Qafny: A Quantum-Program Verifier

Liyi Li
Iowa State University, Ames, IA, USA

Mingwei Zhu
University of Maryland, College Park, MD, USA

Rance Cleaveland
University of Maryland, College Park, MD, USA

Alexander Nicolellis
Iowa State University, Ames, IA, USA

Yi Lee
University of Maryland, College Park, MD, USA

Le Chang
University of Maryland, College Park, MD, USA

Xiaodi Wu
University of Maryland, College Park, MD, USA

Abstract

Because of the probabilistic/nondeterministic behavior of quantum programs, it is highly advisable to verify them formally to ensure that they correctly implement their specifications. Formal verification, however, also traditionally requires significant effort. To address this challenge, we present Qafny, an automated proof system based on the program verifier Dafny and designed for verifying quantum programs. At its core, Qafny uses a type-guided quantum proof system that translates quantum operations to classical array operations modeled within a classical separation logic framework. We prove the soundness and completeness of our proof system and implement a prototype compiler that transforms Qafny programs and specifications into Dafny for automated verification purposes. We then illustrate the utility of Qafny's automated capabilities in efficiently verifying important quantum algorithms, including quantum-walk algorithms, Grover's algorithm, and Shor's algorithm.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Quantum information theory

Keywords and phrases Quantum Computing, Automated Verification, Separation Logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.24

Related Version *Extended Version*: `https://arxiv.org/abs/2211.06411` [24]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.12`
Software (Source Code): `https://zenodo.org/records/10529900`

Acknowledgements We thank Finn Voichick for his helpful comments and contributions during the work's development. This paper is dedicated to the memory of our dear co-author Rance Cleaveland.

1 Introduction

Quantum computers can be used to program substantially faster algorithms compared to those written for classical computers. For example, Shor's algorithm [48] can factor a number in polynomial time, which is not known to be polynomial-time-computable in the classical setting. Developing more and more comprehensive quantum programs and algorithms is essential for the continued practical development of quantum computing [11, 49]. Unfortunately, because quantum systems are inherently probabilistic and must obey quantum physics laws, traditional validation techniques based on run-time testing are virtually impossible to develop for large quantum algorithms. This leaves *formal methods* as a viable alternative for program checking, and yet these typically require a great effort; for example, four experienced researchers needed two years to formally verify Shor's algorithm [38]. To alleviate the effort required for formal verification, many frameworks have been proposed to verify quantum algorithms [26, 56, 3, 59, 20, 16] using interactive theorem

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 24; pp. 24:1–24:31
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

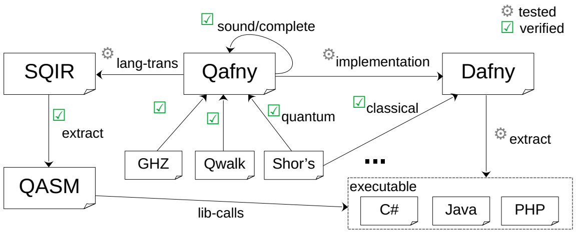


Figure 1 Dafny Development Stages/Key Aspects.

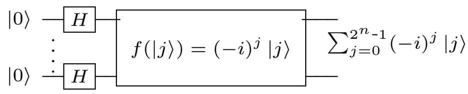


Figure 2 State Preparation Circuit.

provers, such as Isabelle, Coq, and Why3, by building quantum semantic interpretations and libraries. Some attempts towards proof automation have been made by creating new proof systems for quantum data structures such as Hilbert spaces; however, building and verifying quantum algorithms in these frameworks are still time-consuming and require great human effort. Meanwhile, automated verification is an active research field in classical computation with many proposed frameworks [17, 42, 27, 39, 37, 18, 50, 31, 44, 45, 21, 6] showing strong results in reducing programmer effort when verifying classical programs. None of the existing quantum verification frameworks utilize these classical verification infrastructures, however.

We present Qafny, a framework that enables programmers to develop and verify *quantum programs* based on quantum program semantics and classical automated verification infrastructure. It has several elements (Figure 1). The core is a strongly typed, flow-sensitive imperative quantum language Qafny, admitting a classical separation-logic-style proof system, in which users specify quantum programs and input the properties to be verified as pre- and post-conditions and loop invariants, such as GHZ, Quantum Walk, and Shor's algorithm. Qafny programs and specifications are verified via translation to a classical Hoare/separation logic framework implemented in the Dafny program verifier [22]. Qafny programs may also be compiled into quantum circuits and run on a quantum computer via the Qafny to SQIR and SQIR to OpenQASM 2.0 [7] compilers in our technical report (TR) [24] C.6. Quantum programs can be components of *hybrid classical-quantum* (HCQ) programs, so the compiled Qafny code can also be a library function called by an HCQ program defined. For example, one can extract the compiled Dafny program to a programming language, such as C#, PHP, and Java, and utilize quantum programs compiled from Qafny to OpenQASM [21].

A key component of the design of Qafny is to encode quantum states as array-like structures and quantum operations as aggregate array operations on the states. In Figure 2, a quantum state in superposition $\psi = \sum_{j=0}^{2^n-1} 1\,|j\rangle$ is prepared by applying a Hadamard gate to each qubit. Qafny treats ψ as an array containing 2^n elements, one for each indexed basis element in ψ. Each element is a pair of a complex and a natural number (computational basis, essentially a bitstring). For example, Bell pair $\frac{1}{\sqrt{2}}\,|00\rangle + \frac{1}{\sqrt{2}}\,|11\rangle$ can be thought of as a two-element array, with two pairs: $(\frac{1}{\sqrt{2}}, |00\rangle)$ and $(\frac{1}{\sqrt{2}}, |11\rangle)$, where the first one is a complex number and the second one is a bitstring that can be represented as a natural number. Applying a quantum oracle $(f(|j\rangle) = (-i)^j\,|j\rangle)$ on ψ, which evolves each indexed element $1\,|j\rangle$ to $(-i)^j\,|j\rangle$, is similar to an array map function that applies $f'(\alpha_j, j) = ((-1)^j \alpha_j, j)$ to each element j in the 2^n-array. The design and analysis of many quantum algorithms leverage the representation of different groups of qubits in terms of classical arrays [13, 35, 47]. Besides the opportunities for automated reasoning provided by representing quantum states as arrays, Qafny also uses language abstractions such as quantum conditionals and loops, which generalize quantum controlled gates to enable local reasoning in the presence of quantum entanglement. In the prior works above, the usual approach in reasoning about quantum controlled gates such as `CNOT` and controlled-U gates is to transform these operations into a monolithic representation, such as a unitary matrix, not scaling up well for automated verification, because the relations among different entries, representing inductive relation among program constructs, are largely omitted. In Qafny, reasoning about comprehensive constructs, such as controlled gates, amount to building a structural inductive relation among different parts, such as a quantum conditional and its subparts, through deliberately designed proof rules in Sections 3.4 and 4.4; such design permits automated local reasoning.

These designs pose several challenges: 1) quantum operations can be performed on any qubit positions, e.g., f above could apply on arbitrary bits in every bitstring j; 2) performing automated local reasoning requires one to know which states and qubits can be excluded locally, but qubits can form entangled groups that are typically viewed as not separable; and 3) the Qafny proof system should obey quantum physical laws, such as no-cloning and no quantum observer effects. To address these problems, we first introduce different types of quantum-state representations and special data structures (*loci* in Section 3) to partition qubits into disjoint entanglement groups for local reasoning. We then combine a flow-sensitive type system (Section 4.3) with our proof system, capable of 1) statically identifying the quantum state types and tracking entanglement group transformations and 2) performing type-guided quantum-state rewrites in the assertions and automated local operation reasoning on canonicalized quantum states, without violating quantum laws.

The paper's contributions are listed as follows.

- We present the Qafny language, including a big-step semantics and flow-sensitive type system, which provides a simple way of enforcing quantum program properties, such as no-cloning and no observer breakdown. We also prove type soundness for Qafny in Coq.
- The Qafny type-guided proof system permits classical-array-operation views of quantum operations and captures the inductive behaviors of quantum conditionals and loops. Soundness and relative completeness are also proved in Coq. To the best of our knowledge, Qafny provides the first proof-rule definitions for quantum conditionals and for-loops.
- We exhibit a prototype Qafny to Dafny compiler as evidence of connecting quantum-program verification to classical Hoare/separation logic. We verify a number of quantum algorithms (Figure 16) with a high degree of automation. Sections 5.2 and 7 compares proof automation in Qafny with other frameworks.
- We faithfully implement several algorithms, such as GHZ, Shor's, and quantum walk, as case studies in tje paper to demonstrate how Qafny can help programmers to efficiently verify quantum-algorithm implementations. The program operations of these examples are a high-level abstraction of the algorithms' quantum circuit-based description, while the Qafny program state specifications are directly based on the algorithms' state representations based on Dirac notations.

2 Background

Here, we provide background information on quantum computing.

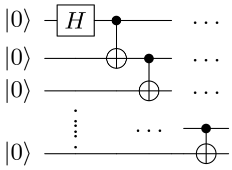
Quantum Value States. A quantum value state[1] consists of one or more quantum bits (*qubits*), which can be expressed as a two-dimensional vector $\binom{\alpha}{\beta}$ where the *amplitudes* α and β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. We frequently write the qubit vector

[1] Most literature mentioned Quantum value states as *quantum states*. Here, we refer to them as quantum value states or quantum values to avoid confusion between program and quantum states.

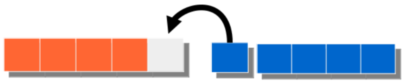
as $\alpha |0\rangle + \beta |1\rangle$ (the Dirac notation [8]), where $|0\rangle = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ and $|1\rangle = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$ are *computational basis-kets*. When both α and β are non-zero, we can think of the qubit being "both 0 and 1 at once," a.k.a. in a *superposition* [35], e.g., $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ represents a superposition of $|0\rangle$ and $|1\rangle$. Larger quantum values can be formed by composing smaller ones with the *tensor product* ($\otimes$) from linear algebra, e.g., the two-qubit value $|0\rangle \otimes |1\rangle$ (also written as $|01\rangle$) corresponds to vector $[\ 0\ 1\ 0\ 0\ ]^T$. However, many multi-qubit values cannot be *separated* and expressed as the tensor product of smaller values; such inseparable value states are called *entangled*, e.g. $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*, which can be rewritten to $\sum_{d=0}^{1} \frac{1}{\sqrt{2}}|dd\rangle$, where dd is a bit string consisting of two bits, each of which must be the same value (i.e., $d = 0$ or $d = 1$). Each term $\frac{1}{\sqrt{2}}|dd\rangle$ is named a *basis-ket* [35], consisting an amplitude ($\frac{1}{\sqrt{2}}$) and a basis vector $|dd\rangle$.

Quantum Computation and Measurement. Computation on a quantum value consists of a series of *quantum operations*, each acting on a subset of qubits in the quantum value. In the standard form, quantum computations are expressed as *circuits*, as in Figure 3a, which depicts a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [12] – an n-qubit entangled value of the form: $|\text{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$, where $|d\rangle^{\otimes n} = \bigotimes_{d=0}^{n-1} |d\rangle$. In these circuits, each horizontal wire represents a qubit, and boxes on these wires indicate quantum operations, or *gates*. The circuit in Figure 3a uses n qubits and applies n gates: a *Hadamard* (`H`) gate and $n-1$ *controlled-not* (`CNOT`) gates. Applying a gate to a quantum value *evolves* it. Its traditional semantics is expressed by multiplying the value's vector form by the gate's corresponding matrix representation: n-qubit gates are 2^n-by-2^n matrices. Except for measurement gates, a gate's matrix must be *unitary* and thus preserve appropriate invariants of quantum values' amplitudes. A *measurement* operation extracts classical information from a quantum value. It collapses the value to a basis state with a probability related to the value's amplitudes (*measurement probability*), e.g., measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ collapses the value to $|0\rangle$ with probability $\frac{1}{2}$, and likewise for $|1\rangle$, returning classical value 0 or 1, respectively. A more general form of quantum measurement is *partial measurement*, which measures a subset of qubits in a qubit array; such operations often have simultaneity effects due to entanglement, *i.e.,* in a Bell pair $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, measuring one qubit guarantees the same outcome for the other – if the first bit is measured as 0, the second bit is too.

Quantum Conditionals. Controlled quantum gates, such as controlled-not gates (`CNOT`), can be thought of as quantum versions of classical operations, where we view a quantum value as an array of basis-kets and apply an array map operation of the classical operation to every basis-ket. This is evident when using Dirac notation. For example, in preparing a two-qubit GHZ state (Figure 3a, also a Bell pair) for qubit array x, the `H` gate evolves the value to $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$ (same as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle$). The quantum conditional maps the classical conditional `if` $(x[0])$ $\{x[1] \leftarrow x[1] + 1\}$ onto the two basis-kets, where the operation $x[1] + 1$ acts as a modulo 2 addition to flip $x[1]$'s bit. Here, we do not flip the $x[1]$ position in the first basis-ket ($\frac{1}{\sqrt{2}}|00\rangle$) due to $x[0] = 0$, and we flip $x[1]$ in the second basis-ket because of $x[0] = 1$. Such behaviors can be generalized to other controlled gates, such as the controlled-U gate appearing in Shor's algorithm (Figure 6), where U refers to a modulo-multiplication operation. The controlled nodes (Boolean guards) in these quantum conditionals can also be generalized to other types of Boolean expressions, e.g., it can be a quantum inequality ($(\kappa < n)$ @ $x[i]$) that compares every basis vector of qubit array κ's value state with the number n and stores the result in qubit $x[i]$, and the controlled node queries $x[i]$ to determine if the conditional body is executed, more in Sections 4 and 6.2.



(a) GHZ Circuit.



(b) GHZ For-loop Analogy.

1 $\{\ x[0,n) \mapsto |\overline{0}\rangle\ \}$

2 $x[0]$ ← `H;`

3 $\{\ x[0,1) \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) * x[1,n) \mapsto |\overline{0}\rangle\ \}$

4 `for` $j \in [1,n)$ `&&` $x[j-1]$

5 $\quad \{\ x[0,j) \mapsto \sum_{d=0}^{1} \frac{1}{\sqrt{2}} |\overline{d}\rangle * x[j,n) \mapsto |\overline{0}\rangle * j \leq n\ \}$

6 $\quad x[j]$ ← $x[j]$ `+ 1;`

7 $\{\ x[0,n) \mapsto \sum_{d=0}^{1} \frac{1}{\sqrt{2}} |\overline{d}\rangle * x[n,n) \mapsto |\overline{0}\rangle\ \}$

8 $\{\ x[0,n) \mapsto \sum_{d=0}^{1} \frac{1}{\sqrt{2}} |\overline{d}\rangle\ \}$

(c) Qafny GHZ Program and Proof.

Figure 3 GHZ Examples. $x[t_1,t_2) \mapsto |\overline{0}\rangle$ means $\overline{0}$ is a bitstring of length t_2-t_1. $x[t_1,t_2) \mapsto \sum_{d=0}^{1} |\overline{d}\rangle$ means $\overline{d}$ is a bitstring of length t_2-t_1 and $d \in [0,1]$ is a bit. $*$ is the separation conjunction. In (c), black parts are Qafny programs, while blue and gray parts are Qafny state predicates.

Quantum Oracles. Quantum algorithms manipulate input information encoded in "oracles," which are callable black-box circuits. Quantum oracles are usually quantum-reversible implementations of classical operations, especially arithmetic operations. Their behavior is defined in terms of transitions between single basis-kets. We can infer its global state behavior based on the single basis-ket behavior through the quantum summation formula below. This resembles an array map operation in Figure 2. oqasm [23] is a language that permits the definitions of quantum oracles with efficient verification and testing facilities using the summation formula.

$$\frac{\forall j.\, |x_j\rangle \longrightarrow f(|x_j\rangle)}{\Sigma_j \alpha_j\, |x_j\rangle \longrightarrow \Sigma_j \alpha_j f(|x_j\rangle)}$$

No Cloning and Observer Effect. The *no-cloning theorem* suggests no general way of copying a quantum value. In quantum circuits, this is related to ensuring the reversible property of unitary gate applications. For example, the Boolean guard and body of a quantum conditional cannot refer to the same qubits, e.g., `if` $(x[1])$ $\{x[1] \leftarrow x[1] + 1\}$ violates the property as $x[1]$ is mentioned in the guard and body. The quantum *observer effect* refers to leaking information from a quantum value state. If a quantum conditional body contains a measurement or classical variable updates, the quantum system breaks down due to the observer effect. Qafny enforces no cloning and no observer breakdown through the syntax and flow-sensitive type system.

3 Qafny Design Principles: Locus, Type, and State

Here, we show the Qafny fundamental design principles for quantum program verification. We use the GHZ example in Figure 3a to highlight these principles, with a proof outline in Figure 3c; x is initialized to an n-qubit `Nor` typed value $|\overline{0}\rangle$ (n number of $|0\rangle$ qubits). After preparing a superposition ($x[0] \leftarrow$ `H`) for a single qubit $x[0]$ in line 2, we execute a *quantum for-loop* that entangles each pair of adjacent qubits in x to prepare the GHZ state. We can unroll each iteration of the loop as a quantum conditional (`if` $(x[j-1])$ $\{x[j] \leftarrow x[j] + 1\}$). When verifying the program in Qafny, it is only needed to provide the program and specifications in blue, with the grayed out parts automatically inferred. We show critical features in our type-guided proof system, making the above verification largely automatic.

3.1 Loci, Types, and States

Figure 4 shows Qafny *loci*, types, and states, which are used for tracking possibly entangled qubits. In GHZ (Figure 3c), each loop step in lines 4-6 entangles the qubit $x[j]$ with $x[0, j)$, i.e., the entangled qubit group is expanded from $x[0, j)$ to $x[0, j+1)$. However, the entanglement here is implicit: the program syntax does not directly tell if $x[j]$ is entangled with $x[0, j)$ but relies on an analysis to resolve the entanglement scopes, which is captured by introducing

1) *loci* (κ) to group possibly entangled qubits,
2) standard *kind environments* (Ω) to record variable *kinds* (explained below), and
3) *locus type environments* (σ) to keep track of both loci and their quantum state types.

Qafny variables may represent one of three *kinds* [2] of values (Figure 4). `C` and `M` kinds are scalars; the former is an integer[3], and the latter is a measurement outcome (r, n) where r is the probability of outcome n. `Q` m kind variables represent a physical m-length qubit array conceptually living in a heap. For simplicity, we assume no aliasing in variable names, no overlapping between qubit arrays referred to by any two different variables, and scalar and qubit array variables are always distinct. Quantum values are categorized into three different types: `Nor`, `Had` and `EN`. A *normal* value (`Nor`) is an array (tensor product) of single-qubit values $|0\rangle$ or $|1\rangle$. Sometimes, a (`Nor`)-typed value is associated with an amplitude z, representing an intermediate partial program state; an example is in Section 6.1. A *Hadamard* (`Had`) typed value represents a collection of qubits in superposition but not entangled, i.e., an n-qubit array $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(r_0)|1\rangle) \otimes ... \otimes \frac{1}{\sqrt{2}}(|0\rangle + \alpha(r_{n-1})|1\rangle)$, can be encoded as $\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + \alpha(r_j)|1\rangle)$, with $\alpha(r_j) = e^{2\pi i r_j}$ ($r_j \in \mathbb{R}$) being the *local phase*, a special amplitude whose norm is 1, i.e., $|\alpha(r_j)| = 1$. The most general form of n-qubit values is the *entanglement* (`EN`) typed value, consisting of a linear combination (represented as an array) of basis-kets, as $\sum_{j=0}^{m} z_j \beta_j \eta_j$, where m is the number of elements in the array. In Qafny, we *extend* traditional basis-ket structures in the Dirac notation to be the above form, so each basis-ket of the above value contains not only an amplitude z_j and a basis β_j but also a frozen basis stack η_j, storing bases not directly involved in the current computation. Here, β_j can always be represented as a single $|c_j\rangle$ by the equation in Figure 4. Every β_j in the array has the same cardinality, e.g., if $|c_0| = n$ ($\beta_0 = |c_0\rangle$), then $|c_i| = n$ ($\beta_j = |c_j\rangle$) for all j.

A Qafny quantum state (φ), representing a quantum heap, maps *loci* to *quantum values*. Loci in a heap φ *partition* it into regions that contain possibly entangled qubits, with the guarantee that cross-locus qubits are not entangled. Each locus is a list of *disjoint ranges* (s), each represented by $x[n, m)$ – an in-place array slice selected from n to m (exclusive) in a physical qubit array x (always being `Q` kind). Ranges in a locus are pairwise disjoint, written as $s_1 \uplus s_2$. For conciseness, we abbreviate a singleton range $x[j, j+1)$ as $x[j]$. At the type level, we maintain a locus type environment (σ) mapping loci to quantum types: any quantum state φ always has an entry in σ, guaranteeing that $\mathtt{dom}(\varphi) = \mathtt{dom}(\sigma)$, i.e., loci mentioned in φ and σ are the same. Locus type environments are stateful, i.e., a statement that starts with the environment σ could end up with a different one σ' because a locus could be modified during the execution. In addition to the locus type environment, we keep a kind environment between variables and their kinds. However, it is scoped and immutable as the kind of any scoped variable does not change.

[2] `C` and `M` kinds are also used as context modes in type checking. See Figure 9.
[3] Any classical values are permitted in our implementation. For simplicity, we only consider integers here.

Basic Terms:

Nat. Num	$m, n \in \mathbb{N}$		Real	$r \in \mathbb{R}$		Amplitude	$z \in \mathbb{C}$		Phase	$\alpha(r)$	$::= e^{2\pi i r}$
Variable	x, y		Bit	$d ::= 0 \mid 1$		Bitstring	$c \in d^{+}$		Basis Vector	β	$::= (\lvert c\rangle)^{*}$

Modes, Kinds, Types, and Classical/Quantum Values:

Mode	g	$::=$	`C`	$\mid$	`M`		
Classical Scalar Value	v	$::=$	n	$\mid$	(r, n)		
Kind	g_k	$::=$	g	$\mid$	`Q` n		
Frozen Basis Stack	γ	$::=$	$(\!\lvert \beta \rvert\!)$				
Full Basis Vector	η	$::=$	$\beta\gamma$				
Basic Ket	w	$::=$	$z\eta$				
Quantum Type	τ	$::=$	`Nor`	$\mid$	`Had`	$\mid$	`EN`
Quantum Value (Forms)	q	$::=$	w	$\mid$	$\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (\lvert 0\rangle + \alpha(r_j)\lvert 1\rangle)$	$\mid$	$\sum_{j=0}^{m} w_j$

Quantum Loci, Environment, and States

Qubit Array Range	s	$::=$	$x[n, m)$		
Locus	κ	$::=$	$\overline{s}$	concatenated op	$\boxplus$
Kind Environment	Ω	$::=$	$x \to g_k$		
Type Environment	σ	$::=$	$\overline{\kappa : \tau}$	concatenated op	$\uplus$
Quantum State (Heap)	φ	$::=$	$\overline{\kappa : q}$	concatenated op	$\uplus$

Syntax Abbreviations and Basis/Locus Equations

$\sum_{j=0}^{0} w_j \simeq w_0 \qquad \sum_{j=0}^{m} w_j \simeq \sum_j w_j \qquad 1\gamma \simeq \gamma \qquad z\beta(\!\lvert \emptyset \rvert\!) \simeq z\beta \qquad z\beta(\!\lvert \beta' \rvert\!) \simeq z\beta\widehat{\beta'}$

$\lvert c_1\rangle \lvert c_2\rangle \equiv \lvert c_1 c_2\rangle \qquad x[n, n) \equiv \emptyset \qquad \emptyset \boxplus \kappa \equiv \kappa \qquad x[n, m) \boxplus \kappa \equiv x[n, j) \boxplus x[j, m) \boxplus \kappa$ `where` $n \le j \le m$

Figure 4 QAFNY element syntax. Each range $x[n, m)$ in a locus represents the number range $[n, m)$ in physical qubit array x. Loci are finite lists, while type environments and states are finite sets. The operations after "concatenated op" are concatenations for loci, type environments, and quantum states.

On the bottom of Figure 4, we show abbreviations ($\simeq$) rules for presentation purposes; $A \simeq B$ means we write B to mean A. The left-most rule shows that `Nor` typed value is a singleton `EN` typed array; see the type-guided state rewrites in Section 3.3. We can also omit the $(\!\lvert\,\rvert\!)$ in a basis-ket presentation and color the basis stack with a hat sign $\widehat{\ }$, e.g., $\frac{1}{\sqrt{2}}\lvert 0\rangle \widehat{\lvert 1\rangle}$ means $\frac{1}{\sqrt{2}}\lvert 0\rangle (\!\lvert\, \lvert 1\rangle \,\rvert\!)$; additionally, $\frac{1}{\sqrt{2}}\lvert 0\rangle \lvert \overline{1}\rangle$ means $\frac{1}{\sqrt{2}}\lvert 0\rangle \lvert \overline{1}\rangle (\!\lvert \emptyset \rvert\!)$. Below the $\simeq$ rules in the figure, we present structural equations ($\equiv$) among bases and loci: 1) the locus concatenation $\boxplus$ holds identity and associativity equational properties; 2) a range ($x[n, n)$) containing 0 qubit is empty; and 3) it is free to split a range ($x[n, m)$) into two ($x[n, j)$ and $x[j, m)$), preserving the disjointness of $\boxplus$.

3.2 Simultaneity for Tracking Qubit Positions and Entanglement Scopes

QAFNY uses locus transformations, captured by the type inference on program operations, to track entanglement scopes. Figure 5 describes the automated proof steps for verifying a loop-step in Figure 3c. The bottom pre-condition contains the quantum values for the two disjoint loci $x[0, j)$ and $x[j, n)$. The quantum conditional's Boolean guard and body are applied to the qubits $x[j\text{-}1]$ and $x[j]$, respectively, appearing in the above two loci. The application entangles $x[j]$ with the locus $x[0, j)$ and transforms the locus to be $x[0, j\text{+}1)$, by appending $x[j]$ to the end of $x[0, j)$. The append, as the first (bottom) proof step in Figure 5, happens *automatically* through rewrites guided by the locus transformations in the type environment σ associated with each proof triple. After the rewrites, we preserve the property of no entangled cross-locus qubits.

In the above example, the static rewrites of a locus in a type environment simultaneously gear and change the rewrites of the locus value form in the associated state. We call this manipulation mechanism *simultaneity*. As shown in Section 1, quantum operations can apply on arbitrary qubit positions, which might seriously harm proof automation, based on previous experiments [16, 3] (Section 5.2), even if they tried hard for automation tactics. It is necessary to statically track qubit positions to permit the canonicalization of quantum state rewrites, allowing a uniform way of defining proof rules for operations.

$$\frac{}{\Omega;\{\kappa_2:\texttt{EN}\}\vdash_{\texttt{M}}\left\{\kappa_2\mapsto\frac{1}{\sqrt{2}}\,|0\rangle\,|\overline{1}\rangle\,\widehat{|1\rangle}\right\}\,e\,\left\{\kappa_2\mapsto\frac{1}{\sqrt{2}}\,|1\rangle\,|\overline{1}\rangle\,\widehat{|1\rangle}\right\}}\ \text{P-ORACLE}$$

$$\frac{}{\Omega;\{\kappa_1:\texttt{EN}\}\vdash_{\texttt{M}}\left\{\kappa_1\mapsto\frac{1}{\sqrt{2}}\,|\overline{1}\rangle\,|0\rangle\,\widehat{|1\rangle}\right\}\,e\,\left\{\kappa_1\mapsto\frac{1}{\sqrt{2}}\,|\overline{1}\rangle\,|1\rangle\,\widehat{|1\rangle}\right\}}\ \text{EQ}$$

$$\frac{}{\Omega;\{\kappa_1:\texttt{EN}\}\vdash_{\texttt{M}}\left\{\mathcal{F}(x[j\text{-}1],\kappa_1)\mapsto\sum_{d=0}^{1}\frac{1}{\sqrt{2}}\,|\overline{d}\rangle\,|0\rangle\right\}\,e\,\left\{\kappa_1\mapsto\frac{1}{\sqrt{2}}\,|\overline{1}\rangle\,|1\rangle\,\widehat{|1\rangle}\right\}}\ \text{M-}\mathcal{F}$$

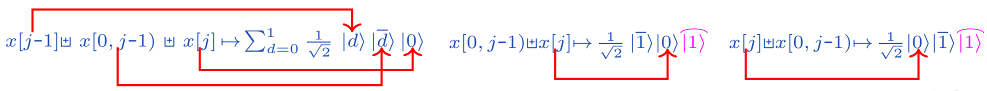
$$\frac{}{\Omega;\{\kappa:\texttt{EN}\}\vdash_{\texttt{C}}\left\{\kappa\mapsto\sum_{d=0}^{1}\frac{1}{\sqrt{2}}\,|\overline{d}\rangle\,|0\rangle\right\}\,\texttt{if}\ (x[j\text{-}1])\ e\left\{U(\neg x[j\text{-}1])\mapsto\sum_{d=0}^{1}\frac{1}{\sqrt{2}}\,|\overline{d}\rangle\,|0\rangle * U(x[j\text{-}1])\mapsto\frac{1}{\sqrt{2}}\,|\overline{1}\rangle\,|1\rangle\,\widehat{|1\rangle}\right\}}\ \text{P-IF}$$

$$\Omega;\sigma\vdash_{\texttt{C}}\left\{x[0,j)\mapsto\sum_{d=0}^{1}\frac{1}{\sqrt{2}}\,|\overline{d}\rangle * x[j,n)\mapsto|\overline{0}\rangle\right\}\,\texttt{if}\ (x[j\text{-}1])\ e\left\{x[0,j\text{+}1)\mapsto\sum_{d=0}^{1}\frac{1}{\sqrt{2}}\,|\overline{d}\rangle * x[j\text{+}1,n)\mapsto|\overline{0}\rangle\right\}$$

$$\kappa = x[j\text{-}1]\uplus\kappa_1 \qquad \kappa_1 = x[0,j\text{-}1)\uplus x[j] \qquad \kappa_2 = x[j]\uplus x[0,j\text{-}1)$$
$$e = x[j]\leftarrow x[j]+1; \qquad \sigma = \{x[0,j):\texttt{EN},\ x[j,n):\texttt{Nor}\} \qquad U(b) = \mathcal{U}(b,\ x[j\text{-}1],\ \kappa)$$

Figure 5 Detailed automated proof for a loop-step in GHZ. Constructed from the bottom up.

To permit automated proof inference, we design the uniformity in Qafny proof rules to require that the locus fragments for qubits that an operation is directly applied always be prefixed. For example in Figure 5 P-If, instead of having locus $x[0,j\text{+}1)$, we rewrite it further to κ ($x[j\text{-}1]\uplus x[0,j\text{-}1)\uplus x[j]$), so the qubit $x[j\text{-}1]$ that the Boolean guard is applied to appears at the start position. These rewrites simultaneously and appropriately rearrange the quantum value associated with the loci. In a Qafny quantum state, qubits in a locus are arranged as a list of indices pointing to qubit positions. The locus indices point to particular qubits in a `Nor` and `Had` typed value since they essentially represent an array of qubits. An `EN` typed value consists of a list of basis-kets; the locus indices refer to the corresponding bases appearing in each basis-ket.



In the three example states above from Figure 5, the first maps the locus κ ($x[j\text{-}1]\uplus x[0,j\text{-}1)\uplus x[j]$) to the pre-state in the bottom of line P-If, where $|\overline{d}\rangle$ is expanded to $|d\rangle\,|\overline{d}\rangle$. In each basis-ket ($d$ is 0 or 1), the first qubit $x[j\text{-}1]$ of the locus κ corresponds to the first basis bit, while the last qubit $x[j]$ corresponds to $|0\rangle$, the last basis bit. Applying an operation on $x[j]$ performs the application on the last basis bit $|0\rangle$ for every basis-ket. We can also refer a consecutive fragment of a locus to its basis bits, e.g., range $x[0,j\text{-}1)$ refers to $|\overline{d}\rangle$, the middle portion of each basis-ket, provided that they have the same cardinality. In this paper, we call the corresponding basis bits of qubits or locus fragments for a value (or a basis-ket set) as the *qubit's/locus's position bases* of the value (or the basis-ket set). A locus's position bases are linked and moved according to the rewrites of the locus, e.g., the middle and right examples above represent the rewrites from locus $\kappa_1 = x[0,j\text{-}1)\uplus x[j]$ to $\kappa_2 = x[j]\uplus x[0,j\text{-}1)$ in the pre-states (bottom to upper) of Figure 5 line EQ.

The rewrite moves $x[j]$ in κ_1 to the front in κ_2; correspondingly, $x[j]$'s position basis ($|0\rangle$) is also moved to the front. These examples show the functionality of *frozen basis stacks*. The two basis-kets' frozen basis stacks both contain a basis $\widehat{|1\rangle}$, which are not referenced by any part of a locus and therefore unreachable qubits. As shown in Section 1, we want local reasoning and preserving quantum theorems, i.e., a quantum state for a program piece does not mention qubits that are not reachable in the piece, e.g., accessing $x[j\text{-}1]$ above inside the conditional body means a violation of no-cloning. However, quantum states can be entangled, so unreachable qubits cannot be separated from the states. Instead, we hide the unreachable qubits, such as $x[j\text{-}1]$, in the frozen stack and retrieve it after jumping out of the conditional body. A comprehensive example is given in Section 6.2. We show how to unfreeze the frozen bases and explain the motivation for having frozen bases shortly below.

3.3 Rewrites based on Locus Type and State Equivalence Relations

The Qafny type system maintains simultaneity through the type-guided state rewrites, formalized as equivalence relations (Section 4.3). Other than the locus qubit position permutation introduced above, the types associated with loci in the environment also play an essential role in the rewrites. In Qafny, a locus represents a possibly entangled qubit group. From the study of many quantum algorithms [2, 4, 35, 48, 1, 43, 30, 14], we found that the establishment of an entanglement group can be viewed as a loop structure of incrementally adding a qubit to the group at a time, representing the entanglement's scope expansion; as the analogy in Figure 3b, qubits in the blue part are added to the orange part one by one. This behavior is similar to splits and joins of array elements if we view quantum states as arrays. However, joining and splitting two `EN`-typed values are hard problems [4]. Another critical observation in studying many quantum algorithms is that the entanglement group establishment usually involves splitting a qubit in a `Nor`/`Had` typed value and joining it to an existing `EN` typed entanglement group. We manage these join and split patterns type-guided equations in Qafny, suitable for automated verification. The GHZ example above (Figure 3c line 5) is an example of `Nor` and an `EN` type state split and join, where in each loop-step in Figure 3c, a `Nor`-typed qubit $x[j]$ is split from locus $x[j,n)$ and moved to the end of the `EN`-typed locus $x[0,j)$. Details are in Section 4.3.

3.4 The Qafny Proof System Glance Via Quantum Conditional Proofs

We integrate our type system with the Qafny proof system, where Qafny's type-guided proof triple ($\Omega;\sigma \vdash_g \{P\}\, e\, \{Q\}$) states that from a pre-condition P, executing e results in a post-condition Q, provided that P and Q are resp. well-formed w.r.t σ and σ', where $\Omega,\sigma \vdash_g e \triangleright \sigma'$ is a valid typing judgment (explained in Section 4.3).

A key design principle for proof automation rules is compositional and rule generalization, i.e., automated proof steps should be compositional, where each proof step is localized regarding a localized state, and the generalization means that automation should not depend on the specific local states. The issue with quantum proof rule designs is entanglement, i.e., a program execution on a local state might have global effects, which force the proof automation system to perform case analyses on the local states to resolve the global effects. For example, in verifying the conditional `if` ($x[j\text{-}1]$) e in the bottom of Figure 5, e can be applied to an entangled qubit state outside the visibility of qubits mentioned in e. Since e can be arbitrarily complicated, the prior work [56] handles the verification of the quantum conditional by expanding it as a whole matrix applied to a whole quantum state and performing case analyses. For proof automation, we need to design a uniform procedure, expressed as proof rules, to derive the verification; such a task is handled by predicate transformers and frozen stacks built on our locus structures.

An example is given at the line P-If in Figure 5, we utilize two locus predicate transformers $\mathcal{F}$ and $\mathcal{U}$ to transform the pre- and post-conditions so that they focus on the loci and basis-kets relevant to the current computation. In verifying the quantum conditional (`if` ($x[j\text{-}1]$) {$x[j] \leftarrow x[j]$+1}), we first apply $\mathcal{F}$ to transform the pre-condition. For the value

[4] The former is a Cartesian product; the latter is $\geq NP$-hard, both unsuitable for automated verification.

$\frac{1}{\sqrt{2}} |\overline{0}\rangle |0\rangle + \frac{1}{\sqrt{2}} |\overline{1}\rangle |0\rangle$, we filter out the basis-ket $\frac{1}{\sqrt{2}} |\overline{0}\rangle |0\rangle$, as the Boolean guard ($x[j\text{-}1]$) is not satisfiable for $x[j\text{-}1]$'s position basis ($|0\rangle$) of the basis-ket. For the remaining basis-ket $\frac{1}{\sqrt{2}} |\overline{1}\rangle |0\rangle$, we freeze $x[j\text{-}1]$'s position basis ($|1\rangle$), by pushing $|1\rangle$ to the frozen stack as an unreachable position, highlighted by $\widehat{|1\rangle}$, since it represents the qubit appearing in the Boolean guard that should not join any computation in e ($x[j] \leftarrow x[j]\text{+}1$). A frozen stack represents the link between the local state and its entangled global state. Each basis-ket in a superposition state can be associated with a single frozen stack, and we utilize a predicate transformer to manipulate all these frozen stacks in a state, recording the side-effects of the entangled global state caused by local state changes.

Notice that the locus is transformed from κ to κ_1 by removing $x[j\text{-}1]$ to preserve simultaneity. The post-condition $\kappa_1 \mapsto \frac{1}{\sqrt{2}} |\overline{1}\rangle |1\rangle \widehat{|1\rangle}$ contains only the computation result of the basis-ket $\frac{1}{\sqrt{2}} |\overline{1}\rangle |0\rangle$, and we want the final post-state to contain all other missing pieces, which is the task of the two $\mathcal{U}$ transformers. $\mathcal{U}(b, x[j\text{-}1], \kappa)$ points to the basis-ket satisfying the Boolean guard ($\frac{1}{\sqrt{2}} |\overline{1}\rangle |0\rangle$), from the above result post-condition. The transformer transforms $x[j\text{-}1]$'s position basis, currently in the stack, back to its normal position. $\mathcal{U}(\neg b, x[j\text{-}1], \kappa)$ represents the basis-ket not satisfying the guard ($\frac{1}{\sqrt{2}} |\overline{0}\rangle |0\rangle$), where we retrieve it from the pre-condition through the transformer. Finally, the two transformers transform and assemble the two states into one as the post-condition at the bottom Figure 5. Section 4.4 contains more details.

4 Qafny Formalism

This section formalizes Qafny's syntax, semantics, type system, proof system, and the corresponding soundness and completeness theorems. Running example in Figure 6 describes quantum order finding, the core component of Shor's algorithm (complete one in TR [24] C.5). The program assumes that an n-qubit `H` gate and an addition ($y[0, n)\text{+}1$) respectively applied to ranges $x[0, n)$ and $y[0, n)$ before line 3. The for-loop entangles range $y[0, n)$ with every qubit in $x[0, n)$, one per loop step, and applies a modulo multiplication in each step. `measure`(y) (partial measurement) in line 8 non-deterministically outputs a classical value $a^t \,\%\, N$ for y, and interconnectively rearranges $x[0, n)$'s quantum state, with all basis kets' bases related to a period value p. We unveil the details along with the section.

4.1 Qafny Syntax

Qafny is a C-like flow-sensitive language equipped with quantum heap mutations, quantum conditionals, and for-loops. We intend to provide users with a high-level view of quantum operations, e.g., viewing `H` and `QFT`$^{[-1]}$ gates as state preparation, quantum oracles (μ in [23]) as quantum arithmetic operations, and controlled gates as quantum conditionals and loops. As in Figure 7, aside from standard forms such as sequence (e ; e) and `SKIP` (`{}`), statements e also include let binding (`let` $x = am$ `in` e), quantum heap mutations (_ ← _), quantum/classical conditionals (`if` (b) e), and loops (`for` $j \in [a_1, a_2)$ `&&` b $\{e\}$). The let statement binds either the result of an arithmetic expression (a) or a *computational basis measurement* operator (`measure`(y)) to an immutable `C`/`M` kind variable x in the body e. This design ensures all classical variables are immutable and lexically scoped to avoid quantum observer breakdown due to mutating a classical variable inside a quantum conditional body.

(1) `int` $u = 0$; `if` ($x[0]$) $u = 1$; ✗ (2) `if` ($x[0]$) `let` $u = 1$ `in {}`; ✓ (3) `let` $u = 1$ `in if` ($x[0]$) `{}`; ✓

Here, case (1) declares u as 0 and changes its value to 1 inside the quantum conditional,

$1 < a < N \qquad E(t) = x[t,n] \mapsto \frac{1}{\sqrt{2^{n-t}}} \bigotimes_{i=0}^{n-t-1} (|0\rangle + |1\rangle) \; * \; x[0,t] \uplus y[0,n] \mapsto \sum_{i=0}^{2^t-1} \frac{1}{\sqrt{2^t}} |i\rangle\, |a^i \,\%\, N\rangle$

$\{x[0,n) \mapsto \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (\|0\rangle + \|1\rangle) * y[0,n) \mapsto \|\overline{0}\rangle\, \|1\rangle\}$	$\{\, x[0,n) : \texttt{Had} ,\ y[0,n) : \texttt{Nor} \,\}$
$\{\ E(0)\ \}$	$\{\, x[0,n) : \texttt{Had} ,\ x[0,0) \uplus y[0,n) : \texttt{EN} \,\}$
`for` $j \in [0,n)$ `&&` $x[j]$	
$\quad\{\ E(j)\ \}$	$\{\, x[j,n) : \texttt{Had} ,\ x[0,j) \uplus y[0,n) : \texttt{EN} \,\}$
$\quad y[0,n) \leftarrow a^{2^j} \cdot y[0,n) \,\%\, N;$	
$\{\ E(n)\ \}$	$\{\, x[0,0) : \texttt{Had} ,\ x[0,n) \uplus y[0,n) : \texttt{EN} \,\}$
$\{\ x[0,n) \uplus y[0,n) \mapsto \sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} \|i\rangle\, \|a^i \,\%\, N\rangle\ \}$	$\{\, x[0,n) \uplus y[0,n) : \texttt{EN} \,\}$
`let` u `=measure(`y`) in` ...	
$\left\{ \begin{array}{l} x[0,n) \mapsto \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \|t+kp\rangle * p = \texttt{ord}(a,N) \\ * \; u = (\frac{p}{2^n}, a^t \,\%\, N) * r = \texttt{rnd}(\frac{2^n}{p}) \end{array} \right\}$	$\{\, x[0,n) : \texttt{EN} \,\}$

Figure 6 Snippets from quantum order finding in Shor's algorithm; full proof in TR [24] C.5. $\texttt{ord}(a, N)$ gets the order of a and N. $\texttt{rnd}(r)$ rounds r to the nearest integer. We interpret integers as bitstrings in $|i\rangle$ and $|a^i \,\%\, N\rangle$. The right column presents the types of all loci involved.

OQASM Expr	μ							
Arith Expr	a	::=	x	$\| x[i,j)$	$\| v$	$\| a_1 + a_2$	$\| a_1 \cdot a_2$	$\| \ldots$
Bool Expr	b	::=	$x[a]$	$\| (a_1 = a_2)$ `@` $x[a]$		$\| (a_1 < a_2)$ `@` $x[a]$		$\| \ldots$
Predicate Locus	K	::=	κ	$\| \mathcal{M}(x,n,\kappa)$	$\| \mathcal{F}(b,\kappa,\kappa)$	$\| \mathcal{U}(b,\kappa,\kappa)$		
Predicate	P,Q,R	::=	$a_1 = a_2$	$\| a_1 < a_2$	$\| K \mapsto q$	$\| P \wedge P$	$\| P * P$	$\| \ldots$
Gate Expr	op	::=	`H`	$\|$ $\texttt{QFT}^{[-1]}$				
C/M Kind Expr	am	::=	a	$\|$ `measure(`y`)`				
Statement	e	::=	`{}`	$\| \kappa \leftarrow op$	$\| \kappa \leftarrow \mu$	$\|$ `let` x `=` am `in` e		
		$\|$	e_1 ; e_2	$\|$ `if` (b) e		$\|$ `for` $j \in [a_1, a_2)$ `&&` b $\{e\}$		

Figure 7 Core QAFNY syntax. Element syntax is in Figure 4 and OQASM is in [23]. $\texttt{QFT}^{[-1]}$ refers to the `QFT` and reversed `QFT`. An arithmetic expression x is a `C/M` kind variable, $x[i,j)$ is a quantum array range, and v is a `C/M` kind value. $x[a]$ is the a-th element of qubit array x.

which creates an observer effect because u's value depends on qubit $x[0]$. Cases `(2)` and `(3)` show that our immutable `let` binding can avoid the issue because the binding in `(2)` can be compiled to `(3)` due to the immutability; thus, u's value does not depend on the qubit.

A quantum heap mutation operation mutates qubit array data by applying to a locus κ either a unitary state preparation operation (op) (one of Hadamard `H`, quantum Fourier transformation `QFT`, and its inverse $\texttt{QFT}^{-1}$) or a unitary oracle operation (μ). [5] Other unitary operations, including quantum diffusion and amplification operations, are in TR [24] C.1.

Quantum reversible Boolean guards b are implemented as OQASM oracle operations, expressed by one of $(a_1 = a_2)$ `@` $x[a]$, $(a_1 < a_2)$ `@` $x[a]$, and $x[a]$, which intuitively amounts to computing $a_1 = a_2$, $a_1 < a_2$ and `false` respectively as b_0 and storing the result of $b_0 \oplus x[a]$ as a binary in qubit $x[a]$.[6] In both conditionals and loops, guards b are used to represent the qubits that are controlling. In addition to the `let` bindings, the quantum for-loop also introduces and enumerates `C`-kind value j over interval $[a_1, a_2)$ with j visible to both the guard b and the loop body e. As a result of immutability, loops in QAFNY are guaranteed to terminate. If all variables in the guard b are classical, the conditional or

[5] μ can define all quantum arithmetics, e.g., $x[j]$+1 (Fig. 3c) & $a^{2^j} y[0,n) \,\%\, N$ (Fig. 6). See [23].
[6] a_1 and a_2 can possibly apply to a range, like $y[0,n)$, in an entangled locus.

$$\text{S-ExpC}\ \frac{(\varphi, e[n/x]) \Downarrow \varphi'}{(\varphi, \texttt{let } x = n \texttt{ in } e) \Downarrow \varphi'} \qquad \text{S-ExpM}\ \frac{(\varphi, e[(r,n)/x]) \Downarrow \varphi'}{(\varphi, \texttt{let } x = (r,n) \texttt{ in } e) \Downarrow \varphi'} \qquad \text{S-OP}\ \frac{\circ = op \vee \circ = \mu}{(\varphi \uplus \{\kappa \uplus \kappa' : q\}, \kappa \leftarrow \circ) \Downarrow \varphi \uplus \{\kappa \uplus \kappa' : [\![\circ]\!]^{|\kappa|} q\}}$$

$$\text{S-If}\ \frac{FV(\Omega, b) = \kappa \qquad [\![b]\!]^{|\kappa|} q = q\langle \kappa, b\rangle + q\langle \kappa, \neg b\rangle \qquad (\varphi \uplus \{\kappa' : S^{|\kappa|}(q\langle \kappa, b\rangle)\}, e) \Downarrow \varphi \uplus \{\kappa' : q'\}}{(\varphi \uplus \{\kappa \uplus \kappa' : q\}, \texttt{if } (b)\ e) \Downarrow \varphi \uplus \{\kappa \uplus \kappa' : P(q') + q\langle \kappa, \neg b\rangle\}}$$

$$\text{S-Loop}\ \frac{n < n' \qquad (\varphi, \texttt{if } (b[n_1/j])\ e[n_1/j]) \Downarrow \varphi' \qquad (\varphi', \texttt{for } j \in [n{+}1, n')\ \texttt{\&\&}\ b\ \{e\}) \Downarrow \varphi''}{(\varphi, \texttt{for } j \in [n, n')\ \texttt{\&\&}\ b\ \{e\}) \Downarrow \varphi''} \qquad \text{S-Loop1}\ \frac{n \geq n'}{(\varphi, \texttt{for } j \in [n, n')\ \texttt{\&\&}\ b\ \{e\}) \Downarrow \varphi}$$

$$\text{S-Seq}\ \frac{(\varphi, e_1) \Downarrow \varphi' \qquad (\varphi', e_2) \Downarrow \varphi''}{(\varphi, e_1\ ;\ e_2) \Downarrow \varphi''} \qquad \text{S-Mea}\ \frac{\kappa = y[0,n) \qquad r = \sum_j |z_j|^2 \qquad (\varphi \uplus \{\kappa' : \sum_j \frac{z_j}{\sqrt{r}} \eta_j\}, e[(r, \{\!\!|c|\!\!\})/x]) \Downarrow \varphi'}{(\varphi \uplus \{\kappa \uplus \kappa' : \sum_j z_j\, |c\rangle\, \eta_j + q\langle \kappa, c \neq \kappa\rangle\}, \texttt{let } x = \texttt{measure}(y) \texttt{ in } e) \Downarrow \varphi'}$$

$$\begin{array}{llll}
[\![\circ]\!]^n(\sum_j z_j\, |c_j\rangle\, \eta_j) & \triangleq & \sum_j z_j([\![\circ]\!]\, |c_j\rangle)\eta_j & \texttt{where}\ \ (\circ = \mu \vee \circ = op \vee \circ = b) \wedge \forall j\, |c_j| = n \\
(\sum_i z_i |c_i\rangle \eta_i + q)\langle \kappa, b\rangle & \triangleq & \sum_i z_i |c_i\rangle \eta_i & \texttt{where}\ \ \forall i.\, |c_i| = |\kappa| \wedge [\![b[c_i/\kappa]]\!] = \texttt{true} \\
S^n(\sum_j z_j\, |c_j\rangle\, \beta_j (\!|\beta'_j|\!)) & \triangleq & \sum_j z_j \beta_j (\!|\, |c_j\rangle\, \beta'_j |\!) & \texttt{where}\ \ \forall j\, |c_j| = n \\
P(\sum_j z_j \beta_j (\!|\, |c_j\rangle\, \beta'_j |\!)) & \triangleq & \sum_j z_j\, |c_j\rangle\, \beta_j (\!|\beta'_j|\!) &
\end{array}$$

Figure 8 Selected semantic rules. $\{\!\!|c|\!\!\}$ turns basis c to an integer.

loop becomes a standard classical one, which is differentiated and definable by our type system, described in TR [24] C. Obviously, users can always view a Qafny program as a quantum sub-component in a Dafny program, which provides better library support for classical conditionals. Predicates and predicate loci in Figure 7 describe quantum state properties in the Qafny proof system, explained in Section 4.4.

4.2 Qafny Semantics

The Qafny semantics is formalized as a big-step transition relation $(\varphi, e) \Downarrow \varphi'$, with φ / φ' being quantum states as described in Figure 4. The judgment relation states that a program e with the pre-state φ transitions to a post-state φ'. A selection of the rules defining $\Downarrow$ may be found Figure 8, and the additional rules are in TR [24] C.2. $FV(\Omega, -)$ produces a locus by unioning all qubits in $-$ with the quantum variable kind information in Ω; its definition is given in TR [24] A.

Assignment and Mutation Operations. Rules S-ExpC and S-ExpM define the behaviors for `C` and `M` kind classical variable assignments, which perform variable-value substitutions. Rule S-OP defines a quantum heap mutation applying a state preparation operation (op) or an oracle expression (μ) to a locus κ for a `EN`-typed state. Here, the locus fragment κ to which the operation is applied must be the very first one in the locus $\kappa \uplus \kappa'$ that refers to the entire quantum state q. If not, we will first apply equivalence rewrites to be explained in Section 4.3 to move κ to the front. With κ preceding the rest fragment κ', the operation's semantic function $[\![\circ]\!]^n$ ($\circ$ being `H` or μ) is then applied to κ's position bases in the quantum value q. More specifically, the function is only applied to the first n (equal to $|\kappa|$) basis bits of each basis-ket in the value while leaving the rest unchanged. The semantic interpretations of the op and μ operations are essentially the quantum gate semantics given in Li et al. [23]. For example, in Figure 5 line P-Oracle, we apply an oracle operation $x[j]$+1 to $x[j]$, the first position of the locus κ_2 ($i.e.$,$x[j] \uplus x[0, j{-}1)$), which transforms the first basis bit to $|1\rangle$.

Before the application, we rewrite the pre-state containing κ_1 below the line EQ in Figure 5, to the form corresponding to the locus κ_2 above the line.

Quantum Conditionals. As in rule S-IF, for a conditional `if` (b) e, we first evaluate the Boolean guard b on κ's position bases ($FV(\Omega, b) = \kappa$) of the quantum value state q to $[\![b]\!]^{|\kappa|}q$[7] because b's computation might have side-effects in changing κ's position bases, as the example in Section 6.2. The quantum value referred by $\kappa \uplus \kappa'$ is further partitioned into $q\langle\kappa, b\rangle + q\langle\kappa, \neg b\rangle$ where $q\langle\kappa, b\rangle$ is a set of basis-kets whose κ's position bases satisfying b and $q\langle\kappa, \neg b\rangle$ is the rest. Since the body e only affects the basis-kets ($q\langle\kappa, b\rangle$) satisfying the guard b, we rule out the basis-kets $q\langle\kappa, \neg b\rangle$ (unsatisfying the guard) in e's computation. We also need to push κ's position bases in $q\langle\kappa, b\rangle$ to the frozen stacks through the S^n operation to maintain the locus-state simultaneity in Section 3.2.

We describe rule S-IF along with an example in Figure 6 line 3-5. Here, the j-th iteration is unrolled to a quantum conditional `if` $(x[j])$ $\{y[0,n) \leftarrow a^{2^j} \cdot y[0,n) \,\%\, N\}$. The loci involved in the computation are $x[j]$ and $x[0,j) \uplus y[0,n)$, and their state transitions are given as:

$$
\begin{array}{ll}
 & \left\{x[j] : \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right\} \uplus \left\{x[0,j) \uplus y[0,n) : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^j}} |i\rangle\, |a^i \,\%\, N\rangle\right\} \\
\equiv & \left\{x[j] \uplus x[0,j) \uplus y[0,n) : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |1\rangle\, |i\rangle |a^i \,\%\, N\rangle + \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |0\rangle\, |i\rangle |a^i \,\%\, N\rangle\right\} \\
\xrightarrow{\text{S-IF}} & \left\{x[j] \uplus x[0,j) \uplus y[0,n) : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |1\rangle\, |i\rangle |(a^i \cdot a^{2^j}) \,\%\, N\rangle + \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |0\rangle\, |i\rangle |a^i \,\%\, N\rangle\right\} \\
\equiv & \left\{x[0,j{+}1) \uplus y[0,n) : \sum_{i=0}^{2^{j+1}-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle\, |a^i \,\%\, N\rangle\right\}
\end{array}
$$

The first equation transition ($\equiv$) merges the two locus states and turns the merged state into two sets (separated by $+$), respectively representing basis-kets where $x[j]$'s position bases are 1 and 0. Since the Boolean guard $x[j]$ has no side-effects, the application $[\![b]\!]^{|\kappa|}$ is an identity. The S-IF application performs a modulo multiplication oracle application on the basis-ket set where $x[j]$'s position bases being 1, while the last equation merges the two sets back to one summation formula. The S-IF application above can be further decomposed into two additional transitions in between:

$$
\begin{array}{ll}
\longrightarrow & \left\{x[0,j) \uplus y[0,n) : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle |a^i \,\%\, N\rangle \widehat{|1\rangle}\right\} \\
\xrightarrow{\text{S-OP}} & \left\{x[0,j) \uplus y[0,n) : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle |(a^i \cdot a^{2^j}) \,\%\, N\rangle \widehat{|1\rangle}\right\}
\end{array}
$$

The first transition removes the $q\langle\kappa, \neg b\rangle$ part, e.g., the basis-ket set where $x[j]$'s position bases are 0. Additionally, for every basis-ket in the $q\langle\kappa, b\rangle$ set, e.g., the basis-ket set where $x[j]$'s position bases being 1, we freeze κ's position bases by pushing the bases into the basis-ket's stacks through the function application $S^{|\kappa|}(q\langle\kappa, b\rangle)$, which finds the first $|\kappa|$ bits in every basis-ket and push them into the basis-ket's stack so that e's application targets locus κ' instead of $\kappa \uplus \kappa'$. As the first transition above, for each basis-ket, we push $x[j]$'s position basis ($|1\rangle$) to the basis-ket's stack, as the $\widehat{|1\rangle}$ part and the pointed-to locus is rewritten to $x[0,j) \uplus y[0,n)$. After applying the body e to the state, for every basis-ket, we pop κ's position bases ($P(q')$) from the basis-ket's stack and relabel the locus of the state to be $\kappa \uplus \kappa'$; in doing so, we also need to add the unmodified basis-kets $q\langle\kappa, \neg b\rangle$ back into the whole state. After applying S-OP on locus $x[0,j) \uplus y[0,n)$ above, we pop $\widehat{|1\rangle}$ from every basis-ket's stack and assemble the unchanged part ($\sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |0\rangle\, |i\rangle |a^i \,\%\, N\rangle$) back to the state of locus $x[j] \uplus x[0,j) \uplus y[0,n)$; the result is shown as the state after the $\xrightarrow{\text{S-IF}}$ application above.

[7] This is defined formally as an oracle, same as μ above.

T-PAR
$$\frac{\sigma \preceq \sigma' \quad \Omega;\sigma' \vdash_g e \triangleright \sigma''}{\Omega;\sigma \vdash_g e \triangleright \sigma''}$$

T-EXPC
$$\frac{x \notin \texttt{dom}(\Omega) \quad \Omega;\sigma \vdash_g e[n/x] \triangleright \sigma'}{\Omega;\sigma \vdash_g \texttt{let } x = n \texttt{ in } e \triangleright \sigma'}$$

T-EXPM
$$\frac{x \notin \texttt{dom}(\Omega) \quad \Omega \vdash a : \texttt{M} \quad \Omega[x \mapsto \texttt{M}];\sigma \vdash_g e \triangleright \sigma'}{\Omega;\sigma \vdash_g \texttt{let } x = a \texttt{ in } e \triangleright \sigma'}$$

T-OP
$$\frac{\circ = op \vee \circ = \mu}{\Omega;\sigma \uplus \{\kappa \boxplus \kappa' : \texttt{EN}\} \vdash_g \kappa \leftarrow \circ \triangleright \sigma \uplus \{\kappa \boxplus \kappa' : \texttt{EN}\}}$$

T-MEA
$$\frac{\Omega(y) = \texttt{Q}\ n \quad x \notin \texttt{dom}(\Omega) \qquad \Omega[x \mapsto \texttt{M}];\sigma \uplus \{\kappa : \texttt{EN}\} \vdash_{\texttt{C}} e \triangleright \sigma'}{\Omega;\sigma \uplus \{y[0,n) \boxplus \kappa : \tau\} \vdash_{\texttt{C}} \texttt{let } x = \texttt{measure}(y) \texttt{ in } e \triangleright \sigma'}$$

T-IF
$$\frac{FV(\Omega,b) = \kappa \quad FV(\Omega,e) \subseteq \kappa' \quad \Omega;\sigma \uplus \{\kappa' : \texttt{EN}\} \vdash_{\texttt{M}} e \triangleright \sigma \uplus \{\kappa' : \texttt{EN}\}}{\Omega;\sigma \uplus \{\kappa \boxplus \kappa' : \texttt{EN}\} \vdash_g \texttt{if } (b)\ e \triangleright \sigma \uplus \{\kappa \boxplus \kappa' : \texttt{EN}\}}$$

T-SEQ
$$\frac{\Omega;\sigma \vdash_g e_1 \triangleright \sigma_1 \quad \Omega;\sigma_1 \vdash_g e_2 \triangleright \sigma_2}{\Omega;\sigma \vdash_g e_1 \,;\, e_2 \triangleright \sigma_2}$$

T-LOOP
$$\frac{x \notin \texttt{dom}(\Omega) \quad \forall j \in [n_1,n_2)\,.\,\Omega;\sigma[j/x] \vdash_g \texttt{if } (b[j/x])\ e[j/x] \triangleright \sigma[j{+}1/x]}{\Omega;\sigma[n_1/x] \vdash_g \texttt{for } x \in [n_1,n_2)\ \texttt{\&\&}\ b\ \{e\} \triangleright \sigma[n_2/x]}$$

Figure 9 QAFNY type system. $FV(\Omega,-)$ gets a locus containing qubits in $-$ w.r.t. Ω (TR [24] A).

Quantum Measurement. A measurement ($\texttt{let } x = \texttt{measure}(y) \texttt{ in } e$) collapses a qubit array y, binds a M-kind outcome to x, and restricts its usage in e. Rule S-MEA shows the partial measurement behavior [8]. Assume that the locus containing the qubit array y is $y[0,n) \boxplus \kappa'$, the measurement is essentially a two-step array filter: (1) the basis-kets of the EN typed value is partitioned into two sets (separated by $+$): $(\sum_{j=0}^{m} z_j \,|c\rangle\,|c_j\rangle) + q\langle \kappa, c \neq \kappa\rangle$ with $\kappa = y[0,n)$, by randomly picking a $|\kappa|$-length basis c where every basis-ket in the first set have κ's position basis c; and (2) we create a new array value by removing all the basis-kets not having c as prefixes (the $q\langle \kappa, c \neq \kappa\rangle$ part) and also removing the κ's position basis in every remaining basis-ket; thus, the quantum value becomes $\sum_{j=0}^{m} \frac{z_j}{\sqrt{r}} \eta_j$. Notice that the element size of the post-state m+1 is smaller than the size of the pre-state before the measurement. Since the amplitudes of basis-kets must satisfy $\sum_i |z_i|^2 = 1$, we need to normalize the amplitude of each element in the post-state by multiplying a factor $\frac{1}{\sqrt{r}}$, with $r = \sum_{j=0}^{m} |z_j|^2$ as the sum of the amplitude squares appearing in the post-state. When proving quantum program properties, the amplitudes appearing in basis-kets usually follow a periodic pattern that users can provide, so computing r will be relatively simple, see Section 4.4. In Figure 6, the measurement (line 8) transitions the state from lines 7 to 9. Locus $y[0,n)$'s position basis is $|a^i \,\%\, N\rangle$ for each basis-ket in $\sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} \,|i\rangle\,|a^i \,\%\, N\rangle$. We then randomly pick the basis value $a^t \,\%\, N$ as a measurement result, stored in u, and the probability of the pick is $\frac{p}{2^n}$ where p is the order of a and N. The probability is computed solely based on p because it represents the period of the factorization in Shor's algorithm. The number (r) of remaining basis-kets in range $x[0,n)$ is computed by rounding $\frac{2^n}{p}$.

4.3 Qafny Locus Type System

The QAFNY typing judgment $\Omega;\sigma \vdash_g e \triangleright \sigma'$ states that e is well-typed under the context mode g (the syntax of kind g is reused as context modes) and environments Ω and σ. The kind environment Ω is populated through `let` and `for` loops that introduce `C` and `M` kind variables, while `Q`-kind variable mappings in Ω are given as a global environment. Selected type rules are in Figure 9; the rules not mentioned are similar and given in TR [24] C.3. For every type rule, well-formed domains ($\Omega \vdash \texttt{dom}(\sigma)$) are required but hidden from the rules,

---

[8] A complete measurement is a special case of a partial measurement when κ' is empty in S-MEA

$$
\begin{aligned}
\sigma &\preceq \sigma \\
\{\emptyset : \tau\} \uplus \sigma &\preceq \sigma \\
\{\kappa : \tau\} \uplus \sigma &\preceq \{\kappa : \tau'\} \uplus \sigma \\
&\quad \texttt{where } \tau \sqsubseteq \tau' \\
\{\kappa_1 \boxplus s_1 \boxplus s_2 \boxplus \kappa_2 : \tau\} \uplus \sigma &\preceq \{\kappa_1 \boxplus s_2 \boxplus s_1 \boxplus \kappa_2 : \tau\} \uplus \sigma \\
\{\kappa_1 : \tau\} \uplus \{\kappa_2 : \tau\} \uplus \sigma &\preceq \{\kappa_1 \boxplus \kappa_2 : \tau\} \uplus \sigma \\
\{\kappa_1 \boxplus \kappa_2 : \tau\} \uplus \sigma &\preceq \{\kappa_1 : \tau\} \uplus \{\kappa_2 : \tau\} \uplus \sigma
\end{aligned}
$$

(a) Environment Equivalence.

$$
\begin{aligned}
\varphi &\equiv \varphi \\
\{\emptyset : q\} \uplus \varphi &\equiv \varphi \\
\{\kappa : q\} \uplus \varphi &\equiv \{\kappa : q'\} \uplus \varphi \\
&\quad \texttt{where } q \equiv_{|\kappa|} q' \\
\{\kappa_1 \boxplus s_1 \boxplus s_2 \boxplus \kappa_2 : q\} \uplus \varphi &\equiv \{\kappa_1 \boxplus s_2 \boxplus s_1 \boxplus \kappa_2 : q'\} \uplus \varphi \\
&\quad \texttt{where } q' = q^{|\kappa_1|}\langle |s_1| \asymp |s_2| \rangle \\
\{\kappa_1 : q_1\} \uplus \{\kappa_2 : q_2\} \uplus \varphi &\equiv \{\kappa_1 \boxplus \kappa_2 : q'\} \uplus \varphi \\
&\quad \texttt{where } q' = q_1 \bowtie q_2 \\
\{\kappa_1 \boxplus \kappa_2 : \varphi\} \uplus \sigma &\equiv \{\kappa_1 : \varphi_1\} \uplus \{\kappa_2 : \varphi_2\} \uplus \sigma \\
&\quad \texttt{where } \varphi_1 \bowtie \varphi_2 = \varphi \wedge |\varphi_1| = |\kappa_1|
\end{aligned}
$$

(b) State Equivalence.

Permutation:

$$(q_1 \bigotimes q_2 \bigotimes q_3 \bigotimes q_4)^n \langle i \asymp k \rangle \triangleq q_1 \bigotimes q_3 \bigotimes q_2 \bigotimes q_4 \quad \texttt{where} \quad |q_1| = n \wedge |q_2| = i \wedge |q_3| = k$$

$$(\textstyle\sum_j z_j \, |c_j\rangle \, |c'_j\rangle \, |c''_j\rangle \, \eta_j)^n \langle i \asymp k \rangle \triangleq \textstyle\sum_j z_j \, |c_j\rangle \, |c''_j\rangle \, |c'_j\rangle \, \eta_j \quad \texttt{where} \quad |c_j| = n \wedge |c'_j| = i \wedge |c''_j| = k$$

Join Product:

$$z_1 \, |c_1\rangle \bowtie z_2 \, |c_2\rangle \triangleq (z_1 \cdot z_2) \, |c_1\rangle \, |c_2\rangle \qquad \textstyle\sum_{j=0}^{n} z_j |c_j\rangle \bowtie \sum_{k=0}^{m} z_k \, |c_k\rangle \triangleq \sum^{n \cdot m} z_j \cdot z_k \, |c_j\rangle \, |c_k\rangle$$

$$|c_1\rangle \bowtie \textstyle\sum_j z_j \eta_j \triangleq \sum_j z_j \, |c_1\rangle \, \eta_j \qquad (|0\rangle + \alpha(r) \, |1\rangle) \bowtie \sum_j z_j \eta_j \triangleq \sum_j z_j \, |0\rangle \, \eta_j + \sum_j (\alpha(r) \cdot z_j) \, |1\rangle \, \eta_j$$

Figure 10 QAFNY type/state relations. $\cdot$ is math mult. Term $\sum^{n \cdot m} P$ is a summation omitting the indexing details. $\bigotimes$ expands a `Had` array, as $\frac{1}{\sqrt{2^{n+m}}} \bigotimes_{j=0}^{n+m-2} q_j = (\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} q_j) \bigotimes (\frac{1}{\sqrt{2^m}} \bigotimes_{j=0}^{m-1} q_j)$.

such that every variable used in all loci of σ must appear in Ω, while $\Omega \vdash a : \texttt{M}$ judges that the expression a is well-formed and returns an `M` kind; see TR [24] A and B. The type system enforces three properties below.

No Cloning and Observer Breakdown. We enforce no cloning by disjointing qubits mentioned in a quantum conditional Boolean guard and its body. In rule T-IF, κ and κ' are disjoint unioned, and the two FV side-conditions ensure that the qubits mentioned in the Boolean guard and conditional body are respectively within κ and κ'; thus, they do not overlap. QAFNY is a *flow-sensitive* language, as we enforce no observer breakdown by ensuring no classical variable assignments through the QAFNY syntax and no measurements inside a quantum conditional through context restrictions. Each program begins with the context mode `C`, which permits all QAFNY operations. Once a type rule switches the mode to `M`, as in T-IF, measurement operations are suspended in this scope, as T-MEA is valid only if the context mode is `C`. For instance, let's imagine that the measurement in Figure 6 line 8 lives inside the for-loop in line 5, which our type system would forbid because type checking through T-LOOP calls rule T-IF that marks the context mode to `M`, while the application of rule T-MEA requires a `C` mode context to begin with.

Guiding Locus Equivalence and Rewriting. The semantics in Section 4.2 assumes that the loci in quantum states can be in ideal forms, e.g., rule S-OP assumes that the target locus κ are always prefixed. This step is valid if we can rewrite (type environment partial order $\preceq$) the locus to the ideal form through rule T-PAR, which interconnectively rewrites the locus appearing in the state, through our state equivalence relation ($\equiv$), as the locus state simultaneity enforcement (Section 3.2). The state equivalence rewrites have two components.

First, the type and quantum value forms have simultaneity, i.e., given a type τ_1 for a locus κ in a type environment (σ), if it is a subtype ($\sqsubseteq$) of another type τ_2, κ's value q_1 in a state (φ) can be rewritten to q_2 that has the type τ_2 through state equivalence rewrites ($\equiv_n$) where n is the number of qubits in q_1 and q_2. Both $\sqsubseteq$ and $\equiv_n$ are reflexive and types Nor and Had are subtypes of EN, which means that a Nor typed value ($|c\rangle$) and a Had typed value ($\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + \alpha(r_j)\,|1\rangle)$) can be rewritten to an EN typed value (TR [24] C.3). For example, range $x[0,n)$'s Had typed value $\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + |1\rangle)$ in Figure 6 line 1 can be rewritten to an EN type as $\sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}}\,|i\rangle$. If such a rewrite happens, we correspondingly transform $x[0,n)$'s type to EN in the type environment.

Second, type environment partial order ($\preceq$) and state equivalence ($\equiv$) also have simultaneity – in a proof judgment, we associate the state predicate, representing a state φ, with the type environment σ by sharing the same domain, i.e., $\mathtt{dom}(\varphi) = \mathtt{dom}(\sigma)$. Thus, the environment rewrites ($\preceq$) happening in σ gear the state rewrites in φ, e.g., the bottom proof step of Figure 5 transforms locus $x[0,j)$ in σ to locus κ ($x[j-1] \uplus x[0,j-1) \uplus x[j]$) above it, and the state rewrites in the pre-condition predicate happen accordingly as (left to right):

$$\begin{array}{llllll} \{x[0,j) : \sum_{d=0}^{1} \frac{1}{\sqrt{2}}\,|\overline{d}\rangle\} \uplus \{x[j] : |0\rangle\} & \equiv & \{x[0,j+1) : \sum_{d=0}^{1} \frac{1}{\sqrt{2}}\,|\overline{d}\rangle\,|0\rangle\} & \equiv & \{\kappa : \sum_{d=0}^{1} \frac{1}{\sqrt{2}}\,|\overline{d}\rangle\,|0\rangle\} \\ \{x[0,j) : \mathtt{EN}\} \quad\quad \uplus \{x[j] : \mathtt{Nor}\} & \preceq & \{x[0,j+1) : \mathtt{EN}\} & \preceq & \{\kappa : \mathtt{EN}\} \end{array}$$

Here, we add qubit $x[j]$ ($|0\rangle$) to the end of locus $x[0,j)$ and transform locus $x[0,j+1)$ to κ, so the upper proof step (P-If) in Figure 5 can proceed. The above rewrites are derived by the rules in Figure 10, where the rules in environment partial order and state equivalence are one-to-one corresponding. The first three lines describe the properties of reflective, identity, and subtyping equivalence. The fourth line enforces that the environment and state are close under locus permutation. After the equivalence rewrite, the position bases of ranges s_1 and s_2 are mutated by applying the function $q^{|\kappa_1|}\langle|s_1| \asymp |s_2|\rangle$. One example is the locus rewrite in Figure 6 line 7 from left to right, as:

$$\begin{array}{lll} \left\{x[0,n) \uplus y[0,n) : \mathtt{EN}\right\} & \preceq & \left\{y[0,n) \uplus x[0,n) : \mathtt{EN}\right\} \\ \left\{x[0,n) \uplus y[0,n) : \sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}}\,|i\rangle\,|a^i \mathbin{\%} N\rangle\right\} & \equiv & \left\{y[0,n) \uplus x[0,n) : \sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}}\,|a^i \mathbin{\%} N\rangle\,|i\rangle\right\} \end{array}$$

The last two lines in Figures 10a and 10b describe locus joins and splits, where the latter is an inverse of the former but much harder to perform practically. In the most general form, joining two EN-type states computes the Cartesian product of their basis-kets, shown in the bottom of Figure 10, which is practically hard for proof automation. Fortunately, the join operations in most quantum algorithms are between a Nor/Had typed and an EN-typed state, Joining a Nor-typed and EN-typed state puts extra qubits in the right location in every basis-ket of the EN-typed state as discussed in Section 3.3. Joining a Had-typed qubit (single qubit state) and EN-typed state duplicates the EN-typed basis-kets. In every loop step in Figure 6 line 3-5, we add a Had-typed qubit $x[j]$ to the middle of an EN-typed locus $x[0,j) \uplus y[0,n)$, transform the state to:

$$\left\{x[0,j{+}1) \uplus y[0,n) : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^j}}\,|i\rangle\,|0\rangle\,|a^i \mathbin{\%} N\rangle + \sum_{j=0}^{2^j} \frac{1}{\sqrt{2^j-1}}\,|i\rangle\,|1\rangle\,|a^i \mathbin{\%} N\rangle\right\}$$

The state can be further rewritten to the one in Figure 6 by merging the above two parts (separated by $+$). Notice that the basis-kets are still all distinct because the two parts are distinguished by $x[j]$'s position basis, i.e., $|0\rangle$ and $|1\rangle$. TR [24] F shows practical ways to perform additional state joins and splits, including an upgraded dependent type system to permit a few cases of splitting EN typed values.

Approximating Locus Scope. The type system approximates locus scopes. In rule T-If, we use $\kappa \uplus \kappa'$ as the approximate locus large enough to describe all possible qubits directly and indirectly mentioned in b and e. Such scope approximation might be over-approximated, which does not cause incorrectness in our proof system, while under-approximation is forbidden.

P-Frame
$$\frac{\texttt{dom}(\sigma) \cap FV(\Omega, R) = \emptyset \quad FV(\Omega, e) \subseteq \texttt{dom}(\sigma) \qquad \Omega; \sigma \vdash_g \left\{P\right\} e \left\{Q\right\}}{\Omega; \sigma \uplus \sigma' \vdash_g \left\{P * R\right\} e \left\{Q * R\right\}}$$

P-Con
$$\frac{\sigma \preceq \sigma' \qquad P \Rightarrow P' \qquad \Omega; \sigma' \vdash_g \left\{P'\right\} e \left\{Q'\right\} \qquad Q' \Rightarrow Q}{\Omega; \sigma \vdash_g \left\{P\right\} e \left\{Q\right\}}$$

P-OP
$$\frac{\circ = op \vee \circ = \mu}{\Omega; \{\kappa \uplus \kappa' : \texttt{EN}\} \vdash_g \left\{\kappa \uplus \kappa' \mapsto q\right\} \kappa \leftarrow \circ \left\{\kappa \uplus \kappa' \mapsto [\![\circ]\!]^{|\kappa|} q\right\}}$$

P-ExpC
$$\frac{\Omega; \sigma \vdash_g \left\{P\right\} e[n/x] \left\{Q\right\}}{\Omega; \sigma \vdash_g \left\{P\right\} \texttt{let } x = n \texttt{ in } e \left\{Q\right\}}$$

P-Mea
$$\frac{x \notin \texttt{dom}(\Omega) \qquad \Omega[x \mapsto \texttt{M}]; \sigma \uplus \{\kappa : \texttt{EN}\} \vdash_{\texttt{C}} \left\{P[\mathcal{M}(x, n, \kappa)/y[0, n) \uplus \kappa]\right\} e \left\{Q\right\}}{\Omega; \sigma \uplus \{y[0, n) \uplus \kappa : \texttt{EN}\} \vdash_{\texttt{C}} \left\{P\right\} \texttt{let } x = \texttt{measure}(y) \texttt{ in } e \left\{Q\right\}}$$

P-If
$$\frac{FV(\Omega, b) = \kappa \qquad \Omega; \{\kappa' : \texttt{EN}\} \vdash_{\texttt{M}} \left\{P[\mathcal{F}(b, \kappa, \kappa')/\kappa \uplus \kappa']\right\} e \left\{Q\right\}}{\Omega; \{\kappa \uplus \kappa' : \texttt{EN}\} \vdash_g \left\{P\right\} \texttt{if } (b)\ e \left\{P[\mathcal{U}(\neg b, \kappa, \kappa \uplus \kappa')/\kappa \uplus \kappa'] * Q[\mathcal{U}(b, \kappa, \kappa \uplus \kappa')/\kappa']\right\}}$$

P-Loop
$$\frac{n < n' \qquad \Omega; \sigma \vdash_g \left\{P(j) \wedge j < n'\right\} \texttt{if } (b)\ e \left\{P(j+1)\right\}}{\Omega; \sigma[n/j] \vdash_g \left\{P(n)\right\} \texttt{for } j \in [n, n') \texttt{ \&\& } b\ \{e\} \left\{P(n')\right\}}$$

P-Seq
$$\frac{\Omega; \sigma \vdash_g e_1 \triangleright \sigma_1 \quad \Omega; \sigma \vdash_g \left\{P\right\} e_1 \left\{P'\right\} \qquad \Omega; \sigma_1 \vdash_g \left\{P'\right\} e_2 \left\{Q\right\}}{\Omega; \sigma \vdash_g \left\{P\right\} e_1 \texttt{ ; } e_2 \left\{Q\right\}}$$

Figure 11 Select proof rules.

For example, if we combine two Had-typed qubits in our system and transform the value to EN-type as $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$, this is an over-approximation since the two qubits are not entangled. Partially measuring the first qubit leaves the second qubit's value unchanged as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

In addition to the above properties, we allow C-kind classical variables introduced by `let` to be evaluated in the type checking stage [9], while tracks M variables in Ω. Rule T-ExpC enforces that a classical variable x is replaced with its assigned value n in e, and classical expressions in e containing x are evaluated, so the proof system can avoid handling constants.

4.4 The Qafny Proof System

Every valid proof judgment $\Omega; \sigma \vdash_g \left\{P\right\} e \left\{Q\right\}$, shown in Figure 11, contains a pre- and post-condition predicates P and Q (syntax in Figure 7) for the statement e, satisfying the type restriction that $\Omega; \sigma \vdash_g e \triangleright \sigma'$; we also enforce that the predicate well-typed restrictions $\Omega; \sigma \vdash P$ and $\Omega; \sigma' \vdash Q$, meaning that all loci mentioned in P must be in the right forms and as elements in $\texttt{dom}(\sigma)$, introduced in TR [24] B. We state the restrictions as the typechecking constraint (TC) below:

$$TC(\sigma, P, Q) \triangleq \Omega; \sigma \vdash_g e \triangleright \sigma_1 \wedge \Omega; \sigma \vdash P \wedge \Omega; \sigma_1 \vdash Q$$

Rule P-Con describes the consequence rule where a well-formed pre- and post-conditions P and Q under σ is replaced by P' and Q', well-formed under σ'. Under the new conditions, we enforce a new type constraint $TC(\sigma', P', Q')$. Rule P-Frame is a specialized separation logic frame rule that separates the locus type environment and the quantum value to support local reasoning on quantum states. Rule M-Frame in Figure 12 describes the predicate semantics

[9] We consider all computation that only needs classical computers is done in the compilation time.

Predicate Model Rules:

$$\begin{array}{lllll}
\text{M-MAP} & \Omega;\psi;\sigma;\varphi \uplus \{\kappa : \sum_j z_j|c_j\rangle(\!|\beta_j|\!)\} \models_g \kappa \mapsto \sum_j z_j|c_j\rangle(\!|\beta'_j|\!) & & & \\
\text{M-LOCAL} & \Omega[x\mapsto \texttt{M}];\psi[x\mapsto (r,v)];\sigma;\varphi & \models_g P & \text{if} & \Omega;\psi;\sigma;\varphi \models_g P[(r,v)/x] \\
\text{M-FRAME} & \Omega;\psi;\sigma\uplus\sigma';\varphi\uplus\varphi' & \models_g P * Q & \text{if} & \Omega;\psi;\sigma;\varphi \models_g P \text{ and } \Omega;\psi;\sigma';\varphi' \models_g Q
\end{array}$$

Transformation Rules:

$$\begin{array}{llcll}
\text{M-}\mathcal{F} & \mathcal{F}(b,\kappa,\kappa') \mapsto q & = & \kappa' \mapsto \sum_j z_j\beta_j(\!|\,|c_j\rangle\,\beta'_j|\!) & \\
 & \multicolumn{4}{l}{\text{where } [\![b]\!]^{|\kappa|} q = q\langle\kappa,b\rangle + q\langle\kappa,\neg b\rangle \ \wedge\ q\langle\kappa,b\rangle = \sum_j z_j|c_j\rangle\beta_j(\!|\beta'_j|\!) \ \wedge\ \forall j.\ |c_j| = |\kappa|} \\
\text{M-}\mathcal{U} & \begin{array}{l}\mathcal{U}(\neg b,\kappa,\kappa') \mapsto \sum_j z_j|c_j\rangle\eta_j + q\langle\kappa,\neg b\rangle \\ *\,\mathcal{U}(b,\kappa,\kappa') \mapsto \sum_j z'_j\beta_j(\!|\,|c_j\rangle\,\beta'_j|\!)\end{array} & = & \kappa' \mapsto \sum_j z'_j|c_j\rangle\beta_j(\!|\beta'_j|\!) + q\langle\kappa,\neg b\rangle & \text{where } \forall j.\ |\kappa| = |c_j| \\
\text{M-}\mathcal{M} & \mathcal{M}(x,n,\kappa) \mapsto \sum_j z_j|c\rangle\eta_j + q\langle\kappa, c\neq\kappa\rangle & = & \kappa \mapsto \sum_j \frac{z_j}{\sqrt{r}}\eta_j * x = (r, \{\!|c|\!\}) & \text{where } n = |c|
\end{array}$$

Figure 12 Predicate semantics.

of the separating conjunction $*$, where ψ is a local store mapping from `M`-kind variables to `M`-kind values (r, n); we require $\texttt{dom}(\psi) \subseteq \texttt{dom}(\Omega)$ and `M`-kind variables modeled by M-LOCAL. Besides predicate well-formedness, the predicate semantic judgment $(\Omega;\psi;\sigma;\varphi \models_g P)$ also ensures the states (φ) being well-formed $(\Omega;\sigma \vdash_g \varphi)$, defined as follows:

▶ **Definition 1** (Well-formed QAFNY state). A state φ is *well-formed*, written as $\Omega;\sigma \vdash_g \varphi$, iff $\texttt{dom}(\sigma) = \texttt{dom}(\varphi)$, $\Omega \vdash \texttt{dom}(\sigma)$ (all variables in φ are in Ω), and:

- For every $\kappa \in \texttt{dom}(\sigma)$, s.t. $\sigma(\kappa) = \texttt{Nor}$, $\varphi(\kappa) = z\,|c\rangle\,(\!|\beta|\!)$ and $|\kappa| = |c|$ and $|z| \leq 1$; specifically, if $g = \texttt{C}$, $\beta = \emptyset$ and $|z| = 1$. [10]
- For every $\kappa \in \texttt{dom}(\sigma)$, s.t. $\sigma(\kappa) = \texttt{Had}$, $\varphi(\kappa) = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + \alpha(r_j)\,|1\rangle)$ and $|\kappa| = n$.
- For every $\kappa \in \texttt{dom}(\sigma)$, s.t. $\sigma(\kappa) = \texttt{EN}$, $\varphi(\kappa) = \sum_{j=0}^{m} z_j|c_j\rangle(\!|\beta_j|\!)$, and for all j, $|\kappa| = |c_j|$ and $\sum_{j=0}^{m} |z|^2 \leq 1$; specifically, if $g = \texttt{C}$, for all j, $\beta_j = \emptyset$ and $\sum_{j=0}^{m} |z|^2 = 1$.

Here, an `M` mode state, representing a computation living in an `M` mode context, has a relaxed well-formedness, where $\sum_{j=0}^{m} |z|^2 \leq 1$ and $\beta_j \neq \emptyset$. This is needed for describing the state inside the execution of a conditional body in rule S-IF in Section 4.2, where unmodified basis-kets are removed before the execution. There is a trick to utilizing the frozen stacks for promoting proof automation, as the modeling rule M-MAP equates two quantum values by discarding the frozen stack qubits, and we will see an example in Section 6.1.

The predicate syntax (Figure 7) introduces three locus predicate transformers $\mathcal{F}$, $\mathcal{U}$, and $\mathcal{M}$ in the locus syntax category, but their semantics (Figure 12) essentially transform quantum states in the predicates, as we define them in equational style, explained below.

Assignment and Heap Mutation Operations. Rule P-EXPC describes `C`-kind variable substitutions. Rule P-OP is a classical separation logic style heap mutation rule for state preparations $\kappa \leftarrow op$ and oracles $\kappa \leftarrow \mu$, which analogize such operations as classical array map operations mentioned in Figure 2. Here, we discuss the cases when the state of the target loci $\kappa \uplus \kappa'$ is of type `EN`, while some other cases are introduced in TR [24] C.4. Each element in the array style pre-state q, for locus $\kappa \uplus \kappa'$, represents a basis-ket $z_j\,|c_j\rangle\,\eta_j$, with $|\kappa| = |c_j|$. Here, we first locate κ's position basis $|c_j\rangle$ in each basis-ket of q, and then apply the operations *op* or μ to $|c_j\rangle$.

Quantum Conditionals. As in Section 3.4 and Figure 5, the key in designing a proof rule for a quantum conditional `if` (b) `{`e`}` with its locus scope $\kappa \uplus \kappa'$, is to encode two transformers: $\mathcal{F}$ and $\mathcal{U}$. In rule P-IF (Figure 11), we require the σ only contains locus $\kappa \uplus \kappa'$, which can be done

[10] $|\kappa|$ and $|c|$ are the lengths of κ and c, and $|z|$ is the norm.

through P-Frame. We then utilize $\mathcal{F}(b, \kappa, \kappa')$ to finish two tasks: (1) it computes b's side-effects on the κ's position bases ($[\![b]\!]^{|\kappa|}q$), and (2) it freezes all basis-kets that are irrelevant when reasoning about the body e. This freezing mechanism modeled by the equation M-$\mathcal{F}$ (Figure 12) is accomplished at two levels: stashing all kets unsatisfying b ($q\langle\kappa, \neg b\rangle$) and moving κ's position bases to basis stacks for the rest of basis-kets. After substituting $\kappa \uplus \kappa'$ for $\mathcal{F}(b, \kappa, \kappa')$, besides expelling the parts not satisfying b, we also shrink the locus $\kappa \uplus \kappa'$ to κ', which in turn marked the κ's position basis in every basis-ket inaccessible as κ is now invisible in the locus type environment.

After the body e's proof steps, the post-state Q describes the computation result for κ' without the frozen parts. To reinstate the state for $\kappa \uplus \kappa'$ by retrieving the frozen parts, we first substitute locus $\kappa \uplus \kappa'$ for $\mathcal{U}(\neg b, \kappa, \kappa \uplus \kappa')$ in P, which represents the unmodified part, unsatisfying b, in the pre-state. We also substitute κ' for $\mathcal{U}(b, \kappa, \kappa \uplus \kappa')$ in Q, which represents the part satisfying b, evolved due to the execution of e. Rule M-$\mathcal{U}$ (Figure 12) describes the predicate transformation, empowered by the locus construct $\mathcal{U}$, that utilizes the innate relation of separating conjunction and logical complement to assemble the previously unmodified and the evolved parts. Rule P-Loop proves a `for` loop where $P(j)$ is the loop invariant parameterized over the loop counter j. Other rules are introduced in TR [24] C.4.

Measurement. A measurement (`let` x `=measure(`y`) in` e) collapses a qubit array y, binds an `M` kind outcome to x and restricts its usage in e. These statements usually appear in periodical patterns in many quantum algorithms, which users formalize as predicates to help verify algorithm properties. In rule P-Mea, we first select an n-length prefix bitstring c from one of range $y[0, n)$'s position bases; it then computes the probability r and assigns $(r, \{\!|c|\!\})$ to variable x. We then replace the locus $y[0, n) \uplus \kappa$ in P with a locus predicate transformer $\mathcal{M}(x, n, \kappa)$ and update the type state Ω and σ by replacing $y[0, n) \uplus \kappa$ with κ. The construct $\mathcal{M}(x, n, \kappa)$, with its transformation rule M-$\mathcal{M}$ (Figure 12), is introduced to do exactly the two steps in Section 4.2 for describing measurement operations, i.e., we remove basis-kets not having c as $y[0, n)$'s position bases ($q\langle\kappa, c \neq \kappa\rangle$) and truncate $y[0, n)$'s position bases in the rest basis-kets.

$$\frac{\Omega[u \mapsto \mathtt{M}]; \{x[0,n) : \mathtt{EN}\} \vdash_{\mathtt{C}} \left\{\mathcal{M}(u, n, x[0,n)) \mapsto C\right\} \{\} \left\{x[0,n) \mapsto D * E\right\}}{\Omega; \{y[0,n) \uplus x[0,n) : \mathtt{EN}\} \vdash_{\mathtt{C}} \left\{y[0,n) \uplus x[0,n) \mapsto C\right\} \mathtt{let}\ u\, \mathtt{=measure(}y\mathtt{)\ in}\ \{\} \left\{x[0,n) \mapsto D * E\right\}}$$

$$C \triangleq \sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |a^j \,\%\, N\rangle\, |j\rangle \quad D \triangleq \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |t + kp\rangle \quad E \triangleq p = \mathtt{ord}(a, N) * u = (\tfrac{p}{2^n}, a^t \,\%\, N) * r = \mathtt{rnd}(\tfrac{2^n}{p})$$

We show a proof fragment above for the partial measurement in Figure 6 line 8. The proof applies rule P-Mea by replacing locus $y[0, n) \uplus x[0, n)$ with $\mathcal{M}(u, n, x[0, n))$. On the top, the pre- and post-conditions are equivalent, as explained below. In locus $y[0, n) \uplus x[0, n)$'s state, for every basis-ket, range $y[0, n)$'s position basis is $|a^j \,\%\, N\rangle$; the value j is range $x[0, n)$'s position basis for the same basis-ket. Randomly picking a basis value $a^t \,\%\, N$ also filters a specific j in range $x[0, n)$, i.e., we collect any j having the relation $a^j \,\%\, N = a^t \,\%\, N$. Notice that modulo multiplication is a periodic function, which means that the relation can be rewritten $a^{t+kp} \,\%\, N = a^t \,\%\, N$, and p is the period order. Thus, the post-measurement state for range $x[0, n)$ can be rewritten as a summation of k: $\frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |t + kp\rangle$. The probability of selecting $|a^j \,\%\, N\rangle$ is $\frac{r}{2^n}$. In the Qafny implementation, we include additional axioms for these periodical theorems to grant this pre- and post-condition equivalence so that we can utilize Qafny to verify Shor's algorithm.

4.5 Qafny Metatheory

We now present Qafny's type soundness and its proof system's soundness and relative completeness. These results have all been verified in Coq. We prove our type system's soundness with respect to the semantics, assuming well-formedness (TR [24] Definition 5 and Definition 1). The type soundness shows that our type system ensures the three properties in Section 4.3 and that the "in-place" style Qafny semantics can describe all different quantum operations without losing generality because we can always use the equivalence rewrites to rewrite the locus state in ideal forms.

▶ **Theorem 2** (Qafny type soundness)**.** If $\Omega;\sigma \vdash_g e \triangleright \sigma'$ and $\Omega;\sigma \vdash_g \varphi$, then there exists φ' such that $(\varphi, e) \Downarrow \varphi'$ and $\Omega;\sigma' \vdash_g \varphi'$.

Our proof system is sound and relatively complete w.r.t. its semantics for well-typed Qafny programs. Our system utilizes a subset of separation logic admitting completeness by excluding qubit array allocation and pointer aliasing. Since every quantum program in Qafny converges, the soundness and completeness refer to the total correctness of the Qafny proof system. $\psi(e)$ refers to that we substitute every variable $x \in \mathtt{dom}(\psi)$ in e with $\psi(x)$.

▶ **Theorem 3** (proof system soundness)**.** For any program e, such that $\Omega;\sigma \vdash_g e \triangleright \sigma'$ and $\Omega;\sigma \vdash_g \{P\}\, e\, \{Q\}$, and for every ψ and φ, such that $\Omega;\sigma \vdash_g \varphi$ and $\Omega;\psi;\sigma;\varphi \models_g P$, there exists a state φ', such that $(\varphi, \psi(e)) \Downarrow \varphi'$ and $\Omega;\sigma' \vdash_g \varphi'$ and $\Omega;\psi;\sigma';\varphi' \models_g Q$.

▶ **Theorem 4** (proof system relative completeness)**.** For a well-typed program e, such that $\Omega;\sigma \vdash_g e \triangleright \sigma'$, $(\varphi, e) \Downarrow \varphi'$ and $\Omega;\sigma \vdash_g \varphi$, and for all predicates P and Q such that $\Omega;\emptyset;\sigma;\varphi \models_g P$ and $\Omega;\emptyset;\sigma';\varphi' \models Q$, we have $\Omega;\sigma \vdash_g \{P\}\, e\, \{Q\}$.

5 Qafny Compilation and Implementation Evaluation

Here, we focus on the Qafny proof system compilation to a subset of separation logic.

5.1 Translation from Qafny to Separation Logic

The Qafny types and loci are extra annotations associated with Qafny programs and predicates for proof automation. In the Qafny implementation in Dafny, these annotations are not present – qubits are arranged as simple array structures without extra metadata such as locus types. This section shows how Qafny annotations can be safely erased with no loss of expressiveness, e.g., loci are represented as virtual-level dynamic sequences without types, and equational rewrites are compiled with extra operations in the compiled language. We present a compilation algorithm that converts from Qafny to Sep, a C-like language admitting a subset of an array separation logic proof system.

Target Compilation Language. Sep is based on a variant of the separation logic introduced by Yang and O'Hearn [55], which is sound and complete. Mainly, we utilize the allocation (`alloc`), heap lookup and mutation operations (`mutate`) in the work, with three additional operations (marked red in Figure 13). In Sep, program states are divided into virtual and physical levels in Figure 14. Every Sep program starts with a physical qubit array, analogous to physical heap structures. The program operations are applied to qubit sequences of array indices (A), representing a collection of physical qubit locations that live at the virtual level. Sep permits immutable program variables (x) representing these sequences.

$$
\begin{array}{lll}
A, B & ::= & \overline{n} \\
\widetilde{op} & ::= & \mu \mid op \\
\widetilde{e} & ::= & x = \texttt{alloc}(A) \\
& \mid & \texttt{mutate}(n, \widetilde{op}, x) \\
& \mid & (r, n) = \texttt{pick}(x, m) \\
& \mid & \texttt{filter}(x, b) \\
& \mid & \texttt{amp}(x, r) \\
& \mid & \ldots
\end{array}
$$

Figure 13 Sep Syntax.

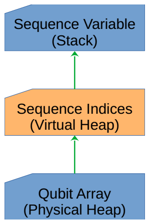


Figure 14 Heap Layout.

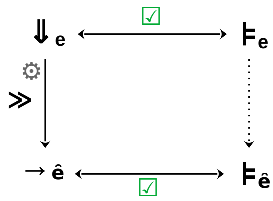


Figure 15 Compilation Proof Diagram.

An allocation $x = \texttt{alloc}(A)$ allocates a new array x that copies A's content and has the same length as A, while a heap mutation $\texttt{mutate}(n, \widetilde{op}, x)$ mutates the first n elements of the array pointed to by x, by applying the operation $\widetilde{op}$. Operations pick, filter, and amp are variations of heap lookups and mutations. $(r, n) = \texttt{pick}(x, m)$ measures the first m qubits in x, with the outcome $n = \{\!|c|\!\}$ and its probability r, similar to the computation in S-Mea (Figure 8). $\texttt{filter}(x, b)$ mutates x's pointed-to quantum value by filtering out basis-kets that are not satisfying b, and $\texttt{amp}(x, r)$ multiples r to every basis-kets in x's quantum value. Yang and O'Hearn maintain completeness by carefully designing the proof rules so heap mutations do not modify pointer references. Sep ensures the same property by immutable variables, i.e., if a sequence changes, we allocate a new array and a new variable pointing to the array. For example, to join two loci represented by two sequences A and B, respectively pointed to by variables x and y, we allocate a new space for the two sequences' concatenation $(A@B)$; so we compile the join to $u = \texttt{alloc}(A@B)$. We must abandon using x and y after the join and only refer to u in the following computation.

Compilation Procedure. As shown in Figure 15, we compile the Qafny language to Sep and achieve the proof system compilation through the proof system completeness in Qafny and Sep. For every Qafny program, we translate the program and states to Sep. Then, every provable triple in Qafny can be translated to a provable Sep triple through the language translation from Qafny to Sep as the diagram (Figure 15). The compilation is defined by extending Qafny's typing judgment thusly: $\Omega; \sigma \vdash_g (\theta, \varphi, e) \gg (\theta, \widetilde{\psi}, \widetilde{\varphi}, \widetilde{e})$. We include an initial Qafny state φ, the output local store $(\widetilde{\psi})$, mapping variables to qubit location sequences, and state $(\widetilde{\varphi})$, mapping from locations to quantum values. θ and θ' are maps from locus locations in φ to Sep qubit locations in $\widetilde{\varphi}$.

Here, we explain the rules for compilation by examples of compiling the Qafny operations to Sep. The locus rewrites (Section 4.3) are compiled to the array allocations, such as the join operation above. Additionally, the split of a locus is compiled to two consecutive allocations of two sequences, respectively representing the two split result loci. In compiling a measurement statement (let x = measure(y) in ...), where y locates in the locus $y[0, n) \uplus \kappa$, let's assume that the locus is mapped in θ by sequence $[0, n+m)$, pointed to by u in $\widetilde{\psi}$, while the range $y[0, n)$ is mapped by the sequence $[0, n)$; the operation is compiled to:

$$(r, p) = \texttt{pick}(u, m) \,;\, \texttt{filter}(u, u[0, n) = p) \,;\, \texttt{amp}(u, \sum_j \frac{z_j}{\sqrt{r}}) \,;\, t = \texttt{alloc}([n, n{+}m))$$

We first pick a key p, filter out the basis-kets whose $u[0, n)$'s position bases are not p, normalize the amplitudes of the remaining basis-kets (Section 4.4), and allocate a new space t for the quantum residue after the measurement. We also update κ in θ to map to the newly allocated space of t instead of $[n, n{+}m)$. We compile an operation $x[0, n) \uplus y[0] \leftarrow (x < 5) \; @ \; y[0]$ with its initial state φ $(C = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + |1\rangle))$ to Sep, with $D = \sum_{j=0}^{2^n-1} |j\rangle\, |j < 5\rangle$. Such an operation computes the Boolean comparison of $x < 5$ and stores the value to $y[0]$.

`Algorithm`	Qafny		QBricks		SQIR	
	`Runtime (sec)`	`LOC`	`Runtime (sec)`	`LOC`	`Runtime (sec)`	`LOC`
`GHZ`	14.2	16	-	-	141	119
`Deutsch-Jozsa`	8.3	13	74	108	163	408
`Grover's search`	26.7	27	253	233	148	1018
`Shor's algorithm`	36.3	36	1328	1163	1244	8464

Figure 16 Running time (include theory loading) & LOC comparison, in an i7 Ubuntu Mach. 8G RAM; `-`: no data.

`Algorithm`	`Runtime (sec)`	`LOC`
`Controlled GHZ`	6.4	12
`Quantum Walk`	43.1	49

Figure 17 QAFNY data for case studies in Section 6.

$$\begin{array}{l} \varphi = \{x[0,n) : C, y[0] : |0\rangle\} \\ \quad x[0,n) \uplus y[0] \leftarrow (x < 5) \,@\, y[0] \\ \varphi' = \{x[0,n) \uplus y[0] : D\} \end{array} \quad \gg \quad \begin{array}{l} \widetilde{\psi} = \{p : [0,n), t : \{n\}\}, \widetilde{\varphi} = \{[0,n) : C * \{n\} : |0\rangle\} \\ \quad u = \mathtt{alloc}([0,n{+}1))\,;\,\mathtt{mutate}(n{+}1, u[0,n) < 5@u[n], u) \\ \widetilde{\psi}' = \widetilde{\psi} \cup \{u : [n+1,\ 2n+2)\}, \widetilde{\varphi}' = \widetilde{\varphi} \cup \{[n+1,\ 2n+2) : D\} \end{array}$$

After the compilation, we create two local variables p and t to represent the loci x and y, mapping to sequences $[0, n)$ and $\{n\}$. We then add $u = \mathtt{alloc}([0, n+1))$ allocating a new space $[n{+}1, 2n+2)$ to join the two loci. The post-state contains a new variable u, pointing to the concatenated new sequence $[n+1, 2n+2)$. The old arrays p and t are still in the stores, but we refer to the locus $x[0, n) \uplus y[0]$ as the newly allocated space in the following computation by mapping the locus to the new space in θ'. As a future work, we will prove the proof system compilation correctness from QAFNY to SEP, proof strategy in Figure 15.

5.2 Implementation and Comparison to Existing Quantum Verifications

We have implemented a prototype QAFNY to Dafny compiler, which faithfully respects the presented QAFNY to SEP compilation algorithm. To validate the soundness of the compiler implementation, we create many test cases for the compiler. We then insert a number of bugs in these test cases; Qafny has been able to detect all of them. Dafny's proof engine cannot be used to verify arbitrary separation-logic assertions because it only has an implicit frame rule implementation that allows users to set up variables that can be modified in a function. However, QAFNY only requires a subset of separation logic, and the QAFNY loci disjoint property and non-aliasing guarantee ensures that the QAFNY separation conjunctions are captured by Dafny's implicit frames when we compile QAFNY to Dafny. We utilize the QAFNY to Dafny compiler as an automated verification framework to verify six quantum programs, shown in Section 6 and Figure 17.

There were two main approaches to verifying quantum programs: program and measurement-based. The former views quantum program transitions as a state machine and verifies the inductive relations among transitions, while the latter focuses on the relations between quantum program measurements and the post-processing classical components – they typically view quantum components as black boxes with specifications.

QAFNY is program-based and other program-based mechanized frameworks for formally verifying quantum programs, including Qwire [41], SQIR [16, 15] (an upgrade of Qwire), and QBricks [3], provide libraries in an interactive proof assistant to guide users for building inductive proofs for quantum programs; each has verified 7-10 quantum programs. The core of SQIR and QBricks, as well as other executable quantum verification platforms, is a circuit-description language. Verifying a program in these frameworks inductively builds a unitary or density matrix as the program's quantum circuit semantic interpretation. For example, to verify Shor's algorithm, both QBricks and SQIR require inductive proofs based on elementary circuit gates to derive the unitary or density matrix semantics of different subcomponents, including state preparations, oracles, and $\mathtt{QFT}^{-1}$ gates. Additionally, program verification in these frameworks requires the development of theories and tactics to capture program properties, which usually involves the proof of additional theorems. This approach is qualitatively different from Qafny, where program verification involves embedding assertions in a program for completing a proof. Qafny identifies a few program structures, such as oracles and quantum conditionals, and formulates inductive patterns involving these quantum components as proof rules. These rules interact with quantum program operations and states for deriving verification proofs, so proofs are largely automated based on this small set of structures.

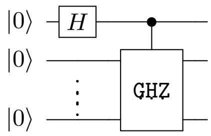
Figure 16 shows a quantitative comparison of Qafny, regarding theory/proof statement running time and numbers of lines (LOC), with respect to QBricks and SQIR for verifying several quantum programs. The results show that Qafny has the shortest running time and LOCs for verifying programs with our automated proof engine. The QBricks verification has better LOCs but a similar running time compared to SQIR. SQIR provides a complete verification [38] by proving every mathematical theorem involved in the verification, so its verification proofs are longer than QBricks; QBricks performs better by providing some automatic tactics for sequence operations (e ; e) and taking many math theorems as assumptions without proof. In testing the two frameworks, we found that the previous claim [16, 15] that rigorous quantum proofs are one-time cost is problematic because inductive theorem provers update constantly. Once an update happens, users might need to fix the proofs (not programs or specifications) in their history code, e.g., our researcher spent three days fixing minor bugs in the proofs in SQIR and QBricks due to Coq and Why3 version issues. Moreover, a new program verification in SQIR typically required detailed proofs of additional theorems beyond the program specifications.

Qafny provides fast prototyping, where we apply the automated verification mechanisms in many classical systems [40, 22] to verify quantum programs and save programmers' effort. Verifying programs in many inductive theorem provers takes weeks and months to finish, while the same tasks in Qafny cost researchers a few days due to the Qafny features mentioned above. The fast prototyping in Qafny can also help users to explore and understand new quantum program patterns such as the two case studies in Section 6, whose running time and LOCs are shown in Figure 17. Compared to the data of well-known algorithms in Figure 16, the data for verifying the new programs do not show a significant difference, showing Qafny's ability to explore new algorithm behaviors without proving many new theories, which usually appears in the above quantum verification frameworks.

6 Case Studies

With two examples, we show Qafny as a rapid prototyping tool for quantum programs.

6.1 Controlled GHZ: Composing Quantum Algorithms from Others



1 $\{ x[0] \mapsto |0\rangle , y[0,n) \mapsto |\overline{0}\rangle \}$
2 $x[0] \leftarrow$ `H;`
3 $\{ x[0] \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) * y[0,n) \mapsto |\overline{0}\rangle \}$
4 $\Rightarrow \{ x[0] \uplus y[0,n) \mapsto \sum_{d=0}^{1} |d\rangle|\overline{0}\rangle \}$
5 `if (`$x[0]$`) ghz(`y`);`
6 $\{ x[0] \uplus y[0,n) \mapsto \frac{1}{\sqrt{2}} |\overline{0}\rangle + \frac{1}{2} |1\rangle |\overline{0}\rangle + \frac{1}{2} |\overline{1}\rangle \}$

Figure 18 Controlled GHZ circuit and proof. ghz(y) is in Fig. 3. Lines 3-4 automatically inferred.

Automated verification frameworks such as Dafny encourage programmers to build program proofs based on the reuse of subprogram proofs. However, this perspective is more or less overlooked in previous quantum proof systems. In SQIR, for example, verifying the correctness of a *controlled GHZ*, a simple circuit constructed by extending GHZ with an extra control qubit, requires generalizing the GHZ circuit to any arbitrary inputs. In QAFNY, users do not need to do this, as shown here.

Figure 18 provides a proof of the Controlled GHZ algorithm based on a proven GHZ method in Figure 3c. The focal point is the quantum conditional on line 5. For verifying a GHZ circuit, its input is an n-qubit `Nor`-typed value of all $|0\rangle$, but the given value, in line 4, is an `EN`-typed entanglement $\sum_{d=0}^{1} |d\rangle|\overline{0}\rangle$. Here is where SQIR gets stuck. In QAFNY, we automatically verify the proof by rule P-IF and the equivalence relation to rewrite a singleton `EN` value to a `Nor` one, as $\sum_{j=0}^{0} z_j|c_j\rangle \equiv z_0\,|c_0\rangle$. The detailed proof for the conditional is given below, where $U(b) = \mathcal{U}(b, x[0], \kappa)$ and $\kappa = x[0] \uplus y[0, n)$.

$$\dfrac{\dfrac{\Omega; \{y[0,n) : \texttt{Nor}\} \vdash_{\texttt{M}} \left\{y[0,n) \mapsto |\overline{0}\rangle\overset{\frown}{|1\rangle}\right\} \texttt{ghz}(y) \left\{y[0,n) \mapsto \sum_{d=0}^{1} \frac{1}{\sqrt{2}}\,|\overline{d}\rangle\overset{\frown}{|1\rangle}\right\}}{\Omega; \{y[0,n) : \texttt{EN}\} \vdash_{\texttt{M}} \left\{\mathcal{F}(x[0], y[0,n)) \mapsto \sum_{d=0}^{1} |d\rangle|\overline{0}\rangle\right\} \texttt{ghz}(y) \left\{y[0,n) \mapsto \sum_{d=0}^{1} \frac{1}{\sqrt{2}}\,|\overline{d}\rangle\overset{\frown}{|1\rangle}\right\}}\ \text{EQ}}{\dfrac{\Omega; \{\kappa : \texttt{EN}\} \vdash_{\texttt{C}} \left\{\kappa \mapsto \sum_{d=0}^{1} |d\rangle|\overline{0}\rangle\right\} \texttt{if}\ (x[0])\ \texttt{ghz}(y) \left\{U(\neg x[0]) \mapsto \sum_{d=0}^{1} |d\rangle|\overline{0}\rangle * U(x[0]) \mapsto \sum_{d=0}^{1} \frac{1}{\sqrt{2}}\,|\overline{d}\rangle\overset{\frown}{|1\rangle}\right\}}{\Omega; \{\kappa : \texttt{EN}\} \vdash_{\texttt{C}} \left\{\kappa \mapsto \sum_{d=0}^{1} |d\rangle|\overline{0}\rangle\right\} \texttt{if}\ (x[0])\ \texttt{ghz}(y) \left\{\kappa \mapsto \frac{1}{\sqrt{2}}\,|\overline{0}\rangle + \frac{1}{2}\,|1\rangle\,|\overline{0}\rangle + \frac{1}{2}\,|\overline{1}\rangle\right\}}}\ \text{P-IF}$$

After rule P-IF is applied, locus $y[0, n)$'s value is rewritten to a `Nor` type value on the top as $|\overline{0}\rangle\overset{\frown}{|1\rangle}$, where $|\overline{0}\rangle$ is n qubits and $\overset{\frown}{|1\rangle}$ is frozen in the stack. Since two values are equivalent as QAFNY discards stacks, $|\overline{0}\rangle\overset{\frown}{|1\rangle}$ is equivalent to $|\overline{0}\rangle$, which satisfies the input condition for GHZ, so all proof obligations introduced to invoke the `ghz` method are discharged.

6.2 Case Study: Understanding Quantum Walk

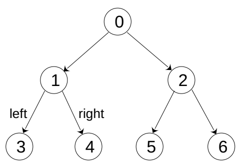


Figure 19 Tree Structure.

Quantum walk [54, 4, 53] is a quantum version of the classical random walk [36] and an important framework for developing quantum algorithms. However, most quantum walk analyses are based on Hamiltonian simulation, which deters many computer programmers from the quantum walk framework. Here, we show that the discrete-time quantum walk, at its very least, is a quantum version of breadth-first search; thus, many algorithms [4] based on Quantum walk can be understood as performing search algorithms in the quantum setting.

Figure 20 lists the proof outline for the core loop of a discrete-time quantum walk algorithm to traverse a complete binary tree (structure in Figure 19); each node has a unique key. The m-depth nodes in the tree have keys $j \in [2^m - 1, 2^{m+1} - 2)$, which form a sequence

$q(j) \triangleq \sum_{i=0}^{2^{(j+1)}-3} z_i \,|\lfloor \log (i{+}1)\rfloor\rangle\,|\texttt{pat}(i,j)\rangle\,|i{+}1\rangle\,|d_i\rangle$ where $\texttt{pat}(i,j) \triangleq |0\rangle^{\otimes \lfloor \log (i+1)\rfloor}\,|1\rangle^{\otimes (j - \lfloor \log (i+1)\rfloor)}$

1 $\left\{x[0,t) \mapsto \frac{1}{\sqrt{2^t}}\bigotimes_{i=0}^{t-1}(|0\rangle+|1\rangle) * y[0,m) \mapsto |\overline{0}\rangle * h[0,n) \mapsto |\overline{0}\rangle * u[0] \mapsto |0\rangle * m = 2^t \cdot m < n\right\}$

2 $\Rightarrow \left\{x[0,t) \uplus y[0,0) \uplus h[0,n) \uplus u[0] \mapsto \sum_{k=0}^{2^t-1}\frac{1}{\sqrt{2^t}}|k\rangle\,|\overline{0}\rangle\,|0\rangle * y[0,m) \mapsto |\overline{0}\rangle * m = 2^t\cdot m < n\right\}$

3 `for` $j \in [0,m)$ `&&` $(x[0,t) < j+1)$ `@` $y[j]$

4 $\left\{x[0,t)\uplus y[0,j)\uplus h[0,n)\uplus u[0] \mapsto q(j) + \sum_{k=j}^{2^t-1}\frac{1}{\sqrt{2^t}}|k\rangle\,|\overline{0}\rangle\,|\overline{0}\rangle\,|0\rangle * y[j,m)\mapsto|\overline{0}\rangle\right\}$

5 { $u[0] \leftarrow$ `H`;

6 `if` ($u[0]$) `left`($\lfloor\log(j{+}1)\rfloor$, $h[0,n)$);

7 `if` ($\neg u[0]$) `right`($\lfloor\log(j{+}1)\rfloor$, $h[0,n)$); }

8 $\left\{x[0,t)\uplus y[0,m)\uplus h[0,n)\uplus u[0]\mapsto q(m)\right\}$

Figure 20 Quantum walk reachable node verification for a complete binary tree. `left` and `right` reach corresponding children. $q(j)$ is a quantum value with var j. $i+1$ is a node key in a tree.

from left to right in depth m-th, such as the sequence $3,4,5,6$ in depth 2 in Figure 19. Thus, a node (key j) has a depth $m = \lfloor \log (j{+}1)\rfloor$, and its `left` and `right` children have keys $2\cdot j+1$ and $2\cdot j+2$, respectively representing the `left` and `right` operation semantics in Figure 20, i.e., for any basis $|j\rangle$, with m being the depth and j a node key, the outputs of applying `left` and `right` are $|2\cdot j+1\rangle$ and $|2\cdot j+2\rangle$, respectively [11].

The algorithm in Figure 20 requires four quantum ranges: a t-qubit range x in superposition, an m-qubit range where $y[j]$'s position bases keep the result of evaluating $x[0,t) < j+1$ for j-th loop step, an n-qubit node register h storing the node keys, and a single qubit u acting as the random walk coin and determining the moving direction of the next step (1 for the left and 0 for the right). In line 2, we merge the ranges x, u, and h as the locus $x[0,t)\uplus y[0,0)\uplus h[0,n)\uplus u[0]$ ($y[0,0)$ is empty); at each loop step (lines 3-7), we entangle a qubit in the range y into the locus. Finally, at line 8, the loop program entangles all these ranges together as a locus $x[0,t)\uplus y[0,m)\uplus h[0,n)\uplus u[0]$.

In the j-th loop step, we abbreviate locus $x[0,t)\uplus y[0,j)\uplus h[0,n)\uplus u[0]$ as $\kappa\langle j\rangle$, and locus $\kappa\langle j\rangle$'s state value is split into two basis-ket sets, separated by $+$ in Figure 20 line 4. To verify a step, we first split the $y[j]$ qubit, having position basis $|0\rangle$, from range $y[j,m)$, and merge the qubit into $\kappa\langle j\rangle$. The split rewrites are given as:

$$\left\{\kappa\langle j\rangle \mapsto \sum_{i=0}^{2^{(j+1)}-3} z_i\,|\lfloor\log(i{+}1)\rfloor\rangle\,|\texttt{pat}(i,j)\rangle\,|i{+}1\rangle\,|d_i\rangle + \sum_{k=j}^{2^t-1}\frac{1}{\sqrt{2^t}}|k\rangle\,|\overline{0}\rangle\,|\overline{0}\rangle\,|0\rangle * y[j,m)\mapsto|\overline{0}\rangle\right\}$$
$$\equiv \left\{\kappa\langle j{+}1\rangle \mapsto \sum_{i=0}^{2^{(j+1)}-3} z_i\,|\lfloor\log(i{+}1)\rfloor\rangle\,|\texttt{pat}(i,j)\rangle\,|0\rangle\,|i{+}1\rangle\,|d_i\rangle + \sum_{k=j}^{2^t-1}\frac{1}{\sqrt{2^t}}|k\rangle\,|\overline{0}\rangle\,|0\rangle\,|\overline{0}\rangle\,|0\rangle * y[j{+}1,m)\mapsto|\overline{0}\rangle\right\}$$
$$\equiv \left\{\kappa\langle j{+}1\rangle \mapsto q'(|0\rangle) + \sum_{k=j}^{2^t-1}\frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle * y[j{+}1,m)\mapsto|\overline{0}\rangle\right\}$$

At the last rewrite above, we abbreviate the first part of $\kappa\langle j{+}1\rangle$'s value to be $q'(|0\rangle)$, and the second part to be $\sum_{k=j}^{2^t-1}\frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle$ where $\delta\langle k,c\rangle = |k\rangle\,|\overline{0}\rangle\,|c\rangle\,|\overline{0}\rangle\,|0\rangle$. Below, we show the proof steps (only the pre-condition transitions) for a conditional step in Figure 20 line 3, which can be divided into two small steps. Here, e is the conditional body in lines 5-7, and we apply P-Frame to frame out locus $y[j{+}1,m)$ from the states, so the bottom state only refers to the locus $\kappa\langle j{+}1\rangle$. We further split the second part above ($\sum_{k=j}^{2^t-1}\frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle$) into two basis-ket sets: $\frac{1}{\sqrt{2^t}}\delta\langle j,0\rangle$ and $\sum_{k=j+1}^{2^t-1}\frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle$.

[11] The tree structure is a simplification; comprehensive implementations use Szegedy walk encoding [30].

$$\dfrac{\dfrac{\Omega; \{h[0,n) \uplus u[0] : \texttt{EN}\} \vdash_{\texttt{M}} \left\{h[0,n) \uplus u[0] \mapsto \sum_{i=0}^{2^{(j+1)}-3} z_i \,|i{+}1\rangle\, |d_i\rangle\, \widehat{\beta} + \frac{1}{\sqrt{2^t}} \,|\overline{0}\rangle\, |0\rangle\, \widehat{\beta'}\right\} e \left\{\dots\right\}}{\Omega; \{\kappa\langle j{+}1\rangle : \texttt{EN}\} \vdash_{\texttt{M}} \left\{\kappa\langle j{+}1\rangle \mapsto q'(|1\rangle) + \frac{1}{\sqrt{2^t}}\delta\langle j,1\rangle + \sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle\right\} \texttt{if}\ ((x[0,t) < j+1)\ \texttt{@}\ y[j])\ e \left\{\dots\right\}}}{\Omega; \{\kappa\langle j{+}1\rangle : \texttt{EN}\} \vdash_{\texttt{M}} \left\{\kappa\langle j{+}1\rangle \mapsto q'(|0\rangle) + \frac{1}{\sqrt{2^t}}\delta\langle j,0\rangle + \sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle\right\} \texttt{if}\ ((x[0,t) < j+1)\ \texttt{@}\ y[j])\ e \left\{\dots\right\}}$$

We split the P-If proof step (Line 3 in Figure 20) into two small steps above. The bottom step represents the first half of the $\mathcal{F}$ transformer application (Figure 11) in P-If. It views the Boolean guard $(x[0,t) < j+1)$ @ $y[j]$ as an oracle application and for every basis-ket in the locus $\kappa\langle j{+}1\rangle$, we compute the Boolean value $x[0,t) < j+1$ and store it to $y[j]$'s position bases. Unlike the simple Boolean guards appearing in Figures 3c and 6, the Boolean guard here has side-effects that modify $y[j]$'s position bases. $\kappa\langle j{+}1\rangle$'s value is split into three basis-ket sets separated by $+$. In the set $q\langle 0\rangle$, range $x[0,t)$'s position bases have the form $|\lfloor \log\,(i{+}1)\rfloor\rangle$ (the depth of a node key $i{+}1$) and the bases' natural number interpretations are smaller than $j{+}1$; in the sets $\frac{1}{\sqrt{2^t}}\delta\langle j,0\rangle$ and $\sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle$, range $x[0,t)$'s position bases are $|j\rangle$ and $|k\rangle$ $(j < k)$. The former's natural number interpretation is less than $j{+}1$, while the latter is not. Thus, we flip $y[j]$'s position bases of the two sets after applying the bottom rule above while leaving the third set unchanged.

The middle step in the above P-If proof step rules out the basis-ket set $\sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}}\delta\langle k,0\rangle$, because the $y[j]$'s position bases are all 0; then, we push bases $|\lfloor \log\,(i{+}1)\rfloor\rangle\, |\texttt{pat}(i,j)\rangle\, |0\rangle$ and $|k\rangle\, |\overline{0}\rangle\, |0\rangle$ to the frozen stacks as $\widehat{\beta}$ and $\widehat{\beta'}$, respectively for the remaining two sets.

$$\begin{array}{l}
\Omega; \{u[0] \uplus h[0,n) : \texttt{EN}\} \vdash_{\texttt{M}} \\
\{u[0] \uplus h[0,n) \mapsto \sum_{i=0}^{2^{(j+1)}-3} z_i \,|d_i\rangle\, |i{+}1\rangle\, \widehat{\beta} + \frac{1}{\sqrt{2^t}} \,|0\rangle\, |\overline{0}\rangle\, \widehat{\beta'}\} \\
\quad u[0]\ \leftarrow\ \texttt{H} \\
\{u[0] \uplus h[0,n) \mapsto \sum_{i=0}^{2^{(j+1)}-3} \frac{1}{\sqrt{2}} z_i \,|d_i\rangle\, |i{+}1\rangle\, \widehat{\beta} + \sum_{i=0}^{2^{(j+1)}-3} \frac{1}{\sqrt{2}} z_i \,|d_i{+}1\rangle\, |i{+}1\rangle\, \widehat{\beta} + \frac{1}{\sqrt{2^{t+1}}} \,|0\rangle\, |\overline{0}\rangle\, \widehat{\beta'} + \frac{1}{\sqrt{2^{t+1}}} \,|1\rangle\, |\overline{0}\rangle\, \widehat{\beta'}\}
\end{array}$$

For the H application in line 5 (Figure 20), we first rewrite the locus, in the pre- and post-states, from $h[0,n) \uplus u[0]$ to $u[0] \uplus h[0,n)$; shown as the proof triple above. There is a hidden uniqueness assumption [12] for all basis-kets in $q(j)$ (Figure 20): $\forall z\beta\, |d_i\rangle \in q(j) \Rightarrow \forall z'\,.\, z'\beta\, |d_i{+}1\rangle \notin q(j)$, i.e., if we truncate the $u[0]$ qubit, every basis is still unique in $q(j)$. For each basis-ket, the H application duplicates the non-$u[0]$ part, with the flip of $u[0]$'s position bases. During the process, the amplitude of each basis-ket is reduced by $\frac{1}{\sqrt{2}}$. The `left` and `right` in lines 6 and 7 then move the node key (range $h[0,n)$'s position basis) of each basis-ket to its `left` and `right` child, depending on the coin bit stored as $u[0]$'s position basis; thus, the uniqueness property is preserved (left and right children always have different keys). Remember that the number of basis-kets is doubled in the H gate application; after the j-th loop step, all $(j{-}1)$-th depth nodes become j-th depth and the two root node basis-kets $(\frac{1}{\sqrt{2^{t+1}}}\,|0\rangle\,|\overline{0}\rangle\widehat{\beta'}$ and $\frac{1}{\sqrt{2^{t+1}}}\,|1\rangle\,|\overline{0}\rangle\widehat{\beta'})$ become 1-st depth nodes; thus, the state, after j-th loop step, is in superposition containing all nodes up to j-th depth, except the root node.

The above applications also show the necessity of frozen basis stacks. When applying the conditional, we hide $x[0,t)$'s position bases to frozen basis stacks, and there are $2^{(j+1)}-3$ different such stacks. We need to record the position bases in the frozen basis stacks; so, 1) when we apply the H gate, we know which basis-kets are associated with a specific position basis; 2) once the conditional is over, the position bases can be retrieved.

The verification in Figure 20 describes the basic property of the quantum walk algorithm framework. The biggest advantage of the framework is to permit the manipulation of different quantum applications on different tree nodes in each loop step, which is why many algorithms [5, 4, 28, 1] have been developed based on it.

[12] In the Dafny implementation, this needs to be an explicit assumption given by the users.

7 Related Work

This section gives related work beyond the discussion in Section 5.2.

Measurement-based Quantum Proof Systems. Except for the works in Section 5.2, previous quantum proof systems are measurement-based, including quantum Hoare logic [56, 26, 10, 57], quantum separation logic (QSL) [20, 59], quantum relational logic [25, 52], and probabilistic Hoare logic for quantum programs [19], informed the Qafny development. They differ from Qafny in three main respects, however: 1) their conditionals are solely classical, while Qafny has quantum conditionals; 2) they mainly focus on the probabilistic relations between the quantum measurement results and classical components and view quantum program components as black boxes specified by Hilbert spaces or density matrices; and 3) most of them have no mechanized implementations, and they do not have a quantum program compiler. The verification procedure in these frameworks, to some extent, shows the possibility of verifying mainly hybrid classical-quantum (HCQ) programs by requiring the input of black-boxed and verified quantum program components.

QSL [20] develops a new separation logic theory (with no executable proof examples, however) for Hilbert spaces and classical controls, mainly for verifying HCQ programs by black-boxing quantum components. This differs from the Qafny system, based on classical separation logic for classical array operations. QSL is based on a notion of frame rules that split a tensor product state into two parts, similar to our `Had` typed state split equation. However, they do not specify when and how a quantum state separation may happen. As in Section 3, in many cases, quantum state separation is not trivial and might not be automatically inferred by a proof engine.

Liu et al. [26] contains an example verification for Grover's search algorithm based on the SQIR inductive verification style, albeit with worse proof automation (3184 LOC vs. 1018 LOC in Figure 16). CoqQ [60] provides a mechanized automated verification framework for HCQ programs. However, their proof automation is to connect quantum and classical components, i.e., they view quantum circuit components as black-boxes. By giving pre- and post-conditions, they perform proof automation on verifying HCQ programs that view the quantum components as sub-procedures. There are some examples in CoqQ to verify quantum components, but they are handled in the same style as SQIR and QBricks above. See TR [24] D.

Classical Proof Systems. We are informed by separation logic, as articulated in the classic paper [42], and other papers as well [18, 50, 34, 58, 27, 46]. Primarily, we show a compilation from Qafny to Sep, representing a subset of separation logic admitting sound and completeness [55], which was also studied by [51, 9]. The Qafny implementation is compiled to Dafny [21], a language designed to simplify writing correct code. The natural proof methodology [27, 37, 29] informs the Qafny development, where it embeds the proofs of data-structures to a recursive search problem.

8 Conclusion and Future Work

We present Qafny, a system for expressing and automatically verifying quantum programs that can be compiled into quantum circuits. We develop a proof system that views quantum operations as classical array aggregate operations, e.g., viewing quantum measurements as array filters, so that we can map the proof system, which is sound and complete with respect to the Qafny semantics for well-typed programs, to classical verification infrastructure. We implement a prototype compiler in Dafny and use it to verify several quantum programs. We believe that programmers can utilize Qafny to develop quantum algorithms and verify them through our automated verification engine with a significant saving of human effort, as demonstrated in Section 6. The Qafny language is universal in terms of the power of expressing quantum programs since all gates in the universal RzQ gate set $\{\texttt{H}, \texttt{X}, \texttt{RZ}, \texttt{CNOT}\}$ [33] are definable. However, being able to define all possible quantum programs does not mean that we can utilize Qafny to verify all quantum programs, especially HCQ programs, automatically. Verifying HCQ programs requires the full power of quantum mixed states, i.e., users might want to know the probabilistic output of executing a quantum program with a quantum input state being associated with a probability. Verifying all such programs requires a powerful classical probability distribution library beyond the current scope of Qafny, although the existing Qafny implementation does include a restricted library for reasoning about probability distributions [32] that can verify some HCQ programs.

In future work, we plan to build and verify a complete Qafny implementation in Dafny; especially, we intend to enhance the probability distribution libraries to automatically verify more HCQ programs. We also want to show the soundness proof of the implementation as well as the circuit compilation correctness from Qafny to SQIR (TR [24] C.6). We will further investigate integrating Qafny with other tools, such as CoqQ [60], to verify HCQ programs automatically.

References

1 A. Ambainis. Quantum walk algorithm for element distinctness. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31, 2004. doi:10.1109/FOCS.2004.54.

2 Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003.

3 Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 148–177. Springer, 2021. doi:10.1007/978-3-030-72019-3_6.

4 Andrew Childs, Ben Reichardt, Robert Spalek, and Shengyu Zhang. Every NAND formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a Quantum Computer. *CoRR*, 2007. doi:/10.48550/arXiv.quant-ph/0703015.

5 Andrew M. Childs. On the Relationship Between Continuous- and Discrete-Time Quantum Walk. *Communications in Mathematical Physics*, 294(2):581–603, October 2009.

6 Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

7 Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language. *arXiv e-prints*, July 2017. arXiv:1707.03429.

8 Paul Adrien Maurice Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35:416–418, 1939.

9 Mahmudul Faisal Al Ameen and Makoto Tatsuta. Completeness for recursive procedures in separation logic. *Theoretical Computer Science*, 631:73–96, 2016. `doi:10.1016/j.tcs.2016.04.004`.

10 Yuan Feng and Mingsheng Ying. Quantum hoare logic with classical variables. *ACM Transactions on Quantum Computing*, 2(4), December 2021. `doi:10.1145/3456877`.

11 Sukhpal Singh Gill, Oktay Cetinkaya, Stefano Marrone, Daniel Claudino, David Haunschild, Leon Schlote, Huaming Wu, Carlo Ottaviani, Xiaoyuan Liu, Sree Pragna Machupalli, Kamalpreet Kaur, Priyansh Arora, Ji Liu, Ahmed Farouk, Houbing Herbert Song, Steve Uhlig, and Kotagiri Ramamohanarao. Quantum computing: Vision and challenges, 2024. `arXiv:2403.02240`.

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

13 Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79:325–328, July 1997. `doi:10.1103/PhysRevLett.79.325`.

14 Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using 2n + 2 qubits with toffoli based modular multiplication. *Quantum Info. Comput.*, 17(7–8):673–684, June 2017.

15 Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. Proving quantum programs correct. In *Proceedings of the Conference on Interative Theorem Proving (ITP)*, June 2021.

16 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2021.

17 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. `doi:10.1145/363235.363259`.

18 Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 944–959, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3453483.3454087`.

19 Yoshihiko Kakutani. A logic for formal verification of quantum programs. In Anupam Datta, editor, *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*, pages 79–93, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

20 Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. A quantum interpretation of separating conjunction for local reasoning of quantum programs based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. `doi:10.1145/3498697`.

21 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

22 K. Rustan M. Leino and Michał Moskal. Co-induction simply. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, pages 382–398, Cham, 2014. Springer International Publishing.

23 Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. Verified compilation of quantum oracles. In *OOPSLA 2022*, 2022. `doi:10.48550/arXiv.2112.06700`.

24 Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicolellis, Yi Lee, Le Chang, and Xiaodi Wu. Qafny: A quantum-program verifier, 2024. `arXiv:2211.06411`.

25 Yangjia Li and Dominique Unruh. Quantum Relational Hoare Logic with Expectations. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 136:1–136:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ICALP.2021.136`.

26 Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum hoare logic. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 187–207, Cham, 2019. Springer International Publishing.

27 Christof Löding, P. Madhusudan, and Lucas Peña. Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158098.

28 Guang Hao Low and Isaac L. Chuang. Optimal Hamiltonian Simulation by Quantum Signal Processing. *Physical Review Letters*, 118(1), January 2017.

29 Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. *SIGPLAN Not.*, 47(1):123–136, January 2012. doi:10.1145/2103621.2103673.

30 Frédéric Magniez, Miklos Santha, and Mario Szegedy. Quantum Algorithms for the Triangle Problem. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 1109–1117, USA, 2005. Society for Industrial and Applied Mathematics.

31 Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

32 Microsoft. Probabilistic Z3, 2014. URL: https://github.com/ProbabilisticZ3/src.

33 Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, May 2018. doi:10.1038/s41534-018-0072-4.

34 Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. Invariant synthesis for incomplete verification engines. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 232–250, Cham, 2018. Springer International Publishing.

35 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, USA, 10th anniversary edition, 2011.

36 KARL PEARSON. The problem of the random walk. *Nature*, 72(1865):294–294, July 1905. doi:10.1038/072294b0.

37 Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 440–451, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2594291.2594325.

38 Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A formally certified end-to-end implementation of Shor's factorization algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023. doi:10.1073/pnas.2218775120.

39 Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. *SIGPLAN Not.*, 48(6):231–242, June 2013. doi:10.1145/2499370.2462169.

40 Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 231–242. ACM, 2013. doi:10.1145/2491956.2462169.

41 Robert Rand. *Formally verified quantum programming.* PhD thesis, University of Pennsylvania, 2018.

42 J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

43 Gustavo Rigolin. Quantum teleportation of an arbitrary two-qubit state and its relation to multipartite entanglement. *Physical Review A*, 71(3), March 2005. doi:10.1103/physreva.71.032303.

44 Grigore Roşu and Andrei Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM, 2011. doi:doi:10.1145/1985793.1985928.

45 Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-Path Reachability Logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

46 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: Automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454036.

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

48 P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi:10.1109/SFCS.1994.365700.

49 Matt Swayne. What Are The Remaining Challenges Of Quantum Computing?, 2023. URL: https://thequantuminsider.com/2023/03/24/quantum-computing-challenges/.

50 Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. Automated mutual explicit induction proof in separation logic, 2016. doi:10.48550/arXiv.1609.00919.

51 Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. Completeness and expressiveness of pointer program verification by separation logic. *Information and Computation*, 267:1–27, 2019. doi:10.1016/j.ic.2019.03.002.

52 Dominique Unruh. Quantum relational hoare logic. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290346.

53 Salvador Elías Venegas-Andraca. Quantum walks: a comprehensive review. *Quantum Information Processing*, 11(5):1015–1106, July 2012. doi:10.1007/s11128-012-0432-5.

54 Thomas G. Wong. Unstructured search by random and quantum walk. *Quantum Information and Computation*, 22(1&2):53–85, January 2022. doi:10.26421/qic22.1-2-4.

55 Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 402–416, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

56 Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.*, 33(6), January 2012. doi:10.1145/2049706.2049708.

57 Mingsheng Ying. Toward automatic verification of quantum programs. *Form. Asp. Comput.*, 31(1):3–25, February 2019. doi:10.1007/s00165-018-0465-3.

58 Bohua Zhan. Efficient verification of imperative programs using auto2. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 23–40, Cham, 2018. Springer International Publishing.

59 Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. A Quantum Interpretation of Bunched Logic and Quantum Separation Logic. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1109/LICS52264.2021.9470673.

60 Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. Coqq: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571222.