Proving Quantum Programs Correct

Kesha Hietala
University of Maryland, College Park, MD, USA

Robert Rand
University of Chicago, IL, USA

Shih-Han Hung
University of Maryland, College Park, MD, USA

Liyi Li
University of Maryland, College Park, MD, USA

Michael Hicks
University of Maryland, College Park, MD, USA

Abstract

As quantum computing progresses steadily from theory into practice, programmers will face a common problem: How can they be sure that their code does what they intend it to do? This paper presents encouraging results in the application of mechanized proof to the domain of quantum programming in the context of the SQIR development. It verifies the correctness of a range of a quantum algorithms including Grover's algorithm and quantum phase estimation, a key component of Shor's algorithm. In doing so, it aims to highlight both the successes and challenges of formal verification in the quantum context and motivate the theorem proving community to target quantum computing as an application domain.

2012 ACM Subject Classification Hardware → Quantum computation; Software and its engineering → Formal software verification

Keywords and phrases Formal Verification, Quantum Computing, Proof Engineering

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.21

Related Version *Extended Version*: `https://arxiv.org/abs/2010.01240`

Supplementary Material *Software*: `https://github.com/inQWIRE/SQIR`
archived at `swh:1:dir:2aed711638324c36b815c91d1f8b9ad7ca46a300`

Funding 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 and the Air Force Office of Scientific Research under Grant No. FA95502110051.

Acknowledgements We thank Yuxiang Peng for ongoing contributions to the SQIR codebase and pointing out a bug in our original specification for QPE. We thank Xiaodi Wu for discussions about SQIR and follow-on projects.

1 Introduction

Quantum computers are fundamentally different from the "classical" computers we have been programming since the development of the ENIAC in 1945. This difference includes a layer of complexity introduced by quantum mechanics: Instead of a deterministic function from inputs to outputs, a quantum program is a function from inputs to a *superposition* of outputs, a notion that generalizes probabilities. As a result, quantum programs are strictly more expressive than probabilistic programs and even harder to get right. While we can test the output of a probabilistic program by comparing its observed distribution to the desired one, doing the same on a quantum computer can be prohibitively expensive and may not fully describe the underlying quantum state.

This challenge for quantum programming is an opportunity for formal methods. We can use formal methods to *prove*, in advance, that the code implementing a quantum algorithm does what it should for all possible inputs and configurations.

In prior work [17], we developed a formally verified optimizer for quantum programs (VOQC), implemented and proved correct in the Coq proof assistant [8]. VOQC transforms programs written in SQIR, a *small quantum intermediate representation*. While we designed SQIR to be a compiler intermediate representation, we quickly realized that it was not so different from languages used to write *source* quantum programs, and that the design choices that eased proving optimizations correct could ease proving source programs correct, too.

To date, we have proved the correctness of implementations of a number of quantum algorithms, including quantum teleportation, Greenberger–Horne–Zeilinger (GHZ) state preparation [15], the Deutsch-Jozsa algorithm [10], Simon's algorithm [32], the quantum Fourier transform (QFT), quantum phase estimation (QPE), and Grover's algorithm [16]. QPE is a key component of Shor's prime-factoring algorithm [31], today's best-known, most impactful quantum algorithm, with Grover's algorithm for unstructured search being the second. Our implementations can be extracted to code that can be executed on quantum hardware or simulated classically, depending on the problem size and hardware limitations.

While SQIR was first introduced as part of VOQC, this paper offers two new contributions. First, it presents a detailed discussion of how SQIR's design supports proofs of correctness. After presenting background on quantum computing (Section 2) and reviewing SQIR (Section 3), Section 4 discusses key elements of SQIR's design and compares and contrasts them to design decisions made in the related tools QWIRE [25], QBRICKS [7], and the Isabelle implementation of quantum Hoare logic [18]. SQIR's overall benefit over these tools is its flexibility, supporting multiple semantics and approaches to proof. As a second contribution, this paper presents the code, formal specification, and proof sketch of Grover's algorithm, QFT, and QPE, which are the most sophisticated algorithms that we have verified so far (Section 5). We comment on the proofs of simpler algorithms in Appendix B of the extended version of this paper. We believe there is ripe opportunity for further application of formal methods to quantum computing and we hope this paper, and our work on SQIR, paves the way for new research; we sketch open problems in Section 6.

SQIR is implemented in just over 3500 lines of Coq, with an additional 3700 lines of example SQIR programs and proofs; it is freely available on Github.[1]

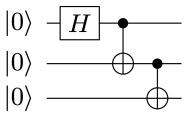
2 Background

We begin with a light background on quantum computing; for a full treatment we recommend the standard text on the subject [22].

2.1 Quantum States

A quantum state consists of one or more *quantum bits*. A quantum bit (or *qubit*) can be expressed as a two dimensional vector $\binom{\alpha}{\beta}$ such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write this vector as $\alpha|0\rangle + \beta|1\rangle$ where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are *basis states*. When both α and β are non-zero, we can think of the qubit as being "both 0 and 1 at once," a.k.a. a *superposition*. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is an equal superposition of $|0\rangle$ and $|1\rangle$.

[1] https://github.com/inQWIRE/SQIR



(a) Quantum circuit

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

(b) SQIR assembly

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

(c) SQIR meta-program

```
box (x,y,z) ⇒
  gate x     ← H x;
  gate (x,y) ← CNOT (x,y);
  gate (y,z) ← CNOT (y,z);
  output (x,y,z).
```

(d) $\mathcal{Q}$WIRE

```
SEQ(SEQ(PAR(H, PAR(I, I)),
        PAR(CNOT, I)),
    PAR(I, CNOT))
```

(e) $\mathcal{Q}$BRICKS-DSL

```
q1 := H q1;
q1,q2 := CNOT q1,q2;
q2,q3 := CNOT q2,q3
```

(f) QWhile

$$(I \otimes CNOT) \times (CNOT \otimes I) \times (H \otimes I \otimes I)$$

(g) Matrix expression

Figure 1 Example quantum program: GHZ state preparation.

We can join multiple qubits together by means of the *tensor product* ($\otimes$) from linear algebra. For convenience, we write $|i\rangle \otimes |j\rangle$ as $|ij\rangle$ for $i, j \in \{0, 1\}$; we may also write $|k\rangle$ where $k \in \mathbb{N}$ is the decimal interpretation of bits ij. We use $|\psi\rangle$ to refer to an arbitrary quantum state. Sometimes a multi-qubit state cannot be expressed as the tensor of individual qubits; such states are called *entangled*. One example is the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*.

2.2 Quantum Programs

Quantum programs are composed of a series of *quantum operations*, each of which acts on a subset of qubits in the quantum state. In the standard presentation, quantum programs are expressed as *circuits*, as shown in Figure 1(a). In these circuits, each horizontal wire represents a qubit and boxes on these wires indicate quantum operations, or *gates*. The circuit in Figure 1(a) uses three qubits and applies three gates: the *Hadamard* (`H`) gate and two *controlled-not* (`CNOT`) gates. The semantics of a gate is a *unitary matrix* (a matrix that preserves the unitarity invariant of quantum states); applying a gate to a state is tantamount to multiplying the state vector by the gate's matrix. The matrix corresponding to the circuit in Figure 1(a) is shown in Figure 1(g), where I is the 2×2 identity matrix, $CNOT$ is the matrix corresponding to the `CNOT` gate, and H is the matrix corresponding to the `H` gate.

A special, non-unitary *measurement* operation is used to extract classical information from a quantum state (often, when a computation completes). Measurement collapses the state to one of the basis states with a probability related to the state's amplitudes. For example, measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ will collapse the state to $|0\rangle$ with probability $\frac{1}{2}$ and likewise for $|1\rangle$, returning classical values 0 or 1, respectively. The semantics of a program involving measurement amounts to a probability distribution over quantum states; such a distribution is called a *mixed state*. In our example above, measurement produces a mixed state that is a uniform distribution over $|0\rangle$ and $|1\rangle$. By contrast, *pure states* like $|0\rangle$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ can be produced without measurement. Section 3.3 discusses non-unitary semantics further.

3 SQIR: A Small Quantum Intermediate Representation

SQIR is a simple quantum language deeply embedded in the Coq proof assistant. This section presents SQIR's syntax and semantics. We defer a detailed discussion of SQIR's design rationale to the next section.

3.1 Unitary SQIR: Syntax

SQIR's *unitary fragment* is a sub-language of full SQIR for expressing programs consisting of unitary gates. (The full SQIR language extends unitary SQIR with measurement.) A program in the unitary fragment has type `ucom` (for "unitary command"), which we define in Coq as follows:

```
Inductive ucom (U: ℕ → Set) (d : ℕ) : Set :=
| useq  : ucom U d → ucom U d → ucom U d
| uapp1 : U 1 → ℕ → ucom U d
| uapp2 : U 2 → ℕ → ℕ → ucom U d
```

The `useq` constructor sequences two commands; we use notational shorthand `p1 ; p2` for `useq p1 p2`. The two `uappn` constructors indicate the application of a quantum gate to n qubits, where n is 1 or 2. Qubits are identified as numbered indices into a *global qubit register* of size d, which stores the quantum state. Gates are drawn from parameter `U`, which is indexed by a gate's size. For writing and verifying programs, we use the following `base` set for `U`, inspired by IBM's OpenQASM [9]:[2]

```
Inductive base : ℕ → Set :=
  | U_R (θ ϕ λ : R) : base 1
  | U_CNOT          : base 2.
```

That is, we have a one-qubit gate `U_R` (which we write U_R when using math notation), which takes three real-valued arguments, and the standard two-qubit *controlled-not* gate, `U_CNOT` (written $CNOT$ in math notation), which negates the second qubit wherever the first qubit is $|1\rangle$, making it the quantum equivalent of a *xor* gate. The `U_R` gate can be used to express any single-qubit gate (see Section 3.2). Together, `U_R` and `U_CNOT` form a *universal* gate set, meaning that they can be composed to describe any unitary operation [3].

Example: SWAP

The following Coq function produces a unitary SQIR program that applies three controlled-not gates in a row, with the effect of exchanging two qubits in the register. We define `CNOT` as shorthand for `uapp2 U_CNOT`.

```
Definition SWAP d a b : ucom base d := CNOT a b; CNOT b a; CNOT a b.
```

[2] It is helpful for proofs to keep `U` small because the number of cases in the proof about a value of type `ucom U d` will depend on the number of gates in `U`. In our work on VOQC [17], we define optimizations over a larger gate set that includes common gates like Hadamard, but convert these gates to our `base` set for proof.

Example: GHZ

Figure 1(b) is the SQIR representation of the circuit in Figure 1(a), which prepares the three-qubit GHZ state [15]. We describe *families* of SQIR circuits by meta-programming in the Coq host language. The Coq function in Figure 1(c) produces a SQIR program that prepares the n-qubit GHZ state, producing the program in Figure 1(b) when given input 3. In Figures 1(b–c), `H` and `I` apply the `U_R` encodings of the Hadamard and identity gates.

3.2 Unitary SQIR: Semantics

Each k-qubit quantum gate corresponds to a $2^k \times 2^k$ unitary matrix. The matrices for our `base` set are:

$$[\![U_R(\theta,\phi,\lambda)]\!] = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix}, \qquad [\![CNOT]\!] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Conveniently, the U_R gate can encode any single-qubit gate [22, Chapter 4]. For instance, two commonly-used single-qubit gates are X ("not") and H ("Hadamard"). The former has the matrix $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ and serves to flip a qubit's α and β amplitudes; it can be encoded as $U_R(\pi, 0, \pi)$. The H gate has the matrix $\frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$, and is often used to put a qubit into superposition (it takes $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$); it can be encoded as $U_R(\pi/2, 0, \pi)$. Multi-qubit gates are easily produced by combinations of $CNOT$ and U_R; we show the definition of the three-qubit "Toffoli" gate in Section 4.6. Keeping our gate set small simplifies the language and enables easy case analysis – and does not complicate proofs. We rarely unfold the definition of gates like X or the three-qubit Toffoli, instead providing automation to directly translate these gates to their intended denotations. Hence, X is translated directly to $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. Users can thereby easily extend SQIR with new gates and denotations.

A unitary SQIR program operating on a size-d register corresponds to a $2^d \times 2^d$ unitary matrix. Function `uc_eval` denotes the matrix corresponding to program `c`.

```
Fixpoint uc_eval {d} (c : ucom base d) : Matrix (2^d) (2^d) := ...
```

We write $[\![\texttt{c}]\!]_d$ for `uc_eval d c`. The denotation of composition is simple matrix multiplication: $[\![\texttt{U1; U2}]\!]_d = [\![\texttt{U2}]\!]_d \times [\![\texttt{U1}]\!]_d$. The denotation of `uapp1` is the denotation of its argument gate, but padded with the identity matrix so it has size $2^d \times 2^d$. To be precise, we have:

$$[\![\texttt{uapp1 U q}]\!]_d = \begin{cases} I_{2^q} \otimes [\![U]\!] \otimes I_{2^{d-q-1}} & q < d \\ 0_{2^d} & \text{otherwise} \end{cases}$$

where I_n is the $n \times n$ identity matrix. In the case of our `base` gate set, $[\![U]\!]$ is the U_R matrix shown above. The denotation of any gate applied to an out-of-bounds qubit is the zero matrix, ensuring that a circuit corresponds to a zero matrix if and only if it is ill-formed. We likewise prove that every well-formed circuit corresponds to a unitary matrix.

As our only two-qubit gate in the `base` set is `U_CNOT`, we specialize our semantics for `uapp2` to this gate. To compute $[\![\texttt{CNOT q1 q2}]\!]_d$, we first decompose the $CNOT$ matrix into $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right) \otimes I_2 + \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right) \otimes X$. We then pad the expression appropriately, obtaining the following when $q_1 < q_2 < d$:

$$I_{2^{q_1}} \otimes \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right) \otimes I_{2^{q_2-q_1-1}} \otimes I_2 \otimes I_{2^{d-q_2-1}} \;+\; I_{2^{q_1}} \otimes \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right) \otimes I_{2^{q_2-q_1-1}} \otimes X \otimes I_{2^{d-q_2-1}}.$$

When $q_2 < q_1 < d$, we obtain a symmetric expression, and when either qubit is out of bounds, we get the zero matrix. Additionally, since the two inputs to $CNOT$ cannot be the same, if $q_1 = q_2$ we also obtain the zero matrix.

Example: Verifying SWAP

We can prove in Coq that `SWAP 2 0 1`, which swaps the first and second qubits in a two-qubit register, behaves as expected on two unentangled qubits:

```
Lemma swap2: ∀ (ϕ ψ : Vector 2), WF_Matrix ϕ → WF_Matrix ψ →
  ⟦SWAP 2 0 1⟧₂ × (ϕ ⊗ ψ) = ψ ⊗ ϕ.
```

`WF_Matrix` says that ϕ and ψ are well-formed vectors [29, Section 2]. This proof can be completed by simple matrix multiplication. In the full development we prove the correctness of `SWAP d a b` for arbitrary dimension `d` and qubits `a` and `b`.

3.3 Full SQIR: Adding Measurement

The full SQIR language adds a branching measurement construct inspired by Selinger's QPL [30]. This construct permits measuring a qubit, taking one of two branches based on the measurement outcome. Full SQIR defines "commands" `com` as either a unitary sub-program, a no-op `skip`, branching measurement, or a sequence of these.

```
Inductive com (U: ℕ → Set) (d : ℕ) : Set :=
| uc   : ucom U d → com U d
| skip : com U d
| meas : ℕ → com U d → com U d → com U d
| seq  : com U d → com U d → com U d.
```

The command `meas q` P_1 P_2 measures qubit `q` and performs P_1 if the outcome is 1 and P_2 if it is 0. We define non-branching measurement and resetting to a zero state in terms of branching measurement:

```
Definition measure q := meas q skip skip.
Definition reset q := meas q (X q) skip.
```

As before, we use our `base` set of unitary gates for full SQIR.

Example: Flipping a Coin

It is simple to generate a random coin flip with a quantum computer: Use the Hadamard gate to put a qubit into equal superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and then measure it.

```
Definition coin : com base 1 := H 0; measure 0.
```

Density Matrix Semantics

As discussed in Section 2.2, measurement induces a probabilistic transition, so the semantics of a program with measurement is a probability distribution over states, called a mixed state. As is standard [25, 34], we represent such a state using a *density matrix*. The density matrix of a pure state $|\psi\rangle$ is $|\psi\rangle\langle\psi|$ where $\langle\psi| = |\psi\rangle^\dagger$ is the conjugate transpose of $|\psi\rangle$. The density matrix of a mixed state is a sum over its constituent pure states. For example, the density matrix corresponding to the uniform distribution over $|0\rangle$ and $|1\rangle$ is $\frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|$.

The semantics $\{\!|\texttt{P}|\!\}_d$ of a full SQIR program `P` is a function from density matrices to density matrices. Naturally, $\{\!|\texttt{skip}|\!\}_d\ \rho = \rho$ and $\{\!|\texttt{P1 ; P2}|\!\}_d = \{\!|\texttt{P2}|\!\}_d \circ \{\!|\texttt{P1}|\!\}_d$. For unitary subroutines, we have $\{\!|\texttt{uc U}|\!\}_d\ \rho\ = [\![\texttt{U}]\!]_d \rho [\![\texttt{U}]\!]_d^\dagger$: Applying a unitary matrix to a state vector is equivalent to applying it to both sides of its density matrix. Finally, using $|i\rangle_q\langle j|$ for $I_{2^q} \otimes |i\rangle\langle j| \otimes I_{2^{d-q-1}}$, the semantics for $\{\!|\texttt{meas q P1 P2}|\!\}_d\ \rho$ is

$$\{\!|P_1|\!\}_d(|1\rangle_q\langle 1|\,\rho\,|1\rangle_q\langle 1|) + \{\!|P_2|\!\}_d(|0\rangle_q\langle 0|\,\rho\,|0\rangle_q\langle 0|)$$

which corresponds to probabilistically applying `P1` to ρ with the specified qubit projected to $|1\rangle\langle 1|$ or applying `P2` to a similarly altered ρ.

Example: A Provably Random Coin

We can now prove that our `coin` circuit above produces the $|1\rangle\langle 1|$ or $|0\rangle\langle 0|$ density matrix (corresponding to the $|1\rangle$ or $|0\rangle$ pure state), each with probability $\frac{1}{2}$.

```
Lemma coin_dist : {|coin|}_1 |0⟩⟨0| = 1/2 |0⟩⟨0| + 1/2 |0⟩⟨0|.
```

The proof proceeds by simple matrix arithmetic. $\{\!|\texttt{H}|\!\}\ |0\rangle\langle 0|$ is $H\,|0\rangle\langle 0|\,H^\dagger = \frac{1}{2}\left(\begin{smallmatrix}1 & 1\\ 1 & 1\end{smallmatrix}\right)$. Calling this ρ_{12}, applying `measure` yields $|1\rangle\langle 1|\,\rho_{12}\,|1\rangle\langle 1| + |0\rangle\langle 0|\,\rho_{12}\,|0\rangle\langle 0|$, which can be further simplified using the fact $\langle 1|\,\rho_{12}\,|1\rangle = \langle 0|\,\rho_{12}\,|0\rangle = \left(\begin{smallmatrix}\frac{1}{2}\end{smallmatrix}\right)$, yielding $\frac{1}{2}\,|1\rangle\langle 1| + \frac{1}{2}\,|0\rangle\langle 0|$ as desired.

Measurement plays a key role in many quantum algorithms; we discuss further examples and an alternative semantics in Appendix A of the extended version of this paper.

4 SQIR's Design

This section describes key elements in the design of SQIR and its infrastructure for verifying quantum programs. To place those decisions in context, we first introduce several related verification frameworks and contrast SQIR's design with theirs. In summary, SQIR benefits from the use of *concrete indices into a global register* (a common feature in the tools we looked at), support for *reasoning about unitary programs in isolation* (supported by one other tool), and the *flexibility to allow different semantics and approaches to proof* (best supported in SQIR).

4.1 Related Approaches

Several prior works have had the goal of formally verifying quantum programs. In 2010, Green [14] developed an Agda implementation of the Quantum IO Monad, and in 2015 Boender et al. [5] produced a small Coq quantum library for reasoning about quantum "programs" directly via their matrix semantics (e.g. see Figure 1(g)). These were both proofs of concept, and were only capable of verifying basic protocols. More recently, Bordg et al. [6] took a step further in verifying quantum programs expressed as matrix products (Figure 1(g)), providing a library for reasoning about quantum computation in Isabelle/HOL and verifying more interesting protocols like the n-qubit Deutsch-Jozsa algorithm (shown in SQIR in Appendix B of the extended version of this paper).

In this section, we compare SQIR's design against three other tools for verified quantum programming that have been used to verify interesting, parameterized quantum programs: QWIRE [27] (implemented in Coq [8]); quantum Hoare logic [19] (in Isabelle/HOL [23]); and QBRICKS [7] (in Why3 [11]). We do not include Bordg et al. [6], despite its recency, because it operates one level below the surface programming language, so many issues considered here do not apply. Bordg et al.'s library is similar to the quantum libraries developed for

QWIRE and the quantum Hoare logic. All matrix formalisms provided by Bordg et al. are also available in QWIRE's library, which we re-use and extend (by $\sim$ 3000 lines [17, Section 2.2]) in SQIR.

QWIRE

The QWIRE language [25, 27] originated as an embedded circuit description language in the style of Quipper [13] but with a more powerful type system. Figure 1(d) shows the QWIRE equivalent of the SQIR program in Figure 1(b). QWIRE uses variables from the host language Coq to reference qubits, an instantiation of higher-order abstract syntax [26]. In Figure 1, the QWIRE program uses variables `x`, `y`, and `z`, while the SQIR program uses indices 0, 1, and 2 to refer to the first, second, and third qubits in the global register. QWIRE does not distinguish between unitary and non-unitary programs, and thus uses density matrices for its semantics. QWIRE has been used to verify simple randomness generation circuits and a few textbook examples [28].

QBRICKS

QBRICKS [7] is a quantum proof framework implemented in Why3 [11], developed concurrently with SQIR. QBRICKS provides a domain-specific language (DSL) for constructing quantum circuits using combinators for parallel and sequential composition (among others). Figure 1(e) presents the GHZ example written in QBRICKS' DSL. The semantics of QBRICKS are based on the *path-sums* formalism by Amy [1, 2], which can express the semantics of unitary programs in a form amenable to proof automation. QBRICKS extends path-sums to support parameterized circuits. QBRICKS has been used to verify a variety of quantum algorithms, including Grover's algorithm and Quantum Phase Estimation (QPE).

Quantum Hoare Logic

Quantum Hoare logic (QHL) [34] has been formalized in the Isabelle/HOL proof assistant [18]. QHL is built on top of the quantum while language (QWhile), which is the quantum analog of the classical while language, allowing looping and branching on measurement results. Figure 1(f) presents the GHZ example written in QHL. QWhile does not use a fixed gate set; gates are instead described directly by their unitary matrices. As such, the program in Figure 1(f) could instead be written as the application of a single gate that prepares the 3-qubit GHZ state. Given that measurement is a core part of the language, QWhile's semantics are given in terms of (partial) density matrices. A density matrix is *partial* when it may represent a sub-distribution – that is, a subset of the outcomes of measurement.

QHL has been used to verify Grover's algorithm [18]. An earlier effort by Liu et al. [20] to formalize QHL claimed to prove correctness of QPE, too. However, the approach used a combination of Isabelle/HOL and Python, calling out to Numpy to solve matrix (in)equalities; as such, we consider this only a partial verification effort. We cannot find a proof of QPE in the associated Github repository[3] and believe that this approach was abandoned in favor of Liu et al. [18].

[3] `https://github.com/ijcar2016/propitious-barnacle`

4.2 Concrete Indices into a Global Register

The first key element of SQIR's design is its use of concrete indices into a fixed-sized global register to refer to qubits. For example, in our `SWAP` program (end of Section 3.1), `a` and `b` are natural numbers indexing into a global register of size `d`. Expressing the semantics of a program that uses concrete indices is simple because concrete indices map directly to the appropriate rows and columns in the denoted matrix. Moreover, it is easy to check relationships between operations – `X a` and `X b` act on the same qubit if and only if `a` = `b`. Keeping the register size fixed means that the denoted matrix's size is known, too.

On the other hand, concrete indices hamper programmability. The `ghz` example in Figure 1(c) only produces circuits that occupy global qubits 0...`n`; we could imagine further generalizing it to add a lower bound `m` (so the circuit uses qubits `m` ... `n`), but it is not clear how it could be generalized to use non-contiguous wires. A natural solution, employed by QWIRE, is to use host-level variables to refer to *abstract* qubits that can be freely introduced and discarded, simplifying circuit construction and sub-program composition. Unfortunately, abstract qubits significantly complicate formal verification. To translate circuits to operations on density matrices, variables must be mapped to concrete matrix indices. Each time a qubit is discarded, indices undergo a de Bruijn-style shifting.

Similar to SQIR's use of concrete indices, QBRICKS-DSL's compositional structure makes it easy to map programs to their denotation: The "index" of a gate application can be computed by its nested position in the program. However, this syntax is even less convenient than SQIR's for programming: Although QBRICKS provides a utility function for defining `CNOT` gates between non-adjacent qubits, their underlying syntax does not support this, meaning that expressions like `CNOT 7 2` are translated into large sequences of `CNOT` gates. QHL is presented as having variables (e.g. `q1` in Figure 1(f)), but these variables are fixed before a program is executed and persist throughout the program. In the Isabelle formalization, they are represented by natural numbers, making them comparable to SQIR concrete indices.

4.3 Extensible Language around a Unitary Core

Another key aspect of SQIR's design is its decomposition into a unitary sub-language and the non-unitary full language. While the full language (with measurement) is more powerful, its density matrix-based semantics adds unneeded complication to the proof of unitary programs. For example, given the program $U_1; U_2; U_3$, its unitary semantics is a matrix $U_3 \times U_2 \times U_1$ while its density matrix semantics is a function $\rho \mapsto U_3 \times U_2 \times U_1 \times \rho \times U_1^\dagger \times U_2^\dagger \times U_3^\dagger$. The latter is a larger term, with a type that is harder to work with. This added complexity, borne by QWIRE and QHL, lacks a compelling justification given that many algorithms can be viewed as unitary programs with measurement occurring implicitly at their conclusion (see Section 4.7).

On the other hand, QBRICKS' semantics is based on (higher-order) path-sums, which cannot describe mixed states, and thus cannot give a semantics to measurement. SQIR's design allows for a "best of both worlds," utilizing a unitary semantics when possible, but supporting non-unitary semantics when needed. Furthermore, as we show in Section 4.6, abstractions like path-sums can be easily defined on top of SQIR's unitary semantics.

4.4 Semantics of Ill-typed Programs

We say that a SQIR program is well-typed if every gate is applied to indices within range of the global register and indices used in each multi-qubit gate are distinct. This second condition enforces quantum mechanics' *no-cloning theorem*, which disallows copying an arbitrary quantum state, as would be required to evaluate an expression like `CNOT q q`. For example, `SWAP d a b` is well-typed if $a < d$, $b < d$, and $a \neq b$.

QWIRE addresses this issue through its linear type system, which also guarantees that qubits are never reused. However, well-typedness is a (non-trivial) extrinsic proposition in QWIRE, meaning that many proofs require an assumption that the input program is well-typed and must manipulate this typing judgment within the proof. QBRICKS avoids the issue of well-typedness through its language design: It is not possible to construct an ill-typed circuit using sequential and parallel composition. The Isabelle implementation of QHL uses a well-typedness predicate to enforce some program restrictions (e.g. the gate in a unitary application is indeed a unitary matrix), but the issue of gate argument validity is enforced by Isabelle's type system: Gate arguments are represented as a set (disallowing duplicates) where all elements are valid variables.

In SQIR, ill-typed programs are denoted by the zero matrix. This often means that we do not need to explicitly assume or prove that a program is well-typed in order to state a property about its semantics, thereby removing clutter from theorems and proofs. For example, we can prove symmetry of `SWAP`, i.e. `SWAP d a b` ≡ `SWAP d b a`, without any well-typedness constraint because either both sides of the equation are well-typed or both are ill-typed. However, we cannot always avoid well-typedness preconditions. Say we want to prove transitivity of `SWAP`, i.e. `SWAP d a c` ≡ `SWAP d a b` ; `SWAP d b c`. In this case the left-hand side may be well-typed while the right-hand side is ill-typed. To verify this equivalence, we (minimally) need the precondition `b` < `d` ∧ `b` ≠ `a` ∧ `b` ≠ `c`. We capture these in our `uc_well_typed` predicate, which resembles the `WF_Matrix` predicate (used in the `SWAP` example in Section 3.2) that guarantees that a matrix's non-zero entries are all within its bounds [17, Section 3.3]. Both conditions are easily checked via automation.

4.5 Automation for Matrix Expressions

The SQIR development provides a variety of automation techniques for dealing with matrix expressions. Most of this automation is focused on simplifying matrix terms to be easier to work with. The best example of this is our `gridify` tactic [17, Section 4.5], which rewrites terms into *grid normal form* where matrix addition is on the outside, followed by tensor product, with matrix multiplication on the inside, i.e., $((.. \times ..) \otimes (.. \times ..)) + ((.. \times ..) \otimes (.. \times ..))$. Most of the circuit equivalences available in SQIR (e.g. ∀ `a`, `b`, `c`. `CNOT a c` ; `CNOT b c` ≡ `CNOT b c` ; `CNOT a c`) are proved using `gridify`. This style of automation is available in other verification tools too; `gridify` is similar to Liu et al.'s Isabelle tactic for matrix normalization [18, Section 5.1]. QBRICKS avoids the issue by using path-sums; they provide a matrix semantics for comparison's sake, but do not discuss automation for it.

Some of our automation is aimed at alleviating difficulties caused by our use of *phantom types* [29] to store the dimensions of a matrix, the rationale of which is explained in our prior work [17, Section 3.3]. In our development, matrices have the type `Matrix m n`, where `m` is the number of rows and `n` is the number of columns. One challenge with this definition is that the dimensions stored in the type may be "out of sync" with the structure of the expression itself. For example, due to simplification, rewriting, or declaration, the expression $|0\rangle \otimes |0\rangle$ may be annotated with the type `Vector` 4, although rewrite rules expect it to be of the form `Vector` $(2 * 2)$. We provide a tactic `restore_dims` that analyzes the structure of a term and rewrites its type to the desired form, allowing for more effective automated simplification.

4.6 Vector State Abstractions

To verify that the `SWAP` program has the intended semantics, we can unfold its definition (`CNOT a b`; `CNOT b a`; `CNOT a b`) and compute the associated matrix expression. However, while this proof is made simpler by automation like `gridify`, it is still fairly complicated considering

that `SWAP` has a simple classical (non-quantum) purpose. In fact, this operation is much more naturally analyzed using its action on basis states. A *(computational) basis state* is any state of the form $|i_1 \dots i_d\rangle$ for $i_1, \dots, i_d \in \{0, 1\}$ (so $|00\rangle$ and $|11\rangle$ are basis states, while $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is not). The set of all d-qubit basis states form a basis for the underlying d-dimensional vector space, meaning that any $2^d \times 2^d$ unitary operation can be uniquely described by its action on those basis states.

Using basis states, the reasoning for our `SWAP` example proceeds as follows, where we use $|\dots x \dots y \dots\rangle$ as informal notation to describe the state where the qubit at index a is in state x and the qubit at index b is in state y.

1. Begin with the state $|\dots x \dots y \dots\rangle$.
2. $CNOT\ a\ b$ produces $|\dots x \dots (x \oplus y) \dots\rangle$.
3. $CNOT\ b\ a$ produces $|\dots (x \oplus (x \oplus y)) \dots (x \oplus y) \dots\rangle = |\dots y \dots (x \oplus y) \dots\rangle$.
4. $CNOT\ a\ b$ produces $|\dots y \dots (y \oplus (x \oplus y)) \dots\rangle = |\dots y \dots x \dots\rangle$.

In our development, we describe basis states using `f_to_vec d f` where `d` : $\mathbb{N}$ and `f` : $\mathbb{N} \to \mathbb{B}$. This describes a d-qubit quantum state where qubit i is in the basis state $f(i)$, and `false` corresponds to 0 and `true` to 1. We also sometimes describe basis states using `basis_vector d i` where $i < 2^d$ is the index of the only 1 in the vector. We provide methods to translate between the two representations (simply converting between binary and decimal encodings). For the remainder of the paper, we will write |f⟩ for `f_to_vec n f` and |i⟩ for `basis_vector n i`, omitting the `n` parameter when it is clear from the context.

We prove a variety of facts about the actions of gates on basis states. For example, the following succinctly describe the behavior of the $CNOT$ and $Rz(\theta)$ gates, where $Rz(\theta) = U_R(0, 0, \theta)$:

```
Lemma f_to_vec_CNOT : ∀ (d i j : ℕ) (f : ℕ → 𝔹),
  i < d → j < d → i ≠ j →
  let f' := update f j (f j ⊕ f i) in
  ⟦CNOT i j⟧_d × |f⟩ = |f'⟩.

Lemma f_to_vec_Rz: ∀ (d j : ℕ) (θ : R) (f : ℕ → 𝔹),
  j < d →
  ⟦Rz θ j⟧_d × |f⟩ = e^(iθ(f j)) * |f⟩.
```

Above, `update f i v` updates the value of `f` at index `i` to be `v` (i.e. for the resulting function f', $f'(i) = v$ and $f'(j) = f(j)$ for all $j \neq i$). So $CNOT\ i\ j$ has the effect of updating the j^{th} entry of the input state to the exclusive-or of its i^{th} and j^{th} entries. $Rz\ \theta\ j$ updates the *phase* associated with the input state.

There are several advantages to applying these rewrite rules instead of unfolding the definitions of $[\![\texttt{CNOT i j}]\!]_d$ and $[\![\texttt{Rz } \theta \texttt{ j}]\!]_d$. For example, these rewrite rules assume well-typedness and do not depend on the ordering of qubit arguments, avoiding the case analysis needed in `gridify` [17, Section 4.5]. In addition, the rule for $CNOT$ above is simpler to work with than the general unitary semantics ($CNOT \mapsto \_ \otimes \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right) \otimes \_ \otimes I_2 \otimes \_ + \_ \otimes \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right) \otimes \_ \otimes \sigma_x \otimes \_$).

As a concrete example of where vector-based reasoning was critical, consider the three-qubit Toffoli gate, which implements a *controlled-controlled-not*, and can be thought of as the quantum equivalent of an *and* gate. It is frequently used in algorithms, but (like all n-qubit gates with $n > 2$) rarely supported in hardware, meaning that it must be decomposed into more basic gates before execution. In practice, we found `gridify` too inefficient to verify the standard decomposition of the gate [22, Chapter 4], shown below.

```
Definition TOFF {d} a b c : ucom base d :=
  H c ; CNOT b c ; T† c ; CNOT a c ; T c ; CNOT b c ; T† c ;
  CNOT a c ; CNOT a b ; T† b ; CNOT a b ; T a ; T b ; T c ; H c.
```

However, like SWAP, the semantics of the Toffoli gate is naturally expressed through its action on basis states:

```
Lemma f_to_vec_TOFF : ∀ (d a b c : ℕ) (f : ℕ → 𝔹),
  a < d → b < d → c < d →
  a ≠ b → a ≠ c → b ≠ c →
  let f' := update f c (f c ⊕ (f a && f b)) in
  ⟦TOFF a b c⟧_d × |f⟩ = |f'⟩.
```

The proof of f_to_vec_TOFF is almost entirely automated using a tactic that rewrites using the f_to_vec lemmas shown above, since T and T† are Rz (PI / 4) and Rz (−PI / 4), respectively.

The f_to_vec abstraction is simple and easy to use, but not universally applicable: Not all quantum algorithms produce basis states, or even sums over a small number of basis states, and reasoning about 2^d terms of the form $|i_1 \dots i_d\rangle$ is no easier than reasoning directly about matrices. To support more general types of quantum states we define indexed sums and tensor (Kronecker) products of vectors.

```
Fixpoint vsum {d} n (f: ℕ → Vector d) : Vector d := ...
Fixpoint vkron n (f: ℕ → Vector 2) : Vector 2^n := ...
```

As an example of a state that uses these constructs, the action of n parallel Hadamard gates on the state $|f\rangle$ can be written as

vkron n (fun i ⇒ $\frac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(i)}|1\rangle)$) or $\frac{1}{\sqrt{2^n}} *$ (vsum 2^n (fun i ⇒ $(-1)^{\texttt{to\_int}(f)\bullet i} * |i\rangle$)),

both commonly-used facts in quantum algorithms. For the remainder of the paper, we will write $\sum_{i=0}^{n-1} f(i)$ for vsum n (fun i ⇒ f i) and $\bigotimes_{i=0}^{n-1} f(i)$ for vkron n (fun i ⇒ f i).

Relation with Path-sums

Our vsum and vkron definitions share similarities with the *path-sums* [1, 2] semantics used by QBRICKS [7]. In the path-sums formalism, every unitary transformation is represented as a function of the form

$$|x\rangle \to \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} e^{2\pi i P(x,y)/2^m} |f(x,y)\rangle$$

where $m \in \mathbb{N}$, P is an arithmetic function over x and y, and f is of the form $|f_1(x,y)\rangle \otimes \cdots \otimes |f_m(x,y)\rangle$ where each f_i is a Boolean function over x and y. For instance, the Hadamard gate H has the form $|x\rangle \to \frac{1}{\sqrt{2}} \sum_{y=0}^{1} e^{2\pi i x y/2} |y\rangle$. Path-sums provide a compact way to describe the behavior of unitary matrices and are closed under matrix and tensor products, making them well-suited for automation. They can be naturally described in terms of our vkron and vsum vector-state abstractions:

```
Definition path_sum (m : ℕ) P f x :=
  vsum 2^m (fun y ⇒ e^(2πiP(x,y)/2^m) * (vkron m (fun i ⇒ f i x y))).
```

As above, P is an arithmetic function over x and y and f i is a Boolean function over x and y for any i.

4.7 Measurement Predicates

The proofs in Section 5 do not use the non-unitary semantics directly, but instead describe the probability of different measurement outcomes using predicates probability_of_outcome and prob_partial_meas.

```
(* Probability of measuring φ given input ψ. *)
Definition probability_of_outcome {n} (φ ψ : Vector n) : R :=
  let c := (φ† × ψ) 0 0 in |c|².

(* Probability of measuring φ on the first n qubits given (n+m) qubit input ψ. *)
Definition prob_partial_meas {n m} (φ : Vector 2^n) (ψ : Vector 2^(n+m)) :=
  ‖ (φ† ⊗ I_(2^m)) × ψ ‖².
```

Above, $\|v\|$ is the 2-norm of vector v and $|c|$ is the complex norm of c. In formal terms, the "probability of measuring φ" is the probability of outcome φ when measuring a state in the basis $\{\varphi \times \varphi^\dagger, \mathtt{I}_{2^n} - \varphi \times \varphi^\dagger\}$.

The *principle of deferred measurement* [22, Chapter 4] says that measurement can always be deferred until the end of a quantum computation without changing the result. However, we included measurement in Section 3.3 because it is an important feature of quantum programming languages that is used in a variety of constructs like repeat-until-success loops [24] and error-correcting codes [12]. QBRICKS also uses measurement predicates, but unlike SQIR does not support a general measurement construct.

5 Proofs of Quantum Algorithms

In this section we discuss the formal verification of two classic quantum algorithms: Grover's algorithm [22, Chapter 6] and quantum phase estimation [22, Chapter 5]. We present additional, simpler examples in Appendices A and B of the extended version of this paper. All proofs and specifications follow the corresponding textbook arguments.

5.1 Grover's Algorithm

Overview

Given a circuit implementing Boolean oracle $f : \{0,1\}^n \to \{0,1\}$, the goal of Grover's algorithm is to find an input x satisfying $f(x) = 1$. Suppose that $n \geq 2$. In the classical (worst-)case where $f(x) = 1$ has a unique solution, finding this solution requires $O(2^n)$ queries to the oracle. However, the quantum algorithm finds the solution with high probability using only $O(\sqrt{2^n})$ queries.

The algorithm alternates between applying the oracle and a "diffusion operator." Individually, these operations each perform a reflection in the two-dimensional space spanned by the input vector (a uniform superposition) and a uniform superposition over the solutions to f. Together, they perform a rotation in the same space. By choosing an appropriate number of iterations i, the algorithm will rotate the input state to be suitably close to the solution vector. The SQIR definition of Grover's algorithm is shown in Figure 2.

The SQIR version of Grover's algorithm is 15 lines, excluding utility definitions like `control` and `npar`. The specification and proof are around 770 lines. The proof took approximately one person-week.

Proof Details

The statement of correctness says that after i iterations, the probability of measuring a solution is $\sin^2((2i+1)\theta)$ where $\theta = \arcsin(\sqrt{k/2^n})$ and k is the number of satisfying solutions to f. Note that this implies that the optimal number of iterations is $\frac{\pi}{4}\sqrt{\frac{2^n}{k}}$.

```
(* Controlled-X with target (n-1) and controls 0, 1, ..., n-2. *)
Fixpoint generalized_Toffoli' n0 : ucom base n :=
  match n0 with
  | 0 | S 0 ⇒ X (n - 1)
  | S n0' ⇒ control (n - n0) (generalized_Toffoli' n0')
  end.
Definition generalized_Toffoli := generalized_Toffoli' n.

(* Diffusion operator. *)
Definition diff : ucom base n :=
  npar n H; npar n X ;
  H (n - 1) ; generalized_Toffoli ; H (n - 1) ;
  npar n X; npar n H.

(* Main program (iterates applying Uf and diff). *)
Definition body := Uf ; cast diff (S n).
Definition grover i := X n ; npar (S n) H ; niter i body.
```

Figure 2 Grover's algorithm in SQIR. `control` performs a unitary program conditioned on an input qubit, `npar` performs copies of a unitary program in parallel, `cast` is a no-op that changes the dimension in a `ucom`'s type, and `niter` iterates a unitary program.

We begin the proof by showing that the uniform superposition can be rewritten as a sum of "good" states (ψg) that satisfy f and "bad" states (ψb) that do not satisfy f.

```
Definition ψ := 1/√2^n Σ_{k=0}^{2^n-1} |k⟩.
Definition θ := asin (√(k/2^n)).
Lemma decompose_ψ : ψ = (sin θ) ψg + (cos θ) ψb.
```

We then prove that `Uf` and `diff` perform the expected reflections (e.g. $[\![\texttt{diff}]\!]_n = -2\left|\psi\right\rangle\left\langle\psi\right| + I_{2^n}$) and the following key lemma, which shows the output state after i iterations of `body`.

```
Lemma loop_body_action_on_unif_superpos : ∀ i,
  [[body]]^i_{n+1} (ψ ⊗ |-⟩) =
    (-1)^i (sin ((2 * i + 1) * θ) ψg + cos ((2 * i + 1) * θ) ψb) ⊗ |-⟩.
```

This property is straightforward to prove by induction on `i`, and implies the desired result, which specifies the probability of measuring any solution to f.

```
Lemma grover_correct : ∀ i,
  Rsum 2^n (fun z ⇒  if f z
                     then prob_partial_meas |z⟩ ([[grover i]]_{n+1} × |0⟩^{n+1})
                     else 0) =
  (sin ((2 * i + 1) * θ))^2.
```

That is, the sum over the probability of all possible outcomes z such that $f(z)$ is true is $\sin^2((2i+1)\theta)$. Above, `Rsum` is a sum over real numbers.

5.2 Quantum Phase Estimation

Overview

Given a unitary matrix U and eigenvector $\left|\psi\right\rangle$ such that $U\left|\psi\right\rangle = e^{2\pi i\theta}\left|\psi\right\rangle$, the goal of quantum phase estimation (QPE) is to find a k-bit representation of θ. In the case where θ can be exactly represented using k bits (i.e. $\theta = z/2^k$ for some $z \in \mathbb{Z}$), QPE recovers θ

exactly. Otherwise, the algorithm finds a good k-bit approximation with high probability. QPE is often used as a subroutine in quantum algorithms, most famously Shor's factoring algorithm [31].

The SQIR program for QPE is shown in Figure 3. For comparison, the standard circuit diagrams for QPE and the quantum Fourier transform (QFT), which is used as a subroutine in QPE, are shown in Figure 4. The SQIR version of QPE is around 40 lines and the specification and proof in the simple case ($\theta = z/2^k$) is around 800 lines. The fully general case ($\theta \neq z/2^k$) adds about 250 lines. The proof of the simple case was completed in about two person-weeks. When working out the proof of the general case, we found that we needed some non-trivial bounds on trigonometric functions (for $x \in \mathbb{R}$, $|\sin(x)| \leq |x|$ and if $|x| \leq \frac{1}{2}$ then $|2 * x| \leq |\sin(\pi x)|$). Laurent Théry kindly provided proofs of these facts using the Coq Interval package [21].

Proof Details

The correctness property for QPE in the case where θ can be described exactly using k bits ($\theta = z/2^k$) says that the QPE program will exactly recover z. It can be stated in SQIR's development as follows.

```
Lemma QPE_correct_simplified: ∀ k n (u : ucom base n) z (ψ : Vector 2^n),
  n > 0 → k > 1 → uc_well_typed u → WF_Matrix ψ →
  let θ := z / 2^k in
  ⟦u⟧_n × ψ = e^{2πiθ} * ψ →
  ⟦QPE k n u⟧_{k+n} × (|0⟩^k ⊗ ψ) = |z⟩ ⊗ ψ.
```

The first four conditions ensure well-formedness of the inputs. The fifth condition enforces that input ψ is an eigenvector of c. The conclusion says that running the `QPE` program computes the value z, as desired.

In the general case where θ cannot be exactly described using k bits, we instead prove that `QPE` recovers the best k-bit approximation with high probability (in particular, with probability $\geq 4/\pi^2$).

```
Lemma QPE_semantics_full : ∀ k n (u : ucom base n) z (ψ : Vector 2^n) (δ : R),
  n > 0 → k > 1 → uc_well_typed u → Pure_State_Vector ψ →
  -1 / 2^{k+1} ≤ δ < 1 / 2^{k+1} → δ ≠ 0 →
  let θ := z / 2^k + δ in
  ⟦u⟧_n × ψ = e^{2πiθ} * ψ →
  prob_partial_meas |z⟩ (⟦QPE k n u⟧_{k+n} × (|0⟩^k ⊗ ψ)) ≥ 4 / π^2.
```

`Pure_State_Vector` is a restricted form of `WF_Matrix` that requires a vector to have norm 1.

As an example of the reasoning that goes into proving these properties, consider the QFT subroutine of QPE. The correctness property for `controlled_rotations` says that evaluating the program on input $|x\rangle$ will produce the state $e^{2\pi i(x_0 \cdot x_1x_2...x_{n-1})/2^n}|x\rangle$ where x_0 is the highest-order bit of x represented as a binary string and $x_1x_2...x_{n-1}$ are the lower-order $n-1$ bits.

```
Lemma controlled_rotations_correct : ∀ n x,
  n > 1 → ⟦controlled_rotations n⟧_n × |x⟩ = e^{2πi(x_0 · x_1x_2...x_{n-1})/2^n}|x⟩.
```

We can prove this property via induction on n. In the base case ($n = 2$) we have that x is a 2-bit string x_0x_1. In this case, the output of the program is $e^{2\pi i(x_0 \cdot x_1)/2^2}|x_0x_1\rangle$, as desired. In the inductive step, we assume that:

$$\llbracket \texttt{controlled\_rotations n} \rrbracket_n \times |x_1x_2...x_{n-1}\rangle = e^{2\pi i(x_0 \cdot x_1x_2...x_{n-1})/2^n}|x_1x_2...x_{n-1}\rangle.$$

```
(* Controlled rotation cascade on n qubits. *)
Fixpoint controlled_rotations n : ucom base n :=
  match n with
  | 0 | 1 ⇒ SKIP
  | S n'  ⇒ controlled_rotations n' ; control n' (Rz (2π / 2^n) 0)
  end.

(* Quantum Fourier transform on n qubits. *)
Fixpoint QFT n : ucom base n :=
  match n with
  | 0    ⇒ SKIP
  | S n' ⇒ H 0 ; controlled_rotations n ; map_qubits (fun q ⇒ q + 1) (QFT n')
  end.

(* The output of QFT needs to be reversed before further processing. *)
Definition reverse_qubits n := ...
Definition QFT_w_reverse n := QFT n ; reverse_qubits n.

(* Controlled powers of u. *)
Fixpoint controlled_powers' {n} (u : ucom base n) k kmax : ucom base (kmax+n) :=
  match k with
  | 0    ⇒ SKIP
  | S k' ⇒ controlled_powers' u k' kmax ; niter 2^k' (control (kmax - k' - 1) u)
  end.
Definition controlled_powers {n} (u : ucom base n) k := controlled_powers' u k k.

(* QPE circuit for program u.
   k = number of bits in resulting estimate
   n = number of qubits in input state *)
Definition QPE k n (u : ucom base n) : ucom base (k + n) :=
  npar k H ;
  controlled_powers (map_qubits (fun q ⇒ k + q) u) k;
  invert (QFT_w_reverse k).
```

Figure 3 SQIR definition of QPE. Some type annotations and calls to `cast` have been removed for clarity. `control`, `map_qubits`, `niter`, `npar`, and `invert` are Coq functions that transform SQIR programs; we have proved that they have the expected behavior (e.g. $\forall\ u.\ [\![\text{invert } u]\!]_n = [\![u]\!]_n^\dagger$).

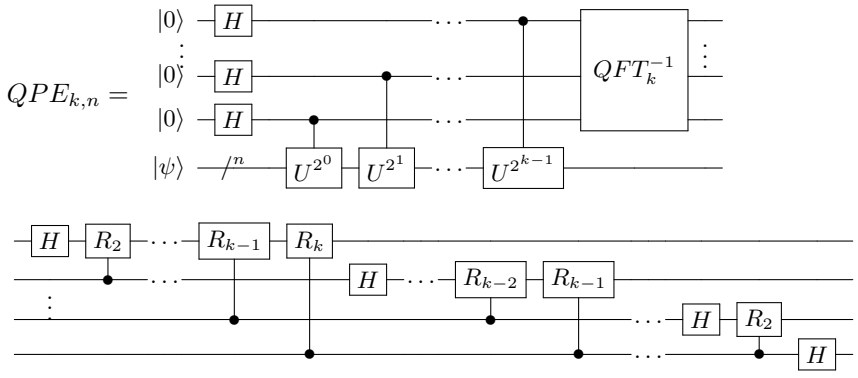


Figure 4 Circuit for quantum phase estimation (QPE) with k bits of precision and an n-qubit input state (top) and quantum Fourier transform (QFT) on k qubits (bottom). $|\psi\rangle$ and U are inputs to QPE. R_m is a z-axis rotation by $2\pi/2^m$.

$$
\begin{aligned}
&[\![\texttt{controlled\_rotations (n+1)}]\!]_{n+1} \times |x\rangle \\
&\quad = [\![\texttt{control } x_n \ (\texttt{Rz } (2\pi/2^{n+1})\ 0]\!]_{n+1} \times [\![\texttt{controlled\_rotations n}]\!]_{n+1} \times |x\rangle \\
&\quad = [\![\texttt{control } x_n \ (\texttt{Rz } (2\pi/2^{n+1})\ 0]\!]_{n+1} \times e^{2\pi i(x_0 \,\cdot\, x_1x_2\ldots x_{n-1})/2^n} \, |x_1x_2\ldots x_{n-1}x_n\rangle \\
&\quad = e^{2\pi i(x_0 \,\cdot\, x_n)/2^{n+1}} e^{2\pi i(x_0 \,\cdot\, x_1x_2\ldots x_{n-1})/2^n} \, |x_1x_2\ldots x_{n-1}x_n\rangle \\
&\quad = e^{2\pi i(x_0 \,\cdot\, x_1x_2\ldots x_n)/2^{n+1}} \, |x_1x_2\ldots x_{n-1}x_n\rangle
\end{aligned}
$$

Figure 5 Reasoning used in the proof of `controlled_rotations`. The first step unfolds the definition of `controlled_rotations`; the second step applies the inductive hypothesis; the third step evaluates the semantics of `control`; and the fourth step combines the exponential terms.

We then perform the simplifications shown in Figure 5, which complete the proof.

Our correctness property for `QFT n` (shown below) can similarly be proved by induction on n, and relies on the lemma `controlled_rotations_correct`.

```
Lemma QFT_semantics : ∀ n x, n>0 → [[QFT n]]_n × |x⟩ = 1/√2^n ⨂_{j=0}^{n-1} (|0⟩ + e^{2πix/2^{n-j}} |1⟩).
```

6 Open Problems and Future Work

We previously presented SQIR as the intermediate representation in a verified circuit optimizer [17]. In this paper, we presented SQIR as a source language for quantum programming and discussed how our design choices (e.g. concrete indices, unitary core, vector state abstractions) ease proofs about SQIR programs. But there is still work to be done.

So far, work on formally verified quantum computation has been limited to textbook quantum algorithms like QPE and Grover's. Although these algorithms are a useful stress-test for tools, they do not accurately reflect the types of quantum programs that are expected to run on near-term machines. Near-term algorithms are usually *approximate*. They do not implement the desired operation exactly, but rather perform an operation "close" to what was intended. Our `probability_of_outcome` and `prob_partial_meas` predicates can be used to express distance between vector states, but we currently do not have support for reasoning about distance between general quantum operations.

Another issue is that near-term algorithms often need to account for hardware errors. Thus, verifying these algorithms may require considering their behavior in the presence of errors. So far, most of our work in SQIR has revolved around the unitary semantics and vector-based state abstractions because we find these simpler to work with. However, it is more natural to describe states subject to error using density matrices, since noisy states are mixtures of pure states [22, Chapter 8].

On another front, there is important work to be done on describing quantum algorithms and correctness properties at a higher level of abstraction. The proofs and definitions in this paper follow the standard textbook presentation, but are still lower-level than similar proofs about classical programs. Rather than working from the circuit model, used in QWIRE, SQIR, QBRICKS, and (to some extent) QWhile, it would be interesting to verify programs written in higher-level languages like Silq [4] or Q# [33].

We hope that SQIR's extensible design and flexible semantics, developed while verifying circuit optimizations and textbook quantum programs, will serve as a solid foundation for the proposed verification efforts above and those to come.

References

1 Matt Amy. *Formal Methods in Quantum Circuit Design*. PhD thesis, University of Waterloo, 2019.

2 Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018*, June 2018.

3 Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, November 1995. doi:10.1103/PhysRevA.52.3457.

4 Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 2020.

5 Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In Chris Heunen, Peter Selinger, and Jamie Vicary, editors, *Proceedings of the 12th International Workshop on Quantum Physics and Logic, Oxford, U.K., July 15-17, 2015*, volume 195 of *Electronic Proceedings in Theoretical Computer Science*, pages 71–83. Open Publishing Association, 2015. doi:10.4204/EPTCS.195.6.

6 Anthony Bordg, Hanna Lachnitt, and Yijun He. Certified quantum computation in Isabelle/HOL. *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-020-09584-7.

7 Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. Toward certified quantum programming. *arXiv e-prints*, 2020. arXiv:2003.05841.

8 The Coq Development Team. The coq proof assistant, version 8.10.0, 2019. doi:10.5281/zenodo.3476303.

9 Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv e-prints*, July 2017. arXiv:1707.03429.

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

11 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, Lecture Notes in Computer Science, 2013.

12 Daniel Gottesman. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, volume 68, pages 13–58, 2010.

13 Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 333–342, 2013.

14 Alexander S Green. *Towards a formally verified functional quantum programming language*. PhD thesis, University of Nottingham, 2010.

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

16 Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, pages 212–219, 1996.

17 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(37), 2021.

18 Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum hoare logic. In *Computer Aided Verification - 31st International Conference, CAV 2019, New*

York City, NY, USA, July 15-18, 2019, Proceedings, Part II, pages 187–207, 2019. doi:10.1007/978-3-030-25543-5_12.

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

20 Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications. *arXiv preprint arXiv:1601.03835*, 2016.

21 Guillaume Melquiond. Interval package for coq, 2020. URL: https://gitlab.inria.fr/coqinterval/interval.

22 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

23 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

24 Adam Paetznick and Krysta M Svore. Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. *Quantum Information & Computation*, 14(15-16):1277–1301, 2014.

25 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009894.

26 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM. doi:10.1145/53990.54010.

27 Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania, 2018.

28 Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. doi:10.4204/EPTCS.266.8.

29 Robert Rand, Jennifer Paykin, and Steve Zdancewic. Phantom types for quantum programs. The Fourth International Workshop on Coq for Programming Languages, January 2018.

30 Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, August 2004.

31 P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, FOCS '94, 1994.

32 DR Simon. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 116–123, 1994.

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

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