```

[3]For the sake of readability, we have elided a coercion from `ucom n'` to `ucom n` in the recursive case. We have done the same for `cpar` below.

The proof for the density matrix semantics is simple: We compute the products using a matrix solver, and do some simple (automated) arithmetic to show that the output matrix has the desired form. While short, this proof does not give much intuition about how quantum teleportation works.

For the non-deterministic semantics the proof is more involved, but also more illustrative of the inner workings of the teleport 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.
```

Since the non-deterministic semantics does not rescale outcomes, we merely require that every outcome is proportional to ($\propto$) the intended outcome. 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 ⟩) ⇓ |0 , 0 ⟩ ⊗ ψ
```

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 $CNOT$s 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 ψ. Finally, resetting the already-measured qubits is deterministic, and leaves us in the desired state.

5.4 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 $f(\bot) = 0$ or $f(\bot) = 1$ 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 ≡ uskip → boolean 1 u
  | boolean_X : ∀ u, u ≡ X 0 → boolean 1 u
```

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

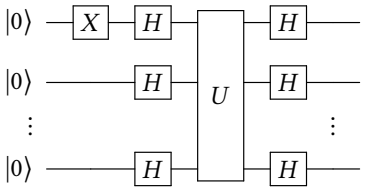


Fig. 13. The Deutsch-Jozsa algorithm in SQIR and as a circuit.

```
  | boolean_U : ∀ dim u u1 u2,
                boolean dim u1 →
                boolean dim u2 →
                ⟦u⟧ᵤ = ⟦u1⟧ᵤ ⊗ |0⟩⟨0| .+ ⟦u2⟧ᵤ ⊗ |1⟩⟨1| →
                boolean (1 + dim) u.
```

`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 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 dim} (P : boolean dim U) : ℙ :=
  count P = 2 ^ (dim - 2).
Definition constant {dim : ℕ} {U : ucom 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 fig. 13. 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 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} (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) (P : boolean dim U), constant P → accept P.
Theorem deutsch_jozsa_balanced_correct :
  ∀ (dim : ℕ) (U : ucom) (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} (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 ≡ uskip` or `U ≡ X 0`. Observing that $\langle\psi| HX^b H |1\rangle = (1-2b)\langle\psi|1\rangle$, we can complete the proof by direct calculation on matrices. For the inductive step, the induction hypothesis says that, for any Boolean function g of $dim-1$ bits,

$$\langle\psi, 0^{dim-1}|\ H^{\otimes dim} U_g H^{\otimes dim}\ |1, 0^{dim-1}\rangle = \left(1 - \frac{|S(g)|}{2^{dim-2}}\right)\langle\psi|1\rangle,$$

where $|S(g)|$ is the number of inputs on which g evaluates to 1. Therefore, for dim-bit Boolean function f, since $|S(f)| = |S(f_0)| + |S(f_1)|$,

$$\begin{aligned}
&\langle\psi, 0^{dim}|\ H^{\otimes(1+dim)} U_f H^{\otimes(1+dim)}\ |1, 0^{dim}\rangle \\
&\quad= \langle\psi, 0^{dim}|\ H^{\otimes(1+dim)} (U_{f_0} \otimes |0\rangle\langle 0| + U_{f_1} \otimes |1\rangle\langle 1|) H^{\otimes(1+dim)}\ |1, 0^{dim}\rangle \\
&\quad= \frac{1}{2}\langle\psi, 0^{dim-1}|\ H^{\otimes dim} U_{f_0} H^{\otimes(dim)}\ |1, 0^{dim-1}\rangle + \frac{1}{2}\langle\psi, 0^{dim-1}|\ H^{\otimes dim} U_{f_1} H^{\otimes dim}\ |1, 0^{dim-1}\rangle \\
&\quad= \left(1 - \frac{|S(f)|}{2^{dim-1}}\right)\langle\psi|1\rangle.
\end{aligned}$$

6 RELATED WORK

In section 3.3, we compare SQIR as a programming language to existing languages like Quipper [Green et al. 2013], QWIRE [Paykin et al. 2017] and PyQuil [Rigetti Computing 2019], noting the abstractions that SQIR gives up for the sake of easy verification. However, we should also address the other two bodies of related work: formal verification of quantum computing and compilation of quantum programs.

Verified Quantum Computing. The earliest attempts to formally verify quantum computations in a proof assistant were Green's Agda implementation of the Quantum IO Monad [Green 2010] and a small Coq quantum library by Boender et al. [2015]. Unfortunately, these were both proofs of concept, and neither developed beyond verifying basic protocols.

A more ambitious project in this domain, from which this work draws heavily, was the QWIRE language. QWIRE was originally conceived by Paykin et al. [2017] as a Quipper-like language with linear and dependent types, paralleling the development of ProtoQuipper [Rios and Selinger 2017; Ross 2015]. Later, QWIRE was embedded in the Coq proof assistant and used to verify a variety of simple programs [Rand et al. 2017], assertions regarding ancilla qubits [Rand et al. 2018], and its own metatheory [Rand 2018].

Amy's metaQASM [Amy 2019], while not directly targeted at verification, extends OpenQASM [Cross et al. 2017] with support for metaprogramming and also gives both languages operational semantics. This opens up the possibility of directly verifying properties of OpenQASM or metaQASM programs.

A parallel line of work, pioneered by D'Hondt and Panangaden [2006] and Ying [2011], uses program logics to reason about quantum programs. These logics allow you to prove a variety of program properties inside a formal deductive system. Two of the most exciting recent developments in this area involved the use of proof assistants. Liu et al. [2019] implemented Ying's quantum Hoare logic inside the Isabelle proof assistant and used it to prove the correctness of Grover's algorithm and Unruh [2019] developed a *relational* quantum Hoare logic and built a Isabelle-based tool to prove the security of quantum cryptosystems.

Quantum Circuit Compilation and Optimization. While both the Quipper [Green et al. 2013]) and Scaffold [Javadi-Abhari et al. 2012] programming languages were developed to estimate the resources required by quantum programs, Quipper developed in the direction of high level quantum programming (since writing quantum programs is hard) and Scaffold developed in the direction of optimizing quantum programs (since running quantum programs is hard). Scaffold's ScaffCC compiler [Javadi-Abhari et al. 2014] implemented a number of peephole transformations that apply simple circuit equivalences to reduce gate count, of the sort described in section 4. A number of optimizing compilers, including IBM's ScaffCC-inspired Qiskit [Aleksandrowicz et al. 2019], Project Q [Steiger et al. 2018] and the Nam et al. [2018] compiler, quickly followed in its footsteps.

While reducing gate counts is crucial for near term quantum computing an even more important task is satisfying architecture requirements, like mapping to a particular gate set or qubit topology. 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 [Staff 2018]. Compilers for quantum programs can even help drive the development of quantum architectures by providing quick feedback about the efficiency of different gate set and topology design choices [Murali et al. 2019].

Verified Quantum Compilation. In terms of verified compilers, Amy et al. [2017] developed a proved-correct optimizing compiler from source Boolean expressions to reversible circuits, but did not handle general quantum programs. Rand et al. [2018] developed a similar compiler for quantum circuits but without optimizations, using the QWIRE language mentioned above.

The line of work that most closely aligns with our own, in terms of reliably compiling quantum circuits, revolves around the ZX-calculus. ZX [Coecke and Duncan 2011] is a formalism for describing quantum computation based upon categorical quantum mechanics [Abramsky and Coecke 2009]. The ZX-calculus is characterized by a small set of rewrite rules that allow translation of a diagram to any other diagram representing the same computation. The Quantomatic tool [Fagan and Duncan 2018] automatically rewrites ZX diagrams according to this set of rules, but suffers from two flaws as a quantum compiler: Its rewriting procedures aren't guaranteed to terminate and not every ZX diagram corresponds to a valid quantum circuit. PyZX [Kissinger and van de Wetering 2019] addresses both of these flaws, 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.

7 CONCLUSIONS AND FUTURE WORK

This paper has presented VOQC, the first verified optimizer for quantum circuits. A key component of VOQC is SQIR, a simple, low-level quantum language deeply embedded in Gallina, the language of the Coq proof assistant. Compiler passes are expressed as Gallina functions which are proved to preserve the semantics of their input SQIR programs. We developed several peephole optimizations aimed to remove redundant gates (e.g., through commutation and cancellation). These were inspired by, and in some cases generalized, optimizations in industrial compilers, but in VOQC are proved correct. When applied to a benchmark suite of 29 circuit programs, we found VOQC performed better than IBM's Qiskit compiler at optimization level 2, reducing gate counts on average by 8.4% compared to 4.7% (and doing better for rotation gates: 16.1% vs. 8.3%).

SQIR is ultimately not too different from modern circuit-oriented languages/frameworks, such as PyQuil [Rigetti Computing 2019] and Qiskit [Aleksandrowicz et al. 2019], and as such is suitable for writing and proving properties of (small) source programs. As demonstration of this, we have written and proven correct standard programs of interest, including GHZ state preparation, quantum teleportation, superdense coding, and Deutsch-Joza. Indeed, we believe that SQIR is easy to learn and straightforward to use and thus may be a good candidate for pedagogy. We are hope that SQIR will prove useful for teaching concepts of quantum computing and verification in the style of the popular Software Foundations textbook [Pierce et al. 2018].

Moving forward, we plan to flesh out VOQC in support of a full-featured verified compliation stack for quantum programs, following the vision of a recent Computing Community Consortium report [Martonosi and Roetteler 2019]. We can implement and verify additional, known optimizations (e.g., those not yet verified in Table 1). We can implement verified parsers [Jourdan et al. 2012], e.g., for OpenQASM, and then verify their translation to SQIR (e.g., using metaQASM's semantics [Amy 2019]) or other programming languages. We can also add support for hardware-specific transformations that compile to a particular gate set, or perform *error-aware* optimization with the goal of reducing the likelihood of an incorrect answer [Rand et al. 2019].

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040.

REFERENCES

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

Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, 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. (2019). https://doi.org/10.5281/zenodo.2562110

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

Matthew Amy. 2019. Sized Types for Low-Level Quantum Metaprogramming. In *Reversible Computation*, Michael Kirkedal Thomsen and Mathias Soeken (Eds.). Springer International Publishing, Cham, 87–107.

Matthew Amy, Dmitri Maslov, and Michele Mosca. 2013. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33 (03 2013). https://doi.org/10.1109/TCAD.2014.2341953

Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified compilation of space-efficient reversible circuits. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer. https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/

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

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

Coq Development Team. 2019. The Coq Proof Assistant Reference Manual, Version 8.9. (2019). Electronic resource, available from https://coq.inria.fr/refman/.

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

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.

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.

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

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*.

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.

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

Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going Beyond Bell's Theorem.* Springer Netherlands, Dordrecht, 69–72. https://doi.org/10.1007/978-94-017-0849-4_10

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

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

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

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)*.

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*.

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

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

Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Quantum Hoare Logic. *Archive of Formal Proofs* (March 2019). http://isa-afp.org/entries/QHLProver.html, Formal proof development.

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

Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. 2019. Full-stack, Real-system Quantum Computer Studies: Architectural Comparisons and Design Insights. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 527–540. https://doi.org/10.1145/3307650.3322273

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

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

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

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Software Foundations*. Electronic textbook. Version 5.6. https://softwarefoundations.cis.upenn.edu/.

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

Robert Rand. 2019. (2019). Personal communication.

Robert Rand, Kesha Hietala, and Michael Hicks. 2019. Formal Verification vs. Quantum Uncertainty. In *Summit on Advances in Programming Languages, SNAPL 2019, Providence, Rhode Island, 16-17 May 2019*. forthcoming.

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*.

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

Rigetti Computing. 2019. Pyquil Documentation. (2019). http://pyquil.readthedocs.io/en/latest/

Francisco Rios and Peter Selinger. 2017. A categorical model for a quantum circuit description language. In *Proceedings of the 14th International Conference on Quantum Physics and Logic, QPL 2017*.

Neil J. Ross. 2015. *Algebraic and Logical Methods in Quantum Computation*. Ph.D. Dissertation. Dalhousie University.

Mehdi Saeedi, Robert Wille, and Rolf Drechsler. 2011. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing* 10, 3 (01 Jun 2011), 355–377. https://doi.org/10.1007/s11128-010-0201-2

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

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

IBM Research Editorial Staff. 2018. We Have Winners! ... of the IBM Qiskit Developer Challenge. (Aug 2018). https://www.ibm.com/blogs/research/2018/08/winners-qiskit-developer-challenge/

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

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

The Cirq Developers. 2019. Cirq: A python library for NISQ circuits. (2019). https://cirq.readthedocs.io/en/stable/

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

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

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

Alwin Zulehner, Alexandru Paler, and Robert Wille. 2017. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *arXiv e-prints*, Article arXiv:1712.04722 (Dec 2017), arXiv:1712.04722 pages. arXiv:quant-ph/1712.04722

A QWIRE VS. SQIR

When we first set out to build VOQC, we thought to do it using QWIRE [Paykin et al. 2017], another formally verified quantum programming language embedded in Coq. However, we were surprised to find that we had tremendous difficulty proving that even simple transformations were correct. This experience led to the development of SQIR, and raised the question: Why does SQIR seem to make proofs easier, and what do we lose by using it rather than QWIRE?

As mentioned in section 3.3, 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 [Pfenning and Elliott 1988] to take advantage of Coq's variable binding and function composition facilities. 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 specific 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 [de Bruijn 1972]. 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.

As informal evidence of the difficulties of QWIRE's representation on proof, we note that while proving the correctness of a simple `inPar` function in SQIR (see appendix B) took a matter of hours, there is no correctness proof for the corresponding function in QWIRE, despite many months of trying [Rand 2019].

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.

Another important difference between QWIRE and SQIR is that QWIRE 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 QWIRE circuit (only) associates to the right. As such, rewriting using SQIR identities is substantially easier.

There are other noteworthy difference between the two languages. QWIRE's standard denotation function is in terms of superoperators over density matrices, which are harder to work with than simple unitary matrices. QWIRE also provides additional useful tools for quantum programming, like wire types and support for *dynamic lifting*, which passes the control flow to a classical computer before resuming a quantum computation.

The differences between these tools stem from the fact that QWIRE was developed as a programming language for quantum computers [Paykin et al. 2017], and was later used as a verification tool [Rand et al. 2018, 2017]. By contrast, SQIR is mainly a tool for verifying quantum programs, ideally compiled from another language such as Q# [Svore et al. 2018], Quipper [Green et al. 2013] or even QWIRE itself.

B COMPOSITION IN SQIR

SQIR was not designed to be compositional. As such, describing the composition of SQIR programs can be difficult. To begin, consider the following function, which composes two SQIR programs in parallel, which relies on two auxiliary functions.

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

Fixpoint cast {dim} (c : ucom dim) dim' : ucom dim' :=
  match c with
  | uskip ⇒ uskip
  | 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 {dim1 dim2} (c1 : ucom dim1) (c2 : ucom dim2) :=
  (cast c1 (dim1 + dim2)); (cast (map_qubits (fun q ⇒ q + dim1) 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⟧u = ⟦c1⟧u ⊗ ⟦c2⟧u.
```

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 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).