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 qir, 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 qir 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.

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 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!\(^1\)

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 qir (“squire”). While qir 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 qir programs, ultimately producing a result that is compatible with the specified quantum architecture. For added convenience, voqc provides translators between qir and OpenQASM, the de facto standard format for quantum circuits [12]. (Section 2.)

Like CompCert, voqc is implemented using the Coq proof assistant. qir 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 qir 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 qir 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 qir, including GHZ state preparation, quantum teleportation, superdense coding, and the Deutsch-Jozsa algorithm. Generally speaking, qir is designed to make proofs as easy as possible. For example, we initially contemplated qir 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 qir 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 qir programs to satisfy constraints on how qubits may interact.
We begin with a brief background on quantum programs, work, Shi et al. [40] developed CertiQ, which uses symbolic verification in mind, and prior circuit-level optimizations with different objectives, e.g., that aim to reduce mizations with different objectives, e.g., that reduce 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 $\alpha, \beta$ such that $|\alpha|^2 + |\beta|^2 = 1$. The vector $\ket{1, 0}$ represents the state $\ket{0}$ while vector $\ket{0, 1}$ represents the state $\ket{1}$. A state written $\ket{\psi}$ is called a ket, following Dirac’s notation. We say a qubit is in a superposition of $\ket{0}$ and $\ket{1}$ when both $\alpha$ and $\beta$ 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 $\ket{0}$ or $\ket{1}$. 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 $\ket{0}$ yields $\ket{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}}$, also written as $\ket{+}$. 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 $H\ket{+} = \ket{0}$.

A quantum state with $N$ qubits is represented as vector of length $2^N$. For example a 2-qubit state is represented as a matrix multiplication. 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 $\ket{0}$ then both qubits are mapped to themselves, but if the first qubit is $\ket{1}$ then the second qubit is negated, e.g., $CNOT\ket{00} = \ket{00}$ while $CNOT\ket{10} = \ket{11}$.

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

An important non-unitary quantum operator is projection onto a subspace. For example, $\ket{0}\bra{0}$ (in matrix notation $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$) projects a qubit onto the subspace where that qubit is in the $\ket{0}$ state. Projections are useful for describing quantum states after measurement has been performed. We sometimes use $\ket{i}_q\bra{q}$ as shorthand for applying the projection $\ket{i}$ 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 Verified Optimizer for Quantum Circuits

Figure 1. Example quantum program: GHZ state preparation

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

(a) Quantum Circuit

(b) Quil

(c) PyQuil (arbitrary # qubits)

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.
The semantics of unitary \texttt{qir} programs, assuming a global register of dimension $d$. The \texttt{applyy} 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, \texttt{applyy}(X, q, d) = I_{2^d} \otimes \sigma_x \otimes I_{2(d-\ell-i)}$, where $\sigma_x$ is the matrix interpretation of the $X$ gate.

### 3.1 Unitary Core

\texttt{qir} is a language for describing quantum programs that is deeply embedded in the Coq proof assistant. In \texttt{qir}, a qubit is referred to by a natural number that indices into a global register of quantum bits. Unitary \texttt{qir} 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 \texttt{qir} 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 \texttt{qir} 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 \texttt{qir} programs over the gate set $\{R_{\theta, \phi, \lambda}, \text{CNOT}\}$ where $R_{\theta, \phi, \lambda}$ is a general single-qubit rotation parameterized by three real-valued rotation angles and \texttt{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

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

and the matrix interpretation of the \texttt{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 \texttt{qir} programs. For example the \texttt{SWAP} operation, which swaps two qubits, is a sequence of three \texttt{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 $U_1 \equiv U_2$, when there exists a $\theta$ such that $\|U_1\|_d = e^{i\theta} \|U_2\|_d$.

### 3.2 Adding Measurement

To describe general quantum programs $P$, we extend unitary \texttt{qir} with a \texttt{branching measurement} operation.

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

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

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

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 $\rho$, $\|P_1\|_d(\rho) = \|P_2\|_d(\rho)$.

### 3.3 Example

Recall the GHZ preparation example from Section 2. The following Coq function \texttt{ghz} recursively constructs an $n$-qubit
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 $R_{0,\phi}$ 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 $\text{CNOT}$ gate. The result of $\text{GHZ} 3$ is equivalent to the circuit shown in Figure 1(a).

### 3.4 Designing $\text{sqr}$

$sqr$ was designed to be expressive while facilitating mechanically checked proofs of quantum programs. We conclude this section by discussing key features of $\text{sqr}$’s design.

**Expressiveness** As a deeply embedded, domain-specific circuit language, $\text{sqr}$ 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 $\text{sqr}$’s host language, Coq, to prove properties about its programs. For example, we can prove that every circuit generated by $\text{GHZ} n$, above, produces the corresponding state $|\text{GHZ}^n\rangle$ given in Section 2.2 when applied to $|0 \ldots 0\rangle$.

As a demonstration of $\text{sqr}$’s expressiveness as a source language, Appendix A presents proof of this property along with other examples of $\text{sqr}$ 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** $\text{sqr}$’s design has two key features that facilitate verification. First, unlike most quantum languages, $\text{sqr}$ 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 $\text{voqc}$) 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 $\text{sqr}$ 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 $\text{sqr}$ 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 reason about gates acting on different parts of the circuit in isolation).

The alternative, used by Quipper [17] and $\text{QW}$ [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 $\text{sqr}$ Programs

The voqc compiler takes as input a $\text{sqr}$ program and attempts to reduce its total gate count by applying a series of optimizations. This section describes optimizations on unitary $\text{sqr}$ programs. The next section discusses how voqc optimizes full $\text{sqr}$ 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 $\text{sqr}$ 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 $\{H, X, R_{z4}, \text{CNOT}\}$ where $R_{z/4}(k)$ describes rotation about the $z$-axis by $k \cdot \pi/4$ for $k \in Z$. 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 $R_{z/2^n}$ for higher $n$ if finer-grained rotations are needed.

Programs in the list representation are deemed equivalent if their back-converted $\text{sqr}$ programs are equivalent, per
the definition in Section 3.1. Conversion translates voqc’s gates $H$, $X$, and $R_{\pi/4}(k)$ into base gates $R_{\pi/2,0,\pi}$, $R_{\pi,0,\pi}$, and $R_{0,0,\pi/4}$, 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

$$X n; \text{CNOT } m n \equiv \text{CNOT } m n; X n$$

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

$$\text{apply}_1(X, n, d) \times \text{apply}_2(\text{CNOT}, m, n, d) =$$

$$\text{apply}_2(\text{CNOT}, m, n, d) \times \text{apply}_1(X, n, d), \quad (1)$$

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 $q = d - n - 1$:

$$\text{apply}_1(X, n, d) = I_m \otimes \sigma_x \otimes I_{q}$$

$$\text{apply}_2(\text{CNOT}, m, n, d) = I_{2m} \otimes |1\rangle \langle 1| \otimes I_{2p} \otimes \sigma_x \otimes I_{2q} + I_{2m} \otimes |0\rangle \langle 0| \otimes I_{2p} \otimes I_2 \otimes I_{2q}$$

Here, $\sigma_x$ 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., $((\times \ldots) \otimes (\times \ldots)) + ((\times \ldots) \otimes (\times \ldots))$.

\[
X q; H q \equiv H q; Z q \\
Z q; H q \equiv H q; X q \\
X q; R_{\pi/4}(k) q \equiv R_{\pi/4}(8 - k) q; X q \\
X q; \text{CNOT } q; q_1 \equiv \text{CNOT } q; q_2; X q; X q_2 \\
X q_2; \text{CNOT } q; q_1 \equiv \text{CNOT } q; q_2; X q_2 \\
Z q; R_{\pi/4}(k) q \equiv R_{\pi/4}(k) q; Z q \\
Z q_1; \text{CNOT } q; q_2 \equiv \text{CNOT } q; q_2; Z q_1 \\
Z q_2; \text{CNOT } q_1 q_2 \equiv \text{CNOT } q_1 q_2; Z q_1; Z q_2 \\
\]

**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, $I_{2p}$ would be replaced by $I_{2m} \otimes I_2 \otimes I_{2q}$ 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

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

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
rules switch between propagating $X$ and $Z$ (= $R_{\pi/4}(9)$) 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 \( H, R_{\pi/4}, \text{CNOT} \) 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 \texttt{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 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.](image)

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

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

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 \( \{X, R_{\pi/4}, \text{CNOT}\} \) 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 $\{x_1, \ldots, x_n\}$, $C$ will produce the state

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

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

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

is a linear combination of affine boolean functions. $p(x_1, \ldots, x_n)$ 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 $f_i(x_1, \ldots, x_n) = f_j(x_1, \ldots, x_n)$ for some $i \neq j$, then the corresponding $i$ and $j$ rotations can be merged.

As an example, consider the two circuits shown below:

$$Rz_{\pi/4}(k') \quad \Rightarrow \quad Rz_{\pi/4}(k + k')$$

To prove that these circuits are equivalent, we can consider their behavior on basis state $\{x_1, x_2\}$. 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:

$$|x_1, x_2\rangle \rightarrow e^{i(k\pi/4)x_1} |x_1, x_2\rangle$$

$$\quad \rightarrow e^{i(k\pi/4)x_1} |x_1, x_1 \oplus x_2\rangle$$

$$\quad \rightarrow e^{i(k\pi/4)x_2} |x_2, x_1 \oplus x_2\rangle$$

$$\quad \rightarrow e^{i(k\pi/4)x_1} e^{i(k\pi/4)x_2} |x_2, x_1 \oplus x_2\rangle.$$ 

Whereas evaluation of the right-hand circuit produces

$$|x_1, x_2\rangle \rightarrow e^{i((k+k')\pi/4)x_1} |x_1, x_1 \oplus x_2\rangle$$

$$\quad \rightarrow e^{i((k+k')\pi/4)x_2} |x_2, x_1 \oplus x_2\rangle$$

The two resulting states are equal because $e^{i(k\pi/4)x_1} e^{i(k\pi/4)x_2} = e^{i((k+k')\pi/4)x_1}$. 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 $\text{meas} P_1 \ P_2$ whose children $P_1$ and $P_2$ are lists of blocks. Equivalence is defined in terms of the density matrix semantics of the sgn 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
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:

- \( R_z(\pi/4)(k) \) preserves the classical state of \( q \).
- \( X \) q flips the classical state of \( q \).
- If \( i = 0 \) then CNOT \( q q' \) is removed, and if \( i = 1 \) then CNOT \( q q' \) becomes X \( q' \).
- \( \text{meas} \ P_i \) 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

\[
|i\rangle_q \ (|i\rangle \times \rho \times |i\rangle)_q (|i\rangle = \rho,
\]

which says that projecting state \( \rho \) 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 9. Two-qubit gate connections on IBM’s Tenerife machine [21].](image)

For example, consider the connectivity of IBM’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 sgrir 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 \( H \ a; H \ b; \neg \text{CNOT} \ a \ b; H \ a; H \ b \equiv \text{CNOT} \ b \ a \).

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. [3] 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...

![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.](image)
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 \( R_{\pi/4}(1) \) is the familiar \( T \) gate and \( R_{\pi/4}(7) \) is its inverse \( T^\dagger \).

\[
\text{TOFF } a \ b \ c := [ H \ c \ ; \ \text{CNOT } b \ c \ ; \ R_{\pi/4}(7) \ c \ ; \ \text{CNOT } a \ c \ ; \ R_{\pi/4}(1) \ c \ ; \ \text{CNOT } b \ c \ ; \ R_{\pi/4}(7) \ c \ ; \ \text{CNOT } a \ c \ ; \ \text{CNOT } a \ b \ ; \ R_{\pi/4}(1) \ a \ ; \ R_{\pi/4}(1) \ b \ ; \ R_{\pi/4}(1) \ c \ ; \ H \ c ].
\]

**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, \( L \) 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 \( \checkmark \); we write \( \checkmark^* \) 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 \( \{ u_1, u_2, u_3, \text{CNOT} \} \) 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 \( \{ H, X, u_1, \text{CNOT} \} \) 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.

**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 SQR 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).
<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Original</th>
<th>Nam (L)</th>
<th>Nam (H)</th>
<th>Qiskit A</th>
<th>Qiskit B</th>
<th>VOQC</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder_8</td>
<td>900</td>
<td>646</td>
<td>606</td>
<td>869</td>
<td>805</td>
<td>752</td>
</tr>
<tr>
<td>barenc_tof_10</td>
<td>450</td>
<td>294</td>
<td>264</td>
<td>427</td>
<td>394</td>
<td>380</td>
</tr>
<tr>
<td>barenc_tof_3</td>
<td>58</td>
<td>42</td>
<td>40</td>
<td>56</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>barenc_tof_4</td>
<td>114</td>
<td>78</td>
<td>72</td>
<td>109</td>
<td>100</td>
<td>98</td>
</tr>
<tr>
<td>barenc_tof_5</td>
<td>170</td>
<td>114</td>
<td>104</td>
<td>162</td>
<td>149</td>
<td>145</td>
</tr>
<tr>
<td>csla_mux_3</td>
<td>170</td>
<td>161</td>
<td>155</td>
<td>168</td>
<td>156</td>
<td>160</td>
</tr>
<tr>
<td>csum_mux_9</td>
<td>420</td>
<td>294</td>
<td>266</td>
<td>420</td>
<td>382</td>
<td>308</td>
</tr>
<tr>
<td>gf2^4_mult</td>
<td>225</td>
<td>187</td>
<td>187</td>
<td>213</td>
<td>206</td>
<td>192</td>
</tr>
<tr>
<td>gf2^5_mult</td>
<td>347</td>
<td>296</td>
<td>296</td>
<td>327</td>
<td>318</td>
<td>291</td>
</tr>
<tr>
<td>gf2^6_mult</td>
<td>495</td>
<td>403</td>
<td>403</td>
<td>465</td>
<td>454</td>
<td>410</td>
</tr>
<tr>
<td>gf2^7_mult</td>
<td>669</td>
<td>555</td>
<td>555</td>
<td>627</td>
<td>614</td>
<td>549</td>
</tr>
<tr>
<td>gf2^8_mult</td>
<td>883</td>
<td>712</td>
<td>712</td>
<td>819</td>
<td>804</td>
<td>705</td>
</tr>
<tr>
<td>gf2^9_mult</td>
<td>1095</td>
<td>891</td>
<td>891</td>
<td>1023</td>
<td>1006</td>
<td>885</td>
</tr>
<tr>
<td>gf2^10_mult</td>
<td>1347</td>
<td>1070</td>
<td>1070</td>
<td>1257</td>
<td>1238</td>
<td>1084</td>
</tr>
<tr>
<td>gf2^16_mult</td>
<td>3435</td>
<td>2707</td>
<td>2707</td>
<td>3179</td>
<td>3148</td>
<td>2695</td>
</tr>
<tr>
<td>gf2^32_mult</td>
<td>13562</td>
<td>10601</td>
<td>10601</td>
<td>12569</td>
<td>12506</td>
<td>10577</td>
</tr>
<tr>
<td>gf2^64_mult</td>
<td>61629</td>
<td>41563</td>
<td>N/A</td>
<td>49659</td>
<td>49532</td>
<td>41515</td>
</tr>
<tr>
<td>mod5_4</td>
<td>63</td>
<td>51</td>
<td>51</td>
<td>62</td>
<td>58</td>
<td>57</td>
</tr>
<tr>
<td>mod_mult_55</td>
<td>119</td>
<td>91</td>
<td>91</td>
<td>117</td>
<td>106</td>
<td>90</td>
</tr>
<tr>
<td>mod_red_21</td>
<td>278</td>
<td>184</td>
<td>180</td>
<td>261</td>
<td>227</td>
<td>214</td>
</tr>
<tr>
<td>qcla_adder_10</td>
<td>521</td>
<td>411</td>
<td>399</td>
<td>512</td>
<td>469</td>
<td>474</td>
</tr>
<tr>
<td>qcla_com_7</td>
<td>443</td>
<td>284</td>
<td>284</td>
<td>428</td>
<td>398</td>
<td>335</td>
</tr>
<tr>
<td>qcla_mod_7</td>
<td>884</td>
<td>636</td>
<td>624</td>
<td>853</td>
<td>793</td>
<td>764</td>
</tr>
<tr>
<td>rc_adder_6</td>
<td>200</td>
<td>142</td>
<td>140</td>
<td>195</td>
<td>170</td>
<td>167</td>
</tr>
<tr>
<td>tof_10</td>
<td>255</td>
<td>175</td>
<td>175</td>
<td>247</td>
<td>222</td>
<td>215</td>
</tr>
<tr>
<td>tof_3</td>
<td>45</td>
<td>35</td>
<td>35</td>
<td>44</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>tof_4</td>
<td>75</td>
<td>55</td>
<td>55</td>
<td>73</td>
<td>66</td>
<td>65</td>
</tr>
<tr>
<td>tof_5</td>
<td>105</td>
<td>75</td>
<td>75</td>
<td>102</td>
<td>92</td>
<td>90</td>
</tr>
<tr>
<td>vbe_adder_3</td>
<td>150</td>
<td>89</td>
<td>89</td>
<td>146</td>
<td>138</td>
<td>103</td>
</tr>
</tbody>
</table>

**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], quile [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 = 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 s\texttt{qr}, 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 s\texttt{qr} 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 s\texttt{qr} (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


with continuous parameters. npj Quantum Information 4, 1 (2018), 23. https://doi.org/10.1038/s41534-018-0072-4


A Verified Optimizer for Quantum Circuits

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 11. Circuit for superdense coding.

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 \( \text{qir} \) program corresponding to the unitary \( H \) 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 \( \text{qir} \) description of this circuit can be produced by the recursive function in Figure 1(d).

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' => \( \frac{1}{\sqrt{2}} \) * nket n |0\rangle + \( \frac{1}{\sqrt{2}} \) * nket n |1\rangle
end.

Above, nket \( n \) \(|i\rangle \) 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 \(|+\rangle\)), 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 \( \text{qir} \) description of this circuit can be produced by the recursive function in Figure 1(d).

The function \( \text{GHZ} \) describes a family of \( \text{qir} \) circuits: For every \( n \), \( \text{GHZ} n \) is a valid \( \text{qir} \) program and quantum circuit.\(^5\) We aim to show via an inductive proof that every circuit

\[^5\text{For the sake of readability, Figure 1(d) elides a coercion from \( \text{qcom base n} \) to \( \text{qcom base n} \) in the recursive case.}\]

generated by \( \text{GHZ} n \) produces the corresponding \( \text{ghz} n \) vector when applied to \(|0\ldots0\rangle\).

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

Lemma superdense_correct : \( \forall \ b_1 \ b_2, \ |	ext{superdense} b_1 b_2\rangle \times |0,0\rangle = |b_1,b_2\rangle \).

A.2 GHZ State Preparation

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

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

The circuit in Figure 13 corresponds to the following \( \text{qir} \) program:

Definition bell : ucom base 3 := H 1; CNOT 1 2.
Definition alice : com base 3 :=
\( \text{CNOT} 0 \ 1 \ ; \ \text{H} \ 0 \ ; \ \text{measure} \ 0 \ ; \ \text{measure} \ 1 \).
Definition bob : com base 3 :=
\( \text{CNOT} 1 \ 2 \ ; \ \text{CZ} \ 0 \ 2 \ ; \ \text{reset} \ 0 \ ; \ \text{reset} \ 1 \).
Definition teleport : com base 3 := bell; alice; bob.
The bell circuit prepares a bell pair on qubits 1 and 2, which are respectively sent to Alice and Bob. Alice applies CNOT from qubit 0 to qubit 1 and then measures both qubits and (implicitly) sends them to Bob. Finally, Bob performs operations controlled by the (now classical) values on 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 $\rho$, teleport takes the state $\rho \otimes |0\rangle \otimes |0\rangle$ to the state $|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes \rho$. Formally,

**Lemma teleport_correct :** $\forall (\rho : \text{Density 2}), \ WF\_Matrix \ \rho \rightarrow \ \|\text{teleport}\| \rho = |0\rangle \otimes |0\rangle \otimes |0\rangle \otimes \rho$.

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 $\psi$, unitary program $u$ will (deterministically) evaluate to $|u\rangle_4 \times \psi$. However, meas $q_1 \ p_2$ may evaluate to $|1\rangle \times \psi$ or $p_2$ applied to $|0\rangle \times \psi$. 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 Cqg), 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 qvoQ 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 :**

$\forall (\psi : \text{Vector} (2^4)) (\psi' : \text{Vector} (2^3))$, \n
$WF\_Matrix \ \psi \rightarrow \ \text{teleport} / (\psi \otimes |0\rangle) \downarrow \psi' \rightarrow \ \psi' \propto (|0\rangle \otimes \psi)$.

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 $\psi'$ and hence all possible paths to $\psi'$. If instead we simply claimed that

$\text{teleport} / (\psi \otimes |0, 0\rangle) \downarrow 1/2 \times (|\psi\rangle \otimes |\psi\rangle)$,

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 $f : \{0, 1\}^n \rightarrow \{0, 1\}$ 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 qvoQ, we must first give a $\text{squr}$ 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 $f_0(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, b)$ is a Boolean function on $(n - 1)$ bits for $b \in \{0, 1\}$. Similarly, an oracle can be written as $U_f = U_{f_0} \otimes |0\rangle \oplus U_{f_1} \otimes |1\rangle$ for $U_f : |y, x_1, \ldots, x_{n-1}, b\rangle = U_{f_0} |y, x_1, \ldots, x_{n-1}\rangle \oplus U_{f_1} |y, x_1, \ldots, x_{n-1}\rangle |b\rangle$. In the base case ($n = 0$), a Boolean function is a constant function of the form $f(\_\_\_\_\_\_\_) = 0$ or $f(\_\_\_\_\_\_) = 1$ 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 $4 \times 4$ unitary matrices:

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

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

**Inductive boolean :** $\forall \text{dim}, \text{ucom} \text{ dim} \rightarrow \text{Set} :=$

$| \text{boolean}_I : \forall \text{ u}, \text{ u } \equiv I \text{ 0 } \rightarrow \text{ boolean } 1 \text{ u} |

| \text{boolean}_X : \forall \text{ u}, \text{ u } \equiv X \text{ 0 } \rightarrow \text{ boolean } 1 \text{ u} |

| \text{boolean}_U : \forall \text{ dim u u1 u2}, \text{ boolean dim u1 } \rightarrow |$

A2
We define balanced and constant oracles in $\text{sqr}$ as follows.

**Definition balanced** $(\text{dim} : \mathbb{N}) (U : \text{ucom base dim})$ $(P : \text{boolean dim U}) : \mathbb{N} :=$

$$\text{count P} = 2 \cdot (\text{dim} - 1).$$

**Definition constant** $(\text{dim} : \mathbb{N}) (U : \text{ucom base dim}) (P : \text{boolean dim U}) : \mathbb{N} :=$

$$\text{count P} = 0 \lor \text{count P} = 2 \cdot (\text{dim} - 1).$$

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 $\ket{0}$ state, and prepares the input state $\ket{\sim} \otimes \text{nk} \text{et dim} \ket{\sim}$. 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 $\text{sqr}$ in Figure 14. Note the use of Coq function $\text{cpar}$, which constructs a $\text{sqr}$ 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 $\Pi = I \otimes \ket{0}$ and “reject” when the output state is orthogonal to $\Pi$. We can express this in Coq as follows:

**Definition accept** $(\text{dim} : \mathbb{N}) (U : \text{ucom base dim}) (P : \text{boolean dim U}) : \mathbb{N} :=$

$$\forall (\psi : \text{Matrix 2 dim}) \exists (\psi' : \text{Matrix 2 1}), ((\psi \otimes \text{nk} \text{et dim} \ket{0}) \dagger \times \text{deutsch_jozsa dim U} u \times \text{nk} \text{et dim} \ket{0}) \otimes 0 = 1.$$
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

\[
\frac{1}{2} \left( 1 - \frac{s_0}{2^{dim-2}} \right) + \frac{1}{2} \left( 1 - \frac{s_1}{2^{dim-2}} \right) = 1 - \frac{s_0 + s_1}{2^{dim-1}}. \tag{2}
\]

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

For a balanced function, \( \text{count } P \) is equal to \( 2^\dim - 2 \). For a constant function, \( \text{count } P \) is either 0 or \( 2^\dim - 1 \). Thus the factor \( 1 - \frac{\text{count } P}{2^\dim} \) is 0 for a balanced function and \( \pm 1 \) for a constant function. For a balanced function, no input state \( \psi \) 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 \textit{Qwire} vs. \textit{Sqr}

When we first set out to build voqc, we thought to do it using \textit{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 \textit{Sqr}, and raised the question: Why does \textit{Sqr} seem to make proofs easier, and what do we lose by using it rather than \textit{Qwire}?

As discussed in Section 3.4, the fundamental difference between \textit{sqr} and \textit{Qwire} is that \textit{sqr} relies on a global register of qubits. Every operation is applied to an explicit set of qubits within the global register. By contrast, \textit{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 \textit{Qwire}

\textit{Qwire} circuits have the following form:

**Inductive** Circuit (w : WType) : Set :=

| output : Pat w → Circuit w |
| gate : \( \forall \{w1 \, w2\}, \text{Gate } w1 \, w2 \rightarrow \text{Pat } w1 \rightarrow \text{Pat } w2 \rightarrow \text{Circuit } w \) |
| lift : Pat Bit → (\( B \rightarrow \text{Circuit } w \)) → Circuit w |

Patterns Pat type the variables in \textit{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 \textit{continuation} of the form \( \text{Pat } w2 \rightarrow \text{Circuit } w \), 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)

\[
(f : \text{Pat } w1 \rightarrow \text{Circuit } w2) \mapsto \text{Circuit } w2 := \text{match } c \text{ with}
\]

| output p \Rightarrow f p |

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 \textit{boxed} circuits:

**Inductive** Box w1 w2 : Set :=

box : (\( \text{Pat } w1 \rightarrow \text{Circuit } w2 \)) → 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 \Rightarrow let p2 \leftarrow \text{unbox } c1 p1; unbox c2 p2. |

**Definition** inPar (w1 w2 w1' w2') (c1 : Box w1 w2)

(c2 : Box w1' w2') : Box (w1 \oplus w1') (w2 \oplus w2') :=

| box (p1,p2) \Rightarrow let p1' \leftarrow \text{unbox } c1 p1; let p2' \leftarrow \text{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 \textit{sqr}

Composing two \textit{sqr} 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 \textit{sqr} programs in parallel.

**Fixpoint** map_qubits (U dim) (f : N → N)

(c : \text{ucom } U dim) : \text{ucom } U dim :=

| c1; c2 \Rightarrow \text{map_qubits } f \, c1; \text{map_qubits } f \, c2 |
| uapp1 u n \Rightarrow uapp1 u \, (f n) |
| uapp2 u m n \Rightarrow uapp2 u \, (f m) \, (f n) |

end.
The inPar function requires an intermediate gadget. And if we want to transform the former into the latter.

\[
\text{Fixpoint \ cast\ \{U\ \text{dim}\}\ (c : \text{ucom\ U\ dim})\ \text{dim}' :
\quad : \text{ucom\ U\ dim'} :=
\quad \text{match \ c\ with}
\quad \quad |\ c1;\ c2 \Rightarrow \text{cast\ c1\ dim'}\ ;\ \text{cast\ c2\ dim'}
\quad \quad |\ \text{uapp1}\ u\ n \Rightarrow \text{uapp1}\ u\ n
\quad \quad |\ \text{uapp2}\ u\ m\ n \Rightarrow \text{uapp2}\ u\ m\ n
\quad \text{end.}
\]

**Definition inPar\ \{U\ \text{dim1}\ \text{dim2}\}\ (c1 : \text{ucom\ U\ dim1})
\quad (c2 : \text{ucom\ U\ dim2}) :=
\quad \text{cast\ c1\ \{dim1\ \text{+}\ \text{dim2}\}};
\quad \text{cast\ \{map_qubits\ \text{fun} q \Rightarrow \text{dim1}\ \text{+}\ q\ \text{c2}\\} (\text{dim1}\ \text{+}\ \text{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 : \forall\ c1\ c2\ d1\ d2,\ uc\_\text{well\_typed}\ d1\ c1 \rightarrow
\quad \|\text{inPar}\ c1\ c2\ d1\|_{d1\text{+}d2} = \|c1\|_{d1} \otimes \|c2\|_{d2}.

The inPar function is relatively simple, but more involved than the corresponding Qwire definition because it requires relabeling the qubits in program c2.

General composition in qir requires even more involved relabeling functions that are less straightforward to describe. For example, consider the composition expressed in the following Qwire program:

\[
\text{box}\ (ps,\ q) \Rightarrow
\quad \text{let}\ (x,\ y,\ z) \leftarrow \text{unbox}\ c1\ ps;
\quad \text{let}\ (q,\ z) \leftarrow \text{unbox}\ c2\ (q,\ z);
\quad (x,\ y,\ z,\ q).
\]

This program connects the last output of program c1 to the second input of program c2. This operation is natural in Qwire, but describing this type of composition in qir 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 c1 and c2 to indices in the new register (for example, the first qubit in c2 might be mapped to the fourth qubit in the new global register). When qir programs are written directly, this puts extra burden on the programmer. When qir 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 c1 and c2 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 qir’s representations on proof, we note that while proving the correctness of the inPar function in qir 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 qir’s equivalent. qir’s inPar function does not require every qubit within the global register to be used – any gaps will be filled by identity matrices. Also, qir does not allow introducing or discarding qubits, which we suspect will make ancilla management difficult.

**Quantum Data Structures** qir 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 qir, which has neither quantum data structures nor typed circuits, this issue does not present itself.

**Dynamic Lifting** qir 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 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.

The differences between these tools stem from the fact that QWire 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 QWire itself.