Proving Quantum Programs Correct

Kesha Hietala[★] Robert Rand[†] Shih-Han Hung[★] Liyi Li[‡] Michael Hicks[★]
[★]University of Maryland [†]University of Chicago [‡]University of Illinois

Abstract

As quantum computing steadily progresses from theory to practice, programmers are faced with 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 Simon's algorithm, 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.

1 Introduction

Quantum computers are fundamentally different than 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 them to *prove*, in advance, that the code implementing a quantum algorithm does what it should for all possible inputs and configurations.

In prior work [20], we developed a formally verified optimizer for quantum programs, which we implemented and proved correct in the Coq proof assistant [10]. Our optimizer transforms programs written in SQIR, a *small quantum intermediate representation.* While we designed SQIR to be a compiler IR, we soon 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 also ease proving source programs correct.

To date, we have proved the correctness of implementations of a half-dozen quantum algorithms, including quantum teleportation, Greenberger–Horne–Zeilinger (GHZ) state preparation [17], the Deutsch-Jozsa algorithm [12], Simon's algorithm [37], the quantum Fourier transform (QFT), quantum phase estimation (QPE), and Grover's algorithm [18]. QPE is the key component of Shor's prime-factoring algorithm [36], the best known and most impactful quantum algorithm, while Grover's algorithm for unstructured search is the second. All of these implementations can be extracted to code that can (in concept, though not in practice, due to resource constraints) be executed on quantum hardware.

While SQIR has been presented previously, this paper offers several new contributions. First, after reviewing SQIR (Section 2), we present a detailed discussion of how SQIR's design enables proofs of correctness, highlighting the benefit of techniques used for automation and for reasoning about a program's action on classical states (Section 3). Second, we present the code, formal specification, and proof sketch of Simon's algorithm, Grover's algorithm, QFT, and QPE (Section 4). Next, we present a detailed comparison of our approach to those of *Q*WIRE [26], *Q*BRICKS [9], and the quantum Hoare Logic (QHL) [21], which constitute the most closely related work. Of these, *Q*WIRE is the most sophisticated as a programming language, but its expressiveness hampers formal reasoning in practice. Like SQIR, both QHL and *Q*BRICKS sacrifice language expressiveness for ease of proof, but both take a more rigid approach to proof than SQIR (via the path-sum semantics [2] in *Q*BRICKS and a deductive logical system in QHL). We believe there is ripe opportunity for further application of formal methods to quantum computing; we sketch several 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. All materials are freely at https://github.com/inQWIRE/SQIR.

2 SQIR: A Small Quantum IR

SQIR is a simple circuit-oriented language deeply embedded in the Coq proof assistant. This section presents SQIR and some basics of quantum computing. We defer a detailed discussion of SQIR's design rationale to the next section.

2.1 Background: 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 $|1\rangle = \binom{0}{1}$. When α or β is 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 $|1\rangle$ since they share coefficients.

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 $|ij\rangle$, where $i, j \in \{0, 1\}$; we may also write $|k\rangle$ where

$k \in \mathbb{N}$ is the decimal interpretation of bits ij. We will 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 $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which is known as a *Bell pair*.

We will introduce further concepts in quantum computing as they arise; for a full treatment we recommend the standard text on the subject [23].

2.2 Unitary SQIR: Syntax

Quantum programs operate on quantum states, transforming an input state into an output state. Such programs are typically decomposed into a series of *quantum gates*, with the program depicted as a *circuit* and input/output qubits depicted as wires. An example is shown in Figure 1. SQIR is a programming language, deeply embedded in Coq, that can express such circuits.

A typical quantum gate's semantics is denoted by a *unitary matrix* (a matrix that preserves quantum states); applying the gate to a state is tantamount to multiplying the state vector by the gate's matrix. SQIR's *unitary fragment* is a sublanguage of full SQIR that can express circuits consisting of a composition of unitary gates. The full SQIR language also includes *measurement*, which is carried out by a special, non-unitary operator; it is used to extract information from the quantum state.

A program in the unitary fragment has type `ucom` (for "unitary command"), which we define as follows in Coq:

```
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
| uapp3 : U 3 → ℕ → ℕ → ℕ → ucom U d.
```

The `useq` constructor sequences two commands; we use notational shorthand `p1 ; p2` for `useq p1 p2`. The three `uapp`*i* constructors indicate the application of a quantum gate to i qubits (where i is 1, 2, or 3). Qubits are identified as numbered indices into a *global register* of size `d`; the global register is the quantum state being operated on. The 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`, used by IBM's OpenQASM [11]:[1]

```
Inductive base : ℕ → Set :=
  | U_R (r1 r2 r3 : 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.

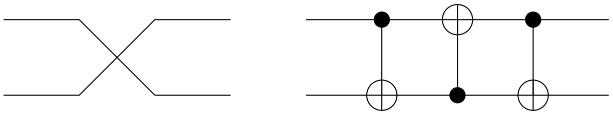

Figure 1. A traditional swap gate (left) and our equivalent `SWAP` circuit (right).

Example: SWAP. The following Coq function produces a unitary SQIR program that applies three controlled-not gates in a row, for the purpose 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.
```

In Figure 1, we show the result of calling `SWAP 2 0 1`, i.e., a circuit that swaps qubits 0 and 1 in a two-qubit register.

2.3 Unitary SQIR: Semantics

Each k-qubit quantum gate corresponds to a $2^k \times 2^k$ unitary matrix, which for our `base` gates `U_R` and `CNOT` is the following.

$$\llbracket U_R(\theta, \phi, \lambda) \rrbracket = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix}$$

$$\llbracket CNOT \rrbracket = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Conveniently, gate U_R can encode any other single-qubit gate. 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 $U_R(\pi/2, 0, \pi)$. 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 (e.g., it takes $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$); it can be encoded as $U_R(\pi, 0, \pi)$. Multi-qubit gates (e.g., the three-qubit "Toffoli" gate) are easily produced by combinations of $CNOT$ and U_R.

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

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

We write $\llbracket \texttt{c} \rrbracket_d$ for `uc_eval d c`. The denotation of `uapp1` is the denotation of its argument gate, but padded with the identity matrix I so it has size $2^d \times 2^d$. To be precise, we have:

$$\llbracket \texttt{uapp1 U q} \rrbracket_d = \begin{cases} I_{2^q} \otimes \llbracket U \rrbracket \otimes I_{2^{d-q-1}} & q < d \\ 0_{2^d} & \text{otherwise} \end{cases}$$

[1] The `U` parameter is most important when SQIR programs are subject to compiler transformations, since the compiler needs to know what gates it can include in an output program.

where I_n is the $n \times n$ identity matrix. 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 proper (well-typed) circuit corresponds to a unitary transformation.

For $[\![\texttt{CNOT q1 q2}]\!]_d$, we decompose $CNOT$ into $|0\rangle\langle 0| \otimes I_2 + |1\rangle\langle 1| \otimes X$, where $|0\rangle\langle 0| = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ and $|1\rangle\langle 1| = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$. We then pad the expression appropriately, obtaining the following when $q_1 < q_2 < d$:

$$I_{2^{q_1}} \otimes |0\rangle\langle 0| \otimes I_{2^r} \otimes I_2 \otimes I_{2^s} + I_{2^{q_1}} \otimes |1\rangle\langle 1| \otimes I_{2^r} \otimes X \otimes I_{2^s}.$$

Here $r = q_2 - q_1 - 1$ and $s = d - q_2 - 1$. When $q_2 < q_1 < d$, we obtain a symmetric expression, and when $q_1 = q_2$ or either qubit is out of bounds, we again obtain the zero matrix.

Finally, sequential composition corresponds to simple matrix multiplication:

$$[\![\texttt{U1; U2}]\!]_d = [\![\texttt{U2}]\!] \times [\![\texttt{U1}]\!]$$

Example: Proving SWAP works. we can prove in Coq that `SWAP 2 0 1` 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` here says that ϕ and ψ are well-formed vectors of length 2 (see Section 3.3).

This proof can be completed via simple matrix multiplication. In Section 3 we will show how to prove the correctness of `SWAP d p q` for arbitrary dimension d and qubits p and q.

2.4 Full sqir: Adding measurement

As mentioned earlier, to extract information from a quantum state we must measure it, and measurement is non-unitary. In particular, measuring a single qubit returns either 0 or 1 with probability corresponding to the square of the respective amplitudes of $|0\rangle$ and $|1\rangle$, and moreover modifies the state of the qubit to be either $|0\rangle$ or $|1\rangle$, to match what is returned. Hence $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ becomes $|0\rangle$ (and returns 0) with probability $1/2$, and similarly for $|1\rangle$. For a multi-qubit state we combine the terms associated with a specific outcome and renormalize the result: Measuring the first qubit of $\frac{1}{\sqrt{3}}(|001\rangle + |010\rangle + |100\rangle)$ produces the state $\frac{1}{\sqrt{2}}(|001\rangle + |010\rangle)$ with probability $2/3$ and the state $|100\rangle$ with probability $1/3$.

To support measurement, full sqir defines *commands* `com` as either a unitary command, a no-op `skip`, a simple branching measurement command inspired by QPL [35], or the sequencing 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 depending on the result either performs P_1 or P_2. We can then 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 will use our `base` set for sqir `com`s in this paper.

Example: Flipping a Coin It is easy to generate a truly random coin flip with a quantum computer: Simply put a qubit into the equal superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and then measure it. In full sqir, we can express this as follows:

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

2.5 Full sqir semantics

Since measurement induces a probabilistic transition, we need to generalize our semantics to encode probabilities. For every quantum state vector $|\psi\rangle$, we have the *density matrix* representation $|\psi\rangle\langle\psi|$ where $\langle\psi| = |\psi\rangle^\dagger$ is the *adjoint* of $|\psi\rangle$ (the transpose with the imaginary terms negated). This represents a point distribution over quantum states. To encode probabilities, we can combine density matrices: For any two density matrices ρ_1 and ρ_2 and $r \in (0, 1)$, $r\rho_1 + (1 - r)\rho_2$ is a valid density matrix, corresponding to ρ_1 with probability r and ρ_2 with probability $1 - r$. If $\rho_1 \neq \rho_2$ we call this a *mixed state*.

The semantics $\{|\texttt{P}|\}_d$ of a sqir program with measurement 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 the 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

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

which corresponds to probabilistically applying `P1` to ρ with the specified qubit replaced by $|1\rangle\langle 1|$ 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, each with probability $\frac{1}{2}$.

```
Lemma coin_dist : ⦃coin⦄ |0⟩⟨0| = 1/2|1⟩⟨1| + 1/2|0⟩⟨0|.
```

To see this, recall that command composition is just function composition. $\{|\texttt{H}|\}\ |0\rangle\langle 0|$ 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` gets us:

$$|1\rangle\langle 1|\, \rho_{12}\, |1\rangle\langle 1| + |0\rangle\langle 0|\, \rho_{12}\, |0\rangle\langle 0|$$

$\langle 1|\, \rho_{12}\, |1\rangle = \langle 0|\, \rho_{12}\, |0\rangle = (\,1/2\,)$ so we can simplify this to $\frac{1}{2}\,|1\rangle\langle 1| + \frac{1}{2}\,|0\rangle\langle 0|$ as desired.

3 Designing sqir

This section describes key elements in the design of sqir and its infrastructure for verifying quantum programs. Section 4 discusses the proofs of several interesting programs.

3.1 Concrete Indices into a Global Register

As noted in the previous section, unitary and non-unitary commands take in a *dimension* d. This represents the size of the global register of qubits. Fixing a global register allows us to use *concrete indices* to refer to qubits. For example, in our SWAP program, 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 simple 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 unchanged too.

In principle, a fixed-size register is a limiting factor: We can neither introduce new qubits nor discard qubits. A natural alternative, used by languages like Quipper [15] and *Q*WIRE [26], is to use variables refer to *abstract* qubits. Using abstract qubits eases programmability, but we find that it complicates formal proof (see Section 5.1). Moreover, a fixed register reflects the limitations of existing quantum computers: They have a set number of qubits and the programmer must take this number into consideration when writing programs.

3.2 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 linearity and thereby quantum mechanic's *no-cloning theorem*, which disallows copying an arbitrary quantum state. As an example, SWAP d a b is well-typed if $a < d$, $b < d$, and $a \neq b$.

We assign ill-typed gate applications the denotation of the zero matrix. A result of this is that the denotation of a unitary program is a unitary matrix if and only if the program is well-typed, and the denotation is the zero matrix if and only if the program is not well-typed (it is impossible to obtain a non-unitary, non-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 that 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.

3.3 Working with Phantom Types

We use the matrix library developed for *Q*WIRE [33], which defines matrices as functions from pairs of natural numbers to complex numbers.

```
Definition Matrix (m n : ℕ) := ℕ → ℕ → ℂ.
```

The arguments m and n, which are the dimensions of the matrix, are *phantom types* [34]—they do not appear on the righthand side of the definition. Phantom types are useful to define operations on matrices that depend on their dimensions, e.g. Kronecker product and matrix multiplication, and there is no proof burden internal to the matrices themselves. But unlike dependent types, there is no obligation to prove that matrices have the desired dimensions, allowing us to declare that $[\![CNOT\ q_1\ q_2]\!]_d$ is a square matrix of length 2^d rather than $2^{q_1} * 2 * 2^r * 2 * 2^s$, as its decomposition (Section 2.3) suggests. This allows us to apply $[\![SWAP\ d\ a\ b]\!]_d$ to a 2^d vector without getting a type error.

However, it is often necessary, for instance when rewriting $A \times I_n = A$, to show that a matrix is well-formed within its specified bounds by means of an external predicate:

```
Definition WF_Matrix {m n} (M : Matrix m n) : P :=
    ∀ i j, i ≥ m ∨ j ≥ n → M i j = 0.
```

We use and expand upon *Q*WIRE's hint database wf_db that includes facts about well-formedness (e.g. the product of two well-formed matrices is well-formed), to make well-formedness proofs almost entirely automated.

Another challenge with this definition of matrices 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 (per above), the expression $|0\rangle \otimes |0\rangle$ may be annotated with the type Vector 4, even though rewrite rules expect it to be of the form Vector (2 * 2). To account for this, we provide a tactic restore_dims that analyzes the structure of a term and rewrites its type to the desired form. These terms will sometimes conflict: $I_4 \times (|0\rangle \otimes |0\rangle)$ expects the term on the left to have dimension 4×4 and the term on the right to have dimension $2 * 2 \times 1$. In such cases, we change the dimensions to maximize the rewriting we can do: In this case we convert I_4 to I_{2*2} allowing us to rewrite by the left identity rule.

3.4 Simplifying Matrix Expressions

If we unfold the definition of $[\![\texttt{SWAP n a b}]\!]_d$, we obtain

$$[\![CNOT\ a\ b]\!]_d \times [\![CNOT\ b\ a]\!]_d \times [\![CNOT\ a\ b]\!]_d.$$

As discussed in Section 2.3, the decomposition of the *CNOT* gate is a sum over two terms that depends on the ordering of a and b (and, for well-typedness, their relationship to d). In the well-typed case (assuming $a < b$) the product above unfolds to

$$(I_{2^a} \otimes |0\rangle\langle 0| \otimes I_{2^s} \otimes I_2 \otimes I_{2^r} + I_{2^a} \otimes |1\rangle\langle 1| \otimes I_{2^s} \otimes X \otimes I_{2^r})$$
$$\times$$
$$(I_{2^a} \otimes I_2 \otimes I_{2^s} \otimes |0\rangle\langle 0| \otimes I_{2^r} + I_{2^a} \otimes X \otimes I_{2^s} \otimes |1\rangle\langle 1| \otimes I_{2^r})$$
$$\times$$
$$(I_{2^a} \otimes |0\rangle\langle 0| \otimes I_{2^s} \otimes I_2 \otimes I_{2^r} + I_{2^a} \otimes |1\rangle\langle 1| \otimes I_{2^s} \otimes X \otimes I_{2^r})$$

where $s = b - a - 1$ and $r = d - b - 1$.

We provide a tactic `gridify` that performs cases analysis on the relationships between arguments to gate applications, immediately solving cases where the circuit is ill-typed ($a = b$, $a \geq n$, and $b \geq d$ above) and rewriting any remaining cases ($a < b$ and $b < a$ above) into *grid normal* form. In grid normal form, each arithmetic expression has addition on the outside, followed by tensor product, with multiplication on the inside, i.e., $((.. \times ..) \otimes (.. \times ..)) + ((.. \times ..) \otimes (.. \times ..))$.

After applying `gridify` and simple automatic rewriting (e.g. $I \times A = A$ and $\langle 0| \times X = \langle 1|$) to $[\![\texttt{SWAP}\ d\ a\ b]\!]_d$, we have

$$I_{2^a} \otimes |0\rangle\langle 0| \otimes I_{2^s} \otimes |0\rangle\langle 0| \otimes I_{2^r} +$$
$$I_{2^a} \otimes |0\rangle\langle 1| \otimes I_{2^s} \otimes |1\rangle\langle 0| \otimes I_{2^r} +$$
$$I_{2^a} \otimes |1\rangle\langle 0| \otimes I_{2^s} \otimes |0\rangle\langle 1| \otimes I_{2^r} +$$
$$I_{2^a} \otimes |1\rangle\langle 1| \otimes I_{2^s} \otimes |1\rangle\langle 1| \otimes I_{2^r}.$$

We can further use `gridify` to simplify applications of this matrix to input states. For example, we can show that

$$[\![\texttt{SWAP}\ n\ a\ b]\!]_d \times |\psi_A\rangle \otimes |\psi_B\rangle \otimes |\psi_C\rangle \otimes |\psi_D\rangle \otimes |\psi_E\rangle =$$
$$|\psi_A\rangle \otimes |\psi_D\rangle \otimes |\psi_C\rangle \otimes |\psi_B\rangle \otimes |\psi_E\rangle$$

for vectors $|\psi_A\rangle$, $|\psi_B\rangle$, $|\psi_C\rangle$, $|\psi_D\rangle$, $|\psi_E\rangle$ of dimensions 2^a, 2, 2^s, 2, and 2^r respectively.

The other important function of `gridify`, not shown above, is to properly *align* terms in preparation for grid normal form. For example, consider the semantics of $[\![CNOT\ a\ b;\ X\ a]\!]_d$:

$$[\![X\ a]\!]_d \times [\![CNOT\ a\ b]\!]_d =$$
$$I_{2^a} \otimes X \otimes I_{2^{d-a-1}}$$
$$\times$$
$$(I_{2^a} \otimes |0\rangle\langle 0| \otimes I_{2^s} \otimes I_2 \otimes I_{2^r} + I_{2^a} \otimes |1\rangle\langle 1| \otimes I_{2^s} \otimes X \otimes I_{2^r}).$$

Our `gridify` tactic rewrites the X term to be $I_{2^a} \otimes X \otimes I_{2^s} \otimes I_2 \otimes I_{2^r}$ to make it compatible with the $CNOT$ term.

3.5 Vector State Abstractions

While the proof of `SWAP d a b` is made simpler by `gridify`, it is still relatively difficult considering that `SWAP` has a simple classical (non-quantum) purpose. In fact, this operation is much more naturally analyzed using its action on classical states. A *classical state* is any state of the form $|i_0 \dots i_d\rangle$ (so $|00\rangle$ and $|11\rangle$ are classical states, while $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is not). The set of all d-qubit classical states is 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 classical states.

Using classical 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 $|\dots x \dots (y \oplus x) \dots\rangle$.
3. *CNOT b a* produces $|\dots y \dots (y \oplus x) \dots\rangle$.
4. *CNOT a b* produces $|\dots y \dots x \dots\rangle$.

In our development, we describe classical vector 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 classical state $f(i)$, and `false` corresponds to 0 and `true` to 1. We also sometimes describe classical 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 (effectively, just converting between binary and decimal encodings).

We prove a variety of facts about the actions of gates on classical 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 →
  ⟦CNOT i j⟧_d × (f_to_vec d f)
      = f_to_vec d (update f j (f j ⊕ f i)).

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

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, which avoids the case analysis needed in `gridify`. Furthermore, these lemmas rarely introduce sums, which significantly increase the size of the proof term (due to subsequent grid normalization). The semantics for $CNOT$ (i.e. $\_ \otimes |1\rangle\langle 1| \otimes \_ \otimes \sigma_x \otimes \_ + \_ \otimes |0\rangle\langle 0| \otimes \_ \otimes I_2 \otimes \_$) tends to make the matrix expressions produced by `gridify` quite large. In contrast, the only `f_to_vec` rule we consider that splits the term into a sum is the rule for the H gate, and both terms of the sum are themselves `f_to_vec` terms.

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. we found `gridify` too inefficient to verify the standard decomposition of the gate, shown below, matches its expected matrix denotation.

```
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 classical 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 →
 ⟦TOFF a b c⟧_d × (f_to_vec d f)
      = f_to_vec d (update f c (f c ⊕ (f a && f b))).
```

The proof of f_to_vec_TOFF is almost entirely automated using a tactic that rewrites using the f_to_vec lemmas discussed above, since T and T† are simply 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 classical states, or even sums over a small number of classical 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 :=
  match n with
  | 0 ⇒ Zero
  | S n' ⇒ vsum n' f .+  f n'
  end.
Fixpoint vkron n (f: ℕ→Vector 2) : Vector (2^n) :=
  match n with
  | 0    ⇒ I 1
  | S n' ⇒ vkron n' f ⊗ f n'
  end.
```

As an example, the action of n parallel Hadamard gates on the state f_to_vec n f can be written as

$$\text{vkron n (fun i} \Rightarrow \tfrac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(i)}\,|1\rangle))$$

or

$$\tfrac{1}{\sqrt{2^n}} * (\text{vsum } 2^n \text{ (fun i} \Rightarrow (-1)^{\text{to\_int}(f)*i} * \text{basis\_vector n i)),}$$

both commonly-used facts in quantum algorithms. Our vsum and vkron definitions share similarities with the *path-sums* approach used in a verification tool by Chareton et al. [9] (see Section 5).

In Section 4 we will write $|f\rangle$ for f_to_vec n f, $|i\rangle$ for basis_vector n i, $\sum_{i=0}^{n-1} f(i)$ for vsum n (fun i ⇒ f i), and $\bigotimes_{i=0}^{n-1} f(i)$ for vkron n (fun i ⇒ f i).

3.6 Measurement Predicates

The proofs in Section 4 do not use the non-unitary semantics directly, but describe the probability of different measurement outcomes using predicates probability_of_outcome and prob_partial_meas (as is done in 𝑄BRICKS [9]).

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

Above, $\|v\|$ is the 2-norm of vector v and $|c|$ is the complex norm of c. We find these predicates sufficient for our use cases, since the programs we verify are purely quantum. That is, they do not use classical subroutines, so we can analyze their outcome purely in terms of the state vector produced.

In fact, the *principle of deferred measurement* says that measurement can always be deferred until the end of a quantum computation without changing the result. However, we included measurement in Section 2.4 because it is a standard feature of quantum programming languages and used in a variety of quantum protocols (see discussion in Section 5.2).

4 Proofs of Quantum Algorithms

In this section we discuss the formal verification of three classic quantum algorithms: Simon's algorithm [23, Chapter 5], Grover's algorithm [23, Chapter 6], and quantum phase estimation [23, Chapter 5]. The proofs in this section all follow the textbook argument.

4.1 Simon's Algorithm

Problem description. Given a function $f : \{0,1\}^n \to \{0,1\}^n$ such that for all $x, y \in \{0,1\}^n$, $f(x) = f(y) \Leftrightarrow x \oplus y \in \{0, s\}$ for unknown $s \in \{0,1\}^n$, the goal of Simon's algorithm is to find s. The inputs to the algorithm are the input size n and a program ("oracle") U_f with the property that $U_f\,|x\rangle\,|y\rangle = |x\rangle\,|f(x) \oplus y\rangle$. If $s = 0$, then the output of Simon's algorithm is a uniform superposition over all n-bit strings (meaning that any string is measured with equal probability). If $s \neq 0$, then the output is a uniform distribution over strings y such that $s \cdot y = 0$, where $x \cdot y$ is the bitwise dot product of x and y modulo 2. The value of s can be determined by $O(n)$ iterations of the algorithm.

The simon function in Figure 2 produces the sqir circuit for the algorithm, which has a simple structure. First, a layer of Hadamard gates prepares a uniform superposition on the first n inputs. Next, U_f encodes information about f in the phase, in essence evaluating the oracle on all possible inputs

```
(* Apply n 1-qubit gates in parallel. *)
Fixpoint npar' d n (U : base 1) : ucom base d :=
  match n with
  | 0 ⇒ ID 0
  | S n' ⇒ npar' d n' U ; uapp1 U n'
  end.
Definition npar n U := npar' n n U.

(* Main program. *)
Definition simon {n} (Uf : ucom base (2 * n)) :=
  npar n H ; Uf ; npar n H.
```

Figure 2. Simon's algorithm in sqir.

at once. Finally, another layer of Hadamard gates brings information in the phase back to the state where it can be measured. The circuit is run on input $|0\rangle^{2*n}$.

Proof effort. The sqir version of Simon's algorithm is two lines (excluding npar), and the specification and proof of correctness are around 540 lines. The proofs were completed in approximately two weeks by a new sqir user.

Proof details. Our statements of correctness for Simon's algorithm say that (1) if s is zero then the probability of measuring any particular output is $1/2^n$, (2) if s is nonzero then the probability of measuring y such that $s \cdot y = 0$ is $1/2^{n-1}$, and (3) if s is nonzero then the probability of measuring y such that $s \cdot y \neq 0$ is 0. We show the full statement of correctness for property (2) below.

```
Lemma simon_nonzero_A :
      ∀ {n : ℕ} (Uf : ucom base (2 * n)) f y s,
  n > 0 → y < 2^n → s < 2^n →
  integer_oracle Uf f →
  (∀ x, x < 2^n → f x < 2^n) →
  (∀ x y, x < 2^n → y < 2^n →
      f x = f y ↔ x ⊕ y = s ∨ x = y)) →
  s ≠ 0 →
  s · y = 0 →
  prob_partial_meas |y⟩ (⟦simon Uf⟧2*n × |0⟩^(2*n)) = 1/2^(n-1).
```

The first three conditions ensure well-formedness of the inputs; the next three describe constraints on f and state that Uf implements f. We call Uf an *integer oracle* because it maps an n-bit number to another n-bit number. The conclusion states that after running the program simon Uf on $|0\rangle^{2*n}$, the probability of measuring y such that $s \cdot y = 0$ is $\frac{1}{2^{n-1}}$.

We begin by showing that for any (well-formed) s and y, prob_partial_meas $|y\rangle$ ($[\![$simon Uf$]\!]_{2*n} \times |0\rangle^{2*n}$) is equal to

$$\left\| \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{x \cdot y} \, |f(x)\rangle \right\|^2 .$$

The proofs of the three properties listed above then amount to showing properties about this norm-sum term.

In the case where $s \neq 0$, f is a two-to-one function, which means that the expression above can be rewritten as a sum over elements in the range of f. In the standard presentation, this expression is simplified as follows.

$$\left\| \frac{1}{2^n} \sum_{z \in \mathrm{range}(f)} \left((-1)^{x_1 \cdot y} + (-1)^{x_2 \cdot y} \right) |z\rangle \right\|^2, \quad f(x_1) = f(x_2) = z$$

$$= \left\| \frac{1}{2^n} \sum_{z \in \mathrm{range}(f)} \left((-1)^{x_1 \cdot y} + (-1)^{(x_1 \oplus s) \cdot y} \right) |z\rangle \right\|^2$$

$$= \left\| \frac{1}{2^n} \sum_{z \in \mathrm{range}(f)} \left((-1)^{x_1 \cdot y} (1 + (-1)^{s \cdot y} \right) |z\rangle \right\|^2$$

From this rewritten form, it is clear that the probability of measuring y such that $s \cdot y = 0$ is $2 * 1/2^n = 1/2^{n-1}$ and the probability of measuring y such that $s \cdot y \neq 0$ is 0.

Our Coq proof essentially follows this structure, although we found it easier to define a function to_injective that takes the two-to-one function f and makes it one-to-one.

```
Definition to_injective n s f x :=
  let y := x ⊕ s in
  if (x <? y) then f x else (2^n + f x).
```

Using this function, we can rewrite the norm-sum term as a sum over vectors of size 2^{n+1}.

$$\left\| \sum_{x=0}^{2^n-1} (-1)^{x \cdot y} \, |f(x)\rangle \right\| =$$

$$\frac{1}{\sqrt{2}} \left\| \sum_{x=0}^{2^n-1} \left((-1)^{x \cdot y} + (-1)^{(x \oplus s) \cdot y} \right) |(\texttt{to\_injective n s f})(x)\rangle \right\|$$

4.2 Grover's Algorithm

Problem description. Given a circuit implementing a Boolean oracle f:$\{0,1\}^n \rightarrow \{0,1\}$, the goal of Grover's algorithm is to find an input x satisfying $f(x) = 1$. Suppose that $n \geq 2$. In the classical case, this problem cannot be solved using fewer than $O(2^n)$ queries to the oracle. However, the quantum algorithm finds a 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 3. For a more detailed discussion see Nielsen and Chuang [23, Chapter 6].

Proof effort. The sqir version of Grover's algorithm is 15 lines. The specification and proof are around 770 lines. The proof took approximately one person-week.

```
(* Controlled-X with target (n-1) and controls
   0, 1, ..., n-2. *)
Fixpoint generalized_Toffoli' n0 : ucom base n :=
  match n0 with
  | O | S O ⇒ 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 U_H; npar n U_X ;
  H (n - 1) ; generalized_Toffoli ; H (n - 1) ;
  npar n U_X; npar n U_H.

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

Figure 3. Grover's algorithm in SQIR. cast is a no-op that changes the dimension in a ucom's type.

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 $\frac{\pi}{4}\sqrt{\frac{2^n}{k}}$.

We begin the proof by showing that the uniform superposition can be rewritten as a sum of "good" states that satisfy f and "bad" states 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 U_f and diff perform the expected reflections (e.g. $[\![\texttt{diff}]\!]_n = -2\,|\psi\rangle\langle\psi| + I_{2^n}$), leading to the main result.

```
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.
```

Above, Rsum is a sum over real numbers.

4.3 Quantum Phase Estimation

Problem description. Given a unitary matrix U and eigenvector $|\psi\rangle$ such that $U\,|\psi\rangle = e^{2\pi i\theta}\,|\psi\rangle$, 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 [36]. For more details on phase estimation see Nielsen and Chuang [23, Chapter 5].

The circuit for QPE is shown in Figure 4. First, a layer of Hadamard gates prepares a uniform superposition. Next, a sequence of controlled U operations encodes information about θ in the phase. Finally, the inverse quantum Fourier transform (QFT) is used to recover the information about θ stored in the phase. Note that the circuits for QPE and QFT both have a recursive structure, making them simple to encode in a functional language. The full SQIR definition of QPE is given in Figure 5.

Proof effort. The SQIR version of QPE is around 40 lines (excluding utility definitions like control and map_qubits), and the specification and proof of correctness 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. We had developed the f_to_vec infrastructure beforehand, but the vsum and vkron abstractions were fleshed out while we worked out the proof of QPE.

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 $\geq 4/\pi^2$).

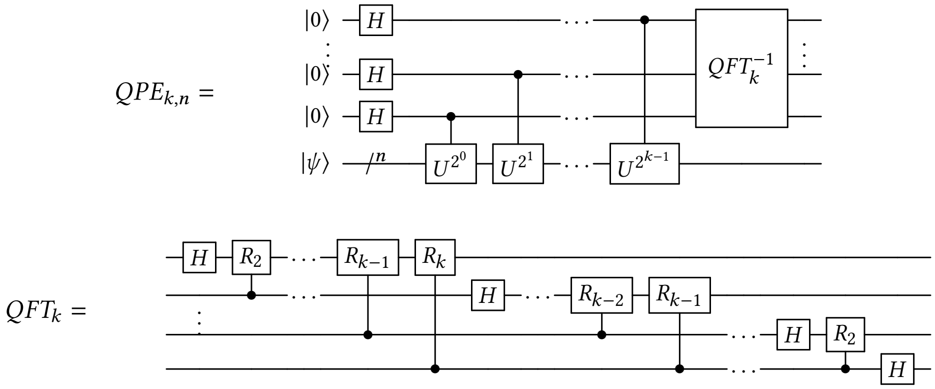

Figure 4. Circuit for quantum phase estimation (QPE) with k bits of precision and an n-qubit input state (top) and quantum Fourier transform (QFT) on k qubits (bottom). $|\psi\rangle$ and U are inputs to QPE. R_m is a z-axis rotation by $2\pi/2^m$.

```
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 = $e^{2\pi i\theta}$ * ψ →
  prob_partial_meas |z⟩ (⟦QPE k n u⟧$_{k+n}$ × (|0⟩$^{\otimes k}$ ⊗ ψ))
     ≥ 4 / $\pi^2$.
```

Pure_State_Vector is more restrictive 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⟩ =
    $e^{2\pi i(x_0\ \cdot\ x_1x_2...x_{n-1})/2^n}$|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 ⟦controlled_rotations n ⟧$_n \times |x_1x_2...x_{n-1}\rangle = e^{2\pi i(x_0\ \cdot\ x_1x_2...x_{n-1})/2^n}|x_1x_2...x_{n-1}\rangle$. We then perform the simplifications shown in Figure 6, which prove our property.

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⟩ = $\frac{1}{\sqrt{2^n}}\bigotimes_{j=0}^{n-1}(|0\rangle + e^{2\pi ix/2^{n-j}}\,|1\rangle)$.
```

It is also often useful to have a version of this lemma in a form similar to the classical definition of Fourier states. This requires reversing the output order of the qubits. We provide a reverse function that does this and verify that it has the desired action on vkron terms ($\bigotimes_{i=0}^{n-1} f(i) = \bigotimes_{i=0}^{n-1} f(n-i-1)$). With this definition, we can state and prove the following correctness property for QFT.

```
Lemma QFT_w_reverse_semantics : ∀ n x, n > 1 →
  ⟦QFT_w_reverse n⟧$_n$ × |x⟩ = $\frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1} e^{2\pi ixk/2^n}\,|k\rangle$.
```

The proof of this lemma relies on QFT_semantics and the (verified) fact that $\bigotimes_{k=0}^{n-1}(|0\rangle + e^{i\alpha 2^{n-k-1}}\,|1\rangle) = \sum_{k=0}^{2^n-1} e^{i\alpha k}\,|k\rangle$.

5 Related Work

The earliest attempts to formally verify quantum programs in a proof assistant were an Agda implementation of the Quantum IO Monad [16] and a small Coq quantum library by Boender et al. [7]. These were both proofs of concept, and neither developed beyond verifying basic protocols.

This section surveys more recent and substantial work on verified quantum programming, paying special attention to three tools: *Q*WIRE [29] (implemented in Coq); quantum Hoare logic (QHL) [22] and quantum relational Hoare logic [39] (implemented in Isabelle [24]); and *Q*BRICKS [9] (implemented in Why3 [13]). These three are the only tools aside from SQIR that have been used to verify interesting, parameterized quantum programs. In particular, QHL has been used to verify Grover's and *Q*BRICKS has been used to verify Grover's and QPE. These tools share several commonalities in design, reflective of the types of challenges encountered when verifying quantum programs.

SQIR itself was previously introduced as part of a verified optimizer for quantum circuits [20]; circuits were represented as SQIR programs. While we developed the SQIR-based verification framework as part of that work, this paper is the first to report the details of SQIR's use in correctness proofs.

```
(* 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
  | 1    ⇒ H 0
  | S n' ⇒ H 0 ; controlled_rotations n ; map_qubits (fun q ⇒ q + 1) (QFT n')
  end.

(* QFT outputs qubits in the wrong order, so the qubits need to be reversed before
   further processing. This can be handled by the classical control hardware or on
     the
   quantum machine with SWAPs, as done here. *)
Fixpoint reverse_qubits' dim n : ucom base dim :=
  match n with
  | 0    ⇒ SKIP
  | S n' ⇒ reverse_qubits' dim n' ; SWAP n' (dim - n' - 1)
  end.
Definition reverse_qubits n := reverse_qubits' n (n/2).
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 5. sqir definition of QPE. Some type annotations and calls to `cast` have been removed for clarity. `control`, `map_qubits`, `niter`, and `invert` are Coq functions that transform sqir programs; we have proved that they have the expected behavior (e.g. $[\![\text{invert } u]\!]_n = [\![u]\!]_n^\dagger$) for any input program.

5.1 *Q*wire

The *Q*wire language [26, 29] originated as an embedded circuit description language in the style of Quipper [15] but with a more powerful type system. The core of *Q*wire is small enough to be presented verbatim:

$$\begin{aligned}
&\llbracket \texttt{controlled\_rotations (n+1)} \rrbracket_{n+1} \times |x\rangle \\
&\quad = \llbracket \texttt{control}\ x_n\ (\texttt{Rz}\ (2\pi/2^{n+1})\ 0 \rrbracket_{n+1} \times \llbracket \texttt{controlled\_rotations n} \rrbracket_{n+1} \times |x\rangle && \text{unfold definitions} \\
&\quad = \llbracket \texttt{control}\ x_n\ (\texttt{Rz}\ (2\pi/2^{n+1})\ 0 \rrbracket_{n+1} \times e^{2\pi i (x_0\ \cdot\ x_1 x_2 \dots x_{n-1})/2^n}\ |x_1 x_2 \dots x_{n-1} x_n\rangle && \text{apply I.H.} \\
&\quad = e^{2\pi i (x_0\ \cdot\ x_n)/2^{n+1}} e^{2\pi i (x_0\ \cdot\ x_1 x_2 \dots x_{n-1})/2^n}\ |x_1 x_2 \dots x_{n-1} x_n\rangle && \text{simplify controlled-}Rz \\
&\quad = e^{2\pi i (x_0\ \cdot\ x_1 x_2 \dots x_n)/2^{n+1}}\ |x_1 x_2 \dots x_{n-1} x_n\rangle && \text{combine exponential terms}
\end{aligned}$$

Figure 6. Reasoning used in the proof of correctness of `controlled_rotations`.

```
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 (written `Pat`) correspond to variables and `WTypes` specify the types of each variable: A `Qubit`, `Bit`, `Unit`, or more complex structure of bits and qubits. The `output` constructor simply outputs a given variable. Gate application takes a gate, a pattern and a *continuation* (a function from a pattern to a circuit) and applies the gate to the given pattern, plugging the output into the continuation. Finally, `lift` adopts the *dynamic lifting* approach to measurement from Quipper, taking in a bit to measure and a function computing the continuation based on the result.

Every Qwire `Circuit` expects a variable argument and hence corresponds to an open term. The `box` constructor produces a closed term from an input pattern and circuit to be applied to that pattern.

```
Inductive Box w1 w2 : Set :=
  box : (Pat w1 → Circuit w2) → Box w1 w2.
```

Unlike in sqir, `w1` and `w2` are not required to be the same: Qwire can freely allocate and deallocate bits and qubits via its `init` and `discard` gates. It can also shuffle them around, as in the following implementation of a swap function:

```
Definition swap: Box (Qubit ⊗ Qubit) (Qubit ⊗ Qubit)
 := box (fun (p,q) ⇒ output (q,p))
```

Unfortunately, Qwire's flexibility proves to be its Achilles' heel when it comes to formal verification. To translate circuits to operations on density matrices, its variables (represented using higher-order abstract syntax [27]) must be mapped to concrete matrix indices. Each time a qubit is discarded, indices undergo a de Bruijn-style shifting, which immensely complicates inductive reasoning about circuits.

Qwire has mainly been used to verify simple randomness generation circuits and a few textbook circuits [33]. It was also used to prove the correct garbage collection of spare qubits and the generation of Boolean oracles [32], though here its design created difficulties. Ancilla management and Boolean compilation both require qubits to be static and maintain their positions, while Qwire introduces a new qubit after every gate application and has no intrinsic notion of position. Hence, this development introduced the circuits `CNOT_at a b` and `Toffoli_at a b c` with concrete indices and admitted their semantics [29].

Challenges in using Qwire to prove non-trivial algorithms correct motivated the simpler, position-based design of sqir. In addition, the difficulty in using Qwire's density matrices motivated the development of sqir's unitary core. sqir began with Qwire's libraries for matrices and complex numbers to describe the semantics of programs. Over the development of sqir, we estimate that we have extended Qwire's libraries with around 3000 LOC of additional linear algebraic lemmas and automation (e.g. `restore_dims` and `gridify`).

5.2 Qbricks

Qbricks [9] is a quantum proof framework based in Why3 [13], developed concurrently with sqir. In Qbricks' highly structured DSL, one can build quantum circuits using parallel and sequential composition of unitary gates. For example, the sqir program `H 0 ; CNOT 0 1; CNOT 1 2` could be written thus

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

Qbricks additionally provides a constructor `ANC(c)` that introduces an ancillary qubit at the end of `c` and discards it immediately after use. Qbricks is entirely unitary (there is no measurement construct), which requires that an ancilla be returned to its original state by the intermediate computation. Similar to sqir's use of concrete indices, Qbricks-DSL's compositional structure makes it simple to map programs to their denotation: The "index" of a gate application can be easily computed by its nested position in the program.

The semantics of Qbricks are based on the *path-sums* formalism of Amy [2, 3], in which every unitary transformation is represented as a function of the form:

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

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 $|x\rangle \rightarrow \frac{1}{\sqrt{2}} \sum_{y=0}^{1} e^{2\pi i x y/2}\ |y\rangle$ and T has the form $|x\rangle \rightarrow e^{2\pi i x/8}\ |x\rangle$. Qbricks' *higher-order path-sums* (HOPS) generalizes path-sums with program variables and size parameters, permitting proofs about families of quantum circuits while maintaining compositionality (path-sums are closed under matrix and tensor products).

A key design goal for *Q*BRICKS, and a motivation for the development of HOPS, is to use deductive reasoning amenable to automation. They achieve this by implementing an verification condition generator within the Why3 framework, which interfaces to an SMT solver. We suspect that integrating with an SMT solver would be advantageous for SQIR too, and largely remove the issues addressed by tactics like `restore_dims` (Section 3.3). The path-sums semantics is not so different from our `vkron` and `vsum` vector-state abstractions, and can be naturally described in their terms:

```
Definition PS (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 its inputs and `f i` is a Boolean function over its inputs, for any `i`. We found working with vector states to be more amenable to automation than working with matrices directly (Sections 3.4 and 3.5); the high degree of automation achieved by *Q*BRICKS seems to reinforce this experience.

*Q*BRICKS' path-sums have proven highly capable of proving the correctness of sophisticated quantum algorithms. Indeed, their announcement of a proof of quantum phase estimation inspired us to attempt such a proof ourselves. However, it remains to be seen whether path-sums are appropriate for describing more complex algorithms, like the seven algorithms implemented in Quipper [15]. By contrast, SQIR uses complex matrices, the standard representation of quantum computing, and layers whatever abstractions needed (e.g. vector states and custom automation) on top, leveraging the substantial Coq ecosystem.

One final similarity between *Q*BRICKS and the SQIR proofs in this paper is that they use predicates to express measurement probability, without explicitly using a measurement operation. Although we find these predicate sufficient for the examples in this paper, SQIR's full semantics do include proper measurement. We do not know if or how *Q*BRICKS intends to handle branching measurement, an integral feature of many quantum constructs like repeat-until-success loops [25] or error-correcting codes [14] and paradigms like measurement-based quantum computation [8]. As quantum algorithms and quantum computers evolve in complexity, we expect measurement to become ever more critical, and therefore important to verify.

5.3 Quantum Hoare Logic

Quantum Hoare logic (QHL), initially developed by Ying [40], has recently been formalized in the Isabelle/HOL proof assistant [21]. QHL uses the quantum while language (QWhile), which has the following syntax:

$$\begin{aligned} S := \texttt{skip} \mid \overline{q} := U\overline{q} \mid S_1; S_2 \mid \texttt{measure}\ M[\overline{q}] : \overline{S} \mid \\ \texttt{while}\ M[\overline{q}] = 1\ \texttt{do}\ S \end{aligned}$$

The first three commands are identical to SQIR's skip, unitary application and sequencing commands. Measurement is generalized to a case statement rather than if-then-else by allowing measurement operators that produce multiple outcomes (rather than binary 0/1). The notable addition to QWhile is its `while` construct, which allows potentially unbounded looping on a (binary) measurement outcome. This last construct is important since QWhile does not support metaprogramming, but likely infeasible on near-term machines. QWhile programs use a finite number of quantum variables, fixed before each run of the program. To convert between variables and qubit indices, Liu et al. [21] define *encode* and *decode* functions following Bentkamp et al. [5]. Thus, QWhile's variables are, in effect, simply program-level names for SQIR's concrete indices.

Given that measurement is a core part of the quantum while language, QWhile's semantics are given in terms of (partial) density matrices. (We call a density matrix *partial* when it may represent a sub-distribution—that is, a subset of the outcomes of measurement.) To support automation, Liu et al. provide a matrix normalization tactic along the lines of SQIR's `gridify` [21, Section 5.1]. Unlike *Q*WIRE and SQIR, which rely on the Kronecker product, Liu et al. use an *abstract* extension operation to extend matrices acting on a subset of qubits to the full space [21, Section 3.4]. We suspect that this approach may be worthwhile implementing in SQIR, and may lead to more effective automation.

Liu et al. [21]'s current work is limited to proofs using QHL—they have not considered reasoning directly about program semantics, as is done in SQIR. It's not clear how broadly applicable this logic is: Zhou et al. [41] needed to revise the logic substantially while restricting its predicates in order to prove the correctness of the Harrow-Hassidim-Lloyd algorithm [19] (on paper). It is interesting future work to consider using the Isabelle framework to reason directly about programs semantics, or alternatively to implement a logic like QHL or Zhou et al.'s aQHL on top of SQIR, to compare the two approaches.

6 Open Problems and Future Work

Verifying near-term algorithms. 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. For example, the version of QFT considered in Section 4 is the textbook presentation. In practice, it is more popular to consider an approximate QFT that removes gates that perform rotations by small angles [23]. 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.

A related point 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 [23]. On-paper proofs about errors typically represent programs using quantum channels in a suitable mathematical form (e.g. Kraus form), which sqir currently does not support.

Verifying the quantum software toolchain. Along with verifying quantum programs, it is equally important to verify the infrastructure used to reason about those programs and turn them into executable code (along the lines of VST for classical software [4]). Our work on sqir and VOQC (optimizing a gate-level intermediate representation) is one part of this toolchain, but there is still much to be done. For example, a proper compiler needs a high-level source language, perhaps along the lines of recent languages like Q# [38] or Silq [6]. Currently, none of the languages formalized in proof assistant are "high-level" (likely due to issues with mapping variables to indices, as discussed throughout this paper). Conversely, there is currently no support for verifying programs below the gate level. The lowest level in the quantum compiler stack is analog *pulse* instructions for the classical control hardware [1]. It might also be useful to verify other components of the standard quantum software toolchain, such as resource estimators or simulators.

Teaching quantum computing. An early version of sqir serves as the basis for Verified Quantum Computing (VQC) [30], an online textbook in the style of Software Foundations [28] introducing readers to quantum computing through the Coq proof assistant. Introducing quantum computing in a proof assistant has proved challenging but rewarding: When unitary matrices (defined as matrices where $UU^\dagger = I$) are introduced, students must immediately show that $U\psi$ always results in a quantum state, a surprisingly straightforward exercise in Coq. VQC has been successfully taught in a formal verification course at the University of Maryland and a tutorial at Principles of Programming Languages [31], but it remains work-in-progress. The techniques and algorithms in this paper (particularly the `f_to_vec` function for succinctly describing classical states) should allow us to cover the material in a standard quantum computing textbook while providing instant feedback to students. This will further strengthen the connection between quantum computing and formal proof, which we expect to prove valuable to programmers in the near future.

Acknowledgments

We thank Yuxiang Peng for contributions to the sqir codebase and Xiaodi Wu for initial contributions to and feedback about sqir and follow-on projects. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040.

References

[1] Thomas Alexander, Naoki Kanazawa, Daniel J. Egger, Lauren Capelluto, Christopher J. Wood, Ali Javadi-Abhari, and David McKay. 2020. Qiskit Pulse: Programming Quantum Computers Through the Cloud with Pulses. *arXiv e-prints* (April 2020). arXiv:quant-ph/2004.06755

[2] Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018.*

[3] Matt Amy. 2019. *Formal Methods in Quantum Circuit Design.* Ph.D. Dissertation. University of Waterloo.

[4] Andrew W. Appel. 2011. Verified Software Toolchain *(ESOP'11/ETAPS'11).*

[5] Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. 2019. A Formal Proof of the Expressiveness of Deep Learning. 63, 2 (2019).

[6] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. 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).*

[7] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. 2015. Formalization of Quantum Protocols using Coq. In *Proceedings of the 12th International Workshop on Quantum Physics and Logic, Oxford, U.K., July 15-17, 2015 (Electronic Proceedings in Theoretical Computer Science),* Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.), Vol. 195. Open Publishing Association, 71–83. https://doi.org/10.4204/EPTCS.195.6

[8] Hans J Briegel, David E Browne, Wolfgang Dür, Robert Raussendorf, and Maarten Van den Nest. 2009. Measurement-based quantum computation. *Nature Physics* 5, 1 (2009), 19–26.

[9] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. 2020. Toward Certified Quantum Programming. *arXiv e-prints* (2020). arXiv:cs.PL/2003.05841

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

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

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

[13] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming (Lecture Notes in Computer Science).*

[14] Daniel Gottesman. 2010. 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, Vol. 68. 13–58.

[15] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 333–342.

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

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

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

[19] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters* 103, 15, Article 150502 (Oct. 2009), 150502 pages. https://doi.org/10.1103/PhysRevLett.103.150502 arXiv:quant-ph/0811.3171

[20] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2019. A Verified Optimizer for Quantum Circuits. *arXiv preprint arXiv:1912.02250* (2019).

[21] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. 187–207. https://doi.org/10.1007/978-3-030-25543-5_12

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

[23] Michael A. Nielsen and Isaac L. Chuang. 2000. *Quantum Computation and Quantum Information*. Cambridge University Press.

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

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

[26] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

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

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

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

[30] Robert Rand. 2019. *Verified Quantum Computing*. http://www.cs.umd.edu/~rrand/vqc/index.html online.

[31] Robert Rand. 2020. Verified Quantum Computing. Principles of Programming Languages (POPL), 2020: Tutorialfest.

[32] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2018. ReQWIRE: Reasoning about Reversible Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*.

[33] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017*. 119–132. https://doi.org/10.4204/EPTCS.266.8

[34] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018. Phantom Types for Quantum Programs. The Fourth International Workshop on Coq for Programming Languages.

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

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

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

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

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

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

[41] Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1149–1162.

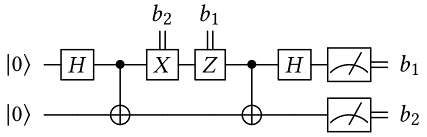

Figure 7. Circuit for superdense coding.

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

Definition bell00 := H a; CNOT a b.

Definition decode := CNOT a b; H a.

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

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

Figure 8. sqir program for the unitary portion of the superdense coding algorithm. We have removed type annotations for clarity, but each sqir program has type `ucom base 2`, which describes a unitary circuits that uses the base gate set and has a global register of size two.

A Additional Examples

sqir's simple structure and semantics allow us to easily verify general properties of quantum programs. In this section we discuss correctness properties of three quantum programs written in sqir: superdense coding, the Deutsch-Jozsa algorithm, and quantum phase estimation. The Deutsch-Jozsa algorithm is verified for *any Boolean oracle with any number of qubits* and quantum phase estimation is verified for *any input program and eigenvector with any number of qubits*, demonstrating that sqir supports parameterized proofs. We also present an alternative proof for quantum teleportation using a non-deterministic semantics; the original is found in **??**, along with our proof of GHZ state preparation.

A.1 Superdense Coding

Superdense coding is a protocol that allows a sender to transmit two classical bits (i.e., Booleans), b_1 and b_2, to a receiver using a single quantum bit. Doing so relies on the sender and receiver sharing a *Bell pair*, as shown in Figure 7. The sender conditionally applies X and Z to her qubit, contingent on the values of her bits, and then transmits it to the receiver, who applies a *Bell measurement* (reversed entangling operation followed by measure) to recover the original bits. The sqir program corresponding to the unitary part of this circuit is produced by the Coq function `superdense`, shown in Figure 8, which is parametrized by the classical input bits b_1 and b_2.

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

```
Lemma superdense_correct :
  ∀ b1 b2, ⟦superdense b1 b2⟧_2 × ∣ 0,0 ⟩ = ∣ b1,b2 ⟩.
```

The proof simply destructs `b1` and `b2` and applies our matrix simplification tactics.

A.2 Teleport with Nondeterministic Semantics

The proof of correctness of quantum teleportation using the density matrix semantics (**??**) is simple, but not particularly useful for understanding *why* the protocol is correct. A more illuminating proof can be carried out using an alternative *nondeterministic semantics* in which evaluation is expressed as a relation. Given a state $|\psi\rangle$, unitary program u will (deterministically) evaluate to $[\![\mathrm{u}]\!]_d \times |\psi\rangle$. However, `meas q p1 p2` may evaluate to either `p1` applied to $|1\rangle\langle 1| \times |\psi\rangle$ or `p2` applied to $|0\rangle\langle 0| \times |\psi\rangle$. At every point in the program, the nondeterministic semantics represents the state using a vector $|\psi\rangle$ rather than a density matrix ρ.

However, because we do not track all possible measurement outcomes, the nondeterministic semantics is only useful for proving properties for which all outcomes lead to the same result. The correctness of the teleport protocol is an example of such a property, because we obtain the original qubit regardless of the measurement outcome. Similarly, the nondeterministic semantics is only useful for verifying transformations of programs involving deterministic outcomes. In practice, we use the density matrix semantics to verify our optimizations.

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

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

This says that on input $|\psi\rangle \otimes |0,0\rangle$, `teleport` will produce a state that is proportional ($\propto$) to $|0,0\rangle \otimes |\psi\rangle$. Note that this statement is quantified over every outcome ψ' and hence all possible paths to ψ'. If instead we simply claimed that `teleport` / (ψ $\otimes$ |0,0⟩) $\Downarrow$ 1/2 $*$ (|0,0⟩ $\otimes$ ψ), where the $1/2$ factor reflects the probability of each measurement outcome ($(1/2)^2 = 1/4$), we would only be stating that some such path exists.

The first half of the circuit is unitary, so the proof simply computes the effect of applying a H gate, two *CNOT* gates and another H gate to the input vector state. The two measurement steps then leave us with four different cases to consider. In each of the four cases, we can use the outcomes of measurement to correct the final qubit, putting it into the

```
Fixpoint npar n (u : ℕ → ucom base n) :=
  match n with
  | 0 ⇒ SKIP
  | S n' ⇒ npar n' u ; u n'
  end.
Definition deutsch_jozsa n (u : ucom base n) :=
  X (n-1) ; npar n H ; u ; npar n H.
```

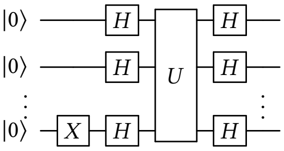

Figure 9. The Deutsch-Jozsa algorithm in sqir and as a circuit. The Coq function npar constructs a sqir program that applies the same operation to every qubit.

state $|\psi\rangle$. Finally, resetting the already-measured qubits is deterministic and leaves us with the desired state.

A.3 Deutsch-Jozsa Algorithm

In the Deutsch-Jozsa problem [12], the goal is to determine whether a Boolean function $f : \{0,1\}^n \longrightarrow \{0,1\}$ is *constant* (always returns the same value) or *balanced* (returns 0 and 1 equally often), given that one is the case. The function f is encoded in an oracle $U : |x, y\rangle \mapsto |x, y \oplus f(x)\rangle$, which is a linear operator over a 2^{n+1} dimensional Hilbert space. In Coq, we express the requirement that program u encodes the function f as follows.

```
Definition boolean_oracle {n} (u : ucom (n + 1)) f :=
  ∀ x y, ⟦u⟧_{n+1} × |x⟩ ⊗ |y⟩ = |x⟩ ⊗ |y ⊕ (f x)⟩.
```

To express that a function is constant or balanced, we can define a function count f n that counts all inputs on which function f (with domain size 2^n) evaluates to true. Then we have:

```
Definition balanced f n := n > 0 ∧ count f n = 2^{n-1}.
Definition constant f n := count f n = 0 ∨  count f n
      = 2^n.
```

As shown in Figure 9, the Deutsch-Jozsa algorithm begins with an all $|0\rangle$ state and prepares the input state $|+\rangle^{\otimes n} \otimes |-\rangle$ by applying an X gate on the last qubit followed by a H gate on every qubit. Next the oracle U is queried, and a H gate is again applied to every qubit in the program. Finally, all qubits are measured in the standard basis. If measuring all the qubits but the last yields an all-zero string (the last qubit is guaranteed to be in the $|1\rangle$ state) then the algorithm outputs “accept,” indicating that the function is constant. Otherwise the algorithm outputs “reject.”

The probability of measuring $|0\rangle$ in the first n qubits and $|1\rangle$ in the last qubit is given by

$$Pr_{accept} = |\,(\langle 0|^{\otimes n} \otimes \langle 1|) \times (\texttt{deutsch\_jozsa n U} \times |0\rangle^{\otimes(n+1)})\,|^2.$$

We define an accept predicate that states that $Pr_{accept} = 1$ and a reject predicate that states that $Pr_{accept} = 0$. Our correctness property is then stated as follows.

```
Lemma deutsch_jozsa_correct :
  ∀ (n : ℕ) (f : ℕ → 𝔹) (u : base_ucom (n + 1)),
  n > 0 → boolean_oracle u f →
  (constant f n → accept u) ∧ (balanced f n →
    reject u).
```

The key lemma in our proof states that Pr_{accept} depends on the number of inputs on which f evaluates to 1, i.e., count f n. In particular, $Pr_{accept} = |1 - \frac{2*(\text{count f n})}{2^n}|^2$. We prove this property using matrix simplification and induction on n. Correctness of the Deutsch-Jozsa algorithm follows directly from this lemma. For a constant function, count f n $= 0$ or count f n $= 2^n$ so $Pr_{accept} = 1$. For a balanced function, count f n $= 2^{n-1}$ so $Pr_{accept} = 0$.

Unlike the GHZ state preparation, quantum teleportation, and superdense coding examples presented so far, the deutsch_jozsa program is parameterized by both the size of the global register n and a sqir program u. Many quantum algorithms are described as such *families* of circuits, so parameterized programs are an important target for verification tools [9]. Proofs about parameterized programs are only possible in tools that manipulate the semantics matrix symbolically.

A.4 Example: Error Correction.

As an example of non-unitary syntax and semantics, consider the following program describing a protocol correcting an X error using the bit-flip code. The goal is to transmit a single-qubit state through a noisy channel which has a bit-flip (X) error on a qubit.

```
Definition encode {n} (q a b : ℕ) : com base n :=
    CNOT q a; CNOT q b.
Definition error {n} (k : ℕ) : com base n := X k.
Definition extraction {n} (q a b s1 s2 : ℕ) : com
    base n := CNOT a s1; CNOT b s1; CNOT q s2; CNOT
    b s2; measure s1; measure s2.
Definition correction {n} (q a b s1 s2 : ℕ) : com
    base n := meas s1 (meas s2 (X b) (X a)) (meas s2
     (X q) skip).
Definition decode {n} (q a b : ℕ) : com base n :=
    CNOT q a; CNOT q b.
Definition ec (k : ℕ): com base 5 := encode 0 1 2;
    error k; extraction 0 1 2 3 4; correction 0 1 2 3
     4; decode 0 1 2.
```

First the program encode maps a logical qubit $|b\rangle$ to a codeword $|\bar{b}\rangle = |bbb\rangle$ for $b \in \{0,1\}$. While in general, the error can be any classical distribution over different locations, it suffices to consider an error at any location k. The error is detected by computing the syndrome into ancilla qubits `s1` `s2` in program `extraction`. The error location is mapped to a quantum state

```
Definition syndrome (k : ℕ) : Matrix 4 4 :=
  match k with
  | 0 ⇒ |0⟩⟨0| ⊗ |1⟩⟨1|
  | 1 ⇒ |1⟩⟨1| ⊗ |0⟩⟨0|
  | 2 ⇒ |1⟩⟨1| ⊗ |1⟩⟨1|
  | _ ⇒ Zero
  end.
```

Then `correction` applies `skip`, `X q`, `X a`, `X b` when the values of `s1 s2` are 00, 01 10, 11 respectively. Finally `decode` maps the codeword $|bbb\rangle$ to $|b00\rangle$.

We describe the correctness of the protocol `ec` as follows.

```
Lemma ec_correct : ∀ k (ρ : Density 2),
    WF_Matrix ρ → (k ≤ 2)%ℕ →
    ⦃ec k⦄_5 (ρ ⊗ |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ |0⟩⟨0|) = (ρ
      ⊗ |0⟩⟨0| ⊗ |0⟩⟨0| ⊗ syndrome k).
```