Verified Optimization in a Quantum Intermediate Representation

Kesha Hietala Robert Rand Shih-Han Hung
Xiaodi Wu Michael Hicks
University of Maryland, College Park, USA
{kesha, rrand, shung, xwu, mwh}@cs.umd.edu

We present sQIRe, a low-level language for quantum computing and verification. sQIRe uses a global register of quantum bits, allowing easy compilation to and from existing "quantum assembly" languages and simplifying the verification process. We demonstrate the power of sQIRe as an intermediate representation of quantum programs by verifying a number of useful optimizations, and we demonstrate sQIRe's use as a tool for general verification by proving several quantum programs correct.

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. These limitations encourage the development of sophisticated algorithms and clever optimizations that are likely to have mistakes. For example, Nam et al. [14] discovered mistakes in both the theory and implementation of optimizing circuit transformations they developed, and found that the optimization library they compared against sometimes produced incorrect results. Unfortunately, we cannot apply standard software assurance techniques to address these challenges: Unit testing and debugging are infeasible due to both the indeterminacy of quantum algorithms and the substantial expense involved in executing or simulating them.

To address these challenges, we can apply rigorous *formal methods* to the development of quantum programs and programming languages. These methods aim to ensure mathematically-proved correctness of code by construction. A notable success in this arena for classical computing is CompCert [11]. CompCert is a *certified compiler* for C programs. It is written and *proved correct* using the Coq proof assistant [4]. CompCert includes sophisticated optimizations whose proofs of correctness are verified to be valid by Coq's type checker. An experimental evaluation of CompCert's reliability provided strong evidence of the validity of Coq's proof checking: While bug finding tools found hundreds of defects in the gcc and LLVM C compilers, no bugs were found in CompCert's verified core [29].

As a first step toward a certified compiler for quantum programs, Rand et al. [21] developed a proved-correct compiler from a source language describing boolean functions to reversible oracles. This work was implemented in Coq as part of the $\mathcal{Q}$WIRE quantum circuit language [15, 19, 22]. Well-formedness constraints on programs in $\mathcal{Q}$WIRE ensure that resources are used properly (e.g., that all qubits are measured exactly once, and that ancillae are returned to their original state by the end of the circuit), and programs are given a mathematical semantics in terms of density matrices, which is the foundation of proofs of correctness.

After compiling from a source language, the certified compiler should optimize the circuit and map it to machine resources. Rather than express these steps in $\mathcal{Q}$WIRE itself, e.g., as source-to-source transformations, this paper proposes that they should be carried out in a simple quantum intermediate repre-

DRAFT April 8, 2019

sentation we call sQIRe (pronounced "squire"). While QWIRE treats wires abstractly as Coq variables via higher order abstract syntax [16], sQIRe accesses qubits via concrete indices into a global register (Sections 2 and 3).

sQIRe's simple design sacrifices some desirable features of high-level languages, such as variable binding to support easy compositionality. We argue that this is not a significant drawback because the language is intended to be an intermediate representation, produced as the output of compiling a higher-level language. Furthermore, low-level languages (like OpenQASM [5] and QUIL [25], both similar to sQIRe) are more practical for programming near-term devices, which must be aware of both the number of qubits available and their connectivity [18]. We discussed the challenges of programming and verifying such devices in a recent position paper [20].

As a demonstration of sQIRe's utility as an intermediate representation, we have written several optimizations/transformations of sQIRe programs that we have proved correct (Section 4): *Skip elimination*, *Not propagation*, and *Circuit layout mapping*. These transformations were prohibitively difficult to prove in QWIRE but were relatively straightforward in sQIRe. Pleasantly, we also find that sQIRe's simple structure and semantics assists in proving correctness properties about sQIRe programs directly. For example, we prove that the sQIRe program to prepare the GHZ state indeed produces the correct state, and show the correctness of quantum teleportation (Section 5). This shows sQIRe's utility for verifying quantum programs.

The problem of quantum program optimization verification has previously been considered in the context of the ZX calculus [6], but, as far as we are aware, our sQIRe-based transformations are the first certified-correct optimizations that can be applied to a realistic quantum circuit language. Amy et al. [2] developed a proved-correct optimizing compiler from source boolean expressions to reversible circuits, but did not handle general quantum programs. (Rand et al. [21] developed a similar compiler for quantum circuits but without optimizations.) Prior low-level quantum languages [5, 25] have not been developed with verification in mind, and prior circuit-level optimizations [1, 9, 14] have not been formally verified. Some recent efforts have examined using formal methods to prove properties of quantum computing source programs, e.g., Quantum Hoare Logic [30]. This line of work is complementary to ours—a property proved of a source program is guaranteed to be preserved by a certified compiler. In addition, sQIRe can also be used to prove properties about quantum programs by reasoning directly about their semantics.

Our work on sQIRe constitutes a step toward developing a full-scale verified compiler toolchain. Next steps include developing certified transformations from high-level quantum languages to sQIRe and implementing more interesting program transformations. We also hope that sQIRe will prove useful for teaching concepts of quantum computing and verification in the style of the popular Software Foundations textbook [17].

All code we reference in this paper can be found at `https://github.com/inQWIRE/SQIRE`.

2 sQIRe: A Small Quantum Intermediate Representation

sQIRe is a low-level language intended to be used as an intermediate representation in compilers for quantum programming languages. It is built on top of the Coq libraries developed for the QWIRE language. The main simplification in sQIRe compared to QWIRE is that sQIRe assumes a global register of qubits. In sQIRe a qubit is referred to by a natural number that indexes into the global register whereas in QWIRE qubits are referred to using standard Coq variables through the use of higher-order abstract syntax [16]. The benefits and drawbacks of this simplification are discussed in Section 2.3.

$$
\begin{aligned}
P \;&\rightarrow \mathit{skip} \\
&\mid P_1;\ P_2 \\
&\mid U\ q_1\ \dots\ q_n \\[1em]
U \;&\rightarrow H \mid X \mid Y \mid Z \mid R_\phi \mid \mathit{CNOT}
\end{aligned}
\qquad
\begin{aligned}
[\![\mathit{skip}]\!]_u^{dim} &= I_{2^{dim}} \\
[\![P_1;\ P_2]\!]_u^{dim} &= [\![P_2]\!]_u^{dim} \times [\![P_1]\!]_u^{dim} \\
[\![U\ q_1\ \dots\ q_n]\!]_u^{dim} &= \begin{cases} \mathit{ueval}(U,\ q_1 \dots q_n) & \text{well-typed} \\ 0_{2^{dim}} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1: sQIRe abstract syntax and semantics. We use the notation $[\![P]\!]_u^{dim}$ to describe the semantics of unitary program P with a global register of size dim. $\mathit{ueval}(U,\ q_1 \dots q_n)$ returns the expected operation (U for single-qubit gate U and $|1\rangle\langle 1| \otimes X + |0\rangle\langle 0| \otimes I$ for CNOT), extended to the correct dimension by applying an identity operation on every other qubit in the system. For example, $\mathit{ueval}(H,\ q) = I_{2^q} \otimes H \otimes I_{2^{dim-q-1}}$.

```
Inductive ucom : Set :=
| uskip : ucom
| useq : ucom → ucom → ucom
| uapp : ∀ {n}, Unitary n → list ℕ → ucom.

Definition in_bounds (l : list ℕ) (max : ℕ) : ℙ :=
    ∀ x, In x l → x < max.

Inductive uc_well_typed : ℕ → ucom → ℙ :=
| WT_uskip : ∀ dim, uc_well_typed dim uskip
| WT_seq : ∀ dim c1 c2,
      uc_well_typed dim c1 → uc_well_typed dim c2 → uc_well_typed dim (c1; c2)
| WT_app : ∀ dim n l (u : Unitary n),
      length l = n → in_bounds l dim → NoDup l → uc_well_typed dim (uapp u l).
```

Figure 2: Coq definitions of unitary programs and well-typedness.

In this section, we present the syntax and semantics of sQIRe programs. To begin, we restrict our attention to the fragment of sQIRe that describes unitary circuits. We describe the full language, which allows measurement and initialization, in Section 3.

2.1 Syntax and Semantics

Unitary sQIRe programs allow three operations: skip, sequencing, and unitary application (of a fixed set of gates), as shown on the left of Figure 1. Unitary application takes a list of indices into the global register. A unitary program is well-typed if every unitary is applied to valid arguments. A list of arguments is valid if the length of the list is equal to the arity of the unitary operator, every element in the list is bounded by the dimension of the global register, and every element of the list is unique. The first two properties ensure standard well-formedness conditions (function arity and index bounds) while the third enforces linearity and thereby quantum mechanics' no-cloning theorem. The Coq definitions of unitary sQIRe programs and well-typedness are shown in Figure 2.

The semantics for unitary sQIRe programs is shown on the right of Figure 1. 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 inappropriate 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, which would result in proofs becoming cluttered with extra reasoning.

We support a fixed set of gates: H, X, Y, Z, R_ϕ, and $CNOT$. R_ϕ represents a phase shift by an arbitrary real number ϕ. In an effort to simplify the denotation function, the only multi-qubit gate we support is $CNOT$. sQIRe can be easily extended with other built-in gates, or new gates can be defined in terms of existing gates. For example, we define the SWAP operation as follows.

```
Definition SWAP (a b : ℕ) : ucom := CNOT a b; CNOT b a; CNOT a b.
```

We can then state and prove properties about the semantics of the defined operations. For example, we can prove that the SWAP program swaps its arguments, as intended.

2.2 Example

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 3. The sQIRe program corresponding to the unitary part of this circuit is shown in Figure 4. In the sQIRe program, note that `encode` is a Coq function that takes two boolean values and returns a circuit. This shows that although sQIRe's design is simple, we can still express interesting quantum programs using help from the host language, Coq. We will see additional examples of this style of metaprogramming in Section 5.

Although sQIRe is intended to be used as an intermediate representation, because it is embedded in Coq and has a defined semantics, we can also prove properties directly about the semantics of sQIRe programs. For example, 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 $|b_1, b_2\rangle$. In our development, we write this as follows.

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

It is worth noting that we are applying this denotation to a vector, rather than a density matrix, and that we use Dirac (bra-ket) notation to represent this vector. In our experience, it is easier to reason about vectors than matrices and it is significantly easier to reason about quantum states in bra-ket notation compared to manually multiplying vectors by matrices. With this is mind, we have added support for bra-ket reasoning to both sQIRe and $\mathcal{Q}$WIRE.

We will present additional examples of verifying correctness of sQIRe programs in Section 5.

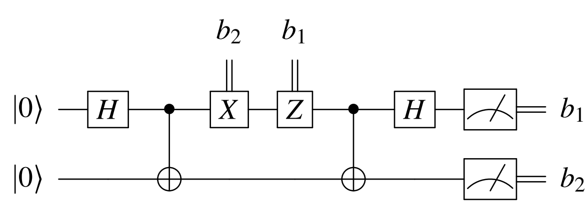


Figure 3: Circuit for the superdense coding algorithm.

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

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

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

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

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

Figure 4: sQIRe program for the unitary portion of the superdense coding algorithm. Note that `U q` is syntactic sugar for applying unitary U to qubit q.

2.3 Discussion

Our use of a global register significantly simplifies proofs about sQIRe programs. Register numbers directly correspond to indices in the vectors that our programs denote. By contrast, $\mathcal{Q}$WIRE's variables map to different indices depending on the local context, which makes it hard to make precise statements about program fragments. We explore this issue in detail in Appendix B.

One downside of using a global register is that it does not allow for easy composition. As discussed in Appendix A, combining separate sQIRe programs requires manually defining a mapping from the global register of one program to the global register of the other. Furthermore, writing sQIRe programs can be tedious because sQIRe is a low-level language that references qubits only through natural numbers. In contrast, writing $\mathcal{Q}$WIRE programs is much like writing programs in any other high-level programming language.$\mathcal{Q}$WIRE naturally allows composition of gates in a manner similar to normal function application.

We believe that sQIRe's lower-level programming style, and the extra work required to perform composition, is not a significant drawback. This is because sQIRe is intended to be used an intermediate representation, and could be produced as the result of compiling a high-level compositional language (like $\mathcal{Q}$WIRE). We expect the compiler from the high-level language to handle the details necessary for composition and produce appropriate sQIRe code.

3 General sQIRe

To describe general quantum programs, we extend sQIRe with operations for initialization and measurement. The command `meas q` measures a qubit and `reset q` measures a qubit and restores it to the $|0\rangle$ state. We present two alternative semantics for general quantum programs: The first is based on density matrices and the second uses a non-deterministic definition to simplify reasoning.

3.1 Density Matrix Semantics

Several previous efforts on verifying quantum programs have defined the semantics of quantum programs as operators on density matrices [15, 30]. We follow this convention here, giving programs their standard

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

Figure 5: sQIRe density matrix semantics. We use the notation $[\![P]\!]_d^{dim}$ to describe the semantics of program P with a global register of size dim. The definition of $ueval$ is given in the caption of Figure 1. 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.

interpretation. The density matrix semantics of general sQIRe programs is given in Figure 5.

We prove the following correspondence between the density matrix semantics (denoted by a subscript d) and the unitary semantics (subscript u) of Figure 1:

```
Lemma c_eval_ucom : ∀ (c : ucom) (dim : ℕ),
    ⟦c⟧_d^dim = fun ρ ⇒ ⟦c⟧_u^dim × ρ × (⟦c⟧_u^dim)† .
```

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.

Note that our language does not include a construct for classical control. This is not a difficult extension, but we chose to keep sQIRe simple to better reflect its intended use as an intermediate representation.

3.2 Non-deterministic Semantics

Our second semantics is based on the observation that quantum computing researchers typically reason about quantum states as vectors rather than density matrices. The non-deterministic semantics allows quantum states to be represented exclusively as vectors by allowing each outcome of a measurement to be reasoned about individually. An illustrative fragment of this semantics is given below.

```
Inductive nd_eval {dim : ℕ} : com → Vector (2^dim) → Vector (2^dim) → ℙ :=
  | nd_app : ∀ n (u : Unitary n) (l : list ℕ) (ψ : Vector (2^dim)),
      app u l / ψ ⇓ ((ueval dim u l) × ψ)
  | nd_meas0 : ∀ n (ψ : Vector (2^dim)),
      let ψ' := pad n dim |0⟩⟨0| × ψ in
      norm ψ' ≠ 0
      meas n / ψ ⇓ ψ'
  | nd_meas1 : ∀ n (ψ : Vector (2^dim)),
      let ψ' := pad n dim |1⟩⟨1| × ψ in
      norm ψ' ≠ 0 →
      meas n / ψ ⇓ ψ'

where "c '/' ψ '⇓' ψ'" := (nd_eval c ψ ψ').
```

Evaluation is given here as a relation. The `nd_app` rule says that, given state ψ, `app u l` evaluates to (`ueval dim u l`) $\times \psi$, as expected. The `nd_meas0` rule says that, if the result of projecting the n^{th} qubit onto the $|0\rangle\langle 0|$ subspace is not the zero matrix, measuring n yields this projection. Note that most quantum states can step via the `nd_meas0` or `nd_meas1` relation. Reset behaves non-deterministically like measurement, but sets the resulting qubit to $|0\rangle$. To simplify the reasoning process, we do not rescale the output of measurement: As is standard in quantum computing proofs, the user may choose to reason about the normalized output of a program or to simply prove a property that is invariant to scaling factors.

Again, we can show that the non-deterministic semantics of the unitary fragment of sQIRe is identical to the unitary semantics, albeit in relational form:

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

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

We give an example of reasoning with both the density matrix semantics and the non-deterministic semantics in Section 5. The end goal of our work on this semantics, and other simplifications that we have made in the the design of sQIRe, is to make sQIRe into a tool that can be used for intuitive reasoning, by both practitioners and pedagogues.

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 sophisticated optimizations to reduce resource usage. These optimizations can be complicated to implement and are vulnerable to programmer error. It is thus important to verify that the implementations of program optimizations are correct. Our work in this section is a first step toward a verified-correct optimizer for quantum programs.

We begin by discussing equivalence of sQIRe programs. We then discuss a simple optimization on unitary programs that removes all possible skip gates. We follow this with a more realistic optimization, which removes unnecessary X gates from a unitary program. Finally, we verify a different type of useful program transformation—circuit layout mapping.

4.1 Equivalence of sQIRe Programs

In general, we will be 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 sQIRe programs is defined as follows:

```
Definition uc_equiv (c1 c2 : ucom) := ∀ dim, ⟦c1⟧_u^dim = ⟦c2⟧_u^dim.
Infix "≡" := uc_equiv.
```

This definition has several nice properties, including the following.

```
Lemma useq_assoc : ∀ c1 c2 c3, ((c1 ; c2) ; c3) ≡ (c1 ; (c2 ; c3)).

Lemma useq_congruence : ∀ c1 c1' c2 c2',
    c1 ≡ c1' →
    c2 ≡ c2' →
    c1 ; c2 ≡ c1' ; c2'.
```

```
Fixpoint rm_uskips (c : ucom) : ucom :=
  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.
```

Figure 6: Skip removal optimization.

Associativity and congruence are both important for proving soundness of transformations. For example, in order to prove that the not propagation optimization (which cancels adjacent X gates) is sound, we need to prove that the program `c` has the same denotation as `X q; X q; c` (for any `c`). We reason as follows: `X q; X q; c` ≡ `(X q; X q); c` by associativity. `(X q; X q); c` ≡ `uskip; c` by applying congruence and using the fact that `X q; X q` ≡ `uskip`. Finally, `uskip; c` ≡ `c` by the identity that says that we can remove a skip on the left without affecting a program's denotation.

4.2 Skip Removal

The skip removal function is shown in Figure 6. To show that this function is semantics-preserving, we prove the following lemma.

```
Lemma rm_uskips_sound : ∀ c, 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`. For example, we can prove that the output of `rm_uskips` is either a single skip operation, or contains no skip operations.

```
Inductive skip_free : ucom → ℙ :=
  | SF_seq : ∀ c1 c2, skip_free c1 → skip_free c2 → skip_free (c1; c2)
  | SF_app : ∀ n l (u : Unitary n), skip_free (uapp u l).

Lemma rm_uskips_correct : ∀ c,
    (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 (c : ucom) : ℕ :=
  match c with
  | c1; c2 ⇒ (count_ops c1) + (count_ops c2)
  | _ ⇒ 1
  end.

Lemma rm_uskips_reduces_count : ∀ c, count_ops (rm_uskips c) ≤ count_ops c.
```

4.3 Not Propagation

We now present a more realistic optimization, which removes unnecessary X gates from a program. This optimization is used as a pre-processing step in a recent quantum circuit optimizer [14]. For each X gate in the circuit, this optimization will propagate the X gate as far right as possible, commuting through the target of $CNOT$ gates, until a cancelling X gate is found. If a cancelling X gate is found, then both gates are removed from the circuit. If no cancelling X gate is found, then the propagated gate is returned to its original position. The structure of this optimization function, and its associated proofs, can be adapted to other propagation-based optimizations (e.g. the "single-qubit gate cancellation" routine from the same optimizer [14]).

We have proven that this optimization is semantics-preserving. The main lemmas that the proof relies on are:

```
Lemma XX_id : ∀ q, uskip ≡ X q; X q.

Lemma X_CNOT_comm : ∀ c t, X t; CNOT c t ≡ CNOT c t ; X t.

Lemma U_V_comm : ∀ (m n : ℕ) (U V : Unitary 1),
  m ≠ n → (U m ; V n) ≡ (V n ; U m).

Lemma U_CNOT_comm : ∀ (q n1 n2 : ℕ) (U : Unitary 1),
  q ≠ n1 → q ≠ n2 → (U q ; CNOT n1 n2) ≡ (CNOT n1 n2 ; U q).
```

The first lemma says that adjacent X gates cancel. The second lemma says that an X gate commutes through the target of a $CNOT$. The third and fourth lemmas says that single-qubit unitary U commutes with any other 1- or 2-qubit gate that accesses distinct qubits. This final lemma is necessitated by our representation of circuits as a list of instructions: In order to discover adjacent X gates, we may need to superficially reorder the instruction list.

4.4 Circuit Mapping

We can also use SQIRe to verify another useful class of transformations—circuit layout 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 [23, 31]. Circuit mapping algorithms take as input an arbitrary circuit and output a circuit that respects the connectivity constraints of some underlying architecture. To our knowledge, no previous circuit mapping algorithm has been developed with verification in mind.

Here we consider a toy architecture and mapping algorithm. We assume 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 : ucom → 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) q, respects_LNN (U q)
  | 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 skip 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 many more SWAPs than the optimal solution, but our verification framework could be applied to optimized implementations as well.

We have proven that this transformation is semantics-preserving, and that the output program satisfies the LNN constraint.

5 sQIRe for General Verification

As we have alluded to several times so far, sQIRe is useful for more than just verifying program optimizations. Its simple structure and semantics also allow us to easily verify general properties of quantum programs. This makes sQIRe a good candidate for introducing students to concepts of verification and quantum computing.

In this section we discuss correctness properties of two quantum programs, written in sQIRe, that could be introduced in an introductory course on quantum computing.

5.1 GHZ State Preparation

The Greenberger-Horne-Zeilinger (GHZ) state [8] is an n-qubit entangled quantum state of the form

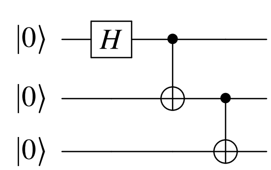
$$|\mathrm{GHZ}\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}). \tag{1}$$

This vector can be defined in Coq as follows:

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

Above, `nket n` $|i\rangle$ 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, prepares a $|+\rangle$ state in the first qubit, and then applies a *CNOT* to every other qubit with the previous qubit as the control. A circuit that prepares the 3-qubit GHZ state is shown below. The sQIRe description of (the unitary portion of) this circuit can be produced by the following recursive function.

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



The function `GHZ` describes a *family* of sQIRe circuits: For every n, `GHZ n` is a valid sQIRe program and quantum circuit. Like in Section 2.2, we use the general-purpose nature of Coq to produce interesting sQIRe circuits. We aim to show via an inductive proof that every circuit generated by `GHZ n` produces the corresponding Coq `ghz n` vector when applied to $|0\ldots0\rangle$. We prove the following theorem:

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

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

5.2 Teleportation

Quantum teleportation is one of the first quantum programs shown in introductory classes on quantum computing. We present it here to highlight the difference between the density matrix semantics and non-deterministic semantics presented in Section 3. The sQIRe program for quantum teleportation is given below.

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

The correctness property of quantum teleportation says that the input qubit is the same as the output qubit. Since sQIRe doesn't permit us to discard wires, instead we reset them back to $|0\rangle$, leaving us with the following theorem to prove under the density semantics:

```
Lemma teleport_correct : ∀ (ρ : Density (2^1)),
  WF_Matrix ρ →
  ⟦teleport⟧³_d (ρ ⊗ |0⟩⟨0| ⊗ |0⟩⟨0|) = |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ ρ.
```

The proof for the density semantics is quite simple: We compute the products with our matrix solver, and do some simple arithmetic to show that the output matrix has the desired form. (This arithmetic is also automated for us.) While short, this proof doesn't give much intuition about how quantum teleportation works.

For the non-deterministic semantics the proof is a bit more complicated, but also more illustrative of the inner workings of the teleport algorithm. For the non-deterministic semantics, the theorem we aim to prove is 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. Having stated the lemma, we can now try to prove it.

The first half of the circuit is unitary, so we can simply compute the effects of applying a Hadamard, two *CNOT*s and another Hadamard to the input state. We can then take both of the 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.

6 Conclusions and Future Work

We have presented sQIRe, a simple, low-level quantum language embedded in Coq. We showed that sQIRe can serve as an intermediate representation for compiled quantum programs and verified several transformations of sQIRe programs. We also showed how to directly verify the correctness of sQIRe programs. Compared to previous languages for verification of quantum programs, sQIRe is easy to learn and straightforward to use and thus, we believe, a good candidate for pedagogy. We hope that sQIRe can be used as the basis for an introduction to quantum computing in the style of Software Foundations [17].

Moving forward, we hope to make more progress toward a full-featured verified compilation stack for quantum programs, from verified transformation of high-level quantum languages to sQIRe code to verified production of circuits that run on real quantum machines, following the vision of a recent Computing Community Consortium report [13]. We also plan to implement additional verified mapping and optimization functions, and transformation functions that take into account other limitations of near-term quantum machines, such as their high rate of error, as envisioned by Rand et al. [20]. We are also looking at extending sQIRe with branching measurements and while loops, in the style of Selinger's QPL [24] or Ying's Quantum While language [30], allowing us to implement and verify the many recently-developed quantum Hoare logics [10, 27, 28, 30]. An Isabelle implementation of the first logic was recently published to the Archive of Formal Proofs [12] but we believe sQIRe is sufficiently versatile to handle a range of interesting logics.

References

[1] Matthew Amy, Dmitri Maslov & 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, doi:10.1109/TCAD.2014.2341953.

[2] Matthew Amy, Martin Roetteler & 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. Available at `https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/`.

[3] 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)*, 75, Elsevier, pp. 381–392.

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

[5] Andrew W. Cross, Lev S. Bishop, John A. Smolin & Jay M. Gambetta (2017): *Open Quantum Assembly Language*. *arXiv e-prints*:arXiv:1707.03429.

[6] Andrew Fagan & 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*.

[7] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & 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, pp. 333–342.

[8] Daniel M. Greenberger, Michael A. Horne & Anton Zeilinger (1989): *Going Beyond Bell's Theorem*, pp. 69–72. Springer Netherlands, Dordrecht, doi:10.1007/978-94-017-0849-4_10. Available at `https://doi.org/10.1007/978-94-017-0849-4_10`.

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

[10] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks & Xiaodi Wu (2019): *Quantitative Robustness Analysis of Quantum Programs*. Proc. ACM Program. Lang. 3(POPL), pp. 31:1–31:29, doi:10.1145/3290344. Available at `http://doi.acm.org/10.1145/3290344`.

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

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

[13] Margaret Martonosi & Martin Roetteler (2019): *Next Steps in Quantum Computing: Computer Science's Role*.

[14] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs & Dmitri Maslov (2018): *Automated optimization of large quantum circuits with continuous parameters*. npj Quantum Information 4(1), p. 23, doi:10.1038/s41534-018-0072-4. Available at `https://doi.org/10.1038/s41534-018-0072-4`.

[15] Jennifer Paykin, Robert Rand & 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, pp. 846–858, doi:10.1145/3009837.3009894.

[16] Frank Pfenning & 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, pp. 199–208, doi:10.1145/53990.54010.

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

[18] John Preskill (2018): *Quantum Computing in the NISQ era and beyond*. Quantum 2, p. 79, doi:10.22331/q-2018-08-06-79. Available at `https://doi.org/10.22331/q-2018-08-06-79`.

[19] Robert Rand (2018): *Formally Verified Quantum Programming*. Ph.D. thesis, University of Pennsylvania.

[20] Robert Rand, Kesha Hietala & 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.

[21] Robert Rand, Jennifer Paykin, Dong-Ho Lee & 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.

[22] Robert Rand, Jennifer Paykin & Steve Zdancewic (2017): *QWIRE Practice: Formal Verification of Quantum Circuits in Coq*, pp. 119–132. doi:10.4204/EPTCS.266.8. Available at `https://doi.org/10.4204/EPTCS.266.8`.

[23] Mehdi Saeedi, Robert Wille & Rolf Drechsler (2011): *Synthesis of quantum circuits for linear nearest neighbor architectures*. Quantum Information Processing 10(3), pp. 355–377, doi:10.1007/s11128-010-0201-2. Available at `https://doi.org/10.1007/s11128-010-0201-2`.

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

[25] Robert S. Smith, Michael J. Curtis & William J. Zeng (2016): *A Practical Quantum Instruction Set Architecture*. arXiv e-prints:arXiv:1608.03355.

[26] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz & 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, p. 7.

[27] Dominique Unruh (2019): *Quantum Hoare Logic with Ghost Variables*. *arXiv preprint arXiv:1902.00325*.

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

[29] Xuejun Yang, Yang Chen, Eric Eide & John Regehr (2011): *Finding and Understanding Bugs in C Compilers*. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, pp. 283–294.

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

[31] Alwin Zulehner, Alexandru Paler & Robert Wille (2017): *An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures*. *arXiv e-prints*:arXiv:1712.04722.

A Composition in SQIRe

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

```
Fixpoint map_qubits (f : ℕ → ℕ) (c : ucom) :=
  match c with
  | uskip ⇒ uskip
  | c1; c2 ⇒ map_qubits f c1; map_qubits f c2
  | uapp u l ⇒ uapp u (map f l)
  end.

Definition inPar (c1 c2 : ucom) (dim1 dim2 : ℕ) :=
  c1; map_qubits (fun q ⇒ q + dim1) c2.
```

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 dim1 dim2,
  uc_well_typed dim1 c1 →
  uc_eval (dim1 + dim2) (inPar c1 c2 dim1 dim2)
      = (uc_eval dim1 c1) ⊗ (uc_eval dim2 c2).
```

The `inPar` function is relatively simple, but more involved than the corresponding $\mathcal{Q}$WIRE definition because it requires relabeling the qubits in program c_2. More general types of composition require more involved relabeling functions that are less straightforward to describe. For example, consider the following $\mathcal{Q}$WIRE program:

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

This program connects the last output of circuit c_1 to the second input of circuit c_2. This operation is natural in $\mathcal{Q}$WIRE, but describing this type of composition in SQIRe 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 sQIRe programs are written directly, this puts extra burden on the programmer. When sQIRe 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 produce manually or automatically).

B 𝒬WIRE vs. sQIRe

The fundamental difference between sQIRe and its sibling, 𝒬WIRE [15], is that sQIRe relies on a global register of qubits and every operation must explicitly apply to a set of qubits within the global register. By contrast, 𝒬WIRE uses Higher Order Abstract Syntax [16] 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. In the definition of `gate`, we provide 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.
```

Here 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 and we can write 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 Bruijin-style variable representation [3]. 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 $\mathcal{Q}$WIRE's representation on proof, we note that while the proof of `inPar`'s correctness in sQIRe (see Appendix A) took a matter of hours, we still lack a correctness proof for the corresponding function in $\mathcal{Q}$WIRE after many months of trying.

Of course, this comparison isn't entirely fair: $\mathcal{Q}$WIRE's `inPar` is quite a bit more powerful than sQIRe's equivalent. sQIRe's `inPar` function asks the user to provide the dimensions of the two input functions and ensure that they only use the correct registers inside each function. It doesn't require the user to use all of the qubits within those dimensions – any gaps will be filled by identity matrices. And, of course, sQIRe doesn't allow us to initialize and discard wires inside these circuits, which makes ancilla management in particular difficult.

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

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

The difference between these tools stem from the fact that $\mathcal{Q}$WIRE was born as a programming language for quantum computers [15], and was later used as a verification tool, with some success stories [22, 21]. By contrast, sQIRe is mainly a tool for verifying quantum programs, ideally compiled from another languages, such as Q# [26], Quipper [7] or $\mathcal{Q}$WIRE itself.