Journal of Circuits, Systems, and Computers
(2024) 2450112 (25 pages)

DOI: 10.1142/S0218126624501123

Formalizing the Semantics of a Classical-Quantum Imperative Language in Coq*

Wenjun Shi †,§, Qinxiang Cao ‡,¶ and Yuxin Deng †,‖

†*Shanghai Key Laboratory of Trustworthy Computing,*
East China Normal University,
3663 North Zhongshan Road, Shanghai 200062, P. R. China

‡*John Hopcroft Center of Computer Science,*
Shanghai Jiao Tong University,
800 Dongchuan Road, Shanghai 200240, P. R. China

§*wjshi@stu.ecnu.edu.cn*
¶*caoqinxiang@sjtu.edu.cn*
‖*yxdeng@sei.ecnu.edu.cn*

Received 5 May 2023
Accepted 17 September 2023
Published 27 October 2023

In order to verify the functional correctness of quantum circuits or algorithms, a prominent approach is to specify them as quantum programs and semi-automatically deduce them in a theorem prover. It is indispensable to first formalize the semantics of the basic quantum language. We formalize in Coq an imperative language which allows for classical and quantum information interactions. We define small-step operational semantics and state-based denotational semantics. Then we prove a consistency theorem between these two semantics. A distribution-based denotational semantics is also defined and related to the state-based one. Finally, we describe a few typical quantum algorithms and utilize the distribution-based denotational semantics to verify their correctness.

Keywords: Quantum programming language; operational semantics; denotational semantics; theorem prover; Coq.

1. Introduction

In 1980, Benioff[1] invented the concept of quantum computation. Since then, many quantum algorithms, e.g., Shor's factorization algorithm,[2] have been put forward to handle some classical problems more efficiently. However, the quantum characteristics that give rise to these advantages, like superposition and entanglement, also make quantum computation complex and uncertain. The design of quantum

*This paper was recommended by Regional Editor Tongquan Wei.
‖Corresponding author.

algorithms is known to be tricky. It is also very difficult to check the correctness of quantum algorithms by executing or simulating quantum programs. Recently, there is increasing interest in reasoning about quantum programs and analyzing quantum algorithms by formal verification methods.

The most well-known formal verification means are model checking as well as theorem proving. Many model checking techniques concerned with quantum computation have flourished in the past twenty years. The primary work mostly aimed at analyzing quantum communication protocols and quantum error correction.[3–11] Quantum automata[12,13] were also investigated to verify some linear-time properties such as reachability.[14,15] In particular, there exist some recent works[16–18] about circuit simulators based on the quantum-dot cellular automata.[19] Model checking techniques for other models like quantum Markov decision processes[20–28] were widely studied. During the last decade, theorem proving techniques related to quantum computation have also drawn a lot of attention, which will be considered detailedly in Sec. 5.

Programming languages are a bridge for the interactions between human beings and computers. Similarly, quantum programming languages are a key element for exploiting quantum computation. In this work, we embed in Coq[29] a programming language with classical and quantum constructs. The semantic theory of this language has been investigated before,[30,31] where case studies are conducted manually. Coq is one of the mainstream theorem provers, and its inductive proof technique helps us to analyze certain algorithms whose state spaces may not be bounded, e.g., the Deutsch–Jozsa algorithm family.

Contributions of this paper

(i) We formalize the syntax, small-step operational semantics and state-based denotational semantics of a classical-quantum imperative language. Additionally, we prove the consistency of these two semantics.
(ii) We lift the denotational semantics from states to distributions and associate the latter with the former. Furthermore, we describe a few typical quantum algorithms and utilize the distribution-based denotational semantics to verify their correctness.

Organization of this paper

In Sec. 2, we recall several fundamental notions concerned with linear algebra as well as quantum theory. In Sec. 3, we formalize the syntax, operational and denotational semantics of a classical-quantum imperative language and establish some relationships between them. In Sec. 4, we describe some practical quantum algorithms and verify their correctness. In Sec. 5, we compare with related work. We summarize the contributions of this paper in Sec. 6.

The Coq code of this work is downloadable at https://github.com/Vickyswj/QIMP.

2. Basics of Quantum Computation

We briefly review several fundamental concepts related to linear algebra and quantum mechanics. To learn additional knowledge, we recommend the textbook written by Nielsen and Chuang.[32]

2.1. *Linear algebra*

The linear algebra associated with quantum computation mainly includes the *Hilbert space* and some *linear operators*.

In a vector space V, the *inner product* is a mapping

$$\langle \cdot \,|\, \cdot \rangle : V \times V \to \mathbb{C}$$

such that for any vectors $|v\rangle$, $|w_i\rangle \in V$, these three properties are satisfied:

- $\langle v \,|\, w \rangle = \langle w \,|\, v \rangle^*$;
- $\langle v | \sum_i c_i |w_i\rangle = \sum_i c_i \langle v \,|\, w_i \rangle$;
- $\langle v \,|\, v \rangle \geq 0$ and the equality holds if $|v\rangle = 0$,

where $\mathbb{C}$ is the complex number set, and for every complex number $c \in \mathbb{C}$, c^* represents the complex conjugate of c. A Hilbert space $\mathcal{H}$ is a complex vector space with an inner product. Two vectors $|v\rangle$ and $|w\rangle$ are *orthogonal* if the inner product of them equals 0, i.e., $\langle v \,|\, w \rangle = 0$. The *norm* of any vector $|v\rangle$ is

$$\||v\rangle\| = \sqrt{\langle v \,|\, v \rangle}. \tag{1}$$

A vector $|v\rangle$ is *normalized* if and only if the norm of it equals 1, i.e., $\||v\rangle\| = 1$. An *orthonormal basis* of a Hilbert space $\mathcal{H}$ is a basis $\{|i\rangle\}$ where every $|i\rangle$ is normalized and each pair of them is orthogonal.

Between two vector spaces V and W, a linear operator is a mapping $A : V \to W$, such that for any vectors $|v_i\rangle$ and scalars a_i the following condition is satisfied:

$$A\left(\sum_i a_i |v_i\rangle\right) = \sum_i a_i A |v_i\rangle. \tag{2}$$

When the vector space V equals W, the linear operator A is defined *on* the vector space V. If $\mathcal{L}(\mathcal{H})$ is the linear operator set on $\mathcal{H}$, for an operator $A \in \mathcal{L}(\mathcal{H})$,

- A is the *identity operator*, if $A|v\rangle = |v\rangle$ with any vector $|v\rangle \in \mathcal{H}$, denoted by $I_{\mathcal{H}}$;
- A is the *zero operator*, if $A|v\rangle = 0$ with any vector $|v\rangle \in \mathcal{H}$, denoted by $0_{\mathcal{H}}$;
- A is a *Hermitian operator* or *Hermitian*, if $A^\dagger = A$;
- A is a *projector*, if $A^2 = A$;
- A is a *normal operator*, if $A^\dagger A = A A^\dagger$;
- A is a *unitary operator*, if $A^\dagger A = A A^\dagger = I_{\mathcal{H}}$;
- A is a *positive operator*, if $\langle v|A|v\rangle \geq 0$ with any vector $|v\rangle \in \mathcal{H}$.

Normal operators have a well-known property called *spectral decomposition*, which indicates that any normal operator on $\mathcal{L}(\mathcal{H})$ is diagonalizable with an orthonormal basis for $\mathcal{H}$. In other words, a normal operator $A \in \mathcal{L}(\mathcal{H})$ satisfies

$$A = \sum_i \lambda_i |i\rangle\langle i|, \tag{3}$$

where the set $\{|i\rangle\}$ forms an orthonormal basis of $\mathcal{H}$, the set $\{\lambda_i\}$ includes the eigenvalues of A and $|i\rangle\langle i|$ is the projector associated with the relevant eigenspace of λ_i. The *trace* of A is written as $\mathrm{tr}(A)$ with $\mathrm{tr}(A) = \sum_i \langle i|A|i\rangle$, where the set $\{|i\rangle\}$ forms an orthonormal basis of $\mathcal{H}$. The trace operation is linear and cyclic.

For two Hilbert spaces $\mathcal{H}_1$ and $\mathcal{H}_2$, and any two vectors $|v\rangle \in \mathcal{H}_1$ and $|w\rangle \in \mathcal{H}_2$, the *tensor product* $\mathcal{H}_1 \otimes \mathcal{H}_2$ is a vector space made up of linear combinations of these vectors $|v\rangle \otimes |w\rangle$, abbreviated as $|vw\rangle$. $\mathcal{H}_1 \otimes \mathcal{H}_2$ is a Hilbert space and the inner product of this Hilbert space satisfies that for any vectors $|v_1\rangle, |v_2\rangle \in \mathcal{H}_1$ and $|w_1\rangle, |w_2\rangle \in \mathcal{H}_2$,

$$\langle v_1 \otimes w_1 | v_2 \otimes w_2 \rangle = \langle v_1 | v_2 \rangle_{\mathcal{H}_1} \langle w_1 | w_2 \rangle_{\mathcal{H}_2}. \tag{4}$$

For any $A \in \mathcal{L}(\mathcal{H}_1)$ and $B \in \mathcal{L}(\mathcal{H}_2)$, a linear operator $A \otimes B \in \mathcal{L}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ satisfies that for any vectors $|\psi\rangle \in \mathcal{H}_1$ and $|\phi\rangle \in \mathcal{H}_2$,

$$(A \otimes B)|\psi\phi\rangle = A|\psi\rangle \otimes B|\phi\rangle. \tag{5}$$

2.2. *Quantum mechanics*

An isolated quantum system is related to a Hilbert space named the *state space* of this system. The simplest quantum system is a two-dimensional state space, namely *qubit*. Let $|0\rangle$ and $|1\rangle$ be an orthogonal basis set of the state space, any state in this state space is represented by the linear combination $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, with two complex numbers α and β satisfying $|\alpha|^2 + |\beta|^2 = 1$. A *pure state* of any quantum system can be expressed as a normalized vector of that system state space. A *mixed state* of any quantum system is expressed as an *ensemble* $\{(p_i, |\phi_i\rangle)\}$, with pure states $|\phi_i\rangle$ and the corresponding probability values p_i. The *density operator* of the quantum system is a positive linear operator, which is denoted by $\rho = \sum_i p_i |\phi_i\rangle\langle\phi_i|$, with $\mathrm{tr}(\rho) = 1$. Mixed states can be described by density operators. The state space of any composite physical system is the tensor product of some state spaces of its components. The simplest example is an entangled state $|\Psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, called EPR state. Entanglement is a crucial aspect related to quantum computation that is unique in the quantum realm.

The evolution of any closed quantum system is expressed as a *unitary transformation*. If the quantum system has state $|\phi\rangle$ at time t and state $|\phi'\rangle$ at time t', for some unitary operator U, we have $|\phi'\rangle = U|\phi\rangle$. Unitary operators can be conveniently understood in the form of their matrix representations. Under matrix multiplication of a $m \times n$ complex unitary matrix U, the unitary operator is

regarded as a mapping of type $\mathbb{C}^n \to \mathbb{C}^m$. Since any unitary operator U satisfies $UU^\dagger = I$, the unitary transformation of a quantum state is reversible, i.e.,

$$U^\dagger U|\phi\rangle = I|\phi\rangle = |\phi\rangle. \tag{6}$$

A common model for quantum computation is *quantum circuit*, which is a unit that converts the initial state into the final state through some quantum operations. Quantum operations are described by using quantum gates. Depending on the number of involved qubits, quantum gates are classified as single-qubit or multi-qubit gates. Taking the Hadamard gate as an example, this single-qubit gate transforms the initial state $|0\rangle$ into $|+\rangle$ and $|1\rangle$ into $|-\rangle$. The most well-known multi-qubit gates are double-qubit gates. Taking the controlled-NOT gate as an example, if the control qubit is $|0\rangle$, it acts as the identity operator; otherwise, the gate performs bit flipping on the target qubit, converting the initial state $|0\rangle$ into $|1\rangle$ and $|1\rangle$ into $|0\rangle$. The matrix representations of Pauli gates I_2, X, Y, Z, Hadamard gate H, and controlled-NOT gate CX are shown as follows:

$$I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

A quantum *measurement* is expressed as a set of measurement operators $\{M_m\}$, with each index m corresponding to each measurement outcome. The measurement operators have the *completeness equation* $\sum_m M_m^\dagger M_m = I$. If a quantum system has state $|\phi\rangle$, after measurement, the probability of obtaining each outcome m is

$$p(m) = \langle\phi|M_m^\dagger M_m|\phi\rangle \tag{7}$$

and the output state is $\frac{M_m|\phi\rangle}{\sqrt{p(m)}}$. If the quantum system has mixed state ρ_1 at time t_1 and state ρ_2 at time t_2, after applying the unitary transformation U, we have $\rho_2 = U\rho_1 U^\dagger$. If a quantum system has a mixed state ρ, after the measurement $\{M_m\}$, the probability of obtaining each outcome m is

$$p(m) = \mathrm{tr}(M_m^\dagger M_m \rho) \tag{8}$$

and the output state is $\frac{M_m \rho M_m^\dagger}{p(m)}$.

3. Classical-Quantum Imperative Language

We formalize the syntax, operational and denotational semantics of a classical-quantum imperative language. The classical part of the language is commonly used,[33] including skip statements, classical assignment statements, conditional branch statements, and while-loop statements. The quantum part of the language

is based on similar constructs in the literature,[34] including quantum unitary transformation and quantum measurement. Since a quantum measurement produces a probability distribution of states, we add branching information to distinguish different reachable states. With this information, we can also prove the consistency of these two semantics. In order to describe classical-quantum programs more intuitively, we propose a distribution-based denotational semantics and relate it to the state-based denotational semantics.

3.1. *Classical-quantum states*

Besides the classical data types **Bool** for booleans and **Int** for integer numbers, we assume an additional type **Qbt** for qubits. We assume type **Cvar** for classical variables, written as lower-case letters, e.g., x and y. To generate them automatically in Coq, we let them be indexed by natural numbers. We assume type **Qvar** for quantum variables. We can also think of them as a quantum register $\overline{q}$, indexing each qubit with a natural number, so quantum variables are written as $q_1, q_2, \ldots, q_n$, with n being the number of involved qubits.[a] Even if **Cvar** and **Qvar** are countably infinite, any given program only mentions a finite number of variables.

A machine state contains the whole variable values at a certain time during program execution. Because the execution of a classical-quantum program involves classical and quantum information, we associate a program with two registers: one to store classical information and the other to store quantum information. Thus, we define the machine state as a pair $(\sigma,\ \rho)$, with a classical state σ as well as a quantum one ρ. A classical state is a mapping $\sigma : \mathbf{Cvar} \to \mathbb{Z}$ from classical variables to integer numbers. The classical state can be updated by changing the integer value corresponding to the classical variable. We use $\sigma[n/x]$ to stand for updating a classical state, which should satisfy

$$\sigma[n/x](y) = \begin{cases} \sigma(y) & \text{if } y \neq x, \\ n & \text{if } y = x. \end{cases} \tag{9}$$

For every quantum variable $q \in \mathbf{Qvar}$, we define the q-system state space as a two-dimensional Hilbert space $\mathcal{H}_q$. With $V \subseteq \mathbf{Qvar}$, we specify

$$\mathcal{H}_V = \otimes_{q \in V} \mathcal{H}_q, \tag{10}$$

which is the tensor product of the state spaces associated with the quantum variables in V. Partial density operators of the space $\mathcal{H}_V$ are considered as quantum states collected in the set $\mathcal{D}(\mathcal{H}_\mathcal{V})$. So we use a specific matrix to formalize a quantum state in Coq, and the default value for each quantum state is the zero state $|0\rangle^{\otimes n}\langle 0|^{\otimes n}$, with n being the number of involved qubits. When applying any quantum operator on a quantum state, we change the entire matrix to update this quantum state.

[a]To simplify the data structure in Coq, we omit q and directly use the corresponding natural number index to represent each qubit to be manipulated.

According to the above discussion, it is easy to formally define in Coq the types of classical states `CState`, quantum states `QState` and machine states `State`.

```
Parameter  n: nat.
Definition CState : Type := Cvar -> Z.
Definition QState : Type := Square (2^n).
Definition State := (CState * QState)%type.
```

The default values of states are as follows:

```
Definition empty_CState : CState := (fun _ => 0%Z).
Definition empty_QState : QState := Qint.
Definition empty_State : State := (empty_CState, empty_QState).
```

We now define the update operations for classical states `update_CState`, quantum states `update_QState` and machine states `update_State`.

```
Definition update_CState (cst : CState)(x0 : Cvar)(z : Z) : CState :=
  (fun x1 => if eqdec x0 x1 then z else cst x1).
Definition update_QState (qst : QState) (U : Square (2^n)) : QState :=
  suop U qst.
Notation "x '!->' v ',' cst" := (update_CState cst x v).
Notation " U '#' qst " := (update_QState qst U).
Definition update_State
  (st : State) (x : Cvar) (z : Z) (U : Square (2^n)) : State :=
    ((x !-> z , (fst st)) , (U # (snd st)))
```

In the definition of `update_QState`, the function `suop` is a super-operator that manipulates linear operators.

```
Definition suop (U : Matrix e f) : Matrix e e -> Matrix f f :=
  fun ρ => U × ρ × U†.
```

3.2. *Syntax*

We first consider the syntax of arithmetic and boolean expressions. Since only classical variables are involved here, their syntax is similar to that in classical imperative languages. Arithmetic expressions can be integers, classical variables, or expressions constructed by applying arithmetic operators to other arithmetic expressions. Boolean expressions can be boolean values, relational expressions, and expressions constructed by applying boolean operators to other boolean expressions. We assume type **Aexp** for arithmetic expressions, denoted by $a, a', \ldots$ and type **Bexp** for boolean expressions, denoted by $b, b', \ldots$. The construction rules of these two kinds of expressions can be described by the following syntax:

- For **Aexp**: $a ::= n \,|\, x \,|\, a + a' \,|\, a - a' \,|\, a \times a'$.
- For **Bexp**: $b ::= \mathbf{true} \,|\, \mathbf{false} \,|\, a = a' \,|\, a \leq a' \,|\, \neg b \,|\, b \wedge b'$.

In Coq, we define **Aexp** by $\mathbb{Z}$, **Cvar** and arithmetic operators $+, -, \times$, and define **Bexp** by the basic boolean values **true**, **false** and boolean operators $\neg, \wedge$. Especially, for any $a, a' \in$ **Aexp**, the relational expressions $a = a'$ and $a \leq a'$ are also boolean expressions.

Then we consider the syntax of commands. Commands are similar to those in classical imperative languages and can be skip statements, classical assignment statements, conditional branch statements, or while-loop statements. Additionally, there are two commands involving quantum data.[b] One is the quantum unitary transformation statement $U[\overline{q}]$, which represents the operation of applying unitary operator U on the quantum system $\overline{q}$. The other is the quantum measurement statement $x := M[\overline{q}]$, which represents the operation of performing a quantum measurement with a measurement operator M on $\overline{q}$ and assigning the measurement result to the classical variable x. We assume type **Com** for commands, denoted by $c, c', \ldots$. The construction rules of commands can be described by the following syntax.

- For **Com**: $c ::=$ **Skip** $\mid x := a | c; c' \mid$ **If** b **then** c **else** c'
  While b **do** $c \mid U[\overline{q}] \mid x := M[\overline{q}]$

The first five commands are commonly found in classical languages and we will not explain them in detail. In Coq, the unitary operator U should be defined by the users. For convenience, we only consider single-qubit operators and double-qubit controlled operators.[c] We need to specify the indices of qubits where the unitary transformation is applied and give in matrix form the associated unitary operator. In addition, we only consider projective measurements and measure one qubit at a time. We need to provide the index of the qubit to be measured and a classical variable to store the corresponding measurement outcome. Below is the definition of commands in Coq.

```
Inductive com : Type :=
  | CQSkip
  | CQAss (x: Cvar) (a : aexp)
  | CQSeq (c0 c1 : com)
  | CQIf (b : bexp) (c0 c1 : com)
  | CQWhile (b : bexp) (c : com)
  | Unit_SG (U : Matrix 2 2) (q : Qvar)
  | Unit_DB (U : Matrix 2 2) (q p : Qvar)
  | Meas (x: Cvar) (q : Qvar).
```

[b]Since we define the default value of a quantum state as the zero state, the quantum initialization command can be omitted.
[c]Single-qubit operators and the double-qubit Controlled-NOT operator are sufficient to simulate arbitrary quantum unitary transformation.[32]

3.3. *Small-step operational semantics*

Small-step operational semantics defines program execution step by step. A triple $\langle e, \sigma, \rho\rangle$, with an expression e and a state (σ, ρ), is referred to as an expression configuration. By utilizing an evaluation relation $\hookrightarrow$ between expression configurations, we use a syntax-oriented way to design the small-step operational semantics of expressions. The evaluation rules about integer variables, arithmetic expressions for multiplication operations, and boolean expressions in the form of $a_0 = a_1$ are shown in Fig. 1. The evaluation rules about other expressions are analogous. We only use the classical state to evaluate expressions, therefore the quantum state in the configuration can be ignored.

We define transitions between command configurations $\langle c, \sigma, \rho\rangle$ for command execution, with a command c and a state (σ, ρ). Since quantum measurements entail probability distributions of states, we write $\xrightarrow{l}$ for the transition relation, where the label l is a sequence of natural numbers to distinguish different reachable states. The transition rules about the small-step operational semantics are displayed in Fig. 2. The particular command **Nil** represents the normal program termination. The execution of a **While** command is defined in terms of a conditional command.

If the behavior of a command is deterministic, e.g., that of the **Skip** statement, classical assignments and quantum unitary transformations, the transitions between command configurations must be unique. In this case, the label l is the empty sequence ϵ. However, the behavior of quantum measurements is probabilistic rather than deterministic. In particular, after applying the quantum measurement M, defined by these measurement operators M_i, the initial state (σ, ρ) can become several states, with not only the classical part being the updated state $\sigma[i/x]$ but

$$\frac{}{\langle x, \sigma\rangle \hookrightarrow \langle \sigma(x), \sigma\rangle}$$

$$\frac{\langle a_0, \sigma\rangle \hookrightarrow \langle a_0', \sigma\rangle}{\langle a_0 \times a_1, \sigma\rangle \hookrightarrow \langle a_0' \times a_1, \sigma\rangle} \qquad \frac{\langle a_1, \sigma\rangle \hookrightarrow \langle a_1', \sigma\rangle}{\langle n \times a_1, \sigma\rangle \hookrightarrow \langle n \times a_1', \sigma\rangle}$$

$$\frac{}{\langle n \times m, \sigma\rangle \hookrightarrow \langle p, \sigma\rangle} \quad \text{if } p \text{ is the product of } n \text{ and } m.$$

$$\frac{\langle a_0, \sigma\rangle \hookrightarrow \langle a_0', \sigma\rangle}{\langle a_0 = a_1, \sigma\rangle \hookrightarrow \langle a_0' = a_1, \sigma\rangle} \qquad \frac{\langle a_1, \sigma\rangle \hookrightarrow \langle a_1', \sigma\rangle}{\langle a_0 = a_1, \sigma\rangle \hookrightarrow \langle n = a_1', \sigma\rangle}$$

$$\frac{}{\langle n = m, \sigma\rangle \hookrightarrow \langle \mathbf{true}, \sigma\rangle} \quad \text{if } n \text{ is equal to } m.$$

$$\frac{}{\langle n = m, \sigma\rangle \hookrightarrow \langle \mathbf{false}, \sigma\rangle} \quad \text{if } n \text{ is not equal to } m.$$

Fig. 1. The evaluation rules about some expressions.

$$\frac{}{\langle \mathbf{Skip}, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle \mathbf{Nil}, \sigma, \rho\rangle}$$

$$\frac{\langle a, \sigma\rangle \hookrightarrow \langle a', \sigma'\rangle}{\langle x := a, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle x := a', \sigma', \rho\rangle} \qquad \frac{}{\langle x := n, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle \mathbf{Nil}, \sigma[n/x], \rho\rangle}$$

$$\frac{\langle c_0, \sigma, \rho\rangle \xrightarrow{l} \langle c_0', \sigma', \rho'\rangle}{\langle c_0; c_1, \sigma, \rho\rangle \xrightarrow{l} \langle c_0'; c_1, \sigma', \rho'\rangle} \qquad \frac{}{\langle \mathbf{Nil}; c_1, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle c_1, \sigma, \rho\rangle}$$

$$\frac{\langle b, \sigma\rangle \hookrightarrow \langle b', \sigma'\rangle}{\langle \mathbf{If}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle \mathbf{If}\ b'\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma', \rho'\rangle}$$

$$\frac{}{\langle \mathbf{If\ true\ then}\ c_0\ \mathbf{else}\ c_1, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle c_0, \sigma, \rho\rangle}$$

$$\frac{}{\langle \mathbf{If\ false\ then}\ c_0\ \mathbf{else}\ c_1, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle c_1, \sigma, \rho\rangle}$$

$$\frac{}{\langle \mathbf{While}\ b\ \mathbf{do}\ c, \sigma, \rho\rangle \xrightarrow{\epsilon} \langle \mathbf{If}\ b\ \mathbf{then}\ (c; \mathbf{While}\ b\ \mathbf{do}\ c)\ \mathbf{else\ Skip}, \sigma, \rho\rangle}$$

$$\frac{}{\langle U[\overline{q}], \sigma, \rho\rangle \xrightarrow{\epsilon} \langle \mathbf{Nil}, \sigma, U\rho U^\dagger\rangle} \qquad \frac{M := \{M_i\}_{i\in I}}{\langle x := M[\overline{q}], \sigma, \rho\rangle \xrightarrow{i} \langle \mathbf{Nil}, \sigma[i/x], M_i\rho M_i^\dagger\rangle}$$

Fig. 2. Small-step operational semantics of commands.

also the quantum part being the updated quantum state $M_i\rho M_i^\dagger$. The label l is i for the i-th successor configuration.

We use the ternary inductive predicate `scstep` instead of a function to formalize the small-step operational semantics in Coq.[d] Below we display part of the definition of `scstep`, including the constructs for classical assignment, quantum unitary transformation and measurement.

```
Inductive scstep : (com * State) -> list Z -> (com * State) -> Prop :=
  | SCS_AssStep : forall st x a0 a1, sastep st a0 a1 ->
      scstep (x ::= a0, st) [] (x ::= a1, st)
  | SCS_Ass : forall st0 st1 x n,
      st1 = ((x !-> n, (fst st0)), snd st0) ->
      scstep (x ::= (ANum n), st0) [] (Skip, st1)
    ...
  | SCS_Unit_SG : forall st0 st1 U q,
      st1 = ((fst st0), ((u_1 U q) # (snd st0))) ->
```

[d]Technically, it is more convenient to define the reflexive and transitive closure of `scstep` in Sec. 3.5.

```
    scstep (Unit_SG U q, st0) [] (Skip, st1)
| SCS_Unit_DB : forall st0 st1 U p q,
    st1 = ((fst st0), ((u_2 U q p) # (snd st0))) ->
    scstep (Unit_DB U q p, st0) [] (Skip, st1)
| SCS_Meas : forall st0 st1 l x q,
    (st1 = ((x !-> 0, (fst st0)), ((Mea0 q) # (snd st0))) ∧ l = [0]) ∨
    (st1 = ((x !-> 1, (fst st0)), ((Mea1 q) # (snd st0))) ∧ l = [1]) ->
    scstep (Meas x q, st0) l (Skip, st1).
```

As mentioned at the end of Sec. 3.2, for unitary transformations, we only consider single-qubit unitary operators $u_1 = I_{2^q} \otimes U \otimes I_{2^{(n-q-1)}}$ and double-qubit controlled unitary operators

$$u_2 = I_{2^q} \otimes (|0\rangle\langle 0| \otimes I_{2^{(p-q-1)}} \otimes I_2 + |1\rangle\langle 1| \otimes I_{2^{(p-q-1)}} \otimes U) \otimes I_{2^{(n-p-1)}}, \quad (11)$$

where U is a specific matrix, such as Pauli operators X, Y and Z or Hadamard operator H. I_k is the k-dimensional identity operator. For measurements, we only use two one-qubit projective measurement operators $\texttt{Mea0} = I_{2^q} \otimes |0\rangle\langle 0| \otimes I_{2^{(n-q-1)}}$ and $\texttt{Mea1} = I_{2^q} \otimes |1\rangle\langle 1| \otimes I_{2^{(n-q-1)}}$, with n being the number of involved qubits and q or p being the qubit index on which the quantum operations are applied.

```
Definition u_1 (U: Matrix 2 2) (q: nat) : Matrix (2^n) (2^n) :=
  (I_{2^q} ⊗ U ⊗ I_{2^{n-p-1}}).
Definition u_2 (U: Matrix 2 2) (q p: nat): Matrix (2^n) (2^n) :=
  (I_{2^q} ⊗ |0⟩⟨0| ⊗ I_{2^{p-q-1}} ⊗ I_2 ⊗ I_{2^{n-p-1}}) .+
  (I_{2^q} ⊗ |1⟩⟨1| ⊗ I_{2^{p-q-1}} ⊗ U ⊗ I_{2^{n-p-1}}).
Definition Mea0 := u_1 |0⟩⟨0|.
Definition Mea1 := u_1 |1⟩⟨1|.
```

3.4. *State-based denotational semantics*

We introduce the state-based denotational semantics to represent the final result after program execution from the given start state.

With a command c and a state (σ, ρ), we use $[\![c]\!]_{(\sigma,\rho)}$ to stand for the interpretation of command c, as a set of triples (σ', ρ', l), indicating that the state (σ', ρ') is reachable from (σ, ρ) by following the label l. The denotational semantics of commands are shown in Fig. 3. We ignore the denotational semantics of expressions e like $[\![e]\!]_\sigma$ because they are nearly identical to those in the classical setting.

We use an extra **Abort** command to terminate the computation prematurely with no outcome and write $\emptyset$ for $[\![\textbf{Abort}]\!]_{(\sigma,\rho)}$. The command (**While** b **do** c) is interpreted as the union of all finite approximations of $[\![\textbf{While } b \textbf{ do } c]\!]_{(\sigma,\rho)}$. The n-th approximation is $[\![(\textbf{If } b \textbf{ then } c \textbf{ else Skip})^n; \textbf{If } b \textbf{ then Abort}]\!]_{(\sigma,\rho)}$, where c^n indicates that a command c is iteratively executed n times. Specifically, c^0 is **Skip**. For simplicity, we abbreviate (**If** b **then** c **else Skip**) as (**If** b **then** c).

Like the operational semantics, after a quantum measurement, a state may evolve into several states and the measurement results are given to the

$$
\begin{aligned}
[\![\mathbf{Skip}]\!]_{(\sigma,\rho)} &= \{(\sigma,\rho,\epsilon)\} \\
[\![\mathbf{Abort}]\!]_{(\sigma,\rho)} &= \emptyset \\
[\![x := a]\!]_{(\sigma,\rho)} &= \{(\sigma[[\![a]\!]_\sigma/x],\rho,\epsilon)\} \\
[\![c_0; c_1]\!]_{(\sigma,\rho)} &= \{(\sigma'',\rho'',l_0 l_1) \mid \exists l_0, l_1, \\
&\qquad (\sigma',\rho',l_0) \in [\![c_0]\!]_{(\sigma,\rho)} \text{ and } (\sigma'',\rho'',l_1) \in [\![c_1]\!]_{(\sigma',\rho')}\} \\
[\![\mathbf{If}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]_{(\sigma,\rho)} &= \begin{cases} [\![c_0]\!]_{(\sigma,\rho)} \text{ if } [\![b]\!]_\sigma = \mathbf{true} \\ [\![c_1]\!]_{(\sigma,\rho)} \text{ if } [\![b]\!]_\sigma = \mathbf{false} \end{cases} \\
[\![\mathbf{While}\ b\ \mathbf{do}\ c]\!]_{(\sigma,\rho)} &= \bigcup_{n\in\mathbb{N}} [\![(\mathbf{If}\ b\ \mathbf{then}\ c)^n;\ \mathbf{If}\ b\ \mathbf{then}\ \mathbf{Abort}]\!]_{(\sigma,\rho)} \\
[\![U[\overline{q}]]\!]_{(\sigma,\rho)} &= \{(\sigma, U\rho U^\dagger, \epsilon)\} \\
[\![x := M[\overline{q}]]\!]_{(\sigma,\rho)} &= \{(\sigma[i/x], M_i \rho M_i^\dagger, i) \mid i \in I\} \\
&\qquad \text{where } M := \{M_i\}_{i\in I}
\end{aligned}
$$

Fig. 3. State-based denotational semantics of commands.

corresponding classical variables. We also use different labels to mark different states. The labels for most commands are easy to understand because of their similarity to those in the operational semantics. Note that, for the sequential execution of two statements, the effect is the accumulation of each statement. In other words, if $(\sigma',\rho',l_0) \in [\![c_0]\!]_{(\sigma,\rho)}$ and $(\sigma'',\rho'',l_1) \in [\![c_1]\!]_{(\sigma',\rho')}$, then $(\sigma'',\rho'',l_0 l_1) \in [\![c_0; c_1]\!]_{(\sigma,\rho)}$.

```
Definition QuaRel.concat (RE0 RE1: State -> list Z -> State -> Prop):
  State -> list Z -> State -> Prop := fun st0 l2 st2 =>
    exists st1 l0 l1, RE0 st0 l0 st1 ∧ RE1 st1 l1 st2 ∧ l2 = l0 ++ l1.
```

In Coq, we first define a few useful ternary relations. For example, the relation `QuaRel.concat` given above formalizes the composition of two ternary relations used to describe sequential statements. Then we rely on them to formalize the state-based denotational semantics of commands as follows:

```
Fixpoint sceval (c : com): State -> list Z -> State -> Prop :=
  match c with
  | CQSkip =>  QuaRel.id
  | CQAss x E => fun st0 l st1 =>
    st1 = ((x !-> (aeval E st0), (fst st0)), snd st0) ∧ l = []
  | CQSeq c1 c2 => QuaRel.concat (sceval c1) (sceval c2)
    ...
```

```
| Unit_SG U q => fun st0 l st1 =>
  st1 = ((fst st0), ((u_1 U q) # (snd st0))) ∧ l = []
| Unit_DB U q p => fun st0 l st1 =>
  st1 = ((fst st0), ((u_2 U q p) # (snd st0))) ∧ l = []
| Meas x q => fun st0 l st1 =>
  (st1 = ((x !-> 0, (fst st0)), ((Mea0 q) # (snd st0))) ∧ l = [0]) ∨
  (st1 = ((x !-> 1, (fst st0)), ((Mea1 q) # (snd st0))) ∧ l = [1])
end.
```

3.5. *Semantic equivalence*

As previously mentioned, the denotational semantics is independent of the small-step operational one. Nevertheless, we are going to show that they are consistent.

In Sec. 3.3, we have defined single-step transitions. Now we are going to define a multi-step transition relation. We write $\langle c, \sigma, \rho\rangle \overset{l}{\Longrightarrow} \langle c', \sigma', \rho'\rangle$, supposing there exist a finite chain of configurations $\langle c_i, \sigma_i, \rho_i\rangle$ and labels l_i such that

$$\langle c, \sigma, \rho\rangle \xrightarrow{l_1} \langle c_1, \sigma_1, \rho_1\rangle \xrightarrow{l_2} \langle c_2, \sigma_2, \rho_2\rangle \xrightarrow{l_3} \cdots \xrightarrow{l_n} \langle c_n, \sigma_n, \rho_n\rangle$$

with $\langle c_n, \sigma_n, \rho_n\rangle = \langle c', \sigma', \rho'\rangle$, $l = l_1 l_2 \cdots l_n$. So the consistency of the denotational semantics and the multi-step operational one is established as follows.

Theorem 1. *For any command c and state (σ, ρ), there exist some l's such that*

$$[\![c]\!]_{(\sigma,\rho)} = \{(\sigma', \rho', l) \mid \langle c, \sigma, \rho\rangle \overset{l}{\Longrightarrow} \langle \mathbf{Nil}, \sigma', \rho'\rangle\}. \tag{12}$$

In Eq. (12), we keep the labels l that are associated with the final state reached by executing c from the state (σ, ρ). In Coq, we first inductively formalize the reflexive and transitive closure of ternary relations below.

```
Inductive refl_trans_clos (RE : com * State -> list Z ->
  com * State -> Prop) :
   com * State -> list Z -> com * State -> Prop :=
| rtc_step : forall cs0 l cs1, RE cs0 l cs1 -> refl_trans_clos
  RE cs0 l cs1
| rtc_refl : forall cs, refl_trans_clos RE cs [] cs
| rtc_trans : forall cs0 cs1 cs2 l0 l1,
  refl_trans_clos RE cs0 l0 cs1 -> refl_trans_clos RE cs1 l1 cs2 ->
  refl_trans_clos RE cs0 (l0 ++ l1) cs2.
```

Then we use the small-step transition relation `scstep` and its reflexive and transitive closure to formalize a multi-step execution of programs.

```
Definition mcstep: com * State -> list Z -> com * State -> Prop :=
  refl_trans_clos scstep.
```

Finally, we state the semantic equivalence and formally prove the consistency result.

```
Theorem oper_deno_equiv: forall c st0 st1 l,
  sceval c st0 l st1 <-> mcstep (c, st0) l (CQSkip, st1).
```

3.6. *Distribution-based denotational semantics*

In the state-based denotational semantics, we view a command as a transformer that changes an input machine state to a probability distribution of successor machine states. Alternatively, we can directly consider a command as a transformer between positive operator valued distributions. Let Γ denote the set of all classical states. Mathematically, a positive operator valued distribution (called *distribution state*) μ is a mapping $\Gamma \to \mathcal{D}(\mathcal{H}_{\mathcal{V}})$ with $\sum_{\sigma\in\lceil\mu\rceil} tr(\mu(\sigma)) \leq 1$, where $V \subseteq \mathbf{Qvar}$ and $\lceil\mu\rceil$ is the support of μ, representing $\{\sigma \in \Gamma \mid \mu(\sigma) \neq 0\}$. Sometimes we write μ in the form $\bigoplus_{i\in I}(\sigma_i, \rho_i)$ where $\mu(\sigma_i) = \rho_i$. We assume type **Dstate** for distribution states.

We indicate the interpretation of command c with distribution state μ by $\{\{c\}\}_\mu$. The distribution-based denotational semantics of commands is shown in Fig. 4. For the denotation of arithmetic and boolean expressions, since quantum states play no role, we omit them and write $\{\{a\}\}_{\sigma_i}$ and $\{\{b\}\}_{\sigma_i}$. Note that for classical assignments and quantum unitary transformations, similar assignments and transformations are applied to each branch of the distribution state. For quantum measurements, all reachable post-measurement states should be collected in the final distribution state. For the conditional statement **If** b **then** c_0 **else** c_1, we split the support of μ into two parts: the states that validate b are applied to c_0, those invalidate b are applied to c_1, and the corresponding results are combined together, which is very different from the denotational semantics in Sec. 3.4.

The following theorem relates the distribution-based denotational semantics to the state-based one.

$$
\begin{aligned}
\{\{\mathbf{Skip}\}\}_\mu &= \mu \\
\{\{\mathbf{Abort}\}\}_\mu &= \varepsilon \\
\{\{x := a\}\}_\mu &= \textstyle\bigoplus_{i\in I}(\sigma_i[\{\{a\}\}_{\sigma_i}/x], \rho_i) \\
\{\{c_0; c_1\}\}_\mu &= \{\{c_1\}\}_{\{\{c_0\}\}_\mu} \\
\{\{\mathbf{If}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\}\}_\mu &= \{\{c_0\}\}_{\bigoplus_i(\sigma_i,\rho_i)} \bigoplus \{\{c_1\}\}_{\bigoplus_j(\sigma_j,\rho_j)} \\
&\qquad \text{where } \{\{b\}\}_{\sigma_i} = \mathbf{true} \text{ and } \{\{b\}\}_{\sigma_j} = \mathbf{false} \\
\{\{\mathbf{While}\ b\ \mathbf{do}\ c\}\}_\mu &= \lim_{n\to\infty} \{\{(\mathbf{If}\ b\ \mathbf{then}\ c)^n;\ \mathbf{If}\ b\ \mathbf{then}\ \mathbf{Abort}\}\}_\mu \\
\{\{U[\overline{q}]\}\}_\mu &= \textstyle\bigoplus_{i\in I}(\sigma, U\rho_i U^\dagger) \\
\{\{x := M[\overline{q}]\}\}_\mu &= \textstyle\bigoplus_{i\in I}(\sigma[i/x], M_i\rho M_i^\dagger) \text{ where } M := \{M_i\}_{i\in I}
\end{aligned}
$$

Fig. 4. Distribution-based denotational semantics of commands, where $\mu = \bigoplus_{i\in I}(\sigma_i, \rho_i)$.

Theorem 2. *For any command c and two distribution states $\bigoplus_{i\in I}(\sigma_i,\rho_i)$ and $\bigoplus_{j\in J}(\sigma_j,\rho_j)$,*

$$\left(\{\{c\}\}_{\bigoplus_{i\in I}(\sigma_i,\rho_i)}=\bigoplus_{j\in J}(\sigma_j,\rho_j)\right)\Rightarrow\left(\forall\, j\in J,\exists\, l_j,\quad(\sigma_j,\rho_j,l_j)\in\bigcup_{i\in I}[\![c]\!]_{(\sigma_i,\rho_i)}\right). \tag{13}$$

In Coq, we represent a distribution in terms of a list.

```
Definition DState : Type := list State.
```

The update operation on the classical part of a distribution state is defined as follows; the update operation on the quantum part as well as on the whole distribution state is similar.

```
Definition DC (st : State) (x: Cvar) (a : aexp) : State :=
  ((x !-> (aeval a st), (fst st)), snd st).
Fixpoint disupc (dst : DState) (x: Cvar) (a : aexp) : DState :=
  match dst with
  | [] => []
  | st :: dst1 => DC st x a :: (disupc dst1 x a)
  end.
```

Unlike the `union` operation for the ternary relation in the state-based denotational semantics, the distribution-based denotational semantics uses the binary relation `merge` to collect all possible distribution states in the denotational semantics. The definitions of the two relations are as follows.

```
Definition union (RE1 RE2: A -> E -> B -> Prop):A -> E -> B -> Prop :=
  fun st0 l st1 => RE1 st0 l st1 ∨ RE2 st0 l st1.
Definition merge (RE1 RE2: DState -> DState -> Prop):
  DState -> DState -> Prop :=
  fun dst0 dst1 => exists d0 d1, RE1 dst0 d0 ∧ RE2 dst0 d1 ∧ dst1 =
  d0 ++ d1.
```

We formalize the distribution-based denotational semantics of commands as follows. The relation `merge` is an important ingredient of `if_sem` and `loop_sem`.

```
Fixpoint dceval (c : com): DState -> DState -> Prop :=
  match c with
  | CQSkip =>  BinRel.id
  | CQAss x E => fun dst0 dst1 => dst1= disupc dst0 x E
  | CQSeq c0 c1 => BinRel.concat (dceval c0) (dceval c1)
  | CQIf b c0 c1 => if_sem b (dceval c0) (dceval c1)
  | CQWhile b c0 c1 => loop_sem b (dceval c0) (dceval c1)
  | Unit_SG U q => fun dst0 dst1 => dst1 = disupq dst0 (u_1 U q)
  | Unit_DB U q p => fun dst0 dst1 => dst1 = disupq dst0 (u_2 U q p)
  | Meas x q => fun dst0 dst1 => dst1 =
    (disups dst0 x 0 (Mea0 q)) ++ (disups dst0 x 1 (Mea1 q))
  end.
```

We formalize the relationship between these two denotational semantics as follows.

```
Lemma EQU : forall c dst0 dst1, dceval c dst0 dst1 ->
  exists dsts: list (list ((list Z) * State)),
    Permutation (map snd (concat dsts)) dst1 ∧
    Forall2 (fun st1 sts => MNP (fst (split sts)) ∧
      forall l st2, sceval c st1 l st2 <-> In (l, st2) sts) dst0 dsts.
```

The functions `Permutation`, `List.concat` and `Forall2` are defined in the Coq standard library. The predicate `MNP` means that the elements of a list are not prefixed with each other, as defined below.

```
Inductive MNP : list (list X) -> Prop :=
  |MNP_nil : MNP []
  |MNP_cons : forall x l, Forall (NoPrefix x) l -> MNP l -> MNP (x::l).
```

4. Case Studies

4.1. *Teleportation*

Quantum teleportation[35] employs quantum entangled states to transmit an unknown quantum state using only classical channels to send classical information. Especially, the procedure applies various operations on the mixed state after two intermediate measurements.

The circuit diagram of quantum teleportation is displayed in Fig. 5. Let the participants be *Alice* and *Bob*, and quantum teleportation is described as follows.

(1) We first create an EPR state $|\beta_{00}\rangle_{q_2,q_3}$ shared by *Alice* and *Bob*, with q_2 assigned to *Alice* and q_3 assigned to *Bob*.
(2) Let *Alice* first apply the controlled-NOT operator on q_1 and q_2 and then apply the Hadamard operator on q_1.
(3) Then *Alice* measures q_1 and q_2 separately and sends the results x and y to *Bob*.
(4) After *Bob* has received the results x and y, he applies the corresponding Pauli operators to obtain the initial state on q_3.

We use our classical-quantum language to describe the quantum teleportation protocol and verify its correctness. The language description of this protocol is as

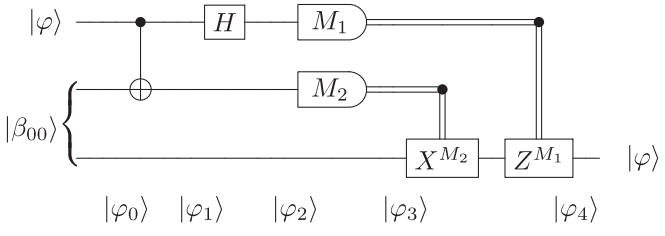

Fig. 5. Teleportation.

follows, which omits the preparation process for the EPR state $|\beta_{00}\rangle$.

$$\begin{aligned}\text{teleportation} \triangleq\ & CX\ 0\ 1; H\ 0;\\ & x := M\ 0; y := M\ 1;\\ & \textbf{If}\ (y = 1)\ \textbf{then}\ X2\ \textbf{else Skip};\\ & \textbf{If}\ (x = 1)\ \textbf{then}\ Z2\ \textbf{else Skip}.\end{aligned}$$

This piece of program is formalized in Coq as follows, which is self-explanatory.

```
Definition tele : com :=
  Unit_DB X 0 1 ;;
  Unit_SG H 0 ;;
  Meas x 0 ;;
  Meas y 1 ;;
  If (y == 1)
    Then (Unit_SG X 2)
    Else Skip EndIf ;;
  If (x == 1)
    Then (Unit_SG Z 2)
    Else Skip EndIf.
```

The input state $|\varphi_0\rangle$ of the above program is tensored by any single-qubit state $|\varphi\rangle$ and the ERP state $|\beta_{00}\rangle$. Here $|\varphi\rangle$ can be defined as $\alpha|0\rangle + \beta|1\rangle$, with two complex numbers α and β satisfying $|\alpha|^2 + |\beta|^2 = 1$. The state $|\beta_{00}\rangle$ stands for $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$.

After the command `tele` is executed, the transition between states is as follows.

```
Lemma tele_cor : dceval tele [(empty_CState, density φ0)]
  [((y !-> 1, x !-> 1), density (/ C2 .* (|11⟩ ⊗ φ)));
   ((y !-> 0, x !-> 1), density (/ C2 .* (|10⟩ ⊗ φ)));
   ((y !-> 1, x !-> 0), density (/ C2 .* (|01⟩ ⊗ φ)));
   ((y !-> 0, x !-> 0), density (/ C2 .* (|00⟩ ⊗ φ)))].
```

Using the distribution-based denotational semantics and some simplification strategies developed in the previous work,[36] the correctness of this transition is easy to prove. The third qubit of each output state in the distribution is $|\varphi\rangle$. Therefore, we achieve the goal of quantum teleportation using only classical channels to send classical information.

4.2. *Grover's search algorithm*

Search algorithms are widely used. Classical methods are inefficient in searching unstructured data, whose general complexity is $O(n)$, where n is the data size. Grover's quantum search algorithm can improve search efficiency, reducing the complexity to $O(\sqrt{n})$.

If the search space N is 2^n and the number of solutions for this search problem is M, Grover's search algorithm consists of one $H^{\otimes n}$ operation and $\sqrt{\frac{N}{M}}$ times of Grover iterations, which is illustrated in Fig. 6. The operation $H^{\otimes n}$ applies n

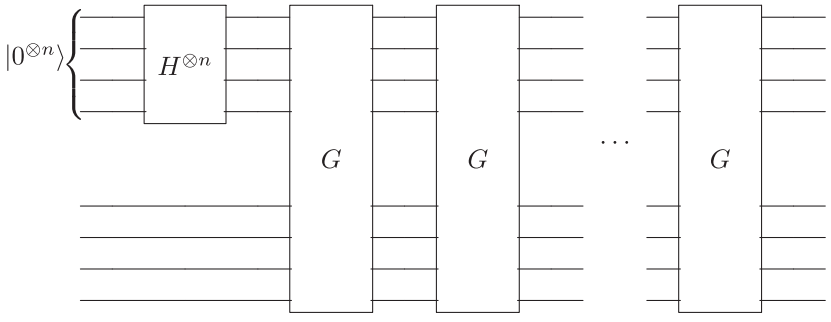

Fig. 6. Grover's search algorithm.

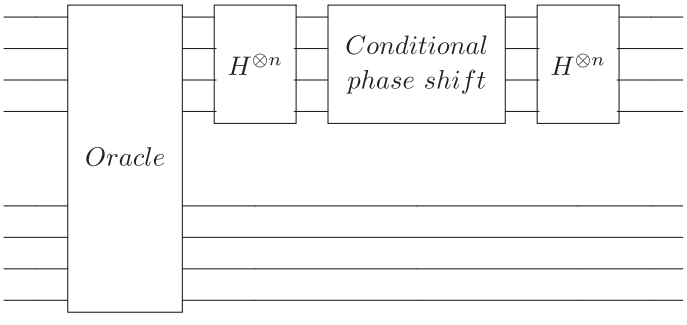

Fig. 7. Grover iteration.

Hadamard gates to n qubits $|0\rangle^{\otimes n}$, respectively, and then generates the maximum superposition state

$$|\phi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle. \tag{14}$$

The Grover iteration G can be divided into four steps as shown in Fig. 7.

(1) First apply the quantum oracle O to transform $|x\rangle$ into $(-1)^{f(x)}|x\rangle$;
(2) Then apply the operator $H^{\otimes n}$ on the first n qubits;
(3) Next, perform a conditional phase shift on $|x\rangle$, that is, when $|x\rangle$ equals $|0\rangle$, make no transformation; otherwise, convert $|x\rangle$ into $-|x\rangle$.
(4) Finally, apply the operator $H^{\otimes n}$ on the first n qubits again.

In the third step, the conditional phase shift is $2|0\rangle\langle 0| - I$. The last three steps can be combined as shown below:

$$H^{\otimes n} \times (2|0\rangle\langle 0| - I) \times H^{\otimes n} = 2|\phi\rangle\langle\phi| - I. \tag{15}$$

Thus, the Grover iteration G is $(2|\phi\rangle\langle\phi| - I) \times O$.

In summary, starting from a maximum superposition state, Grover's search algorithm performs t times of Grover iterations to find the solution to the search

problem. By using our language, we describe the whole algorithm as follows:

$$\begin{aligned}\text{Grover} \triangleq\ & \textbf{While}\ (x \leq (t-1)))\ \textbf{do}\\ & \text{Grover iteration};\\ & x := x+1.\end{aligned}$$

When the search space N is 4 and the search solution is 3, t equals 1. The unitary transformation of the Grover iteration is described below:

$$\begin{aligned}\text{Grover iteration} \triangleq\ & CZ\ 0\ 1;\ H\ 0;\ H\ 1;\ X\ 0;\ X\ 1;\\ & CZ\ 0\ 1;\ X\ 0;\ X\ 1;\ H\ 0;\ H\ 1.\end{aligned}$$

In this case, the algorithm is formalized as follows.

```
Definition gro : com :=
  While (x <= (t-1)) Do
    Unit_DB Z 0 1;;  Unit_SG H 0;;  Unit_SG H 1;;
    Unit_SG X 0;;  Unit_SG X 1;;
    Unit_DB Z 0 1;;  Unit_SG X 0;;  Unit_SG X 1;;
    Unit_SG H 0;;  Unit_SG H 1;;
    x := x + 1
  EndWhile.
```

After the command `gro` is executed, the transition between states is as follows.

```
Lemma gro_cor : dceval gro [((t !-> 1), density  (|+⟩ ⊗ |+⟩))]
  [((t !-> 1, x !-> 1), density |11⟩)].
```

Also using the distribution-based denotational semantics and some simplification strategies,[36] we prove the correctness of this lemma. The above case is the simplest of Grover's search algorithm, which only involves two qubits and only executes the loop body once. With the expansion of the search space, the number of loops will increase, and the quantum operation simulating Grover iteration will become more complex.

5. Related Work

Formal verification for quantum computation developed quickly, particularly in the use of proof assistants. We first discuss some relevant work on quantum circuit languages or compilation verification implemented in the proof assistant Coq. Paykin *et al.*[37] formalized a quantum language QWIRE in Coq. They defined the denotational semantics of programs and formally proved several quantum algorithms stated in that language.[38] Rand *et al.*[39] extended QWIRE to ReQWIRE and developed a verified compiler to convert classical circuits into reversible quantum ones. Hietala *et al.*[40] designed a quantum language SQIR in Coq and developed a quantum circuit optimizer VOQC. For both QWIRE and SQIR, the formalization and verification involving density matrices is based on an explicit matrix representation. Our language is based on the Dirac symbolic representation,[36] which improves the efficiency of the proof process as well as the readability of matrix representations.

In addition, we choose a looser notion of matrix equivalence, which means that two matrices are considered equivalent when they have equal components only in the desired dimensions. Therefore, the proof process can omit the side condition that the involved matrices must be well-formed.

The verification work mentioned above is based on the Coq standard library. There also exists some other work that is implemented by using some specific Coq math libraries. For example, Boender *et al.*[41] turned the generic model of quantum communication into constructive logic in Coq. They proposed a generic framework, based on the Coq repository C-CoRN,[42] to model and analyze some quantum protocols. Cano *et al.*[43] specifically designed a CoqEAL library[44] to develop efficient computer algebra programs with proofs of correctness. They used lists of lists to represent matrices for efficient matrix computations in Coq. Mahmoud and Felty[45] formally defined a quantum language Proto-Quipper in Coq and verified the type soundness property. In the Hybrid system,[46] they proposed a linear logical framework to express and infer Quipper's linear type system.[47] Zhou *et al.*[48] presented a program validator CoqQ for a simple imperative language in Coq. They formalized a program logic and proved its soundness in denotational semantics depending on the MathComp library.[49] Unlike our matrix library built on the Coq standard library, the linear and dependent types of the libraries involved in the aforementioned work are more sophisticated and require much more proof efforts.

Besides Coq, other theorem provers and verification tools can also analyze quantum circuits and programs. Let us review some verification work that adopts Isabelle, another popular theorem prover. Unruh[50] not only designed a relational quantum Hoare logic (QRHL) but also developed an Isabelle-based qrhl-tool, which is embedded in a language with rich types of quantum data. The important use of the qrhl-tool is to verify the safety of post-quantum cryptographic algorithms and protocols. Liu *et al.*[51] formally defined Ying's quantum Hoare logic[52] in Isabelle/HOL.[53] Their formalization contains a denotational semantics of a pure quantum programming language and a Hoare logic that is proved sound and complete for partial correctness with respect to the program semantics. The formalization uses the Jordan_Norm_Form (JNF) library[54] for matrices as well as the Deep_Learning library[55] for tensors. Bordg *et al.*[56] designed an Isabelle formalization to formally verify quantum programs. They not only formalized quantum circuits using the JNF library but also provided a fundamental encoding of the Dirac notation. However, Isabelle cannot flexibly customize high-level proof strategies, so its degree of automation is not as good as Coq.

We finally analyze some related work with other verification tools. Amy *et al.*[57] developed a reversible circuit compiler called ReVerC, which has been formally verified in F*.[58] This verification contained some unitary quantum circuits described by the quantum circuit language LIQ$Ui|\rangle$.[59] Beillahi *et al.*[60] validated quantum circuits with nearly 200 double-qubit operators in HOL Light. It depended on a set of rules regarding complex operations and linear operators that are verified in Mahmoud *et al.*'s formalization[61] for Hilbert spaces in HOL Light. Chareton

et al.[62] designed a validation framework QBRICKS in the deductive verification tool Why3.[63] They used the parametrized path-sums[64] representation to formalize quantum circuits. In the work mentioned above, a quantum state must be in vector form, hence merely pure states can be described. Furthermore, they only consider unitary operators but do not support quantum measurements. In contrast, we can define quantum states as density matrices and support quantum unitary transformations and quantum measurements.

All the languages mentioned above are quantum low-level circuit languages or pure quantum high-level languages which only contain quantum constructs. Our imperative language allows for interactions of classical and quantum information. Deng *et al.*[31] and Feng *et al.*[30] have proposed formal semantics and satisfaction-based or expectation-based Hoare logics of a programming language which also contains classical and quantum variables. But their verification is performed manually. To our best knowledge, we are the first to embed a classical-quantum language in a theorem prover.

6. Conclusion

We formalize the syntax, operational and denotational semantics of a classical-quantum imperative language. Due to the probability distributions of states generated by quantum measurements, we add branching information in our semantics to distinguish all different reachable states. Using this information, we can prove the consistency of these two semantics. In order to describe the program more intuitively, we further directly consider distribution-based denotational semantics and link it to state-based denotational semantics. Using distribution-based denotational semantics and some symbolic reasoning strategies can verify the correctness of quantum algorithms.

So far we have limited ourselves to operational semantics and denotational semantics. In future work, we intend to formally define suitable axiomatic semantics, or Hoare logics for classical-quantum languages in Coq, so to analyze quantum programs.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. Deng would like to acknowledge the support of the National Natural Science Foundation of China (Grant Nos. 62072176 and 61832015), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100), and Shanghai Trusted Industry Internet Software Collaborative Innovation Center.

ORCID

Wenjun Shi https://orcid.org/0009-0005-3110-7825
Qinxiang Cao https://orcid.org/0000-0002-5678-6538
Yuxin Deng https://orcid.org/0000-0003-0753-418X

References

1. P. Benioff, The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines, *J. Stat. Phys.* **22** (1980) 563–591.
2. P. W. Shor, Algorithms for quantum computation: Discrete logarithms and factoring, *Proc. 35th Annual Symp. Foundations of Computer Science* (IEEE, 1994), pp. 124–134.
3. S. J. Gay, R. Nagarajan and N. Papanikolaou, Probabilistic model-checking of quantum protocols, preprint (2005), arXiv:quant-ph/0504007.
4. S. J. Gay, R. Nagarajan and N. Papanikolaou, QMC: A model checker for quantum systems, *Proc. 20th Int. Conf. Computer Aided Verification* (Springer, Princeton, USA, 2008), pp. 543–547.
5. S. J. Gay, R. Nagarajan and N. Papanikolaou, Specification and verification of quantum protocols, *Semantic Techniques in Quantum Computation*, Vol. 1 (Cambridge University Press, 2010), pp. 414–472.
6. P. Baltazar, R. Chadha, P. Mateus and A. Sernadas, Towards model-checking quantum security protocols, *Proc. 1st Int. Conf. Quantum, Nano, and Micro Technologies* (IEEE, Guadeloupe, French Caribbean, 2007), pp. 14–14.
7. P. Baltazar, R. Chadha and P. Mateus, Quantum computation tree logic - model checking and complete calculus, *Int. J. Quantum Inf.* **6** (2008) 219–236.
8. T. A. S. Davidson, Formal verification techniques using quantum process calculus, PhD thesis, University of Warwick, Coventry, UK (2012).
9. T. A. S. Davidson, S. J. Gay, H. Mlnarik, R. Nagarajan and N. Papanikolaou, Model checking for communicating quantum processes, *Int. J. Unconv. Comput.* **8** (2012) 73–98.
10. E. Ardeshir-Larijani, S. J. Gay and R. Nagarajan, Equivalence checking of quantum protocols, *Proc. 19th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Rome, Italy, 2013), pp. 478–492.
11. E. Ardeshir-Larijani, S. J. Gay and R. Nagarajan, Verification of concurrent quantum protocols by equivalence checking, *Proc. 20th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Grenoble, France, 2014), pp. 500–514.
12. A. Kondacs and J. Watrous, On the power of quantum finite state automata, *Proc. 38th Annual Symposium on Foundations of Computer Science* (IEEE, Miami Beach, USA, 1997), pp. 66–75.
13. C. Moore and J. P. Crutchfield, Quantum automata and quantum grammars, *Theor. Comput. Sci.* **237** (1997) 275–306.
14. M. Ying, Y. Li, N. Yu and Y. Feng, Model-checking linear-time properties of quantum systems, *ACM Trans. Comput. Log.* **15** (2014) 1–31.
15. Y. Li and M. Ying, (Un)decidable problems about reachability of quantum systems, *Proc. 25th Int. Conf. Concurrency Theory* (Springer, Rome, Italy, 2014), pp. 482–496.
16. S. Seyedi and N. J. Navimipour, A space-efficient universal and multi-operative reversible gate design based on quantum-dots, *J. Circuits Syst. Comput.* **32** (2023) 1–10.
17. E. haq Shaik, B. R. Mannava, M. S. Shaik and N. Rangaswamy, QCA-based pulse/bit sequence detector using low quantum cost *D*-flip flop, *J. Circuits Syst. Comput.* **32** (2023) 1–21.
18. Q. Han, M. Shi and B. O. Mohammed, A new three-level design of nano-scale subtractor based on coulomb interaction of quantum dots, *J. Circuits Syst. Comput.* **31** (2022) 1–13.

19. G. Grössing and A. Zeilinger, Quantum cellular automata, *Complex Syst.* **2** (1988) 197–208.
20. N. Yu and M. Ying, Reachability and termination analysis of concurrent quantum programs, *Proc. 23rd Int. Conf. Concurrency Theory* (Springer, Newcastle upon Tyne, UK, 2012), pp. 69–83.
21. M. Ying, N. Yu, Y. Feng and R. Duan, Verification of quantum programs, *Sci. Comput. Program.* **78** (2013) 1679–1700.
22. S. Ying, Y. Feng, N. Yu and M. Ying, Reachability probabilities of quantum Markov chains, *Proc. 24th Int. Conf. Concurrency Theory* (Springer, Buenos Aires, Argentina, 2013), pp. 334–348.
23. S. Ying and M. Ying, Reachability analysis of quantum Markov decision processes, *Inf. Comput.* **263** (2018) 31–51.
24. J. Guan, Y. Feng and M. Ying, Decomposition of quantum Markov chains and its applications, *J. Comput. Syst. Sci.* **95** (2018) 55–68.
25. Y. Feng, N. Yu and M. Ying, Model checking quantum Markov chains, *J. Comput. Syst. Sci.* **79** (2013) 1181–1198.
26. Y. Feng, N. Yu and M. Ying, Reachability analysis of recursive quantum Markov chains, *Proc. 38th Int. Symp. Mathematical Foundations of Computer Science* (Springer, Klosterneuburg, Austria, 2013), pp. 385–396.
27. Y. Feng, E. M. Hahn, A. Turrini and L. Zhang, QPMC: A model checker for quantum programs and protocols, *Proc. 20th Int. Symp. Formal Methods* (Springer, Oslo, Norway, 2015), pp. 265–272.
28. Y. Feng, E. M. Hahn, A. Turrini and S. Ying, Model checking omega-regular properties for quantum Markov chains, *Proc. 28th Int. Conf. Concurrency Theory* (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Berlin, Germany, 2017), pp. 1–16.
29. Coq Development Team and Inria, The Coq proof assistant (2023), https://coq.inria.fr.
30. Y. Feng and M. Ying, Quantum Hoare logic with classical variables, *ACM Trans. Quantum Comput.* **2** (2021) 16–60.
31. Y. Deng and Y. Feng, Formal semantics of a classical-quantum language, *Theor. Comput. Sci.* **913** (2022) 73–93.
32. M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th anniversary edn. (Cambridge University Press, 2010).
33. G. Winskel, *The Formal Semantics of Programming Languages: An Introduction* (MIT Press, 1993).
34. P. Selinger, Towards a quantum programming language, *Math. Struct. Comput. Sci.* **14** (2004) 527–586.
35. C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres and W. K. Wootters, Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels, *Phys. Rev. Lett.* **70** (1993) 1895–1899.
36. W. Shi, Q. Cao, Y. Deng, H. Jiang and Y. Feng, Symbolic reasoning about quantum circuits in Coq, *J. Comput. Sci. Technol.* **36** (2021) 1291–1306.
37. J. Paykin, R. Rand and S. Zdancewic, QWIRE: A core language for quantum circuits, *Proc. 44th ACM SIGPLAN Symp. Principles of Programming Languages* (ACM, Paris, France, 2017), pp. 846–858.
38. R. Rand, J. Paykin and S. Zdancewic, QWIRE practice: Formal verification of quantum circuits in Coq, *Proc. 14th Int. Conf. Quantum Physics and Logic*, Nijmegen, The Netherlands, 3–7 July 2017, pp. 119–132.
39. R. Rand, J. Paykin, D. Lee and S. Zdancewic, ReQWIRE: Reasoning about reversible quantum circuits, *Proc. 15th Int. Conf. Quantum Physics and Logic*, Halifax, Canada, 3–7 June 2018, pp. 299–312.

40. K. Hietala, R. Rand, S. Hung, X. Wu and M. Hicks, A verified optimizer for quantum circuits, *Proc. ACM Program. Lang.* **5** (2021) 1–29.
41. J. Boender, F. Kammüller and R. Nagarajan, Formalization of quantum protocols using Coq, *Proc. 12th Int. Workshop on Quantum Physics and Logic* Oxford, UK, 15–17 July 2015, pp. 71–83.
42. L. Cruz-Filipe, H. Geuvers and F. Wiedijk, C-CoRN, the constructive Coq repository at Nijmegen, *Proc. 3th Int. Conf. Mathematical Knowledge Management* (Springer, Bialowieza, Poland, 2004), pp. 88–103.
43. G. Cano, C. Cohen, M. Dénès, A. Mörtberg and V. Siles, Formalized linear algebra over elementary divisor rings in Coq, *Log. Methods Comput. Sci.* **12** (2016), https://doi.org/10.2168/LMCS-12(2:7)2016.
44. G. Cano, C. Cohen, M. Dénès, A. Mörtberg and V. Siles, CoqEAL - the Coq effective algebra library (2023), https://github.com/coq-community/coqeal.
45. M. Y. Mahmoud and A. P. Felty, Formalization of metatheory of the Quipper quantum programming language in a linear logic, *J. Autom. Reason.* **63** (2019) 967–1002.
46. A. P. Felty and A. Momigliano, Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax, *J. Autom. Reason.* **48** (2012) 43–105.
47. A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger and B. Valiron, Quipper: a scalable quantum programming language, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, Seattle, USA, 2013), pp. 333–342.
48. L. Zhou, G. Barthe, P. Strub, J. Liu and M. Ying, CoqQ: Foundational verification of quantum programs, *Proc. ACM Program. Lang.* **7** (2023) 833–865.
49. Mathematical Components Team and Inria, Mathematical components (2023), https://github.com/math-comp/math-comp.
50. D. Unruh, Quantum relational Hoare logic, *Proc. ACM Program. Lang.* **3** (2019) 1–31.
51. J. Liu, B. Zhan, S. Wang, S. Ying, T. Liu, Y. Li, M. Ying and N. Zhan, Formal verification of quantum algorithms using quantum Hoare logic, *Proc. 31st Int. Conf. Computer Aided Verification* (Springer, New York, USA, 2019), pp. 187–207.
52. M. Ying, *Foundations of Quantum Programming* (Morgan Kaufmann Publishers, 2016).
53. T. Nipkow, L. C. Paulson and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic* (Springer, 2002).
54. R. Thiemann and A. Yamada, Formalizing Jordan normal forms in Isabelle/HOL, *Proc. 5th ACM SIGPLAN Conf. Certified Programs and Proofs* (ACM, Saint Petersburg, USA, 2016), pp. 88–99.
55. A. Bentkamp, J. C. Blanchette and D. Klakow, A formal proof of the expressiveness of deep learning, *J. Autom. Reason.* **63** (2019) 347–368.
56. A. Bordg, H. Lachnitt and Y. He, Certified quantum computation in Isabelle/HOL, *J. Autom. Reason.* **65** (2021) 691–709.
57. M. Amy, M. Roetteler and K. M. Svore, Verified compilation of space-efficient reversible circuits, *Proc. 29th Int. Conf. Computer Aided Verification* (Springer, Berlin, Heidelberg, 2017), pp. 3–21.
58. N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue and S. Z. Béguelin, Dependent types and multi-monadic effects in F*, *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages)* (ACM, Petersburg, USA, 2016), pp. 256–270.
59. D. Wecker and K. M. Svore, LIQ$|Ui\rangle$: A software design architecture and domain-specific language for quantum computing, preprint (2014), arXiv:quant-ph/1402.4467.

60. S. M. Beillahi, M. Y. Mahmoud and S. Tahar, A modeling and verification framework for optical quantum circuits, *Form. Asp. Comput.* **31** (2019) 321–351.
61. M. Y. Mahmoud, V. Aravantinos and S. Tahar, Formalization of infinite dimension linear spaces with application to quantum theory, *Proc. 5th International Symposium on NASA Formal Methods* (Springer, Moffett Field, USA, 2013), pp. 413–427.
62. C. Chareton, S. Bardin, F. Bobot, V. Perrelle and B. Valiron, An automated deductive verification framework for circuit-building quantum programs, *Proc. 30th Eur. Symp. Programming Languages and Systems* (Springer, Luxembourg City, Luxembourg, 2021) pp. 148–177.
63. J. Filliâtre and A. Paskevich, Why3 - where programs meet provers, *Proc. 22nd European Symposium on Programming Languages and Systems* (Springer, Rome, Italy, 2013), pp. 125–128.
64. M. Amy, Towards large-scale functional verification of universal quantum circuits, *Proc. 15th Int. Conf. Quantum Physics and Logic*, Halifax, Canada, 3–7 June 2018, pp. 1–21.