Quantum Natural Proof

ANONYMOUS AUTHOR(S)

Quantum program correctness is typically assured by formal verification, but the quantum semantic nature, based on unitary, density matrices, and complex numbers, indicates that such verification is laborious and time-comsuming. In this paper, we proposed quantum natural proof (QNP), an automated proof system for verifying classical quantum hybird algorithms. Natural proofs are a subclass of proofs that are amenable to completely automated reasoning, that provide sound but incomplete procedures, and that capture common reasoning tactics in program verification. The core of QNP is a quantum proof system, named the Qafny proof system, that maps quantum operations to classical array aggregate operations that can be effectively verified in a classical separation logic framework. We have shown the soundness and completeness of the Qafny proof system as well as the soundness of the proof system compilation from Qafny to Dafny. Qafny permits quantum conditionals and we believe that our quantum conditional proof rule is the first quantifier free proof rules for quantum conditional operations. In addition, quantum programs written in Qafny can be compiled to quantum circuits so that every verified quantum program can be run on a quantum machine.

1 INTRODUCTION

Quantum computers offer unique capabilities that can be used to program substantially faster algorithms compared to those written for classical computers. For example, Shor's algorithm [48] can factorize a number in polynomial time (compared to the sub-exponential time for the best known classical algorithm). It is well known that quantum computers provide quantum supremacy. Most quantum algorithms are not classically simulatable because of the property; therefore, they are verified through rigorous *formal methods*. Many frameworks were proposed to verify quantum algorithms [3, 19, 25, 30, 54, 57], which essentially established quantum semantic interpretations and libraries in some interactive theorem provers, such as Isabelle and Coq, to permit quantum program verification, with some tactics for proof automation; but building and verifying quantum algorithms in these frameworks are time-consuming and require human efforts. Not to mention that many of these frameworks have no quantum circuit compilations, i.e., verified programs require additional efforts to convert to circuits in other platforms. On the other hand, automated verification is an active research fields in classical computation with many frameworks being proposed [5, 20, 21, 26, 31, 34, 38, 40, 44, 46, 47, 50], all of which showed strong results in relieving programmers' pain in verifying classical programs. Is there a way to utilize classical automated verification frameworks in verifying quantum programs?

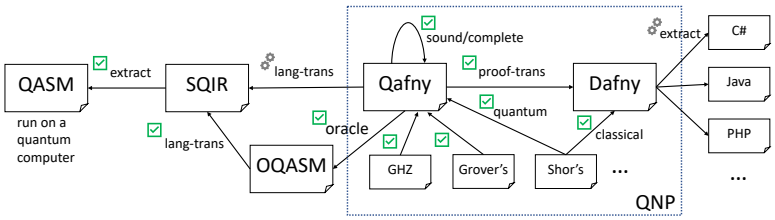


Fig. 1. QNP Development Stages and the Key Aspects

In this paper, we propose *Quantum Natural Proof* (QNP), a framework that help programmers write and verify quantum programs based on the marriage of quantum program semantics and classical automated verification infrastructure. It has several elements, as shown in Figure 1: 1) Using QNP, an quantum program can be specified in a simple, high-level programming language Qafny , which has standard imperative features and can express quantum classical hybrid programs with

high-level operations, such as state preparation, oracle, quantum diffusion, quantum conditionals, and for-loops; 2) QNP provides a quantum classical hybird proof system that allows programmers to automatically verify their programs in Qafny . The quantum portion of the proof system is named the Qafny proof system, which is specified based on the Qafny quantum language semantics, while the classical part is handled by Dafny [27]; 3) The automated verification in Qafny is compiled to Dafny and utilizes its infrastructure in finishing the task, while the classical component is soley relied on the Dafny proof system, e.g., GHZ and Grover's search algorithms are verified by Qafny , while Shor's algorithm is split into quantum and classical components that are handled by Qafny and Dafny, respectively; and 4) The Qafny quantum components can be compiled to quantum circuits and run on a quantum computer via the Qafny to SQIR compiler and SQIR to OpenQASM 2.0 [7] compiler, while the classical components are based on the Dafny infrastructure that can be extracted to several different programming languages, such as C#, Java and PHP.

The key QNP design philosophy leverages the methodology of analogizing quantum operations as classical aggregate operations. An example that motivates the QNP development is the state preparation in Child's quantum Boolean equation algorithm [4], as shown in Figure 2a, where we first prepare a superposition state $\sum_{j=0}^{2^n} |j\rangle$ by applying n Hadamard operations (see Section 2 for quantum background), then apply an oracle operation $f(|\kappa\rangle) = (-i)^\kappa |\kappa\rangle$. The oracle operation can be translated as $f(1, \kappa) = ((-i)^\kappa, \kappa)$, where we take κ in a pair and push the value $(-i)^\kappa$ to the first element. If we view the superposition state as an 2^n element array, with elements being basis states, the oracle application on the state is exactly a map operation that applies the function f to every array element as in Figure 2b. Obviously, quantum algorithms have more complicated structures than a simple array map. Let's analyze the GHZ example in Figure 2e. Before entering the for-loop (line 4-6), the input quantum array is split into conceptually two parts, analogized in Figure 2d. The two parts have different types. Here, the red part is an array of basis states, while the blue part is an array of qubits. In each step, we cut one qubit from the blue part and insert it into the white place in the red part by transforming the qubit type. During this process, the red array structure might vary depending on the inserted qubit state type. Finally, a quantum conditional is applied on the red array. Many quantum algorithms have similar structures as the above scenario, such as GHZ [15] and Shor's algorithms.

QNP is designed to capture the scenario and similar algorithm patterns with two features. First, the language operations in Qafny are designed to be analogized to high level array aggregate operations, so that not only programmers can write programs based on high level operations, but also does the Qafny proof system connect to traditional separation logic; thus, we can then utilize a classical automated proof engine, like Dafny, to automatically verify quantum programs, as our Qafny to Dafny compiler in Section 4.1; while most quantum proof systems, such as QHL [30], QBricks [3], QSL/BI [57], and QSL [24], built proof systems based on quantum computation theories Second, a type system that tracks qubit array bounds and the qubit state type transformation, as the scenario indicates, is designed in QNP. Apparently, tracking bounds in quantum arrays is not as easy as tracking classical array bounds, because qubits from different arrays can be entangled together, i.e., their states are not separable. In QNP, we invented the concept *sessions*, representing groups of qubit array pieces that are possibly entangled with each other. Tracking quantum array bounds is to track the session scopes in our type system that ensures that applications on a session do not interfere with others. We identify several QNP achieves as follows, which are partly indicated in Figure 1.

- We define the Qafny semantics as a small-step semantics, and the Qafny proof system based on viewing quantum operations as array aggregate operations. Especially, we define a quantifier free proof rule for quantum conditionals and for-loops whose Boolean guards

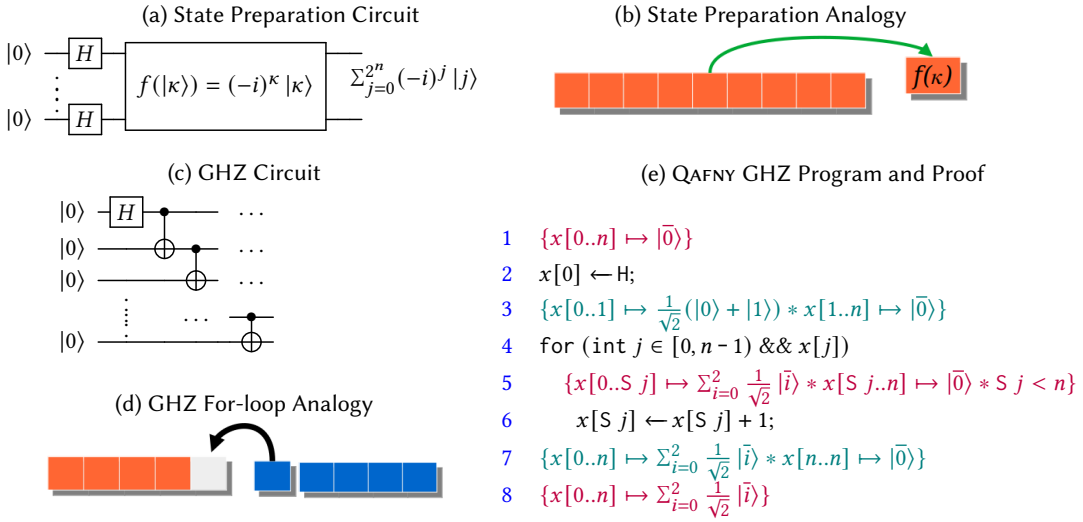


Fig. 2. Motivating Examples. S $j = j + 1$. Assume that we have $\kappa \mapsto \Sigma_{i=0}^{2}|\bar{i}\rangle$, then $|\kappa|$ is the length of κ, and $|\bar{i}\rangle$ refers to m number of $i \in [0, 1]$ bits, where $m = |\kappa|$. $*$ is the separation conjunction in separation logic. In (d), black parts are QAFNY programs, while purple and teal parts are state predicates.

involve quantum variables. To the best of our knowledge, this is the first inference rule definition for quantum conditionals and for-loops.

- We proved in Coq the QAFNY type system soundness as well as the proof system soundness and completeness with respect to the QAFNY semantics on type-correct programs.
- We proved in Coq the QAFNY to separation logic proof system compilation correctness, as a verification for the QAFNY to Dafny compiler. To the best of our knowledge, this is the first work that connects a quantum proof system and classical separation logic proof system.
- We implemented the QAFNY proof system on Dafny, a proof framework based on separation logic, and verified many quantum algorithms (Figure 16) with automation. For example, users do not need to specify any teal parts in Figure 2e and Figure 3, as they are inferred by the QAFNY proof system. Section 5 shows that program specifications in QNP can be a lot shorter than other quantum proof systems and QNP saves human efforts in verifying quantum programs.
- We also show in Section 5 two case studies that QNP can help programmers to construct verification for new quantum algorithms based on the reuse of existing quantum algorithm proofs. Programmers can also learn about the intuitive behaviors of quantum programs that are previously described as physical procedures.
- The circuit compilation from QAFNY to OQASM is verified in Coq, while the one from QAFNY to SQIR is tested by extracting a QAFNY interpreter to Ocaml, and we test program behaviors in the interpreter against the extracted OpenQASM code from SQIR.

2 BACKGROUND

Here, we provide background information for quantum computing and classical proof systems.

Quantum States and Alternative State Representations. A quantum state consists of one or more quantum bits (*qubits*). A qubit can be expressed as a two dimensional vector $\binom{\alpha}{\beta}$ where α, β

are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write the qubit vector as $\alpha\,|0\rangle + \beta\,|1\rangle$ where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are *computational basis states*. When both α and β are non-zero, we can think of the qubit as being "both 0 and 1 at once," a.k.a. a *superposition*. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is an equal superposition of $|0\rangle$ and $|1\rangle$. We can join multiple qubits together to form a larger quantum state with the *tensor product* ($\otimes$) from linear algebra. For example, the two-qubit state $|0\rangle \otimes |1\rangle$ (also written as $|01\rangle$) corresponds to vector $[\ 0\ 1\ 0\ 0\ ]^T$. Sometimes a multi-qubit state cannot be expressed as the tensor of individual states; such states are called *entangled*. One example is the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*.

n-qubit quantum states are typically represented as 2^n dimensional vectors. Alternatively, the states can be represented as different forms. For example, a newly generated qubit typically has a state $|0\rangle$ or $|1\rangle$, which is named *normal typed state* (`Nor`) in QNP. n Qubits that are in superposition but not entangled, such as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes ... \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, can be expressed as a summation of tensor products as $\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$, where $\alpha(r_j) = e^{2\pi i r}$ and $r \in \mathbb{R}$, which is named *Hadamard typed state* (`Had`) in QNP. $\alpha(r_j)$ is named the *local phase* of the state, which are special quantum amplitudes (see below) such that the norm is 1, i.e., $|\alpha(r_j)| = 1$. In the above state, we can view the local phase 1 as e^0, and $\frac{1}{\sqrt{2^n}}e^0$ is the amplitude for every basis state.

The most general representation is to express quantum states as a path-sum formula as: $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$, where $z_j \in \mathbb{C}$ is an amplitude, c_j is an n-length bitstring named *basis*, and $m \leq 2^n$. The path-sum formula is same as $z_0\,|c_0\rangle + ... + z_{m-1}\,|c_{m-1}\rangle$, such that each j-th element $z_j\,|c_j\rangle$ represents a *basis state* in the superposition state. This is named *entanglement typed state* (`CH`) in QNP. For example, the bell pair can be represented as $\Sigma_{j=0}^{2} \frac{1}{\sqrt{2}}\,|c_j\rangle$ with $c_0 = 00$ and $c_1 = 11$. Notice that the amplitudes satisfy the relation $\sum_0^m |z_j|^2 = 1$. However, in some intermediate program evaluation in QNP, we loose the restriction to be $\sum_0^m |z_j|^2 \leq 1$, because a state $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$ can be split into two parts as $\Sigma_{j=0}^{m} z_j\,|c_j\rangle = \Sigma_{i=0}^{m_1} z_i\,|c_i\rangle + \Sigma_{k=0}^{m_2} z_k\,|c_k\rangle$, and we might only want to reason about a portion of the state $\Sigma_{i=0}^{m_1} z_i\,|c_i\rangle$ locally, so that $\sum_0^{m_1} |z_i|^2 < 1$.

In QNP, each quantum state is associated with a *session*, referring to a cluster of quantum array pieces possibly entangled with each other. We can view a session as a virtual quantum array that manages quantum physical qubit arrays living in different locations but are locally connected through entanglement. See Section 3.1. As a shortcut of basis state representations, we might write $|c_1\rangle \otimes |c_2\rangle = |c_1\rangle\,|c_2\rangle = |c_1.c_2\rangle$ where $c_1.c_2$ is the bitstring concatenation and c_1 and c_2 are bitstrings of some sizes that relate to session lengths.

Quantum Computations and Conditionals. High-level quantum programming languages are usually designed to follow the *QRAM model* [23], where quantum computers are used as co-processors to classical computers. The classical computer generates descriptions of circuits to send to the quantum computer and then processes the measurement results. Computation on a quantum state consists of a series of *quantum operations*, each of which acts on a subset of qubits in the quantum state. In the standard presentation, quantum computations are expressed as *circuits*, as shown in Figure 2c, which constructs a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [15], which is an n-qubit entangled quantum state of the form: $|\mathrm{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$.

In these circuits, each horizontal wire represents a qubit and boxes on these wires indicate quantum operations, or *gates*. The circuit in Figure 2c uses n qubits and applies n gates: a *Hadamard* (`H`) gate and $n-1$ *controlled-not* (`CNOT`) gates. Applying a gate to a state *evolves* the state. The Qafny implementation in Figure 2e shows the evolving. In the j-th loop step, the quantum state of array $x[0..j]$ is $\frac{1}{\sqrt{2}}(|\overline{0}\rangle + |\overline{1}\rangle)$ [1], and a qubit $|0\rangle$ is added to the state and transforms the state to

[1] $|\overline{0}\rangle$ and $|\overline{1}\rangle$ have the same length as $x[0..j]$

$\frac{1}{\sqrt{2}}(|\overline{0}\rangle\,|0\rangle + |\overline{1}\rangle\,|0\rangle)$, which adds a bit 0 to the end of every basis state. Then, the quantum conditional (`if (x[j]) x[S j] ←x[S j] + 1`) turns $|0\rangle$ in the second basis state to $|1\rangle$ as $\frac{1}{\sqrt{2}}(|\overline{0}\rangle\,|0\rangle + |\overline{1}\rangle\,|1\rangle)$, because the quantum conditional checks the j-th qubit in every basis state, if it is 1, then the conditional body is applied; otherwise, it does nothing.

Measurement. A special, non-unitary *measurement* operation is used to extract classical information from a quantum state, typically when a computation completes. Measurement collapses the state to one of the basis states with a probability related to the state's amplitudes. For example, measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ will collapse the state to $|0\rangle$ with probability $\frac{1}{2}$ and likewise for $|1\rangle$, returning classical values 0 or 1, respectively.

Quantum Oracles. Quantum algorithms manipulate input information encoded in "oracles," which are callable black box circuits. For example, Grover's algorithm for unstructured quantum search [16, 17] is a general approach for searching a quantum "database," which is encoded in an oracle for a function $f : \{0,1\}^n \to \{0,1\}$. Grover's finds an element $x \in \{0,1\}^n$ such that $f(x) = 1$ using $O(2^{n/2})$ queries, a quadratic speedup over the best possible classical algorithm, which requires $\Omega(2^n)$ queries. Oracles are typically viewed as quantum reversible implementations of classical operations, especially arithmetic operations. OQASM [28] is a language that permits the effective testing of quantum oracles.

Dafny and Natural Proof. Dafny [26] is a language that is designed to make it easy to write correct code. It permits imperative programming with logical specifications which can be automatically verified through the Dafny proof system, a separation logic based system. The natural proof methodology was first proposed by Madhusudan et al. [32, 40], which exploits a fixed set of proof tactics, keeping the expressiveness of powerful logics, retaining the automated nature of proving validity, but giving up on completeness, e.g., the Qafny to Dafny compilation is only sound but not complete. The Qafny implementation of the natural proof methodology identifies a subclass of proofs $\mathcal{N}$ such that (1) a large class of valid verification specifications of near term classical quantum hybrid programs have a proof in $\mathcal{N}$, and (2) searching for a proof in $\mathcal{N}$ is efficiently decidable. In the original natural proof, the subclass identification is based on mapping heap data-structures, such as trees and link lists, to logical invariant properties. In QNP, the identification is through the Qafny type system in classifying quantum sessions and state types so that quantum operation applications on a specific session can be compiled to classical aggregate operations that have rich proof infrastructures in Dafny.

3 QAFNY: A HIGH-LEVEL QUANTUM LANGUAGE ADMITTING A PROOF SYSTEM

We designed Qafny , the core language of QNP, to express quantum programs with high-level operations that are abstracted away low-level circuit gates. The operations in Qafny are analogized to classical array aggregate operations so that automated verification is feasible. Qafny 's type system tracks the transformation of sessions with three types representing the sessions' quantum states. The Qafny proof system captures the analogy of viewing quantum operations as classical aggregate operations by utilizing the type system to ensure the session state forms. All of these features are novel to quantum languages and proof systems. This section presents Qafny states and the language's syntax, typing, semantics, proof system, and soundness/completeness results.

As a running example, we use the Shor's algorithm [49] shown in Figure 3. Given an integer N, Shor's algorithm finds its nontrivial prime factors, which has the following step: (1) randomly pick a number $1 < a < N$ and compute $k = \mathtt{gcd}(a, N)$; (2) if $k \neq 1$, k is the factor; (3) otherwise, a and N are coprime and we find the order p of a and N; (4) if p is even and $a^{\frac{p}{2}} \neq -1\%N$, $\mathtt{gcd}(a^{\frac{p}{2}} \pm 1, N)$ are the factors, otherwise, we repeat the process. Step (2) is the Shor's algorithm's quantum component and Figure 3 and Figure 38 show its automated proof in QNP. In Figure 19, we show the actual

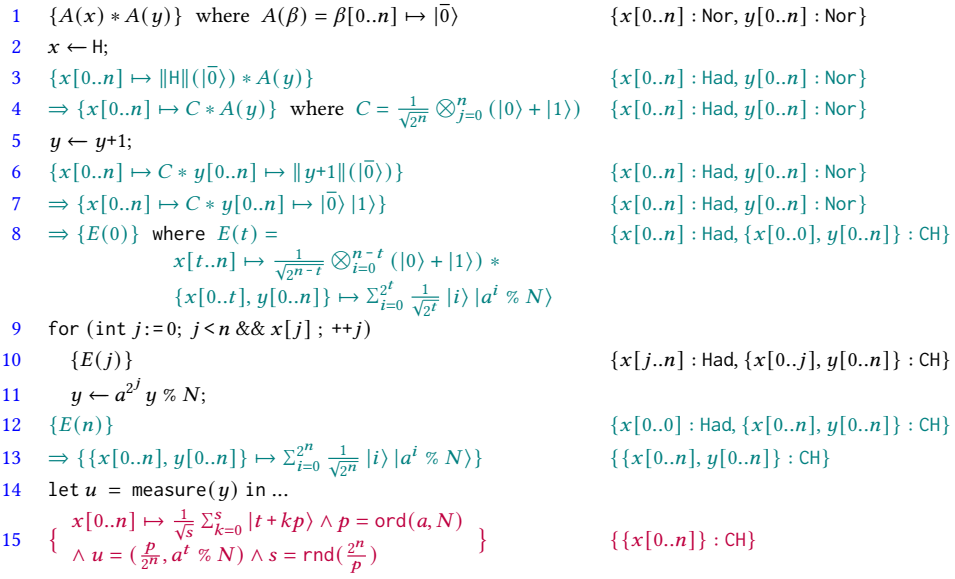


Fig. 3. Pre-measurement quantum steps of the Shor's algorithm. $\texttt{ord}(a, N)$ gets the order of a and N. $\texttt{rnd}(r)$ rounds r to the nearest integer. The right-hand-side contains the types for the sessions involved. $|i\rangle$ is an abbreviation of $|(\!|i|\!)\rangle$. $(\!|i|\!)$ turns a number i to a bitstring.

implementation and proof in the Qafny tool. The Shor's pre-measurement quantum steps in Figure 3 can be analogized as an efficient array filter operation in Figure 43. The steps before line 14 (steps at line 2, 5 and 9-11) create a 2^n-length of pairs, each of which is formed as $(i, a^i\%N)$ where $i \in [0, 2^n)$. The measurement in line 14 filters the array as a new one ($x[0..n]$) with all elements i satisfying $a^i\%N = u$ where u is a randomly picked number. Notice that modulo multiplication $f(i) = a^i\%N$ is a periodic function. All elements in $x[0..n]$ satisfy $a^i\%N = u$, which means that 1) there is a smallest t such that $a^t\%N = u$, and 2) all elements can be rewritten as $i = t + kp$ and p is the period of the modulo multiplication function, which is given as the post-condition on the right of line 15. Notice that only the black and purple parts in Figure 3 are required to input in the Qafny implementation, and the teal parts can be inferred by the Qafny proof system. [2]

We first introduce Qafny states, syntax, and type system. Then, we discuss its semantics and proof system and metatheories.

3.1 Classical and Quantum States

Qafny has three *kinds* [3] of parameters in Figure 4: C-kind classical integers [4], M-kind classical integers (r, n) with a probability characteristic r representing the theoretical probability of the measurement resulting in the natural number n, and (Q n)-kind quantum parameters, where n represents the number of qubits a qubit array piece, represented by a parameter. Quantum parameters are classified as three types: Nor, Had, or CH, representing the three types of quantum values in Section 2. We have subtyping relations over quantum types, i.e., Nor and Had are subtypes of CH, representing that Nor and Had typed quantum values can be rewritten as CH-type state forms.

[2]The purple is also not needed if we verify the whole Shor's algorithm in Figure 19.
[3]C and M kinds are also used as context modes in type checking. See Figure 6.
[4]In the Qafny implementation, one can use any classical types in Dafny. For simplicity, we only allow integers in this paper.

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	β	::=	$(\lvert c\rangle)^+$

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

Mode	g	::=	$\mathtt{C} \mid \mathtt{M}$
Classical Value	v	::=	$n \mid (r, n)$
Kind	$\overline{g}$	::=	$g \mid \mathtt{Q}\ n$
Quantum Type	τ	::=	$\mathtt{Nor} \mid \mathtt{Had} \mid \mathtt{CH}$
Quantum Value	q	::=	$z\beta \mid \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (\lvert 0\rangle + \alpha(r_j) \lvert 1\rangle) \mid \Sigma_{j=0}^{m} z_j \beta_j$

Quantum Sessions, Environment, and States

Range	re	::=	$x[n..m]$		
Session	κ	::=	$\overline{re}$	concatenated op	$\uplus$
Type Environment	σ	::=	$\overline{\kappa : \tau}$	concatenated op	$\cup$
Quantum State	φ	::=	$\overline{\kappa : q}$	concatenated op	$\cup$

Fig. 4. Qafny element syntax. Each range $x[n..m]$ in a session l represents the number range $[n, m)$ in a qubit array piece x. Sessions are finite lists, while type environments and states are finite sets. the operations after "concatenated op" refer to the concatenation operations for session, type environments and quantum states.

Qafny represents qubit arrays as *sessions* (κ), which consist of different *disjoint ranges*, each of which describes an array fragment $x[n..m]$, where x is a parameter representing a qubit array piece and $[n..m]$ represents the array fragment from position n to m (exclusive) in array piece x. For simplicity, we assume that there are no aliasing variables in this paper, i.e., two distinct variables represent disjoint array pieces. We view ranges are subsets of session, so that we can abbreviate a singleton session $\{x[n..m]\}$ as a range $x[n..m]$. Qafny quantum type environments/states are finite maps from sessions to types/states, and their domains do not overlap, i.e., for all $\kappa, \kappa' \in \mathtt{dom}(\sigma)$ (or $\mathtt{dom}(\varphi)$), $\kappa \neq \kappa' \Rightarrow \kappa \cap \kappa' = \emptyset$.

Qafny utilize *equivalence relations* over quantum sessions, quantum values, type environments and states (as shown in Figure 39) to facilitate automated program verification, written as $\equiv$ for state equivalence and $\preceq$ for environment partial order. One example is the state predicate rewrite from line 12 to line 13 in Figure 3, where the session state $x[0..0]$ rewrites to `true`, because range $x[0..0]$ is essentially empty. The common equivalence relations are state type rewrites, permutations, split and joins. State type rewrites transform state forms, such as the rewrite of session $\{x[0..0], y[0..n]\}$ from type `Nor` to `CH` in line 7 and 8 in Figure 3. Another example is to rewrite a `CH`-type state $\Sigma_{j=0}^{1} z_j \beta_j$ to `Nor`-type $z_0 \beta_0$. Permutation equivalence refers to two qubits can mutate their locations. For example, the state in line 13 can be rewritten to $\{y[0..n], x[0..n]\} \mapsto \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} \lvert a^j \% N\rangle \lvert j\rangle$ [5]. State joins merge two sessions together. Merging a `Nor`-type and `CH`-type state is analogized to concatenate two qubit arrays. An example and its explanation are given in Figure 2e line 4-6, where the `Nor`-type qubit $x[\mathtt{S}\ j]$ is added to the end of `CH`-type session $x[0..\mathtt{S}\ j]$. Merging a `Had`-type and `CH`-type state doubles the `CH`-type basis states. In each loop step in Figure 3 line 9-11, we add the `Had` type qubit $x[j]$ to `CH` type session $\{x[0..j], y[0..n]\}$, and the basis states are doubled by making a copy of the current state and inserting $\lvert 0\rangle$ and $\lvert 1\rangle$ in the two copies, as $\Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}} \lvert j\rangle \lvert 0\rangle \lvert a^j \% N\rangle + \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}} \lvert j\rangle \lvert 1\rangle \lvert a^j \% N\rangle$. Merging two `CH`-type states computes the Cartesian product of basis states in the two sessions (Figure 39). State split cuts a session into two individual sessions. The split of `Nor` and `Had` types is no more than an array split, while the split

[5] More aggressively, we can write the state $x[0..2] \mapsto \frac{1}{\sqrt{2}}(\lvert 01\rangle + \lvert 10\rangle)$ to $\{x[1..2], x[0..1]\} \mapsto \frac{1}{\sqrt{2}}(\lvert 10\rangle + \lvert 01\rangle)$.

⒪QASM Expr	μ		
Parameter	l	::=	$x \mid x[a]$
Arith Expr	a	::=	$x \mid v \mid a + a \mid a * a \mid ...$
Bool Expr	b	::=	$x[a] \mid (a = a)@x[a] \mid (a < a)@x[a] \mid ...$
Predicate	P, Q, R	::=	$a = a \mid a < a \mid \kappa \mapsto q \mid P \wedge P \mid P * P \mid ...$
Gate Expr	op	::=	$\mathtt{H} \mid \mathtt{QFT}^{[-1]}$
C/M Mode Expr	e	::=	$a \mid \mathtt{measure}(y)$
Statement	s	::=	$\{\} \mid \mathtt{let}\ x\ =\ e\ \mathtt{in}\ s \mid l \leftarrow op \mid \kappa \leftarrow \mu \mid l \leftarrow \mathtt{dis}$
		|	$s\ ;\ s \mid \mathtt{if}\ (b)\ s \mid \mathtt{for}\ (\mathtt{int}\ j \in [a_1, a_2)\ \&\&\ b)\ s$

Fig. 5. Core QAFNY syntax. ⒪QASM is in Section 3. For an operator OP, $\mathtt{OP}^{[-1]}$ indicates that the operator has a built-in inverse available. $x[a]$ represents the a-th element in the qubit array x, while a quantum variable x represents range $x[0..n]$ and n is the length of x.

$$\textsc{TPar}\ \frac{\sigma \preceq \sigma' \qquad \Omega; \sigma' \vdash_g s \triangleright \sigma''}{\Omega; \sigma \vdash_g s \triangleright \sigma''} \qquad \textsc{TExp}\ \frac{x \notin \Omega}{\Omega[x \mapsto \mathtt{C}]; \sigma \vdash_g s[n/x] \triangleright \sigma'}{\Omega; \sigma \vdash_g \mathtt{let}\ x\ =\ n\ \mathtt{in}\ s \triangleright \sigma'}$$

$$\textsc{TMea}\ \frac{x \notin \Omega \qquad \sigma(y) = \{y[0..j] \uplus \kappa \mapsto \tau\} \qquad \Omega(y) = \mathtt{Q}\ j \qquad \Omega[x \mapsto \mathtt{M}]; \sigma[\kappa \mapsto \mathtt{CH}] \vdash_{\mathtt{C}} s \triangleright \sigma'}{\Omega; \sigma \vdash_{\mathtt{C}} \mathtt{let}\ x\ =\ \mathtt{measure}(y)\ \mathtt{in}\ s \triangleright \sigma'}$$

$$\textsc{TA-CH}\ \frac{FV(\Omega, \mu) = \kappa \qquad \sigma(\kappa \uplus \kappa') = \mathtt{CH}}{\Omega; \sigma \vdash_g \kappa \leftarrow \mu \triangleright \{\kappa \uplus \kappa' : \mathtt{CH}\}} \qquad \textsc{TDis}\ \frac{FV(\Omega, l) = \kappa \qquad \sigma(\kappa \uplus \kappa') = \mathtt{CH}}{\Omega, \sigma \vdash_g \kappa \leftarrow \mathtt{dis} \triangleright \{l \uplus \kappa' : \mathtt{CH}\}}$$

$$\textsc{TSeq}\ \frac{\Omega; \sigma \vdash_g s_1 \triangleright \sigma_1 \qquad \Omega; \sigma[\uparrow \sigma_1] \vdash_g s_2 \triangleright \sigma_2}{\Omega; \sigma \vdash_g s_1\ ;\ s_2 \triangleright\ \sigma_2 \cup \sigma_1|_{\notin \mathsf{dom}(\sigma_2)}}$$

$$\textsc{TIf}\ \frac{FV(\Omega, b) = \kappa \qquad \kappa \cap FV(\Omega, s) = \emptyset \qquad \Omega; \sigma[\kappa \mapsto \mathtt{CH}] \vdash_{\mathtt{M}} s \triangleright \{\kappa' : \mathtt{CH}\}}{\Omega; \sigma[\kappa \uplus \kappa' \mapsto \mathtt{CH}] \vdash_g \mathtt{if}\ (b)\ s \triangleright \{\kappa \uplus \kappa' : \mathtt{CH}\}}$$

$$\textsc{TLoop}\ \frac{\forall j \in [n_1, n_2)\ .\ \Omega[x \mapsto \mathtt{C}]; \sigma[\uparrow \sigma'[j/x]] \vdash_g \mathtt{if}\ (b[j/x])\ s[j/x] \triangleright \sigma'[\mathtt{S}\ j/x]}{\Omega; \sigma[\uparrow \sigma'[n_1/x]] \vdash_g \mathtt{for}\ (\mathtt{int}\ x \in [n_1, n_2)\ \&\&\ b)\ s \triangleright \sigma'[n_2/x]}$$

$$\sigma[\uparrow \sigma'] = \sigma[\forall \kappa : \tau \in \sigma'\ .\ \kappa \mapsto \tau] \qquad \sigma|_{\notin \mathsf{dom}(\sigma')} = \{\kappa : \tau \in \sigma | \kappa \notin \mathsf{dom}(\sigma')\}$$

Fig. 6. QAFNY type system. $\sigma(y) = \{\kappa \mapsto \tau\}$ produces the entry $\kappa \mapsto \tau$ and the range $y[0..|y|] \subseteq \kappa$. $\sigma(\kappa) = \tau$ is an abbreviation of $\sigma(\kappa) = \{\kappa \mapsto \tau\}$. $\sigma[\kappa \mapsto \tau]$ is valid only if $\kappa \in \mathsf{dom}(\sigma)$, we can use rule TPAR to address the domain. $FV(\Omega, -)$ produces a session by union all qubits in $-$ with info in Ω; see Appendix B.1.

of a CH-type is equal to disentanglement, a hard problem. In quantum algorithms, splitting Nor and Had types are more common than disentanglement, and is permitted in QAFNY . For splitting CH-type, we invent an upgraded type system in Appendix C to permit few cases.

3.2 Syntax and Type Sysem

One key QAFNY principle permits programmers thinking of quantum programs as sequences of functional operations that are analogized to array aggregate operations, reflected by the QAFNY syntax in Figure 5. A program consists of a sequence of C-like statements s that end at a skip ($\{\}$). The let operation ($\mathtt{let}\ x\ =\ e\ \mathtt{in}\ s$) in the first row binds a new classical variable x with e, which is used in s. If e is an arithmetic expression (a), it introduces a C/M kind classical variable. $\mathtt{let}\ x\ =\ \mathtt{measure}(y)\ \mathtt{in}\ s$ measures array piece y, stores the result in M-kind variable x that is used in s. The last three of the first row are quantum data-flow operations. $l \leftarrow op$ prepares a quantum superposition state of quantum qubits l through Hadamard H or QFT gates, which are used for state preparation. It is also used to Fourier transform quantum qubit states by a $\mathtt{QFT}^{-1}$ gate in quantum phase estimation and Shor's algorithms. The other gate applications are done through $\kappa \leftarrow \mu$ that performs an ⒪QASM quantum oracle computation μ ([28]) on each basis state of session κ as we unveil in Section 1. Since all quantum reversible arithmetic operations were defined in ⒪QASM; thus, we permit μ's

description in Qafny to be arithmetic operations, similar to expressions a in Figure 5, e.g., fig. 3 line 5 and 11. $l \leftarrow \texttt{dis}$ is a quantum diffusion operation applying on the parameter l, where l may be part of an entangled session. The main functionality is to increase and average the occurrence likelihood of some quantum bases in a quantum state.

The second row of statements in Figure 5 are control-flow operations. s_1 ; s_2 is a sequential operation. $\texttt{if}\ (b)\ s$ is a classical or quantum conditional depending on if b contains quantum parameters. Quantum reversible Boolean guards b are implemented as QASM oracle operations, written as $(a_1 = a_2)@x[a]$, $(a_1 < a_2)@x[a]$, and $x[a]$, meaning that for each quantum basis state, we compute b's value $a_1 = a_2$, $a_1 < a_2$, and $\texttt{true}$ [6], and store the result in the qubit bit $x[a]$ by computing $b \oplus x[a]$. One example quantum Boolean equation is used at the quantum walk algorithm in Section 5.2. $\texttt{for}\ (\texttt{int}\ j \in [a_1, a_2)\ \&\&\ b)\ \{s\}$ is a possible quantum for-loop depending on if b contains quantum parameters. A classical variable j is introduced and initialized as the lower bound a_1, incremented in each loop step by $++j$, and ended at the upper bound a_2. For example, the for-loop in Figure 3 repeatably entangles the Had-type qubit $x[j]$ with the CH-type session $\{x[0..j], y[0..n]\}$ through the modulo multiplication at line 11. [7]

A Quantum Session Type System. In Qafny , typing is with respect to a *kind environment* Ω and a *finite type environment* σ, which map Qafny variables to kinds and map sessions to types, respectively. The typing judgment is written as $\Omega; \sigma \vdash_g s \triangleright \sigma'$, which states that statements s is well-typed under the context mode g and environments Ω and σ, the sessions representing s is exactly the domain of σ' ($\texttt{dom}(\sigma')$), and s transforms types for the sessions in σ to types in σ'. Ω is populated through $\texttt{let}$ expressions that introduce variables,and the type system enforces variable scope; such enforcement is neglected in Figure 6 for simplicity. We assume that variables introduced in $\texttt{let}$ expressions are all distinct through proper alpha conversions, as in TExp and TMea. $\texttt{dom}(\sigma)$ and $\texttt{dom}(\sigma')$ are large enough to describe all sessions containing all qubits mentioned in s. Kinds g are reused as context modes (C and M) for enforcing no quantum information leak in a quantum conditional. Selected type rules are given in Figure 6; the rules not mentioned are similar and listed in Appendix B.

The type system enforces four invariants. First, we place well-formed and context restrictions on quantum programs. Well-formedness refers to the No-cloning theorem, such that qubits mentioned in a quantum conditional Boolean guard cannot be accessed in the conditional body; while context restriction refers to no quantum information leak, such that quantum conditional bodies cannot create and measure ($\texttt{measure}$) qubits. For example, the *FV* checks in rule TIF enforces that the session for the Boolean and the conditional body does not overlap. Context mode C permits most Qafny operations. Once a type rule turns it to M, as in TIF, we disallow measurement operations, as rules TMea is valid only if the input context mode is C. Second, the type system tracks the arrangement of sessions as well as permits the session equivalence relations through rule TPar. In rule TA-CH, the session appearing in μ is κ, which is the prefix of a session $\kappa \uplus \kappa'$. To type check the case when we apply μ on other locations, we utilize the TPar to transform the session structure by the session permutation equivalence. For example, in Figure 2 line 5, we want to apply addition on $x[\texttt{S}\ j]$, but the session is arranged as $x[0..j+2]$. In type checking the statement, we first rewrite the session through rule TPar to $\{x[\texttt{S}\ j..j+2], x[0..\texttt{S}\ j]\}$, then apply the TA-CH rule. Third, the type system enforces that C classical variables can be evaluated to values in the compilation time [8], while tracks M variables. Rule TExp enforces that a classical variable x is replaced with its assignment value n in s, where classical expressions containing x are evaluated. See Appendix B. Finally, the

[6] a_1 and a_2 can possibly be a quantum array piece x in an entangled session.

[7] In the Qafny implementation, $++j$ and $j < a_2$ can be arbitrary monotonic increment and comparison functions. For simplicity, we restrict the two to be $++j$ and $j < a_2$ in this paper.

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

$$\textsc{Frame}\quad \frac{FV(s)\cap FV(R)=\emptyset \qquad FV(s)\subseteq \mathsf{dom}(\sigma) \qquad \sigma\perp\sigma' \qquad \Omega;\sigma\vdash_g \{P\}\,s\,\{Q\}}{\Omega;\sigma\cup\sigma'\vdash_g \{P*R\}\,s\,\{Q*R\}} \qquad \frac{\sigma\perp\sigma' \qquad \varphi\perp\varphi' \qquad \Omega;\sigma;\psi;\varphi\models_g P \qquad \Omega;\sigma';\psi;\varphi'\models_g Q}{\Omega;\sigma\cup\sigma';\psi;\varphi\cup\varphi'\models_g P*Q}$$

$$\textsc{TCon}\quad \frac{\sigma\preceq\sigma' \qquad \Omega;\sigma'\vdash_g \{P'\}\,s\,\{Q'\}}{\Omega;\sigma\vdash_g \{P\}\,s\,\{Q\}} \qquad \frac{\forall j.\ |\kappa|=|c_j|}{\Omega;\sigma;\psi;\varphi[\kappa\mapsto\sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta'_j)]\models_g \sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j)}$$

$$TC(\sigma,P,Q)\triangleq \Omega;\sigma\vdash_g s\triangleright\sigma_1\wedge\Omega;\sigma\vdash P\wedge\Omega;\sigma[\uparrow\sigma_1]\vdash Q$$

Fig. 7. Frame proof rule and type consequence rule and model rules

type system estimates the scopes of entangled sessions. In rule TIF, we use κ', the domain of the post type environment, as the estimated session that is large enough to describe all possible qubits directly and indirectly mentioned in s [9] and should be in the same session. Based on κ', we estimate the conditional's session as $\kappa\uplus\kappa'$. The type system keeps such estimation as small as possible.

3.3 The Qafny Semantics and Proof System

QNP intends to create a proof system that utilizes classical automated reasoning infrastructure in analyzing quantum programs, through prototyping classical separation logic [44], by analogizing quantum operations to array aggregate operations. To materialize such analogy, Qafny sessions play two roles, both as variable names representing quantum arrays and as qubit array structure indicators, through the enforcement of well-formedness and well-typed restrictions on states and proof system predicates. Appendix C.7 describes the well-formedness: well-formed domains ($\Omega\vdash\mathsf{dom}(\sigma)$) mean that every range variables in a session in $\mathsf{dom}(\sigma)$ are mentioned in Ω, and well-formed predicates ($\Omega,\sigma\vdash P$) mean that every sessions mentioned in P are in $\mathsf{dom}(\sigma)$. Qafny semantics is a small-step transition $(\psi,\varphi,s)\longrightarrow(\psi',\varphi',s')$, where ψ and ψ' are stacks storing local classical variables, and φ and φ' are quantum states. A Qafny proof triple is written as: $\Omega;\sigma\vdash_g\{P\}\,s\,\{Q\}$, where P and Q are the pre- and post-conditions, Ω, σ, and g are the type entities. Apparently, any QNP proof judgment, which merges the Qafny type system into proof rules, has a hidden type restriction stated as predicate TC in Figure 40. The restriction is required not only at the bottom proof tree, but also at all levels, e.g., rule TCon describes the typing consequence rule by replacing σ with σ'. We require that P and Q satisfy $TC(\sigma,P,Q)$, as well as P' and Q' satisfy $TC(\sigma',P',Q')$. Similarly, the Frame rule is no more than a separation logic frame rule [10], other than the additional type restrictions, where we separate the type environment into σ and σ'; on the above level of the Frame rule, the conditions P and Q also need to satisfy the type restriction with respect to Ω and σ.

The rule besides rule Frame in Figure 40 is the modeling rule for a separable state P and Q. In QNP, a modeling rule has the judgment: $\Omega;\sigma;\psi;\varphi\models_g P$, with the above well-formed restrictions, additionally requiring $\mathsf{dom}(\psi)\subseteq\mathsf{dom}(\Omega)$, and $\Omega;\sigma\vdash_g\varphi$, defined as follows:

Definition 3.1 (Well-formed Qafny state). A state φ is *well-formed*, written as $\Omega;\sigma\vdash_g\varphi$, iff $\mathsf{dom}(\sigma)=\mathsf{dom}(\varphi)$, $\Omega\vdash\mathsf{dom}(\sigma)$, and:

- For every $\kappa\in\mathsf{dom}(\sigma)$ such that $\sigma(\kappa)=\mathtt{Nor}$, $\varphi(\kappa)=z\,|c\rangle$ and $|\kappa|\le|c|$ [11] and $|z|\le 1$; specifically, if $g=\mathtt{C}$, $|\kappa|=|c|$ and $|z|=1$.

[9] This includes all entangled array pieces whose qubits are mentioned in s.
[10] We use $FV(s)\cap FV(R)=\emptyset$ here because $FV(s)$ produces the session that s modifies in Qafny .
[11] $|\kappa|$ and $|c|$ are the lengths of κ and c.

SA-CH

$$\frac{\varphi(\kappa) = \{\kappa \uplus \kappa' \mapsto \sum_{j=0}^{m} q(c_j)\} \qquad \forall j.\ |c_j| = |\kappa|}{(\varphi, \kappa \leftarrow \mu) \longrightarrow (\varphi[\kappa \uplus \kappa' \mapsto \sum_{j=0}^{m} q([\![\mu]\!](c_j))], \{\})}$$

PA-CH

$$\frac{\sigma(\kappa) = \{\kappa \uplus \kappa' \mapsto \mathsf{CH}\} \qquad \forall j.\ |c_j| = |\kappa|}{\Omega; \sigma \vdash_g \{\kappa \uplus \kappa' \mapsto \sum_{j=0}^{m} q(c_j)\}\ \kappa \leftarrow \mu\ \{\kappa \uplus \kappa' \mapsto \sum_{j=0}^{m} q([\![\mu]\!](c_j))\}}$$

$$q(c_j) = \Sigma_{j=0}^{m} z_j\, |c_j\rangle\, \beta_j \qquad [\![\mu]\!] c_j = z'_j\, |c'_j\rangle$$

Fig. 8. Oracle application rules

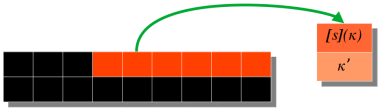


Fig. 9. Conditional Analogy

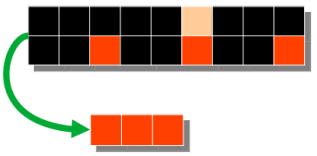


Fig. 10. Measurement Analogy

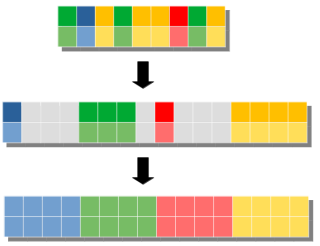


Fig. 11. Diffusion Analogy

- For every $\kappa \in \mathsf{dom}(\sigma)$ such that $\sigma(\kappa) = \mathsf{Had}$, $\varphi(\kappa) = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$ and $|\kappa| = n$.
- For every $\kappa \in \mathsf{dom}(\sigma)$ such that $\sigma(\kappa) = \mathsf{CH}$, $\varphi(\kappa) = \Sigma_{j=0}^{m} z_j\, |c_j\rangle$ and $|\kappa| \leq |c_j|$ and $\sum_{j=0}^{m} |z|^2 \leq 1$ and for $i, k \in [0, m)$, $|c_i| = |c_k|$; specifically, if $g = \mathsf{C}$, $|\kappa| = |c_j|$ and $\sum_{j=0}^{m} |z|^2 = 1$.

The reason that we place a relaxed well-formedness for a M-mode state is that the program location in M-mode is inside a quantum conditional where parts of the sessions and states are frozen, whereas once the quantum conditional finishes execution and the program is transitioned back to C-mode, the frozen parts are added back to the state and the united state is required to satisfy the quantum state principles, which are described as the C-mode restrictions in Definition 3.1. The frozen states are described in Section 3.3. There are other rules in Appendix C.6 that are similar the rules in a separation logic framework with additional Qafny type restrictions. We now discuss few interesting cases.

Oracle Applications. The oracle application $\kappa \leftarrow \mu$ is analogized to classical array map operation as discussed in Figure 2b. Assume that μ works on session $\kappa \uplus \kappa'$. For each element $z_j\, |c_j\rangle\, \beta_j$ in the CH type state, we first find c_j as the corresponding basis state bitstring for the κ session fragment [12], then we apply μ only on c_j, which is described in the semantic rule SA-CH. Rule PA-CH describes the proof rule for capturing the array map analogy, which has the exact behavior as the semantic rule. The other similar rules, such as state preparation rules, can be found in Appendix B.

Rules for Conditionals and For-Loops. Figure 9 describes the analogy of quantum conditionals in Qafny , which are partial map functions that only apply applications on the red parts and freeze the black parts. It contains two levels of freezing. For each basis state in a session $\kappa_b \uplus \kappa_a$, it freezes the κ_b part of the state, which is indicated as the marked black items in the second line in Figure 9. For each basis state in $\Sigma_{j=0}^{m} z_j\, |c_j\rangle\, \beta_j$ with $|c_j| = |\kappa_b|$, we freeze the c_j part by pushing it to the end of the basis state as $z_j \beta_j |c_j\rangle$, which is indicated in rule SIF (Figure 12). During the process,

[12] κ and c_j have the same length and are the prefix session and basis state, respectively.

$$\frac{R \qquad \Omega;\sigma;\varphi \models_g \kappa \mapsto \sum_{j=0}^{m} z_j\beta_j\,|c_j\rangle}{\Omega;\sigma;\varphi \models_g \mathcal{M}(b,\kappa) \mapsto \sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j + q(\kappa,\neg b)}$$

$$\frac{R \qquad \Omega;\sigma,\varphi \models_g \kappa \mapsto \sum_{j=0}^{m'} z'_j\,|c_j\rangle\,\beta'_j + q(\kappa,\neg b)}{\begin{array}{l}\Omega;\sigma,\varphi \models_g \mathcal{U}(\neg b,\kappa) \mapsto \sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j + q(\kappa,\neg b) \\ \qquad *\mathcal{U}(b,\kappa) \mapsto \sum_{j=0}^{m'} z'_j\beta'_j\,|c_j\rangle\end{array}}$$

SIF

$$\frac{R \qquad FV(\emptyset,s) \subseteq \kappa' \qquad (\psi,\varphi[\kappa' \mapsto \sum_{j=0}^{m} z_j\beta_j\,|c_j\rangle],s) \longrightarrow (\psi',\varphi[\kappa' \mapsto \sum_{j=0}^{m'} z'_j\beta'_j\,|c_j\rangle],s')}{(\psi,\varphi[\kappa \uplus \kappa' \mapsto \sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j + q(\kappa,\neg b)],\texttt{if}\ (b)\ s) \longrightarrow (\psi',\varphi[\kappa \uplus \kappa' \mapsto \sum_{j=0}^{m'} z'_j\,|c_j\rangle\,\beta'_j + q(\kappa,\neg b)],\texttt{if}\ (b)\ s')}$$

PIF

$$\frac{\Omega;\{\kappa' : \mathtt{CH}\} \vdash Q' \qquad \Omega;\sigma[\kappa' \mapsto \mathtt{CH}] \vdash_{\mathtt{M}} \{P[\mathcal{M}(b,\kappa')/\kappa \uplus \kappa']\}\ s\ \{Q * Q'\}}{\Omega;\sigma[\kappa \uplus \kappa' \mapsto \mathtt{CH}] \vdash_g \{P\}\ \texttt{if}\ (b)\ s\ \{P[\mathcal{U}(\neg b,\kappa \uplus \kappa')/\kappa \uplus \kappa')] * Q'[\mathcal{U}(b,\kappa \uplus \kappa')/\kappa']\}}$$

SLOOP

$$\frac{n_1 < n_2 \qquad b' = b[n_1/j] \qquad s' = s[n_1/j]}{\begin{array}{l}(\psi,\varphi,\texttt{for}\ (\texttt{int}\ j \in [n_1,n_2)\ \&\&\ b)\ s) \longrightarrow \\ \quad (\psi,\varphi,\texttt{if}\ (b')\ s'\ ;\ \texttt{for}\ (\texttt{int}\ j \in [\mathtt{S}\ n_1,n_2)\ \&\&\ b)\ s)\end{array}}$$

PLOOP

$$\frac{n_1 < n_2 \qquad \Omega;\sigma \vdash_{\mathtt{M}} \{P(j) \wedge j < n_2\}\ \texttt{if}\ (b)\ s\ \{P(\mathtt{S}\ j)\}}{\Omega;\sigma \vdash_g \{P(n_1)\}\ \texttt{for}\ (\texttt{int}\ j \in [n_1,n_2)\ \&\&\ b)\ s\ \{P(n_2)\}}$$

$$q(\kappa,\neg b) = \Sigma_{i=0}^{m} z_i\,|c_i\rangle\,\beta_i \ \ \texttt{where}\ \forall i.\ |c_i| = |\kappa| \wedge [\![\neg b[c_i/\kappa]]\!]$$
$$R = FV(\emptyset,b) = \kappa \wedge \forall j.\ |c_j| = |\kappa| \wedge [\![b[c_j/\kappa]]\!]$$

Fig. 12. Semantic and Proof Rules for Conditionals and For-loops. $\mathcal{M}$ is the frozen function and $\mathcal{U}$ is the unfrozen function. $[\![b[c_j/\kappa]]\!]$ is the interpretation of Boolean guard b by replacing qubits mentioned in κ with bits in bitstring c_j. $P(j)$ is a predicate P with j as a variable.

the state session is transformed from $\kappa_b \uplus \kappa_a$ to κ_a when evaluating the conditional body s. This indicates that the $\kappa_b/|c_j\rangle$ part is frozen, because the session is shorter, the $|c_j\rangle$ part is stored in a location beyond the reachable of session κ_a; therefore, in evaluating s, no operation can touch the $|c_j\rangle$ part. The second level of freezing happens in selecting valid basis states by evaluating the Boolean condition b on basis state bitstrings. Predicate R in ruleSIF reflects the task. Here, we divide the state into two parts as $\Sigma_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j$ and $q(\kappa,\neg b)$, where the first part contains all basis states satisfying b, i.e., if we replace the qubit variables in b with c_j, the evaluation $[\![b[c_j/\kappa]]\!]$ is true; while the second part $q(\kappa,\neg b)$ contains all basis states that evaluate b to false. After the freezing step, we execute s only on the selected unfrozen parts of the state, and merge the execution results (z'_j and β'_j) back to the whole state. The result state element number m' might be different from the pre-execution one (m) because applications in s might increase the state numbers such as applying a quantum diffusion operation.

To design a proof rule for such partial map, we develop the frozen ($\mathcal{M}$) and unfrozen ($\mathcal{U}$) operations with the session types. Both take a Boolean expression b and a quantum state as arguments. As shown on top of Figure 12, $\mathcal{M}$'s modeling materializes the freezing mechanism above. For a state $\Sigma_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j + q(\kappa,\neg b)$, we only keep basis states satisfying b, as shown in predicate R, remove the unsatisfied basis states $q(\kappa,\neg b)$, and push the basis bitstrings c_j to the ends of the basis states, which are similar to stacks that store frozen bitstrings. In the pre-condition manipulation of rule PIF, we substitute $\kappa \uplus \kappa'$ with $\mathcal{M}(b,\kappa')$. During the process, the session type in σ is changed from $\kappa \uplus \kappa'$ to κ'. Function $\mathcal{U}$ unfreezes the state by assembling the result state, of applying s, back to the

$$\frac{|c| = n \qquad \Omega; \sigma; \varphi \models \kappa \mapsto \sum_{j=0}^{m} \frac{z_j}{\sqrt{r}} |c_j\rangle \wedge x = (r, \{\!|c|\!\})}{\Omega; \sigma, \varphi \models \mathcal{F}(x, n, \kappa) \mapsto \sum_{j=0}^{m} z_j |c\rangle |c_j\rangle + q(n, \neq c)}$$

SMEA

$$\frac{\varphi(y) = \{ y[0..n] \uplus \kappa \mapsto \sum_{j=0}^{m} z_j |c\rangle |c_j\rangle + q(n, \neq c) \}}{(\psi, \varphi, \texttt{let}\ x\ =\ \texttt{measure}(y)\ \texttt{in}\ s) \longrightarrow (\psi[x \mapsto (r, \{\!|c|\!\})], \varphi[\kappa \mapsto \sum_{j=0}^{m} \frac{z_j}{\sqrt{r}} |c_j\rangle], s)}$$

PMEA

$$\frac{\Omega[x \mapsto \texttt{M}]; \sigma[\kappa \mapsto \texttt{CH}] \vdash_{\texttt{C}} \{P[\mathcal{F}(x, n, \kappa)/y[0, n] \uplus \kappa]\}\, s\, \{Q\}}{\Omega[y \mapsto \texttt{Q}\ n]; \sigma[y[0, n] \uplus \kappa \mapsto \texttt{CH}] \vdash_{\texttt{C}} \{P\}\ \texttt{let}\ x\ =\ \texttt{measure}(y)\ \texttt{in}\ s\, \{Q\}}$$

$$r = \textstyle\sum_{k=0}^{m} |z_k|^2 \qquad q(n, \neq c) = \textstyle\sum_{k=0}^{m'} z'_k |c'.c'_k\rangle \ \texttt{where}\ c' \neq c$$

Fig. 13. Semantic and Proof Rules for Measurement. $\mathcal{F}$ is the measurement function construct. $\{\!|c|\!\}$ turns bitstring c to an integer, and r is the likelihood that the bitstring c appears in a basis state.

frozen $\kappa \uplus \kappa'$ state (the black part in Figure 9). It is always appeared as a pair with both the b and $\neg b$ cases, referring to the two state divisions above. In the post-condition manipulation, we substitute $\kappa \uplus \kappa'$ with $\mathcal{U}(\neg b, \kappa \uplus \kappa')$ in P, representing the unchanged and frozen state; and substitute κ' with $\mathcal{U}(b, \kappa \uplus \kappa')$ in Q', representing the result of applying s on the unfrozen part, and assemble them together through the separation operation $*$. $\Omega; \{\kappa' : \texttt{CH}\} \vdash Q'$ ensures that Q' only mentions session κ'. $\mathcal{U}$'s modeling in the first line of Figure 12 captures the assemble procedure by merging two $\mathcal{U}$ constructs together. Notice that c_j appears in two $\mathcal{U}$ constructs, meaning that we actually utilize c_j to distinguish the frozen and unfrozen basis states without using b. Rules SLOOP and PLOOP are the semantic and proof rules for a for-loop, which is a simplified version of while-loop in classical separation logic, because for-loop is guaranteed to terminate.

$$\frac{\Omega; \{\kappa : \texttt{CH}\} \vdash_{\texttt{M}} \{\kappa \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle |1\rangle\}\, s\, \{\kappa \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |1\rangle\}}{\Omega; \{\kappa : \texttt{CH}\} \vdash_{\texttt{M}} \{\mathcal{M}(x[j], \kappa) \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle |0\rangle\}\, s\, \{\kappa \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |1\rangle\}}$$

$$\frac{\Omega; \{\kappa : \texttt{CH}\} \vdash_{\texttt{M}} \{\mathcal{M}(x[j], \kappa) \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle |0\rangle\}\, s\, \{\kappa \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |1\rangle\}}{\Omega; \{\kappa_1 : \texttt{CH}\} \vdash_{\texttt{C}} \{\kappa_1 \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle |0\rangle\}\ \texttt{if}\ (x[j])\ s\, \{\mathcal{U}(\neg x[j], \kappa_1) \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle |0\rangle * \mathcal{U}(x[j], \kappa_1) \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |1\rangle\}}$$

$$\frac{\Omega; \{\kappa_1 : \texttt{CH}\} \vdash_{\texttt{C}} \{\kappa_1 \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle |0\rangle\}\ \texttt{if}\ (x[j])\ s\, \{\mathcal{U}(\neg x[j], \kappa_1) \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle |0\rangle * \mathcal{U}(x[j], \kappa_1) \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |1\rangle\}}{\Omega; \sigma \vdash_{\texttt{C}} \{x[0..\texttt{S}\ j] \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle * x[\texttt{S}\ j..n] \mapsto |\bar{0}\rangle\}\ \texttt{if}\ (x[j])\ s\, \{x[0..j+2] \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}} |\bar{i}\rangle * x[j+2..n] \mapsto |\bar{0}\rangle\}}$$

$$\kappa = \{x[0..j], x[\texttt{S}\ j..j+2]\} \quad \kappa_1 = \{x[j..\texttt{S}\ j\} \uplus \kappa \quad s = x[\texttt{S}\ j] \leftarrow x[\texttt{S}\ j] + 1; \quad \sigma = \{x[0..\texttt{S}\ j] : \texttt{CH}, x[\texttt{S}\ j..n] : \texttt{Nor}\}$$

As an example, we show the proof, built from bottom up, for a GHZ loop-step above. We first move a Nor type qubit $x[\texttt{S}\ j]$ to session $x[0..\texttt{S}\ j]$ and use the FRAME rule to disregard session $x[j+2..n]$. We then apply rule PIF to freeze the qubit $x[j]$, with the basis state having $x[j] = 0$ frozen and disregarded, and left with the state $\frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle |1\rangle$, with the purple part also being frozen. Notice that κ_1 has the same qubits as session $x[0..j+2]$ with different arrangement by applying some equivalence state rewrites. We apply rule PA-CH at the top and flip the 0 bit. As the post-condition of rule PIF, we use two $\mathcal{U}$ functions to assemble the state $\frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |1\rangle$ back to the state of session κ_1.

Measurement Rules. As Figure 10 describes, quantum measurement is a two-step array filter: 1) The session is partitioned into two parts, so do all the basis states, and we randomly pick a basis state first part as a key (the pink part); and 2) we create a new array by cutting all elements' first parts with keeping the elements whose original first part is equal to the key. Notice that the the red basis states appear in a periodical pattern in the whole array. This behavior is universally true for quantum operations, and many quantum algorithms utilize the periodical pattern.

$$\textsc{SDis}\ \frac{FV(\Omega, l) = \kappa \qquad \varphi(\kappa) = \{\kappa \uplus \kappa' \mapsto q\}}{(\varphi, l \leftarrow \texttt{dis}) \longrightarrow (\varphi[\kappa \uplus \kappa' \mapsto \mathcal{D}(|\kappa|, q)], \{\})} \qquad \textsc{PDis}\ \frac{FV(\Omega, l) = \kappa \qquad \sigma(\kappa) = \{\kappa \uplus \kappa : \texttt{CH}\}}{\Omega; \sigma \vdash_g \{\kappa \uplus \kappa' \mapsto q\}\ l \leftarrow \texttt{dis}\ \{\kappa \uplus \kappa' \mapsto \mathcal{D}(|\kappa|, q)\}}$$

$$\mathcal{D}(n, \Sigma_{i=0}^{m} \Sigma_{j=0}^{2^n} z_{ij}\, |j\rangle\, |c_{ij}\rangle) = \Sigma_{i=0}^{m} \Sigma_{j=0}^{2^n} (\tfrac{1}{2^{n-1}} \Sigma_{u=0}^{2^n} z_{iu} - z_{ij})\, |j\rangle\, |c_{ij}\rangle$$

Fig. 14. Semantic and Proof Rules for Diffusion Operations

In rule SMea in Figure 13, we pick an n-length bitstring c as the pink key and collect m basis states $\Sigma_{j=0}^{m} z_j\, |c\rangle\, |c_j\rangle$ that has the key c. In the post-state, we update the remaining session κ to $\Sigma_{j=0}^{m} \frac{z_j}{\sqrt{r}}\, |c_j\rangle$ with the adjustment of amplitude $\frac{1}{\sqrt{r}}$, and replace the variable x in the statement s with the value $(r, \{\!|c|\!\})$. In designing the proof rule PMea, a session type operation $\mathcal{F}(x, n, \kappa)$ is invented to do exactly the two steps above by selecting an n-length prefix bitstring c in a basis state for range $y[0..n]$, computing the probability r, and assigning $(r, \{\!|c|\!\})$ to variable x. Rule PMea replaces session $y[0..n] \cup \kappa$ in P with the measurement result session $\mathcal{F}(x, n, \kappa)$ and updates the type state Ω and σ by replacing $y[0..n] \cup \kappa$ with κ.

$$\frac{\Omega[u \mapsto \texttt{M}]; \{x[0..n] : \texttt{CH}\} \vdash_{\texttt{C}} \{\mathcal{F}(u, n, x[0..n]) \mapsto C\}\ \{\}\ \{x[0..n] \mapsto D * E\}}{\Omega; \{\{y[0..n], x[0..n]\} : \texttt{CH}\} \vdash_{\texttt{C}} \{\{y[0..n], x[0..n]\} \mapsto C\}\ \texttt{let}\ u\ =\ \texttt{measure}(y)\ \texttt{in}\ \{\}\ \{x[0..n] \mapsto D * E\}}$$

$$C \triangleq \Sigma_{j=0}^{2^n} \tfrac{1}{\sqrt{2^n}}\, |(|a^j \,\%\, N|).(|j|)\rangle \quad D \triangleq \tfrac{1}{\sqrt{s}} \Sigma_{k=0}^{s} |t + kp\rangle \quad E \triangleq p = \texttt{ord}(a, N) \wedge u = (\tfrac{s}{2^n}, a^t \,\%\, N) \wedge s = \texttt{rnd}(\tfrac{2^n}{p})$$

Here, we show a proof fragment above for the partial measurement in line 14 in Figure 3. The proof applies rule PMea by replacing session $\{x[0..n], y[0..n]\}$ with $\mathcal{F}(u, n, x[0..n])$. On the top, the pre- and post-conditions are equivalent as explained below. In session $\{y[0..n], x[0..n]\}$, range $y[0..n]$ stores the basis state $(|a^j \,\%\, N|)$, which contains value j that represents the basis states for range $x[0..n]$. Selecting a basis state $a^t \,\%\, N$ also filters the 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 periodical function, which means that the relation can be rewritten $a^{t+kp} \,\%\, N = a^t \,\%\, N$, and p is the order. Thus, the $x[0..n]$ state is rewritten as a summation of k: $\frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} |t + kp\rangle$. The probability of selecting $(|a^j \,\%\, N|)$ is $\frac{s}{2^n}$. In Qafny , we set up additional axioms for these periodical theorems to grant this kind of pre- and post-condition equivalence.

Rules for Diffusion. Quantum diffusion operations ($l \leftarrow \texttt{dis}$) reorient the amplitudes of basis states based on the basis state corresponding to l. They are analogized to an aggregate operation of reshape and mean computation, both appeared in some programming languages, such as Python. The aggregate operation first applies a reshape, where elements are regrouped into a normal form, as the first arrow of Figure 11. More specifically, the diffusion function $\mathcal{D}(n, q)$ (Figure 14) first takes an n'-element CH type state $\Sigma^{t=0} z_t\, |c_t\rangle$, where n corresponds to the number of qubits in l. Then, we rearrange the state by extending the element number from n' to $m * n$ with probably adding new elements that originally have zero amplitude (the white elements in Figure 11). Here, let's view a basis c_t as a small-endian (LSB) number $\{\!|c_t|\!\}$. The rearrangement of changing bases c_t (for all t) to $(|j|).c_{ij}$ is analogized to rewrite a number $\{\!|c_t|\!\}$ to be the form $2^n i + j$, with $j \in [0, 2^n)$, i.e., the reshape step rearranges the basis states to be a periodical counting sequence, with 2^n being the order. The mean computation analogy (the second arrow in Figure 11) takes every period in the reshaped state, and for each basis state in a period, we redistribute its amplitude by the formula $(\frac{1}{2^{n-1}} \Sigma_{u=0}^{2^n} z_{iu} - z_{ij})$. Rule SDis is the semantics for diffusion $l \leftarrow \texttt{dis}$, which applies the $\mathcal{D}$ function to session $\kappa \uplus \kappa'$, where κ contains all qubits in l. Proof rule PDis is a separation style rule that does the same as rule SDis. An example of quantum walk algorithm that uses diffusion operations is given in Section 5.2.

3.4 Qafny Metatheory

Here, we show the type soundness and proof system soundness and completeness.

Type Soundness. We prove that well-typed Qafny programs are well defined; i.e., the type system is sound with respect to the semantics, with the well-formedness assumptions (Definition C.1 and Definition 3.1). The Qafny type soundness is stated as two theorems, type progress and preservation . The proofs are done by induction on statements s and mechanized in Coq.

Theorem 3.2 (Qafny type progress). If $\Omega; \sigma \vdash_g s \triangleright \sigma'$ and $\Omega; \sigma \vdash \varphi$, then either $s = \{\}$, or there exists φ' and s' such that $(\varphi, s) \longrightarrow (\varphi', s')$.

Theorem 3.3 (Qafny type preservation). If $\Omega; \sigma \vdash_g s \triangleright \sigma'$, $\Omega; \sigma \vdash \varphi$, and $(\varphi, s) \longrightarrow (\varphi', s')$, then there exists Ω' and σ'', $\Omega'; \sigma'' \vdash_g s' \triangleright \sigma'$ and $\Omega'; \sigma'' \vdash \varphi'$.

Proof System Soundness and Completeness. We prove that the Qafny proof system is sound and complete with respect to it semantics for well-typed Qafny programs. In Qafny , there are three different state representations for a session κ and two sessions can be joined into a large session. Hence, given a statement s and an initial state ψ and φ, the semantic transition $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi', \{\})$ might not be unique, such that there might be different φ' representations. However, any `Nor` and `Had` type state can be represented as a `CH` type state, so that `CH` type states can be viewed as the *most general* state representation. We also have state equivalence relations defined for capturing the behaviors of session permutation, join and split. We define the *most general state representation* of evaluating a statement s in an initial state φ below.

Definition 3.4 (Most general Qafny state). Given $s, \varphi, \Omega, \sigma$, and g, such that $\Omega; \sigma \vdash_g \varphi$, $\Omega; \sigma \vdash_g s \triangleright \sigma^*$, $\Omega; \sigma[\uparrow\ \sigma^*] \vdash \varphi^*$, and $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi^*, \{\})$, φ^* is the most general state representation of evaluating (ψ, φ, s), iff for all σ' and φ', such that $\Omega; \sigma \vdash_g s \triangleright \sigma'$, $\Omega; \sigma[\uparrow\ \sigma'] \vdash \varphi'$ and $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi', \{\})$, $\sigma' \preceq \sigma^*$ and $\varphi' \equiv \varphi^*$.

The Qafny proof system correctness is defined by the soundness and relatively completeness theorems below, which has been formalized and proved in Coq. The Qafny proof system only describes the quantum portion built on top of the Dafny system. Since the quantum portion contains non-terminated programs, the soundness and completeness essentially refers to the partial correctness of the Qafny proof system and the total correctness is achieved by compiling Qafny programs to Dafny. The Qafny proof system correctness is defined in terms of programs being well-typed. The type soundness theorem suggests that any intermediate transitions of evaluating a well-typed Qafny program is also well-typed. Thus, the proof system correctness proof only reason about well-typed predicates in Definition 3.1.

Theorem 3.5 (proof system soundness). For a well-typed program s, such that $\Omega; \sigma \vdash_g s \triangleright \sigma'$, $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$, $\Omega; \sigma; \psi; \varphi \models_g P$, then there exists a state representation φ', such that $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi', \{\})$ and $\Omega; \sigma[\uparrow\ \sigma']; \psi'; \varphi' \models_g Q$, and there is a most general state representation φ^* of evaluating (ψ, φ, s) as $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi^*, \{\})$ and $\varphi' \equiv \varphi*$.

Theorem 3.6 (proof system relative completeness). For a well-typed program s, such that $\Omega; \sigma \vdash_g s \triangleright \sigma'$, $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi', \{\})$ and $\Omega; \sigma \vdash_g \varphi$, there is most general state representation φ^*, such that $(\psi, \varphi, s) \longrightarrow^* (\psi', \varphi', \{\})$ and $\varphi' \equiv \varphi^*$ and $\Omega; \sigma \vdash_g s \triangleright \sigma^*$ and $\Omega; \sigma[\uparrow\ \sigma^*] \vdash_g \varphi^*$, and there are predicates P and Q, such that $\Omega; \sigma; \psi; \varphi \models_g P$ and $\Omega; \sigma[\uparrow\ \sigma^*]; \psi'; \varphi^* \models Q$ and $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$.

4 QAFNY PROOF SYSTEM AND PROGRAM COMPILATION

We discuss the two compilation dimensions in QNP. First, the Qafny proof system is compiled to Dafny and utilizes its facilities for automated verification. Second, the Qafny language is compiled to sqir, a quantum circuit language, so Qafny programs can be executed in a quantum machine.

4.1 Translation from QAFNY to Dafny

We compile the QAFNY proof system to Dafny. The QAFNY proof rules utilize extra type environment to track sessions as well as state representation form transformations. However, there are no Dafny support for automatic equational rewrites of state forms, neither does Dafny predicate variables permit session structures. All of these require additional constructs and annotations to be inserted to the compiled predicates and programs when translating QAFNY to Dafny. This section shows how additional constructs and annotations are inserted, with no loss of expressiveness. Compilation is defined by extending QAFNY 's typing judgment thusly: $\Xi; \Omega; \sigma \vdash_g (P, Q, s) \triangleright (P', Q', s', \sigma')$. We now add the pre- and post-conditions P and Q for s, as well as the compilation result of the Dafny conditions P' and Q', and compiled program s', such that if the proof $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$ is valid in QAFNY , then $\{P'\}s'\{Q'\}$ is valid in Dafny. Ξ is an additional map from sessions to Dafny predicate variables, holding variables used in P' and Q' to represent sessions. We formalize rules for this judgment in Coq and prove the compilation correctness, i.e., every QAFNY quantum program verification can be correctly expressed and proved in a separation logic framework. We also faithfully implement the compiler in Dafny and verify many quantum programs in Section 5. Our compilation algorithm is evidence that the QAFNY proof system is sound with respect to classical separation logic.

Conceptually, the compilation procedure does three items. First, every time there is a change in sessions, such as qubit position permutation and split/join of sessions, we generate additional variables representing transformed sessions. Second, for a program s, if there is a change of state forms, we insert an additional construct to reflect the change so the Dafny proof system can capture the state form transformation. Third, we generate additional Dafny axioms and inference rules for quantum types and state transitions. Assume that $\Omega; \sigma \vdash_g s \triangleright \sigma'$, the compilation essentially is:

$$\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\} \quad \longrightarrow \quad \{A \wedge T \wedge \overline{P}\} s' \{A \wedge T' \wedge \overline{Q}\}$$

Here, s' is the compiled Dafny program with additional constructs inserted; $\overline{P}$ and $\overline{Q}$ are the compiled conditions of P and Q, respectively; T and T' are predicates representing session types in σ and σ', respectively; and A is a set of axioms for capturing the type and state equations as well as the array aggregate operations that support the QAFNY operation semantic rewrites rules, as those shown in Section 3.3.

$$\Omega; \{x[0..n] : \mathsf{Had}, y[0..1] : \mathsf{Nor}\} \vdash_g \{x[0..n] \mapsto C * y[0..1] \mapsto |0\rangle\}\ \kappa \leftarrow x<5@y[0]\ \{\kappa \mapsto \Sigma_{j=0}^{2^n} |j\rangle\,|j<5\rangle\} \quad \longrightarrow \quad \{T_1 *\ u_1 \mapsto C * u_2 \mapsto |0\rangle\}\ \mathtt{lift}(x[0..n])\ ;\ \mathtt{join}(x[0..n], y[0..1])\ ;\ \kappa \leftarrow x<5@y[0]\ \{T_2 \wedge \mathsf{ses}(u_3) = \mathsf{ses}(u_1) \uplus \mathsf{ses}(u_2) * u_3 \mapsto \Sigma_{j=0}^{2^n} |(|j|).(|j<5|)\rangle\}$$

$$\Omega = \{x : \mathtt{Q}\ n, y : \mathtt{Q}\ 1\} \qquad \kappa = \{x[0..n], y[0..1]\} \qquad C = \tfrac{1}{\sqrt{2^n}} \otimes_{j=0}^{n} (|0\rangle + |1\rangle)$$

$$T_1 = \mathsf{ses}(u_1) = x[0..n] \wedge \mathsf{ses}(u_2) = y[0..1] \wedge \mathsf{type}(u_1) = \mathsf{Had} \wedge \mathsf{type}(u_2) = \mathsf{Nor}$$

$$T_2 = \mathsf{ses}(u_1) = x[0..n] \wedge \mathsf{ses}(u_2) = y[0..1] \wedge \mathsf{ses}(u_3) = \kappa \wedge \mathsf{type}(u_1) = \mathsf{type}(u_2) = \mathsf{type}(u_3) = \mathsf{CH}$$

As a highlight of the compilation, we first see how to compile a simple proof of a statement $\{x[0..n], y[0..1]\} \leftarrow x<5@y[0]$ that computes the Boolean comparison of $x<5$ and stores the value to $y[0]$ above. The result of the application is an entanglement state having 2^n basis states. For every basis state, the $y[0]$ bit position stores the the result of computing $x < 5$. Since variable x initially is of type Had, we turn its type to CH by a `lift` statement, and the use a `join` statement to join the CH type (x) and Nor type (y) states. Notice that we use variables u_1, u_2 and u_3 to refer to the sessions $x[0..n]$, $y[0..1]$, and $\{x[0..n], y[0..1]\}$, respectively, and connect variables with sessions by using the ses function that takes a variable and outputs its pointed-to session. In the QAFNY compiler, we use Ξ to track such information. Here, we need to generate a new variable u_3 in Ξ to represent the join session $\{x[0..n], y[0..1]\}$ after the `join` statement. In addition, the compiled result has no type environment, therefore, we use type functions as predicates to track the session types.

In compiling quantum conditionals, not only we need to do the above type conversion, will we also explicitly insert frozen ($\mathcal{M}$) and unfrozen ($\mathcal{U}$) functions; e.g., in computing the conditional `if` $(x[0])$ $y[0]$ with $x[0]$ and $y[0]$ being types of `Had` and `Nor`, respectively, `lift` and `join` functions are added first, then we also add the $\mathcal{M}$ and $\mathcal{U}$ before and after the conditional as:

$$\texttt{lift}(x[0..1])\ ;\ \texttt{join}(x[0..1], y[0..1])\ ;\ \mathcal{M}(x[0], y[0..1])\ ;\ \texttt{if}\ (x[0])\ y[0]\ ;\ \mathcal{U}(x[0], \{x[0..1], y[0..1]\})$$

We implemented the compiler in Dafny and formalize it with the correctness theorem proof in Coq. The target Dafny formalism is separation logic [44], and we implemented its proof rules in Coq with additional axioms for the array aggregate operations for capturing QAFNY operation semantics in Section 3.3.

THEOREM 4.1 (QAFNY TO DAFNY COMPILATION CORRECTNESS). If a proof $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$ is valid to derive in QAFNY , and through the compilation process $\Xi; \Omega; \sigma \vdash_g (P, Q, s) \triangleright (P', Q', s', \sigma')$, the proof $\{P'\}\, s'\, \{Q'\}$ is valid in Dafny.

4.2 Translation from QAFNY to SQIR

QNP translates QAFNY to SQIR by mapping QAFNY qubit arrays to SQIR concrete qubit indices and expanding QAFNY instructions to sequences of SQIR gates. Translation is expressed as the judgment $\Omega; \gamma; n \vdash s \longrightarrow \epsilon$ where Ω maps QAFNY variables to their sizes, ϵ is the output SQIR circuit, γ maps a QAFNY range variable position $x[i]$, in the range $x[j..k]$ where $i \in [j, k)$, to a SQIR concrete qubit index (i.e., offset into a global qubit register), and n is the current qubit index bound. At the start of translation, for every variable x and $i < \texttt{nat}(\Omega(x))$ [13], γ maps $x[i]$ to a unique concrete index chosen from 0 to n. Ω is populated through the QAFNY type checking in Figure 6, while γ and n are populated when hitting a qubit allocation instruction (`init`) as shown in Appendix C.1.

$$\frac{\Omega; \gamma; n \vdash b@x[i] \to \epsilon \qquad \Omega; \gamma; n \vdash s \to \epsilon'}{\Omega; \gamma; n \vdash \texttt{if}\ (b@x[i])\ s \to \epsilon\ ;\ \texttt{ctrl}(\gamma(x[i]), \epsilon')} \qquad \frac{\forall t \in [i, j).\, \Omega; \gamma; n \vdash \texttt{if}\ (b[t/x])\ s[t/x] \to \epsilon_t}{\Omega; \gamma; n \vdash \texttt{for}\ (\texttt{int}\ x \in [i, j)\ \&\&\ b)\ s \to \epsilon_i\ ;\ldots;\ \epsilon_{j-1}}$$

Fig. 15. Select QAFNY to SQIR translation rules (SQIR circuits are marked blue)

Figure 15 depicts two translation rules,[14] while the rest is in Appendix C.8. The first rule describes the translation of a quantum conditional to a controlled operation in SQIR. Here, we first translate the Boolean guard $b@x[i]$ [15], and sequentially we translate the conditional body as an SQIR expression controlled on the $x[i]$ position. `ctrl` generates the controlled version of an arbitrary SQIR program using standard decompositions [37, Chapter 4.3]. The last rule translate QAFNY for-loops. Essentially, a for-loop is compiled to a series of conditionals with different loop step values.

The compiler is implemented in Coq and validated through testing. We extract both the QAFNY semantic interpreter and the QAFNY to SQIR compiler to Ocaml and compile the compiled SQIR programs to OpenQASM [8] via the SQIR compiler. Then, we run test programs in the QAFNY Ocaml interpreter and compiler and see if the results are matched. Overall, we run 135 unit test programs to test individual operations and small composed programs, and all the results from the QAFNY interpreter and compiler are matched.

5 EVALUATION AND CASE STUDIES

We evaluate QNP by (1) demonstrating how quantum algorithms are verified in the framework, and (2) showing that it saves the time for programmers to write and verify quantum programs,

[13] $\Omega(x) = \texttt{Q}\ m$ and $\texttt{nat}(\texttt{Q}\ m) = m$

[14] Translation in fact threads through the typing judgment, but we elide that for simplicity.

[15] Here, b is $a < a$, $a = a$, or `true` referring to the Boolean equation parts of $a < a@x[i]$, $a = a@x[i]$, or $x[i]$.

Algorithm	Run Time (sec)	QBricks Run Time (sec)	# Lines	QBricks # Lines	Human Effort (days)
GHZ	4.2	-	11	-	< 1
Controlled GHZ	6.4	-	7	-	< 1
Deutsch–Jozsa	3.3	79	8	57	< 1
Grover's search	26.7	283	22	193	2
Shor's algorithm	36.3	1380	31	1163	30
Quantum Walk	43.1	-	43	-	3

Fig. 16. Computer running time and program line numbers and human effort for verifying algorithms in QAFNY . Verification running time (Run Time) is measured in a i7 windows computer. QBricks running time is based on [3], and – means no data. The human effort measures the time for a single person to finish programming and verifying an algorithm. The quantum walk algorithm is the core in [53].

especially, it helps them to discover possible new ways of integrating quantum operations. This section presents the facts about verifying quantum programs in QAFNY . Then, we discuss two case studies, including the verification of the controlled GHZ and quantum walk algorithms, as demonstrations of verifying quantum algorithms in QAFNY .

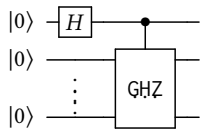
Figure 16 shows the algorithms being verified in QAFNY . When we started the QNP project, we first tried to verify Shor's algorithm directly on Dafny, which spent a researcher 30 days to finish. After that, we built the QAFNY compiler on Dafny, and run a QAFNY version of the Shor's algorithm proof, which is much more cleaner than the code written directly in Dafny. The running time for that proof is 36.3 seconds. In fact, running any QAFNY verification does not take more than a minute to finish, which is relatively comparable with most nowadays automated verification framework [27, 41], and better than the existing quantum automated frameworks, such as QBricks [3], as indicated in Figure 16.

Computer running time is usually a less important factor in verifying programs, while the human effort is more concerning. We believe that QNP saves a great amount of human resources. This fact is indicated by the number of lines in writing the algorithm specifications in QAFNY . As shown in Figure 16, all the algorithms are written with less than 43 lines of code. In contrast, algorithm specifications and verification in other framework requires much more lines. For example, the Grover's search specification in quantum Haore Logic [30] has 3184 lines of code, and the Shor's algorithm specification in QBricks [3] has 1163 lines of code in Figure 16 [16]. In terms of human effort, other than Shor's algorithm, which was done before the QAFNY compiler was fully constructed, most algorithms were written and verified in QAFNY (Figure 16) in three days by a single researcher. As a comparison, the complete Shor's algorithm correctness proof [39] was finished by four researchers that spent two years. Even the oracle in the Shor's algorithm, the modulo multiplication circuit, was verified by three researchers that spent four months. We believe that QNP relieves programmers' pain in reasoning about quantum programs. Below, we show two examples of helping programmings to understand and integrate quantum algorithms by QNP.

5.1 Controlled GHZ Case Study: Building Quantum Algorithms on Others

One important criteria appearing in many automated verification frameworks is to reuse existing verification proofs to synthesize new algorithm verification. However, in most quantum proof systems nowadays, this criteria is neglected. For example, QBricks did not utilize the quantum phase estimation (QPE) proof, which is the core part of Shor's algorithm, to construct their Shor's algorithm proof. In VOQC [19], the QPE reuse in the Shor's proof was done by proving many new

[16]We do not mean to compare the coding lines to other frameworks since the coding line numbers might be varied depending on many factors, but we only provide a hint on the automation in QAFNY .



1 $\{x[0..1] \mapsto |0\rangle, y[0..n] \mapsto |\overline{0}\rangle\}$
2 $x[0] \leftarrow \mathsf{H};$
3 $\{x[0..1] \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) * y[0..n] \mapsto |\overline{0}\rangle\}$
4 $\Rightarrow \{\{x[0..1], y[0..n]\} \mapsto \Sigma_{j=0}^{2} |j\rangle\,|\overline{0}\rangle\}$
5 $\mathtt{if}\ (x[0])\ \mathsf{ghz}(y);$
6 $\{\{x[0..1], y[0..n]\} \mapsto \frac{1}{\sqrt{2}}|\overline{0}\rangle + \frac{1}{2}|1\rangle\,|\overline{0}\rangle + \frac{1}{2}|\overline{1}\rangle\}$

Fig. 17. Controlled GHZ circuit and proof. $\mathsf{ghz}(y)$ is given in Figure 2. Lines 3-4 are automatically inferred.

theorems that were not originally required in the QPE proofs. QNP opens a windows towards the reusable proofs for verifying new algorithms. Figure 17 provides a proof of the Controlled GHZ algorithm based on the GHZ proof in Figure 2e. The focal point is at Figure 17 line 5, where the GHZ function requires input to be a Nor state of all 0 qubits, but the given state, in line 4, is an entanglement $\Sigma_{j=0}^{2}|j\rangle\,|\overline{0}\rangle$. In Qafny , we automatically verify the proof by rule PIF and the equational relation to rewrite a singleton CH state to a Nor state, as $\Sigma_{j=0}^{1} z_j\,|c_j\rangle \equiv z_0\,|c_0\rangle$. The detailed proofs for the conditional is given below, where $\kappa = \{x[0..1], y[0..n]\}$.

$$\frac{\frac{\frac{\Omega; \{y[0..n] : \mathsf{Nor}\} \vdash_{\mathsf{M}} \{y[0..n] \mapsto |\overline{0}\rangle|1\rangle\}\ \mathsf{ghz}(y)\ \{y[0..n] \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}}|\bar{i}\rangle|1\rangle\}}{\Omega; \{y[0..n] : \mathsf{CH}\} \vdash_{\mathsf{C}} \{\mathcal{M}(x[0], y[0..n]) \mapsto \sum_{j=0}^{2}|j\rangle\,|\overline{0}\rangle\}\ \mathsf{ghz}(y)\ \{y[0..n] \mapsto \sum_{i=0}^{2} \frac{1}{\sqrt{2}}|\bar{i}\rangle|1\rangle\}}}{\Omega; \sigma \vdash_{\mathsf{C}} \{\kappa \mapsto \sum_{j=0}^{2}|j\rangle\,|\overline{0}\rangle\}\ \mathtt{if}\ (x[0])\ \mathsf{ghz}(y)\ \{\mathcal{U}(\neg x[0], \kappa) \mapsto \sum_{j=0}^{2}|j\rangle\,|\overline{0}\rangle * \mathcal{U}(x[0], \kappa) \mapsto \sum_{i=0}^{2}\frac{1}{\sqrt{2}}|\bar{i}\rangle|1\rangle\}}}{\Omega; \sigma \vdash_{\mathsf{C}} \{\kappa \mapsto \sum_{j=0}^{2}|j\rangle\,|\overline{0}\rangle\}\ \mathtt{if}\ (x[0])\ \mathsf{ghz}(y)\ \{\kappa \mapsto \frac{1}{\sqrt{2}}|\overline{0}\rangle + \frac{1}{2}|1\rangle\,|\overline{0}\rangle + \frac{1}{2}|\overline{1}\rangle\}}$$

Once rule PIF is applied on the second and third line, session $y[0..n]$'s state is actually a Nor type state $|\overline{0}\rangle|1\rangle$, where $\overline{0}$ is n qubits and $|1\rangle$ is frozen, which is marked purple. Now, the state satisfies the input state for GHZ above, so that we can safely reuse the GHZ proof in Figure 2e and finish the proof. The type of $y[0..n]$: CH on the top of Figure 17 is turned to Nor by extra axioms defined in the Qafny to Dafny compiler to rewrite type predicates. In Appendix C, we implement a new type system that tracks both sessions and basis states, where the type rewrite can automatically happens without extra predicate axioms.

5.2 Case Study: Understanding Quantum Walk

Quantum walk [4, 52, 53] is a quantum version of the classical random walk [43], and an important algorithmic protocol for writing quantum algorithms. However, most quantum walk analyses were based on Hamiltonian simulation evolving, which deters many computer programmers to invent quantum walk algorithms. Here, we show that discrete time quantum walk, at its very least, is a quantum version of breadth first search.

Figure 18 provides the core loop of a discrete time quantum walk algorithm on a complete binary tree. In quantum walk, there are four quantum array pieces: x (size t) is the loop step register in superposition; y (size $m = 2^t$) stores the result of evaluating $x <$ S j in every loop step; u (size 1) is the coin, 1 for the left direction and 0 for the right one, and determines the moving direction of next step ; and w (size n) stores the node keys with $\overline{0}$ being the root node. The loop entangles all these four pieces together as session $\{x[0..t], y[0..j], u[0..1], w[0..n]\}$. Before the j-th step, every basis states are divided into two sets based on the values of range $x[0..t]$, described by $q(j)$'s two parts separated by +, if a basis state has $\{\!|x[0..n]|\!\} < j$, it is in the active set, while the rest ones

1 $\{x[0..t] \mapsto \frac{1}{\sqrt{2^t}} \bigotimes_{i=0}^{t} (|0\rangle + |1\rangle) * y[0..m] \mapsto |\bar{0}\rangle * u[0..1] \mapsto |0\rangle * w[0..n] \mapsto |\bar{0}\rangle * m = 2^t \wedge m < n\}$
2 for (int $j \in [0, m)$ && $x <$ S j @ $y[j]$)
3 $\{\{x[0..t], y[0..j], u[0..1], w[0..n]\} \mapsto q(j) \wedge \texttt{is\_steps}(j, q(j))\}$
4 { $u \leftarrow$ dis;
5 if ($u[0]$) left(w);
6 if ($\neg u[0]$) right(w); }
7 $\{\{x[0..n], y[0..m], u[0..1], w[0..n]\} \mapsto q(m) \wedge \texttt{is\_steps}(m, q(m))\}$

$\texttt{pat}(i,j) = |0\rangle^{\otimes \lfloor \log\ i \rfloor}\, |1\rangle^{\otimes (j - \lfloor \log\ i \rfloor)}$
$q(j) = \Sigma_{i=0}^{2^{(\texttt{S}\ j)}-2} z_i\, |\lfloor \log\ i \rfloor\rangle\, |\texttt{pat}(i,j)\rangle\, |u_i\rangle\, |\texttt{key}(i)\rangle + \Sigma_{k=j}^{2^t} z_k\, |k\rangle\, |\bar{0}\rangle\, |0\rangle\, |\bar{0}\rangle$
$\texttt{is\_steps}(j, q(j)) = \forall i.\ |\lfloor \log\ i \rfloor\rangle\, |\texttt{pat}(i,j)\rangle\, |u_i\rangle\, |\texttt{key(i)}\rangle \in q(j) \Rightarrow \texttt{is\_suc}(j - \lfloor \log\ i \rfloor, \texttt{key}(i))$

Fig. 18. Quantum walk reachable node verification for a complete binary tree. As a circuit, the conditional (if ($\neg u[0]$) s) is interpreted as X($u[0]$); if ($u[0]$) s. left and right reach the left and right children in a tree. $q(j)$ is a quantum state with variable j. key(i) accesses i-th node's key in a tree. is_suc(t, i) judges if i is a t-depth node. $\lfloor r \rfloor$ rounds r down to nearest nat.

are inactive. In the loop, we first compare $x[0..n]$ with S j; move exactly one basis state, the one $\{\!|x[0..n]|\!\} = j$, to the active set; insert $y[j]$ qubit into the four piece session; and store the Boolean result $\{\!|x[0..n]|\!\} <$ S j to $y[j]$. Then, we apply a diffusion operation on the coin of all active set basis states and double the active set size. In each loop step, we have $2^{(\texttt{S}\ j)} - 2$ numbers of active basis states. For every basis state, diffusion redistributes its amplitude to "copy" a new basis state with exactly the same content except that the u coin is now in an opposite direction. The next two statements in Figure 18 line 5-6 transition every active set basis state's node key to its child's keys depending on the coin directions [17].

If the for-loop executes m steps, the result state contains all possible tree nodes up to m-depth except the root node, stored as basis states; stated as the post-condition in Figure 18. Apparently, the amplitudes, which represent the basis state likelihood, are low for each basis state here. In quantum computing, there are amplification operations that serve the opposite of quantum diffusion to increase certain basis state amplitudes. In the full Qafny implementation, we have an amplifier operation $w \leftarrow \texttt{reduce}(\bar{0})$ that reduces the root node amplitude while increases the other basis state amplitudes. We can insert the operation in Figure 18 line 4, so that every time if am active set basis state holds the root node key in range w, we reduce its amplitude. Notice that every new basis state that just becomes active in each loop step starts with the root node, which means that basis states having S j-th depth nodes have a higher amplitudes than basis states having j-th depth nodes, so that leaf nodes having the biggest amplitudes, which is ideal because most tree algorithms are likely to work on the leaves rather than the middle transition nodes.

6 RELATED WORK

Quantum Proof Systems and Verification Frameworks. Previous quantum proof systems, including quantum Hoare Logics [11, 30, 54, 55], quantum separation logics [25, 57], quantum relational logics [29, 51], and probabilistic Hoare logic for quantum programs [22], enlightened the development of QNP. The problems of these works are three: 1) their conditionals are solely classical, while QNP has quantum conditionals; 2) most of them are theoretical works or implemented as tactics in an interactive theorem provers, so that it is unclear if they can be implemented in a classical computer and utilize classical SMT solvers for proof automation for verifying comprehensive

[17] The tree representations and left and right functions can be implemented as data structures based on ©Qasm.

quantum programs; and 3) they did not compile quantum programs to circuits. Most quantum programming languages are either built by meta-programs embedded in a host language, such as Python (for Qiskit [6], Cirq [13], PyQuil [45], and others), Haskell (for Quipper [14]), or Coq (for SQIR and VOQC [19]), or contain some high level operations with the mix of some circuit gates without a compiler, like [2] and [35]. Quantum separation logic [25] discusses a frame rule that indicates a Had type quantum state can be split into two parts, which is similar to our Had type state split equations but different from the frame rule in QNP that is based directly on classical separation logic frame rule and utilizing the QAFNY type system to ensure the separation.

There are works on formally verifying quantum programs includes Qwire [42], SQIR [18], and QBricks [3]. These tools have been used to verify a range of quantum algorithms, from Grover's search to quantum phase estimation. The former two tools provided libraries in a proof assistant to help verify quantum programs and they have circuit compilation models, while the latter one built a proof system on top of a proof assistant to achieve some proof automation without providing a circuit compilation model. The comparison between QBricks and QNP is given in Section 5.

Classical Proof Systems. The QAFNY proof system is enlightened by the classical seapration logic [44], and many others [21, 31, 36, 50, 56]. Especially, the QAFNY proof system is compiled to Dafny [26], a mechanized separation logic system. The methodology of QNP is enlightened by the natural proof methodology [31, 32, 38], which is discussed in Section 2.

7 CONCLUSION

We present QNP, a system for expressing and automatically verifying classical quantum hybrid programs, whose quantum components can be compiled to quantum circuits. QNP's methodology is to 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 to classical verification infrastructure. The key component of QNP is QAFNY , a quantum programming language admitting a proof system, which allows programmers to specify quantum programs and logic properties that are automatically verified against the programs. The QAFNY proof system is sound and complete with respect to the QAFNY semantics for well-typed QAFNY programs. We have verified the soundness of the translator from QAFNY to Dafny, and utilized the Dafny proof infrastructure to verify many quantum programs. We also compile QAFNY programs to SQIR, which allows the quantum components to run on a quantum computer. We have demonstrated the ability of using QNP to develop and verify new quantum algorithms and we believe that the classical interpretation of quantum operations help programmers to understand quantum computation.

REFERENCES

[1] Stephane Beauregard. 2003. Circuit for Shor's Algorithm Using 2n+3 Qubits. *Quantum Info. Comput.* 3, 2 (March 2003), 175–185.

[2] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3385412.3386007

[3] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. An Automated Deductive Verification Framework for Circuit-building Quantum Programs. In *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 (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 148–177. https://doi.org/10.1007/978-3-030-72019-3_6

[4] Andrew Childs, Ben Reichardt, Robert Spalek, and Shengyu Zhang. 2007. Every NAND formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a Quantum Computer. (03 2007).

[5] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order*

Logics, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–42.

[6] Andrew Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. In *APS Meeting Abstracts*.

[7] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. *arXiv e-prints* (Jul 2017). arXiv:1707.03429 [quant-ph]

[8] Andrew W. Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2021. OpenQASM 3: A broader and deeper quantum assembly language. arXiv:2104.14722 [quant-ph]

[9] J. I. den Hartog. 1999. Verifying Probabilistic Programs Using a Hoare like Logic. In *Advances in Computing Science — ASIAN'99*, P. S. Thiagarajan and Roland Yap (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–125.

[10] Thomas G. Draper. 2000. Addition on a Quantum Computer. *arXiv: Quantum Physics* (2000).

[11] Yuan Feng and Mingsheng Ying. 2021. Quantum Hoare Logic with Classical Variables. *ACM Transactions on Quantum Computing* 2, 4, Article 16 (dec 2021), 43 pages. https://doi.org/10.1145/3456877

[12] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (April 2021), 433. https://doi.org/10.22331/q-2021-04-15-433

[13] Google Quantum AI. 2019. Cirq: An Open Source Framework for Programming Quantum Computers. https://quantumai.google/cirq

[14] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 333–342. https://doi.org/10.1145/2491956.2462177

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

[16] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866 arXiv:quant-ph/9605043

[17] Lov K. Grover. 1997. Quantum Mechanics Helps in Searching for a Needle in a Haystack. *Phys. Rev. Lett.* 79 (July 1997), 325–328. Issue 2. https://doi.org/10.1103/PhysRevLett.79.325 arXiv:quant-ph/9706033

[18] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021. Proving Quantum Programs Correct. In *Proceedings of the Conference on Interative Theorem Proving (ITP)*.

[19] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.

[20] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[21] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

[22] Yoshihiko Kakutani. 2009. A Logic for Formal Verification of Quantum Programs. In *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*, Anupam Datta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–93.

[23] Emmanuel Knill. 1996. *Conventions for quantum pseudocode*. Technical Report. Los Alamos National Lab., NM (United States).

[24] Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. 2022. A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 36 (jan 2022), 27 pages. https://doi.org/10.1145/3498697

[25] Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. 2022. A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 36 (jan 2022), 27 pages. https://doi.org/10.1145/3498697

[26] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

[27] K. Rustan M. Leino and Michał Moskal. 2014. Co-induction Simply. In *FM 2014: Formal Methods*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer International Publishing, Cham, 382–398.

[28] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified Compilation of Quantum Oracles. In *OOPSLA 2022*. https://doi.org/10.48550/ARXIV.2112.06700

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

[30] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019. Formal Verification of Quantum Algorithms Using Quantum Hoare Logic. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 187–207.

[31] Christof Löding, P. Madhusudan, and Lucas Peña. 2017. Foundations for Natural Proofs and Quantifier Instantiation. *Proc. ACM Program. Lang.* 2, POPL, Article 10 (dec 2017), 30 pages. https://doi.org/10.1145/3158098

[32] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. 2012. Recursive Proofs for Inductive Tree Data-Structures. *SIGPLAN Not.* 47, 1 (jan 2012), 123–136. https://doi.org/10.1145/2103621.2103673

[33] Igor L. Markov and Mehdi Saeedi. 2012. Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation. *Quantum Info. Comput.* 12, 5–6 (May 2012), 361–394.

[34] Narciso Martí-Oliet and José Meseguer. 2000. Rewriting logic as a logical and semantic framework. In *Electronic Notes in Theoretical Computer Science*, J. Meseguer (Ed.), Vol. 4. Elsevier Science Publishers.

[35] Microsoft. 2017. *The Q# Programming Language.* https://docs.microsoft.com/

[36] Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. 2018. Invariant Synthesis for Incomplete Verification Engines. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 232–250.

[37] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information* (10th anniversary ed.). Cambridge University Press, USA.

[38] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

[39] Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. 2022. A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm. https://doi.org/10.48550/ARXIV.2204.07112

[40] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. *SIGPLAN Not.* 48, 6 (jun 2013), 231–242. https://doi.org/10.1145/2499370.2462169

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

[42] Robert Rand. 2018. *Formally verified quantum programming.* Ph.D. Dissertation. University of Pennsylvania.

[43] Rayleigh. [n.d.]. The Problem of the Random Walk. *Nature* 72 ([n. d.]), 318–318.

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

[45] Rigetti Computing. 2021. PyQuil: Quantum programming in Python. https://pyquil-docs.rigetti.com

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

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

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

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

[50] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. https://doi.org/10.48550/ARXIV.1609.00919

[51] Dominique Unruh. 2019. Quantum Relational Hoare Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 33 (jan 2019), 31 pages. https://doi.org/10.1145/3290346

[52] Salvador Elías Venegas-Andraca. 2012. Quantum walks: a comprehensive review. *Quantum Information Processing* 11, 5 (jul 2012), 1015–1106. https://doi.org/10.1007/s11128-012-0432-5

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

[54] Mingsheng Ying. 2012. Floyd–Hoare Logic for Quantum Programs. *ACM Trans. Program. Lang. Syst.* 33, 6, Article 19 (Jan. 2012), 49 pages. https://doi.org/10.1145/2049706.2049708

[55] Mingsheng Ying. 2019. Toward Automatic Verification of Quantum Programs. *Form. Asp. Comput.* 31, 1 (feb 2019), 3–25. https://doi.org/10.1007/s00165-018-0465-3

[56] Bohua Zhan. 2018. Efficient Verification of Imperative Programs Using Auto2. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 23–40.

[57] Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. A Quantum Interpretation of Bunched Logic and Quantum Separation Logic. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science* (Rome, Italy) *(LICS '21)*. Association for Computing Machinery, New York, NY, USA, Article 75, 14 pages. https://doi.org/10.1109/LICS52264.2021.9470673

```
method Shor ( a : int, N : int, n : int, m : int, x : Q[n], y : Q[n] )
 requires (n > 0)
 requires (1 < a < N)
 requires (N < 2^(n-1))
 requires (N^2 < 2^m ≤ 2 * N^2)
 requires (gcd(a, N) == 1)
 requires ( type(x) = Tensor n (Nor 0))
 requires ( type(y) = Tensor n (Nor 0))
 ensures (gcd(N, r) == 1)
 ensures (p.pos ≥ 4 / (PI ^ 2))
{
  x *= H ;
  y *= cl(y+1); //cl can be omitted.
  for (int i = 0; i < n; x[i]; i ++)
    invariant (0 ≤ i ≤ n)
    invariant (saturation(x[0..i]))
    invariant (type(y,x[0..i]) = Tensor n (ch (2^i) {k | j baseof x[0..i] && k = (a^j mod N,j)}))
    //psum(k=b,M,p(k),b(k)) = sum_{k=b}^M p(k)*b(k)
    invariant ((y,x[0..i]) == psum(k=0,2^i,1,(a^k mod N,k)))
  {
    y *= cl(a^(2^i) * y mod N);
  }

 M z := measure(y);//partial measurement, actually measure(y,r) r is the period
 x *= RQFT;
 M p := measure(x); //p.pos and p.base
 var r := post_period(m,p.base) // ∃ t. 2^m * t / r = p.base
}
```

Fig. 19. Shor's Algorithm in Q-Dafny

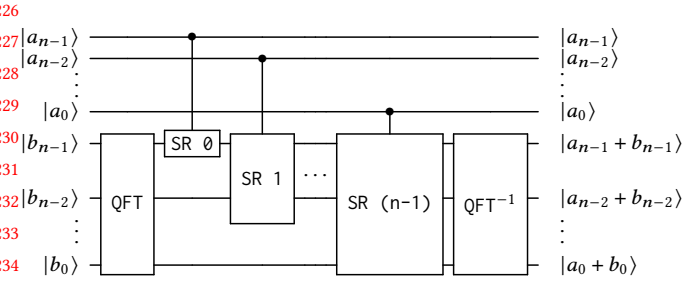
A ℚASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

We designed ℚASM to be able to express efficient quantum oracles that can be easily tested and, if desired, proved correct. ℚASM operations leverage both the standard computational basis and an alternative basis connected by the quantum Fourier transform (QFT). ℚASM's type system tracks the bases of variables in ℚASM programs, forbidding operations that would introduce entanglement. ℚASM states are therefore efficiently represented, so programs can be effectively tested and are simpler to verify and analyze. In addition, ℚASM uses *virtual qubits* to support *position shifting operations*, which support arithmetic operations without introducing extra gates during translation. All of these features are novel to quantum assembly languages.

This section presents ℚASM states and the language's syntax, semantics, typing, and soundness results. As a running example, we use the QFT adder [1] shown in Figure 20. The Coq function `rz_adder` generates an ℚASM program that adds two natural numbers `a` and `b`, each of length `n` qubits.

A.1 ℚASM States

An ℚASM program state is represented according to the grammar in Figure 21. A state φ of d qubits is a length-d tuple of qubit values q; the state models the tensor product of those values. This means that the size of φ is $O(d)$ where d is the number of qubits. A d-qubit state in a language like SQIR is represented as a length 2^d vector of complex numbers, which is $O(2^d)$ in the number of qubits. Our linear state representation is possible because applying any well-typed ℚASM program on any well-formed ℚASM state never causes qubits to be entangled.



(a) Quantum circuit

```
Fixpoint rz_adder' (a b:var) (n:ℕ)
  := match n with
     | 0 ⇒ ID (a,0)
     | S m ⇒ CU (a,m) (SR m b);
              rz_adder' a b m
     end.
Definition rz_adder (a b:var) (n:ℕ)
  := Rev a ; Rev b ; QFT b ;
     rz_adder' a b n;
     QFT⁻¹ b; Rev b ; Rev a.
```

(b) 𝒪QASM metaprogram (in Coq)

Fig. 20. Example 𝒪QASM program: QFT-based adder

Bit	b	::=	$0 \mid 1$
Natural number	n	$\in$	$\mathbb{N}$
Real	r	$\in$	$\mathbb{R}$
Phase	$\alpha(r)$	::=	$e^{2\pi i r}$
Basis	τ	::=	Nor | Phi n
Unphased qubit	$\overline{q}$	::=	$\|b\rangle \mid \|\Phi(r)\rangle$
Qubit	q	::=	$\alpha(r)\overline{q}$
State (length d)	φ	::=	$q_1 \otimes q_2 \otimes \cdots \otimes q_d$

Fig. 21. 𝒪QASM state syntax

Position	p	::=	(x, n) Nat. Num n Variable x
Instruction	ι	::=	ID p | X p | $\text{RZ}^{[-1]}$ n p | ι ; ι
		|	$\text{SR}^{[-1]}$ n x | $\text{QFT}^{[-1]}$ n x | CU p ι
		|	Lshift x | Rshift x | Rev x

Fig. 22. 𝒪QASM syntax. For an operator OP, $\text{OP}^{[-1]}$ indicates that the operator has a built-in inverse available.

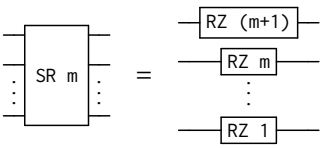


Fig. 23. SR unfolds to a series of RZ instructions

A qubit value q has one of two forms $\overline{q}$, scaled by a global phase $\alpha(r)$. The two forms depend on the *basis* τ that the qubit is in—it could be either Nor or Phi. A Nor qubit has form $|b\rangle$ (where $b \in \{0, 1\}$), which is a computational basis value. A Phi qubit has form $|\Phi(r)\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \alpha(r)\,|1\rangle)$, which is a value of the (A)QFT basis. The number n in Phi n indicates the precision of the state φ. As shown by Beauregard [1], arithmetic on the computational basis can sometimes be more efficiently carried out on the QFT basis, which leads to the use of quantum operations (like QFT) when implementing circuits with classical input/output behavior.

A.2 𝒪QASM Syntax, Typing, and Semantics

[Liyi: add RZ gate back]

Figure 22 presents 𝒪QASM's syntax. An 𝒪QASM program consists of a sequence of instructions ι. Each instruction applies an operator to either a variable x, which represents a group of qubits, or a *position* p, which identifies a particular offset into a variable x.

The instructions in the first row correspond to simple single-qubit quantum gates—ID p, X p, and $\text{RZ}^{[-1]}$ n p —and instruction sequencing. The instructions in the next row apply to whole

$$\text{X}\ \frac{\Omega(x) = \texttt{Nor} \qquad n < \Sigma(x)}{\Sigma; \Omega \vdash \texttt{X}\ (x, n) \triangleright \Omega} \qquad \text{RZ}\ \frac{\Omega(x) = \texttt{Nor} \qquad n < \Sigma(x)}{\Sigma; \Omega \vdash \texttt{RZ}\ q\ (x, n) \triangleright \Omega}$$

$$\text{SR}\ \frac{\Omega(x) = \texttt{Phi}\ n \qquad m < n}{\Sigma; \Omega \vdash \texttt{SR}\ m\ x \triangleright \Omega} \qquad \text{QFT}\ \frac{\Omega(x) = \texttt{Nor} \qquad n \leq \Sigma(x)}{\Sigma; \Omega \vdash \texttt{QFT}\ n\ x \triangleright \Omega[x \mapsto \texttt{Phi}\ n]}$$

$$\text{RQFT}\ \frac{\Omega(x) = \texttt{Phi}\ n \qquad n \leq \Sigma(x)}{\Sigma; \Omega \vdash \texttt{QFT}^{-1}\ n\ x \triangleright \Omega[x \mapsto \texttt{Nor}]} \qquad \text{CU}\ \frac{\Omega(x) = \texttt{Nor} \qquad \texttt{fresh}\ (x, n)\ \iota \qquad \Sigma; \Omega \vdash \iota \triangleright \Omega \qquad \texttt{neutral}(\iota)}{\Sigma; \Omega \vdash \texttt{CU}\ (x, n)\ \iota \triangleright \Omega}$$

$$\text{LSH}\ \frac{\Omega(x) = \texttt{Nor}}{\Sigma; \Omega \vdash \texttt{Lshift}\ x \triangleright \Omega} \qquad \text{SEQ}\ \frac{\Sigma; \Omega \vdash \iota_1 \triangleright \Omega' \qquad \Sigma; \Omega' \vdash \iota_2 \triangleright \Omega''}{\Sigma; \Omega \vdash \iota_1\ ;\ \iota_2 \triangleright \Omega''}$$

Fig. 24. Select ℴQASM typing rules

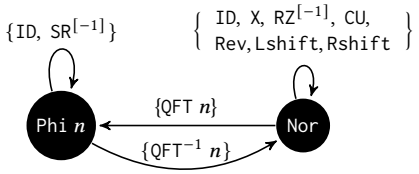


Fig. 25. Type rules' state machine

variables: QFT n x applies the AQFT to variable x with n-bit precision and $\texttt{QFT}^{-1}$ n x applies its inverse. If n is equal to the size of x, then the AQFT operation is exact. $\texttt{SR}^{[-1]}$ n x applies a series of RZ gates (Figure 23). Operation CU p ι applies instruction ι *controlled* on qubit position p. All of the operations in this row—SR, QFT, and CU—will be translated to multiple SQIR gates. Function rz_adder in Figure 20(b) uses many of these instructions; e.g., it uses QFT and $\texttt{QFT}^{-1}$ and applies CU to the mth position of variable a to control instruction SR m b.

In the last row of Figure 22, instructions Lshift x, Rshift x, and Rev x are *position shifting operations*. Assuming that x has d qubits and x_k represents the k-th qubit state in x, Lshift x changes the k-th qubit state to $x_{(k+1)\%d}$, Rshift x changes it to $x_{(k+d-1)\%d}$, and Rev changes it to x_{d-1-k}. In our implementation, shifting is *virtual* not physical. The ℴQASM translator maintains a logical map of variables/positions to concrete qubits and ensures that shifting operations are no-ops, introducing no extra gates.

Other quantum operations could be added to ℴQASM to allow reasoning about a larger class of quantum programs, while still guaranteeing a lack of entanglement. In **??**, we show how ℴQASM can be extended to include the Hadamard gate H, z-axis rotations Rz, and a new basis Had to reason directly about implementations of QFT and AQFT. However, this extension compromises the property of type reversibility (Theorem A.5, Appendix A.3), and we have not found it necessary in oracles we have developed.

Typing. In ℴQASM, typing is with respect to a *type environment* Ω and a predefined *size environment* Σ, which map ℴQASM variables to their basis and size (number of qubits), respectively. The typing judgment is written $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ which states that ι is well-typed under Ω and Σ, and transforms the variables' bases to be as in Ω' (Σ is unchanged). **[Liyi: good?]** Σ is fixed because the number of qubits in an execution is always fixed. It is generated in the high level language compiler, such as ℴQIMP in **??**. The algorithm generates Σ by taking an ℴQIMP program and scanning through all the variable initialization statements. Select type rules are given in Figure 29; the rules not shown (for ID, Rshift, Rev, $\texttt{RZ}^{-1}$, and $\texttt{SR}^{-1}$) are similar.

The type system enforces three invariants. First, it enforces that instructions are well-formed, meaning that gates are applied to valid qubit positions (the second premise in X) and that any control qubit is distinct from the target(s) (the fresh premise in CU). This latter property enforces

$$
\begin{array}{lll}
[\![\texttt{ID}\ p]\!]\varphi & = \varphi & \\
[\![\texttt{X}\ (x,i)]\!]\varphi & = \varphi[(x,i) \mapsto \uparrow \texttt{xg}(\downarrow \varphi(x,i))] & \text{where } \texttt{xg}(|0\rangle) = |1\rangle \quad \texttt{xg}(|1\rangle) = |0\rangle \\
[\![\texttt{CU}\ (x,i)\ \iota]\!]\varphi & = \texttt{cu}(\downarrow \varphi(x,i), \iota, \varphi) & \text{where } \texttt{cu}(|0\rangle, \iota, \varphi) = \varphi \quad \texttt{cu}(|1\rangle, \iota, \varphi) = [\![\iota]\!]\varphi \\
[\![\texttt{RZ}\ m\ (x,i)]\!]\varphi & = \varphi[(x,i) \mapsto \uparrow \texttt{rz}(m, \downarrow \varphi(x,i))] & \text{where } \texttt{rz}(m,|0\rangle) = |0\rangle \quad \texttt{rz}(m,|1\rangle) = \alpha(\tfrac{1}{2^m})\,|1\rangle \\
[\![\texttt{RZ}^{-1}\ m\ (x,i)]\!]\varphi & = \varphi[(x,i) \mapsto \uparrow \texttt{rrz}(m, \downarrow \varphi(x,i))] & \text{where } \texttt{rrz}(m,|0\rangle) = |0\rangle \quad \texttt{rrz}(m,|1\rangle) = \alpha(-\tfrac{1}{2^m})\,|1\rangle \\
[\![\texttt{SR}\ m\ x]\!]\varphi & = \varphi[\forall i \le m.\ (x,i) \mapsto \uparrow |\Phi(r_i + \tfrac{1}{2^{m-i+1}})\rangle] & \text{when } \downarrow \varphi(x,i) = |\Phi(r_i)\rangle \\
[\![\texttt{SR}^{-1}\ m\ x]\!]\varphi & = \varphi[\forall i \le m.\ (x,i) \mapsto \uparrow |\Phi(r_i - \tfrac{1}{2^{m-i+1}})\rangle] & \text{when } \downarrow \varphi(x,i) = |\Phi(r_i)\rangle \\
[\![\texttt{QFT}\ n\ x]\!]\varphi & = \varphi[x \mapsto \uparrow \texttt{qt}(\Sigma(x), \downarrow \varphi(x), n)] & \text{where } \texttt{qt}(i, |y\rangle, n) = \bigotimes_{k=0}^{i-1}(|\Phi(\tfrac{y}{2^{n-k}})\rangle) \\
[\![\texttt{QFT}^{-1}\ n\ x]\!]\varphi & = \varphi[x \mapsto \uparrow \texttt{qt}^{-1}(\Sigma(x), \downarrow \varphi(x), n)] & \\
[\![\texttt{Lshift}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_l(\varphi(x))] & \text{where } \texttt{pm}_l(q_0 \otimes q_1 \otimes \cdots \otimes q_{n-1}) = q_{n-1} \otimes q_0 \otimes q_1 \otimes \cdots \\
[\![\texttt{Rshift}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_r(\varphi(x))] & \text{where } \texttt{pm}_r(q_0 \otimes q_1 \otimes \cdots \otimes q_{n-1}) = q_1 \otimes \cdots \otimes q_{n-1} \otimes q_0 \\
[\![\texttt{Rev}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_a(\varphi(x))] & \text{where } \texttt{pm}_a(q_0 \otimes \cdots \otimes q_{n-1}) = q_{n-1} \otimes \cdots \otimes q_0 \\
[\![\iota_1; \iota_2]\!]\varphi & = [\![\iota_2]\!]([\![\iota_1]\!]\varphi) &
\end{array}
$$

$$
\begin{array}{l}
\downarrow \alpha(b)\overline{q} = \overline{q} \qquad \downarrow (q_1 \otimes \cdots \otimes q_n) = \downarrow q_1 \otimes \cdots \otimes \downarrow q_n \\
\varphi[(x,i) \mapsto \uparrow \overline{q}] = \varphi[(x,i) \mapsto \alpha(b)\overline{q}] \quad \text{where } \varphi(x,i) = \alpha(b)\overline{q_i} \\
\varphi[(x,i) \mapsto \uparrow \alpha(b_1)\overline{q}] = \varphi[(x,i) \mapsto \alpha(b_1 + b_2)\overline{q}] \quad \text{where } \varphi(x,i) = \alpha(b_2)\overline{q_i} \\
\varphi[x \mapsto q_x] = \varphi[\forall i < \Sigma(x).\ (x,i) \mapsto q_{(x,i)}] \\
\varphi[x \mapsto \uparrow q_x] = \varphi[\forall i < \Sigma(x).\ (x,i) \mapsto \uparrow q_{(x,i)}]
\end{array}
$$

Fig. 26. ©QASM semantics

the quantum *no-cloning rule*. For example, we can apply the CU in `rz_adder'` (Figure 20) because position `a,m` is distinct from variable `b`.

Second, the type system enforces that instructions leave affected qubits in a proper basis (thereby avoiding entanglement). The rules implement the state machine shown in Figure 25. For example, QFT n transforms a variable from Nor to Phi n (rule QFT), while $\texttt{QFT}^{-1}$ n transforms it from Phi n back to Nor (rule RQFT). Position shifting operations are disallowed on variables x in the Phi basis because the qubits that make up x are internally related (see Appendix C.7) and cannot be rearranged. Indeed, applying a Lshift and then a $\texttt{QFT}^{-1}$ on x in Phi would entangle x's qubits.

Third, the type system enforces that the effect of position shifting operations can be statically tracked. The `neutral` condition of CU requires that any shifting within ι is restored by the time it completes. For example, CU p (Lshift x) ; X $(x, 0)$ is not well-typed, because knowing the final physical position of qubit $(x, 0)$ would require statically knowing the value of p. On the other hand, the program CU c (Lshift x ; X $(x, 0)$; Rshift x) ; X $(x, 0)$ is well-typed because the effect of the Lshift is "undone" by an Rshift inside the body of the CU.

Semantics. We define the semantics of an ©QASM program as a partial function $[\![\,]\!]$ from an instruction ι and input state φ to an output state φ', written $[\![\iota]\!]\varphi = \varphi'$, shown in Figure 26.

Recall that a state φ is a tuple of d qubit values, modeling the tensor product $q_1 \otimes \cdots \otimes q_d$. The rules implicitly map each variable x to a range of qubits in the state, e.g., $\varphi(x)$ corresponds to some sub-state $q_k \otimes \cdots \otimes q_{k+n-1}$ where $\Sigma(x) = n$. Many of the rules in Figure 26 update a *portion* of a state. We write $\varphi[(x,i) \mapsto q_{(x,i)}]$ to update the i-th qubit of variable x to be the (single-qubit) state $q_{(x,i)}$, and $\varphi[x \mapsto q_x]$ to update variable x according to the qubit *tuple* q_x. $\varphi[(x,i) \mapsto \uparrow q_{(x,i)}]$ and $\varphi[x \mapsto \uparrow q_x]$ are similar, except that they also accumulate the previous global phase of $\varphi(x,i)$ (or $\varphi(x)$). We use $\downarrow$ to convert a qubit $\alpha(b)\overline{q}$ to an unphased qubit $\overline{q}$.

Function xg updates the state of a single qubit according to the rules for the standard quantum gate X. cu is a conditional operation depending on the Nor-basis qubit (x, i). **[Liyi: good?]** RZ (or $\mathtt{RZ}^{-1}$) is an z-axis phase rotation operation. Since it applies to Nor-basis, it applies a global phase. By Theorem A.4, when we compiles it to sqir, the global phase might be turned to a local one. For example, to prepare the state $\sum_{j=0}^{2^n}(-i)^x |x\rangle$ [4], we apply a series of Hadamard gates following by several controlled-RZ gates on x, where the controlled-RZ gates are definable by 𝔒qasm. SR (or $\mathtt{SR}^{-1}$) applies an $m+1$ series of RZ (or $\mathtt{RZ}^{-1}$) rotations where the i-th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$ (or $\alpha(-\frac{1}{2^{m-i+1}})$). qt applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes \cdots \otimes |b_i\rangle$ (assuming $\Sigma(y) = i$) and n is the degree of approximation. If $n = i$, then the operation is the standard QFT. Otherwise, each qubit in the state is mapped to $|\Phi(\frac{y}{2^{n-k}})\rangle$, which is equal to $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(\frac{y}{2^{n-k}})\,|1\rangle)$ when $k < n$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ when $n \leq k$ (since $\alpha(n) = 1$ for any natural number n). $\mathtt{qt}^{-1}$ is the inverse function of qt. Note that the input state to $\mathtt{qt}^{-1}$ is guaranteed to have the form $\bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{n-k}})\rangle)$ because it has type Phi n. $\mathtt{pm}_l$, $\mathtt{pm}_r$, and $\mathtt{pm}_a$ are the semantics for Lshift, Rshift, and Rev, respectively.

A.3 𝔒qasm Metatheory

Soundness. We prove that well-typed 𝔒qasm programs are well defined; i.e., the type system is sound with respect to the semantics. We begin by defining the well-formedness of an 𝔒qasm state.

Definition A.1 (Well-formed 𝔒qasm state). A state φ is *well-formed*, written $\Sigma; \Omega \vdash \varphi$, iff:

- For every $x \in \Omega$ such that $\Omega(x) = \mathtt{Nor}$, for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|b\rangle$.
- For every $x \in \Omega$ such that $\Omega(x) = \mathtt{Phi}\ n$ and $n \leq \Sigma(x)$, there exists a value v such that for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|\Phi(\frac{v}{2^{n-k}})\rangle$.[18]

Type soundness is stated as follows; the proof is by induction on ι, and is mechanized in Coq.

Theorem A.2. [𝔒qasm type soundness] If $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ and $\Sigma; \Omega \vdash \varphi$ then there exists φ' such that $[\![\iota]\!]\varphi = \varphi'$ and $\Sigma; \Omega' \vdash \varphi'$.

Algebra. Mathematically, the set of well-formed d-qubit 𝔒qasm states for a given Ω can be interpreted as a subset $\mathcal{S}^d$ of a 2^d-dimensional Hilbert space $\mathcal{H}^d$,[19] and the semantics function $[\![\,]\!]$ can be interpreted as a $2^d \times 2^d$ unitary matrix, as is standard when representing the semantics of programs without measurement [18]. Because 𝔒qasm's semantics can be viewed as a unitary matrix, correctness properties extend by linearity from $\mathcal{S}^d$ to $\mathcal{H}^d$—an oracle that performs addition for classical Nor inputs will also perform addition over a superposition of Nor inputs. We have proved that $\mathcal{S}^d$ is closed under well-typed 𝔒qasm programs.

[Liyi: good?] Given a qubit size map Σ and type environment Ω, the set of 𝔒qasm programs that are well-typed with respect to Σ and Ω (i.e., $\Sigma; \Omega \vdash \iota \triangleright \Omega'$) form a algebraic structure $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d)$, where $\{\iota\}$ defines the set of valid program syntax, such that there exists Ω', $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ for all ι in $\{\iota\}$; $\mathcal{S}^d$ is the set of d-qubit states on which programs $\iota \in \{\iota\}$ are run, and are well-formed $(\Sigma; \Omega \vdash \varphi)$ according to Appendix C.7. From the 𝔒qasm semantics and the type soundness theorem, for all $\iota \in \{\iota\}$ and $\varphi \in \mathcal{S}^d$, such that $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ and $\Sigma; \Omega \vdash \varphi$, we have $[\![\iota]\!]\varphi = \varphi'$, $\Sigma; \Omega' \vdash \varphi'$, and $\varphi' \in \mathcal{S}^d$. Thus, $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d)$, where $\{\iota\}$ defines a groupoid.

We can certainly extend the groupoid to another algebraic structure $(\{\iota'\}, \Sigma, \mathcal{H}^d)$, where $\mathcal{H}^d$ is a general 2^d dimensional Hilbert space $\mathcal{H}^d$ and $\{\iota'\}$ is a universal set of quantum gate operations.

[18] Note that $\Phi(x) = \Phi(x + n)$, where the integer n refers to phase $2\pi n$; so multiple choices of v are possible.

[19] A *Hilbert space* is a vector space with an inner product that is complete with respect to the norm defined by the inner product. $\mathcal{S}^d$ is a sub*set*, not a sub*space* of $\mathcal{H}^d$ because $\mathcal{S}^d$ is not closed under addition: Adding two well-formed states can produce a state that is not well-formed.

$$\texttt{X}\ (x,n) \xrightarrow{\text{inv}} \texttt{X}\ (x,n) \qquad \texttt{SR}\ m\ x \xrightarrow{\text{inv}} \texttt{SR}^{-1}\ m\ x \qquad \texttt{QFT}\ n\ x \xrightarrow{\text{inv}} \texttt{QFT}^{-1}\ n\ x \qquad \texttt{Lshift}\ x \xrightarrow{\text{inv}} \texttt{Rshift}\ x$$

$$\frac{\iota \xrightarrow{\text{inv}} \iota'}{\texttt{CU}\ (x,n)\ \iota \xrightarrow{\text{inv}} \texttt{CU}\ (x,n)\ \iota'} \qquad \frac{\iota_1 \xrightarrow{\text{inv}} \iota_1' \qquad \iota_2 \xrightarrow{\text{inv}} \iota_2'}{\iota_1\ ;\ \iota_2 \xrightarrow{\text{inv}} \iota_2'\ ;\ \iota_1'}$$

Fig. 27. Select ⟦QASM inversion rules

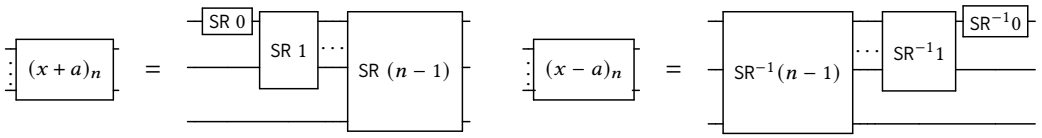


Fig. 28. Addition/subtraction circuits are inverses

Clearly, we have $\mathcal{S}^d \subseteq \mathcal{H}^d$ and $\{\iota\} \subseteq \{\iota'\}$, because sets $\mathcal{H}^d$ and $\{\iota'\}$ can be acquired by removing the well-formed $(\Sigma; \Omega \vdash \varphi)$ and well-typed $(\Sigma; \Omega \vdash \iota \triangleright \Omega')$ definitions for $\mathcal{S}^d$ and $\{\iota\}$, respectively. $(\{\iota'\}, \Sigma, \mathcal{H}^d)$ is a groupoid because every ⟦QASM operation is valid in a traditional quantum language like SQIR. We then have the following two two theorems to connect ⟦QASM operations with operations in the general Hilbert space:

Theorem A.3. $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d) \subseteq (\{\iota\}, \Sigma, \mathcal{H}^d)$ is a subgroupoid.

Theorem A.4. Let $|y\rangle$ be an abbreviation of $\bigotimes_{m=0}^{d-1} \alpha(r_m)\,|b_m\rangle$ for $b_m \in \{0,1\}$. If for every $i \in [0, 2^d)$, $[\![\iota]\!]\,|y_i\rangle = |y_i'\rangle$, then $[\![\iota]\!](\sum_{i=0}^{2^d-1} |y_i\rangle) = \sum_{i=0}^{2^d-1} |y_i'\rangle$.

We prove these theorems as corollaries of the compilation correctness theorem from ⟦QASM to SQIR (??). Theorem A.3 suggests that the space $\mathcal{S}^d$ is closed under the application of any well-typed ⟦QASM operation. Theorem A.4 says that ⟦QASM oracles can be safely applied to superpositions over classical states.[20]

⟦QASM programs are easily invertible, as shown by the rules in Figure 27. This inversion operation is useful for constructing quantum oracles; for example, the core logic in the QFT-based subtraction circuit is just the inverse of the core logic in the addition circuit (Figure 27). This allows us to reuse the proof of addition in the proof of subtraction. The inversion function satisfies the following properties:

Theorem A.5. [Type reversibility] For any well-typed program ι, such that $\Sigma; \Omega \vdash \iota \triangleright \Omega'$, its inverse ι', where $\iota \xrightarrow{\text{inv}} \iota'$, is also well-typed and we have $\Sigma; \Omega' \vdash \iota' \triangleright \Omega$. Moreover, $[\![\iota; \iota']\!]\varphi = \varphi$.

B THE FULL DEFINITIONS OF QAFNY

B.1 QAFNY Session Generation

A type is written as $\bigotimes_n\ t$, where n refers to the total number of qubits in a session, and t describes the qubit state form. A session being type $\bigotimes_n\ \texttt{Nor}\ \overline{d}$ means that every qubit is in normal basis (either $|0\rangle$ or $|1\rangle$), and $\overline{d}$ describes basis states for the qubits. The type corresponds to a single qubit basis state $\alpha(n)\,|\overline{d}\rangle$, where the global phase $\alpha(n)$ has the form $e^{2\pi i \frac{1}{n}}$ and $\overline{d}$ is a list of bit values. Global phases for Nor type are usually ignored in many semantic definitions. In QWhile, we record it because in quantum conditionals, such global phases might be turned to local phases.

[20] Note that a superposition over classical states can describe *any* quantum state, including entangled states.

$$\frac{}{\Omega \vdash x : \Omega(x)} \qquad \frac{\Omega(x) = (x, 0, \Sigma(x))}{\Omega \vdash x[n] : [(x, n, n+1)]} \qquad \frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2}{\Omega \vdash a_1 + a_2 : q_1 \sqcup q_2} \qquad \frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2}{\Omega \vdash a_1 * a_2 : q_1 \sqcup q_2}$$

$$\frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2 \Omega \vdash a_3 : q_3}{\Omega \vdash (a_1 = a_2)@x[n] : q_1 \sqcup q_2 \sqcup q_3} \qquad \frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2 \Omega \vdash a_3 : q_3}{\Omega \vdash (a_1 < a_2)@x[n] : q_1 \sqcup q_2 \sqcup q_3} \qquad \frac{\Omega \vdash b : q}{\Omega \vdash \neg b : q} \qquad \frac{\Omega \vdash e : \zeta_2 \sqcup \zeta_1}{\Omega \vdash e : \zeta_1 \sqcup \zeta_2}$$

$$\zeta_1 \sqcup \zeta_2 = \zeta_1 \uplus \zeta_2 \zeta \sqcup g = \zeta \qquad g \sqcup \zeta = \zeta \qquad \mathtt{C} \sqcup \mathtt{C} = \mathtt{C} \qquad \mathtt{Q} \sqcup \mathtt{C} = \mathtt{Q} \qquad \mathtt{C} \sqcup \mathtt{Q} = \mathtt{Q} \qquad \mathtt{C} \leq \mathtt{Q} \leq \zeta$$
$$\bot \uplus l = l \qquad l \uplus \bot = l \qquad [(x, v_1, v_2)] \uplus [(y, v_3, v_4)] = [(x, v_1, v_2), (y, v_3, v_4)]$$
$$[v_2, v_2] \cap [v_3, v_4] \neq \emptyset \Rightarrow [(x, v_1, v_2)] \uplus [(x, v_3, v_4)] = [(x, \mathtt{min}(v_1, v_3), \mathtt{max}(v_2, v_4))]$$

Fig. 29. Arith, Bool, Gate Mode Checking

$\bigotimes_n$ Had w means that every qubit in the session has the state: $(\alpha_1 |0\rangle + \alpha_2 |1\rangle)$; the qubits are in superposition but they are not entangled. ◯ represents the state is a uniform superposition, while ∞ means the phase amplitude for each qubit is unknown. If a session has such type, it then has the value form $\bigotimes_{k=0}^{m} |\Phi(n_k)\rangle$, where $|\Phi(n_k)\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \alpha(n_k)|1\rangle)$.

All qubits in a session that has type $\bigotimes_n$ CH $m\beta$ are supposedly entangled (eventual entanglement below). m refers to the number of possible different entangled states in the session, and the bitstring indexed set β describes each of these states, while every element in β is indexed by $i \in [0, m)$. β can also be ∞ meaning that the entanglement structure is unknown. For example, in quantum phase estimation, after applying the $\mathtt{QFT}^{-1}$ operation, the state has type $\bigotimes_n$ CH $m\infty$. In such case, the only quantum operation to apply is a measurement. If a session has type $\bigotimes_n$ CH $m\beta$ and the entanglement is a uniform superposition, we can describe its state as $\sum_{i=0}^{m} \frac{1}{\sqrt{m}} \beta(i)$, and the length of bitstring $\beta(i)$ is n. For example, in a n-length GHZ application, the final state is: $|0\rangle^{\otimes n} + |1\rangle^{\otimes n}$. Thus, its type is $\bigotimes_n$ CH $2\{\overline{0}^n, \overline{1}^n\}$, where $\overline{d}^n$ is a n-bit string having bit d.

The type $\bigotimes_n$ CH $m\beta$ corresponds to the value form $\sum_{k=0}^{m} \theta_k |\overline{d_k}\rangle$. θ_k is an amplitude real number, and $\overline{d_k}$ is the basis. Basically, $\sum_{k=0}^{m} \theta_k |\overline{d_k}\rangle$ represents a size m array of basis states that are pairs of θ_k and $\overline{d_k}$. For a session ζ of type CH, one can use $\zeta[i]$ to access the i-th basis state in the above summation, and the length is m. In the Q-Dafny implementation section, we show how we can represent θ_k for effective automatic theorem proving.

The QWhile type system has the type judgment: $\Omega, \mathcal{T} \vdash_g s : \zeta \triangleright \tau$, where g is the context mode, mode environment Ω maps variables to modes or sessions (q in Figure 5), type environment $\mathcal{T}$ maps a session to its type, s is the statement being typed, ζ is the session of s, and τ is ζ's type. The QWhile type system in Figure 35 has several tasks. First, it enforces context mode restrictions. Context mode g is either C or Q . Q represents the current expression lives inside a quantum conditional or loop, while C refers to other cases. In a Q context, one cannot perform M-mode operations, i.e., no measurement is allowed. There are other well-formedness enforcement. For example, the session of the Boolean guard b in a conditional/loop is disjoint with the session in the conditional/loop body, i.e., qubits used in a Boolean guard cannot appear in its conditional/loop body.

Second, the type system enforces mode checking for variables and expressions in Figure 29. In QWhile, C-mode variables are evaluated to values during type checking. In a `let` statement (Figure 35), C-mode expression is evaluated to a value n, and the variable x is replaced by n in s. The expression mode checking (Figure 29) has the judgment: $\Omega \vdash (a \mid b) : q$. It takes a mode environment Ω, and an expression (a, b), and judges if the expression has the mode g if it contains only classical values, or a quantum session ζ if it contains some quantum values. All the supposedly C-mode locations in an expression are assumed to be evaluated to values in the type checking step,

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9
$x[i]$	Nor	Had	Had	Had	Had	Had	Had	CH	CH
y	any	Nor	Nor	Had	Had	CH	CH	CH	CH
y's operation type	any	$\mathcal{X}$	$\mathcal{R}$	$\mathcal{X}$	$\mathcal{R}$	$\mathcal{X}$	$\mathcal{R}$	$\mathcal{X}$	$\mathcal{R}$
Output Type Entangled?	N	Y	N	N	Y	Y	Y	Y	Y

Fig. 30. Control Gate Entanglement Situation

$$\otimes_n \ \mathsf{Nor}\ \overline{d} \sqsubseteq \otimes_n \ \mathsf{CH}\ 1\{\overline{d}\} \qquad \otimes_n \ \mathsf{CH}\ 2^n\beta \sqsubseteq \otimes_n \ \mathsf{CH}\ 2^n\infty \qquad \otimes_n \ \mathsf{Had} \bigcirc \sqsubseteq \otimes_n \ \mathsf{CH}\ 2^n\mathcal{P}(n)$$

Fig. 31. Session Type Subtyping

such as the index value $x[n]$ in difference expressions in Figure 29. It is worth noting that the session computation ($\uplus$) is also commutative as the last rule in Figure 29.

Third, by generating the session of an expression, the QWhile type system assigns a type τ for the session indicating its state format, which will be discussed shortly below. Recall that a session is a list of quantum qubit fragments. In quantum computation, qubits can entangled with each other. We utilize type τ (Figure 21) to state entanglement properties appearing in a group of qubits. It is worth noting that the entanglement property refers to *eventual entanglement*, .i.e. a group of qubits that are eventually entangled. Entanglement classification is tough and might not be necessary. In most near term quantum algorithms, such as Shor's algorithm [48] and Childs' Boolean equation algorithm (BEA) [4], programmers care about if qubits eventually become entangled during a quantum loop execution. This is why the normal basis type ($\otimes_n$ Nor $\overline{d}$) can also be a subtype of a entanglement type ($\otimes_n$ CH $1\{\overline{d}\}$) in our system (Figure 31).

Entanglement Types. We first investigate the relationship between the types and entanglement states. It is well-known that every single quantum gate application does not create entanglement (X, H, and RZ). It is enough to classify entanglement effects through a control gate application, i.e., if $(x[i])$ $e(y)$, where the control node is $x[i]$ and e is an operation applying on y.

A qubit can be described as $\alpha_1 |b_1\rangle + \alpha_2 |b_2\rangle$, where α_1/α_2 are phase amplitudes, and b_1/b_2 are bases. For simplicity, we assume that when we applying a quantum operation on a qubit array y, we either solely change the qubit amplitudes or bases. We identify the former one as $\mathcal{R}$ kind, referring to its similarity of applying an RZ gate; and the latter as $\mathcal{X}$ kind, referring to its similarity of applying an X gate. The entanglement situation between $x[i]$ and y after applying a control statement if $(x[i])$ $e(y)$ is described in Figure 30.

If $x[i]$ has input type Nor, the control operation acts as a classical conditional, i.e., no entanglement is possible. In most quantum algorithms, $x[i]$ will be in superposition (type Had) to enable entanglement creation. When y has type Nor, if y's operation is of $\mathcal{X}$ kind, an entanglement between $x[i]$ and y is created, such as the GHZ algorithm; if the operation is of $\mathcal{R}$ kind, there is not entanglement after the control application, such as the Quantum Phase Estimation (QPE) algorithm.

When $x[i]$ and y are both of type Had, if we apply an $\mathcal{X}$ kind operation on y, it does not create entanglement. An example application is the phase kickback pattern. If we apply a $\mathcal{R}$ operation on y, this does create entanglement. This kind of operations appears in state preparations, such as preparing a register x to have state $\sum_{t=0}^{N} i^{-t} |t\rangle$ in Childs' Boolean equation algorithm [4]. The main goal for preparing such state is not to entanglement qubits, but to prepare a state with phases related to its bases.

The case when $x[i]$ and y has type Had and CH, respectively, happens in the middle of executing a quantum loop, such as in the Shor's algorithm and BEA. Applying both $\mathcal{X}$ and $\mathcal{R}$ kind operations

$$\begin{array}{llll} \mathsf{Nor}\ \infty & \sqsubseteq_n & \mathsf{CH}\ \infty \\ \mathsf{Nor}\ c & \sqsubseteq_n & \mathsf{CH}\ \{c\} \\ \mathsf{CH}\ \overline{c}(1) & \sqsubseteq_n & \mathsf{Nor}\ \overline{c}[0] \\ \mathsf{Had}\ p & \sqsubseteq_n & \mathsf{CH}\ \{(\!|j|\!)|j \in [0,2^n)\}(2^n) \\ \mathsf{CH}\ \{0,1\} & \sqsubseteq_1 & \mathsf{Had}\ \infty \\ \mathsf{CH}\ p & \sqsubseteq_n & \mathsf{CH}\ \infty \end{array}$$

(a) Subtyping

$$\begin{array}{lll} |c\rangle & \equiv_n & \Sigma^1_{j=0}\,|c\rangle \\ \Sigma^1_{j=0}\,z_j\,|c_j\rangle & \equiv_n & |c_0\rangle \\ \frac{1}{\sqrt{2^n}}\bigotimes^n_{j=0}(|0\rangle+\alpha(r_j)\,|1\rangle) & \equiv_n & \Sigma^{2^n}_{j=0}\,\frac{\alpha(\sum^n_{k=0} r_k\cdot(\!|j|\!)[k])}{\sqrt{2^n}}\,|j\rangle \\ \Sigma^2_{j=0}\,z_j\,|c_j\rangle & \equiv_1 & \frac{1}{\sqrt{2}}\bigotimes^1_{j=0}(|0\rangle+\frac{\sqrt{2}z_1}{z_0}\,|1\rangle) \\ & & \text{when}\ \ c_0=0\ \ c_1=1 \end{array}$$

(b) State Equivalence

Fig. 32. Qafny type/state relations. $\overline{c}[n]$ produces the n-th element in set $\overline{c}$. $\{(\!|j|\!)|j \in [0,2^n)\}(2^n)$ defines a set $\{(\!|j|\!)|j \in [0,2^n)\}$ with the emphasis that it has 2^n elements. $\{0,1\}$ is a set of two single element bitstrings 0 and 1. $\cdot$ is the multiplication operation, $(\!|j|\!)$ turns a number j to a bitstring, $(\!|j|\!)[k]$ takes the k-th element in the bitstring $(\!|j|\!)$, and $|j\rangle$ is an abbreviation of $|(\!|j|\!)\rangle$.

result in entanglement. In this narrative, algorithm designers intend to merge an additional qubit $x[i]$ into an existing entanglement session y. $x[i]$ is commonly in uniform superposition, but there can be some additional local phases attached with some bases, which we named this situation as saturation, i.e., In an entanglement session written as $\sum^n_{i=0}|x_l,y,x_r\rangle$, for any fixing x_l and x_r bases, if y covers all possible bases, we then say that the part y in the entanglement is in saturation. This concept is important for generating auto-proof, which will be discussed in Appendix C.3.

When $x[i]$ and y are both of type CH, there are two situations. When the two parties belong to the same entanglement session, it is possible that an $\mathcal{X}$ or $\mathcal{R}$ operation de-entangles the session. Since QWhile tracks eventual entanglement. In many cases, HAD type can be viewed as a kind of entanglement. In addition, the QWhile type system make sure that most de-entanglements happen at the end of the algorithm by turning the qubit type to CH $m\infty$, so that after the possible de-entanglement, the only possible application is a measurement.

If $x[i]$ and y are in different entanglement sessions, the situation is similar to when $x[i]$ having Had and y having CH type. It merges the two sessions together through the saturation $x[i]$. For example, in BEA, The quantum Boolean guard computes the following operation $(z < i)@x[i]$ on a Had type variable z (state: $\sum^{2^n}_{k=0}|k\rangle$) and a Nor type factor $x[i]$ (state: $|0\rangle$). The result is an entanglement $\sum^{2^n}_{k=0}|k,k<i\rangle$, where the $x[i]$ position stores the Boolean bit result $k < i$. [21] The algorithm further merges the $|z,x[i]\rangle$ session with a loop body entanglement session y. In this cases, both $|z,x[i]\rangle$ and y are of CH type.

C A COMPLICATED TYPE SYSTEM

The Qafny element component syntax is represented according to the grammar in Figure 4. In Qafny , there are three kinds of values, two of which are classical ones represented by the two modes: C and M. The former represents classical values, represented as a natural number n, that do not intervene with quantum measurements and are evaluated in the compilation time, the latter represents values, represented as a pair (r,n), produced from a quantum measurement. The real number r is a characteristic representing the theoretical probability of the measurement resulting in the value n. Any classical arithmetic operation does not change r, i.e., $(r,n)+m=(r,n+m)$.

Quantum variables are defined as kind Q n, where n is the number of qubits in a variable representing as a qubit array. Quantum values are more often to be described as sessions (λ) that can be viewed as clusters of possibly entangled qubits, where the number of qubits is exactly the session length, i.e., $|\overline{x[n..m]}|$. Each session consists of different disjoint ranges, connected by the

[21] When $k < i$, $x[i] = 1$ while $\neg(k < i)$, $x[i] = 0$.

$\uplus$ operation (meaning that different ranges are disjoint), represented as $x[n..m]$ that refers the number range $[n, m)$ in a quantum array named x. For simplicity, we assume that different variable names referring to different quantum arrays without aliasing. Sessions have associated equational properties. They are associative and identitive with the identity operation as $\perp$. There are another two equational properties for sessions below:

$$n \leq j < m \Rightarrow x[n, m] \uplus \lambda \equiv_\lambda x[n, j] \uplus x[j, m] \uplus \lambda \qquad x[n, n] \equiv_\lambda \perp$$

Each length-n session is associated to a quantum state that can be one of the three forms (q in Figure 4) that are corresponding to three different types (τ in Figure 4). The first kind of state is of Nor type (Nor (c opt)), having the state form $|c\rangle$, which is a computational basis value. c is of length n and represents a tensor product of qubits, all being 0 or 1. The second kind of state is of Had type (Had ($\bigcirc$ opt)), meaning that qubits in such session are in superposition but not entangled. The state form is $\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$, where $\alpha(r_j)$ is a local phase for the j-th qubit in the session. If $r_j = 0$ for all j, the state can be represented by type Had $\bigcirc$ representing a uniformly distributed superposition; otherwise, we represent the type as Had ∞. The third kind of state is of CH type (CH ($\overline{c}(m)$ opt)), having the state form $\sum_{j=0}^{m} z_j\,|c_j\rangle$, referring to that qubits in such session are possibly entangled. The state $\sum_{j=0}^{m} z_j\,|c_j\rangle$ can be viewed as an m element set of pairs $z_j\,|c_j\rangle$, where z_j and c_j are the j-th amplitude and basis. The well-formed restrictions for the state are three: 1) $\sum_{j=0}^{m} |z_j|^2 = 1$ (z_j is a complex number); 2) length of c_j is n for all j and $m \leq 2^n$; 3) any two bases c_j and c_k are distinct if $j \neq k$.

In Qafny , the quantum types and states are associated through bases and equational properties. For each quantum state q, especially for Nor type state $|c\rangle$ and CH type state $\sum_{j=0}^{m} z_j\,|c_j\rangle$, the type factors are either ∞ meaning no bases can be tracked, or having the form c and $\overline{c}(m)$ that track the bases of the state $|c\rangle$ and $\sum_{j=0}^{m} z_j\,|c_j\rangle$, respectively. For Nor type, this means that the type factor c (in Nor c) and the state qubit format $|c\rangle$ must be equal; for CH type (CH $\overline{c}(m)$), if the state is $\sum_{j=0}^{m} z_j\,|c_j\rangle$, the j-th element $\overline{c}[j]$ is equal to c_j. Additionally, Qafny types permit subtyping relations that correspond to state equivalent relations in Figure 39. Both subtype relation $\sqsubseteq_n$ and state equivalence relation $\equiv_n$ are parameterized by a session length number n, such that they establish relations between two quantum states describing a session of length n. $\sqsubseteq_n$ in Figure 39a describes a type term on the left can be used as a type on the right. For example, a Nor type qubit array Nor c can be used as a single element entanglement type term CH $\{c\}$ [22]. Correspondingly, state equivalence relation $\equiv_n$ describes the two state forms to be equivalent; specifically, the left state term can be used as the right one, e.g., a single element entanglement state $\sum_{j=0}^{1} z_j\,|c_j\rangle$ can be used as a Nor type state $|c_0\rangle$ with the fact that z_0 is now a global phase that can be neglected.

C.1 Type Checking: A Quantum Session Type System

In Qafny , typing is with respect to a *kind environment* Ω and a *finite type environment* σ, which map Qafny variables to kinds and map sessions to types, respectively. The typing judgment is written as $\Omega; \sigma \vdash_g s \triangleright \sigma'$, which states that statements s is well-typed under the context mode g and environments Ω and σ, the sessions representing s is exactly the domain of σ' as $\mathtt{dom}(\sigma')$, and s transforms types for the sessions in σ to types in σ'. Ω describes the kinds for all program variables. Ω is populated through let expressions that introduce variables, and the Qafny type system enforces variable scope; such enforcement is neglected in Figure 35 for simplicity. We also assume that variables introduced in let expressions are all distinct through proper alpha conversions. σ and σ' describe types for sessions referring to possibly entangled quantum clusters pointed to by quantum variables in s. σ and σ' are both finite and the domain of them contain sessions that do

[22] If a qubit array only consists of 0 and 1, it can be viewed as an entanglement of unique possibility.

©QASM Expr	μ		
Parameter	l	::=	$x \mid x[a]$
Arith Expr	a	::=	$x \mid v \mid a + a \mid a * a \mid ...$
Bool Expr	b	::=	$x[a] \mid (a = a)@x[a] \mid (a < a)@x[a] \mid ...$
Predicate	P	::=	$a = a \mid a < a \mid \lambda \mapsto q \mid P \wedge P \mid P * P \mid ...$
Gate Expr	op	::=	$\mathtt{H} \mid \mathtt{QFT}^{[-1]}$
C/M Moded Expr	e	::=	$a \mid \mathtt{init}\ a \mid \mathtt{measure}(y) \mid \mathtt{ret}(y, (r, n))$
Statement	s	::=	$\{\} \mid \mathtt{let}\ x\ =\ e\ \mathtt{in}\ s \mid l \leftarrow op \mid \lambda \leftarrow \mu \mid l \leftarrow \mathtt{dis}$
		|	$s\ ;\ s \mid \mathtt{if}\ (b)\ s \mid \mathtt{for}\ (\mathtt{int}\ j \in [a_1, a_2)\ \&\&\ b)\ s$

Fig. 33. Core QAFNY syntax. ©QASM is in Section 3. For an operator OP, $\mathtt{OP}^{[-1]}$ indicates that the operator has a built-in inverse available. Arithmetic expressions in e are only used for classical operations, while Boolean expressions are used for both classical and quantum operations. $x[a]$ represents the a-th element in the qubit array x, while a quantum variable x represents the qubit group $x[0..n]$ and n is the length of x.

$$\begin{array}{rrcl} & \{\bot : \tau\} \cup \sigma & \preceq & \sigma \\ \tau \sqsubseteq_{|\lambda|} \tau' \Rightarrow & \{\lambda : \tau\} \cup \sigma & \preceq & \{\lambda : \tau'\} \cup \sigma \\ & \{\lambda_1 \uplus l_1 \uplus l_2 \uplus \lambda_2 : \tau\} \cup \sigma & \preceq & \{\lambda_1 \uplus l_2 \uplus l_1 \uplus \lambda_2 : \mathtt{mut}(\tau, |\lambda_1|)\} \cup \sigma \\ & \{\lambda_1 : \tau_1\} \cup \{\lambda_2 : \tau_2\} \cup \sigma & \preceq & \{\lambda_1 \uplus \lambda_2 : \mathtt{mer}(\tau_1, \tau_2)\} \cup \sigma \\ \mathtt{spt}(\tau, |\lambda_1|) = (\tau_1, \tau_2) \Rightarrow & \{\lambda_1 \uplus \lambda_2 : \tau\} \cup \sigma & \preceq & \{\lambda_1 : \tau_1\} \cup \{\lambda_2 : \tau_2\} \cup \sigma \end{array}$$

$\mathtt{pmut}((c_1.i_1.i_2.c_2), n) = (c_1.i_2.i_1.c_2)$ when $|c_1| = n$

$\mathtt{mut}(\mathtt{Nor}\ c, n) = \mathtt{Nor}\ \mathtt{pmut}(c, n)$ $\quad \mathtt{mut}(\mathtt{CH}\ \overline{c}(m), n) = \mathtt{CH}\ \{\mathtt{pmut}(c, n) | c \in \overline{c}(m)\}(m)$ $\quad \mathtt{mut}(\tau, n) = \tau$ [owise]

$\mathtt{mer}(\mathtt{Nor}\ c_1, \mathtt{Nor}\ c_2) = \mathtt{Nor}\ (c_1.c_2)$ $\quad \mathtt{mer}(\mathtt{Had}\ \bigcirc, \mathtt{Had}\ \bigcirc) = \mathtt{Had}\ \bigcirc$ $\quad \mathtt{mer}(T\ \infty, T\ t) = T\ \infty$

$\mathtt{mer}(\mathtt{CH}\ \overline{c_1}(m_1), \mathtt{CH}\ \overline{c_2}(m_2)) = \mathtt{CH}\ (\overline{c_1} \times \overline{c_2})(m_1 * m_2)$

$\mathtt{spt}(\mathtt{Nor}\ c_1.c_2, n) = (\mathtt{Nor}\ c_1, \mathtt{Nor}\ c_2)$ when $|c_1| = n$ $\quad \mathtt{spt}(\mathtt{Had}\ t, n) = (\mathtt{Had}\ t, \mathtt{Had}\ t)$

$\mathtt{spt}(\mathtt{CH}\ \{c_j.c | j \in [0, m) \wedge |c_j| = n\}(m), n) = (\mathtt{CH}\ \{c_j | j \in [0, m) \wedge |c_j| = n\}(m), \mathtt{Nor}\ c)$

Fig. 34. Type environment partial order. We use set union ($\cup$) to describe the type environment concatenation with the empty set operation $\emptyset$. i is a single bit either 0 or 1. The . operation is bitstring concatenation. $\times$ is the Cartesian product of two sets. T is either Nor, Had or CH.

not overlap with each other; $\mathtt{dom}(\sigma)$ is large enough to describe all sessions pointed to by quantum variables in s, while $\mathtt{dom}(\sigma')$ should be the exact sessions containing quantum variables in s. We have partial order relations defined for type environments in Figure 39d, which will be explained shortly. Selected type rules are given in Figure 35; the rules not mentioned are similar and listed in Appendix B.

The type system enforces five invariants. First, well-formed and context restrictions for quantum programs. Well-formedness means that qubits mentioned in the Boolean guard of a quantum conditional cannot be accessed in the conditional body, while context restriction refers to the fact that the quantum conditional body cannot create (`init`) and measure (`measure`) qubits. For example the FV checks in rule TIF enforces that the session for the Boolean and the conditional body does not overlap. Coincidentally, we utilize the modes (g, either C or M) as context modes for the type system. Context mode C permits most QAFNY operations. Once a type rule turns a mode to M, such as in the conditional body in rule TIF, we disallow `init` and `measure` operations. For example, rules TMEA and TMEA-N are valid only if the input context mode is C.

Second, the type system tracks the basis state of every qubit in sessions. In rule TA-CH, we find that the oracle μ is applied on λ belonging to a session $\lambda \uplus \lambda'$. Correspondingly, the session's type is $\mathtt{CH}\ \overline{c}(m)$, for each bitstring $c_1.c_2 \in \overline{c}$, with $|c_1| = |\lambda|$, we apply μ on the c_1 and leave c_2

$$\textsc{TPar}\ \frac{\sigma \preceq \sigma' \qquad \Omega, \sigma' \vdash_g s \triangleright \sigma''}{\Omega, \sigma \vdash_g s \triangleright \sigma''} \qquad \textsc{TExp}\ \frac{\Omega[x \mapsto \mathtt{C}], \sigma \vdash_g s[n/x] \triangleright \sigma'}{\Omega, \sigma \vdash_g \mathtt{let}\ x\ =\ n\ \mathtt{in}\ s \triangleright \sigma'}$$

$$\textsc{TMea}\ \frac{\Omega(y) = \mathtt{Q}\ j \qquad \sigma(y) = \{\, y[0..j] \uplus \lambda \mapsto \tau\}}{\Omega, \sigma \vdash_{\mathtt{C}} \mathtt{let}\ x\ =\ \mathtt{measure}(y)\ \mathtt{in}\ s \triangleright \sigma'}$$

$$\textsc{TA-CH}\ \frac{FV(\mu) = \lambda \qquad \sigma(\lambda \uplus \lambda') = \mathtt{CH}\ \overline{c}(m) \qquad \overline{c'} = \{(\llbracket \mu \rrbracket c_1).c_2 \mid c_1.c_2 \in \overline{c} \wedge |c_1| = |\lambda|\}}{\Omega, \sigma \vdash_g \lambda \leftarrow \mu \triangleright \{\lambda \uplus \lambda' : \mathtt{CH}\ \overline{c'}(m)\}}$$

$$\textsc{TMea-N}\ \frac{\Omega(y) = \mathtt{Q}\ j \qquad \overline{c'} = \{c_2 \mid (\lfloor n \rfloor).c_2 \in \overline{c} \wedge |(\lfloor n \rfloor)| = j\} \qquad \Omega[x \mapsto \mathtt{M}], \sigma[\lambda \mapsto \mathtt{CH}\ \overline{c'}(|\overline{c'}|)] \vdash_{\mathtt{C}} s \triangleright \sigma'}{\Omega, \sigma[\, y[0..j] \uplus \lambda \mapsto \mathtt{CH}\ \overline{c}(m)] \vdash_{\mathtt{C}} \mathtt{let}\ x\ =\ \mathtt{ret}(y, (r, n))\ \mathtt{in}\ s \triangleright \sigma'}$$

$$\textsc{TSeq}\ \frac{\Omega, \sigma \vdash_g s_1 \triangleright \sigma_1 \qquad \Omega, \sigma[\uparrow \sigma_1] \vdash_g s_2 \triangleright \sigma_2}{\Omega, \sigma \vdash_g s_1\ ;\ s_2 \triangleright\ \sigma_2 \cup \sigma_1|_{\notin \mathtt{dom}(\sigma_2)}} \qquad \textsc{TLoop}\ \frac{\forall j \in [n_1, n_2)\ .\ \Omega, \sigma[\uparrow \sigma'[j/x]] \vdash_g \mathtt{if}\ (b)\ s \triangleright \sigma'[\mathtt{S}\ j/x]}{\Omega, \sigma \vdash_g \mathtt{for}\ (\mathtt{int}\ \mathtt{int}\ x := n_1 \in [x < n_2, b)\ \&\&\ {+}{+}x)\ s \triangleright \sigma'[n_2/x]}$$

$$\textsc{TIf}\ \frac{FV(b@x[j]) = \lambda \uplus x[j, \mathtt{S}\ j] \qquad FV(b@x[j]) \cap FV(s) = \emptyset \qquad \sigma(\lambda \uplus x[j, \mathtt{S}\ j] \uplus \lambda_1) = \mathtt{CH}\ \overline{c}(m) \qquad \Omega, \sigma \vdash_{\mathtt{M}} s \triangleright \{\lambda \uplus x[j, \mathtt{S}\ j] \uplus \lambda_1 : \mathtt{CH}\ \overline{c'}(m)\}}{\Omega, \sigma \vdash_g \mathtt{if}\ (b@x[j])\ s \triangleright \{\lambda \uplus x[j, \mathtt{S}\ j] \uplus \lambda_1 : \mathtt{CH}\ \overline{c''}(m)\}}$$

$$\textsc{SLoop-N}\quad (\varphi, \mathtt{for}\ (\mathtt{int}\ j \in [n_1, n_2)\ \&\&\ b)\ s) \longrightarrow (\varphi, \{\})$$

$$\overline{c''} = \{(\lfloor n \rfloor).1.c_2 \mid (\lfloor n \rfloor).d.c_1 \in \overline{c} \wedge (\lfloor n \rfloor).d.c_2 \in \overline{c'} \wedge b[(\lfloor n \rfloor)/\lambda] \oplus d \wedge |(\lfloor n \rfloor)| = |\lambda|\}$$
$$\cup \{(\lfloor n \rfloor).0.c_1 \mid (\lfloor n \rfloor).d.c_1 \in \overline{c} \wedge \neg(b[(\lfloor n \rfloor)/\lambda] \oplus d) \wedge |(\lfloor n \rfloor)| = |\lambda|\}$$
$$\sigma[\uparrow \sigma'] = \sigma[\forall \lambda : \tau \in \sigma'\ .\ \tau/\lambda]$$
$$\sigma|_{\notin \mathtt{dom}(\sigma')} = \{\lambda : \tau | \lambda \notin \mathtt{dom}(\sigma')\}$$

Fig. 35. QAFNY type system. $\llbracket \mu \rrbracket c$ is the ⟦QASM⟧ semantics of interpreting reversible expression μ in Figure 26. Boolean expression b can be $a_1 = a_2, a_1 < a_2$ or true. $b[(\lfloor n \rfloor)/\lambda]$ means that we treat b as a ⟦QASM⟧ μ expression, replace qubits in array λ with bits in bitstring $(\lfloor n \rfloor)$, and evaluate it to a Boolean value. $\sigma(y) = \{\lambda \mapsto \tau\}$ produces the map entry $\lambda \mapsto \tau$ and the range $y[0..|y|]$ is in λ. $\sigma(\lambda) = \tau$ is an abbreviation of $\sigma(\lambda) = \{\lambda \mapsto \tau\}$. $FV(-)$ produces a session by union all qubits appearing in $-$.

unchanged. Here, we utilize the ⟦QASM⟧ semantics that describes transitions from a Nor state to another Nor one, and we generalize it to apply the semantic function on every element in the CH type. During the transition, the number of elements m does not change. Similarly, applying a partial measurement on range $y[0..j]$ of the session $y[0..j] \uplus \lambda$ in rule TMEA-N can be viewed as a array filter, i.e., for an element $c_1.c_2$ in set $\overline{c}$ of the type $\mathtt{CH}\ \overline{c}(m)$, with $|c_1| = j$, we keep only the ones with $c_1 = (\lfloor n \rfloor)$ (n is the measurement result) in the new set $\overline{c'}$ and recompute $|\overline{c'}|$. In QAFNY , the tracking procedure is to generate symbolic predicates that permit the production of the set $\overline{c'}(|c'|)$, not to actually produce such set. If the predicates are not not effectively trackable, we can always use ∞ to represent the set.

[Liyi: may be we can add a rule about turning NOR to HAD so that we can say that the subtyping casting is also useful.] Third, the type system enforces equational properties of quantum qubit sessions through a partial order relation over type environments, including subtyping, qubit position mutation, merge and split quantum sessions. Essentially, we can view two qubit arrays be equivalent if there is a bijective permutation on the qubit positions of the two. To analyze a quantum application on a qubit array, if the array is arranged in a certain way, the semantic definition will be a lot more trivial than other arrangements. For example, in applying a quantum oracle to a session (rule TMEA), we fix the qubits that permits the μ operation to always

live in the front part (λ in $\lambda \uplus \lambda'$). This is achieved by a consecutive application of the mutation rule (mut) in the partial order ($\preceq$) in Figure 39d, which casts the left type environment to the format on the right through rule TPar. Similarly, split (spt) and combination (mer) of sessions in Figure 39d are useful to describe some quantum operation behaviors. the split of a quantum session into two represents the process of disentanglement of quantum qubits. For example, $|00\rangle + |10\rangle$ can be disentangled as $(|0\rangle + |1\rangle) \otimes |0\rangle$. The spt function is a partial one since disentanglement is considered to be a hard problem and it is usually done through case analyses as the ones in Figure 39d. Merging two sessions is valuable for analyzing the behavior of quantum conditionals. In rule TIF, the session ($\lambda_1 \uplus x[j, \mathtt{S}\ j]$) for the Boolean guard ($b@x[j]$) and the session for (λ_2) the body can be two separative sessions. Here, we first merge the two session through the mer rule in Figure 39d by computing the Cartesian product of the two type bases, such that if the two sessions are both CH types $\lambda_1 \uplus x[j, \mathtt{S}\ j] \mapsto \mathtt{CH}\ \overline{c_1}(m_1)$ and $\lambda_2 \mapsto \mathtt{CH}\ \overline{c_2}(m_2)$, the result is of type $\mathtt{CH}\ (\overline{c_1} \times \overline{c_2})(m_1 * m_2)$. After that, the quantum conditional behavior can be understood as applying a partial map function on the size $m_1 * m_2$ array of bitstrings, and we only apply the conditional body's effect on the second part (the $\overline{c_2}$ part) of some bitstrings whose first part is checked to be true by applying the Boolean guard b. **[Liyi: see how to merge the following to above]** Based on the new CH type with the set $\overline{c_1} \times \overline{c_2}$, the quantum conditional creates a new set based on $\overline{c_1} \times \overline{c_2}$, i.e., for each element $(\!|n|\!).d.c$ in the set, with $|(\!|n|\!)| = |\lambda_1|$, we compute Boolean guard b value by substituting qubit variables in b with the bitstring $(\!|n|\!)$, and the result $b[(\!|n|\!)/\lambda_1] \oplus d$ is true or not (d represents the bit value for the qubit at $x[j, \mathtt{S}\ j]$); if it is true, we replace the c bitstring by applying the conditional body on it; otherwise, we keep c to be the same. In short, the quantum conditional behavior can be understood as applying a partial map function on an m array of bitstrings, and we only apply the conditional body's effect on the second part of some bitstrings whose first part is checked to be true by applying the Boolean guard b.

Fourth, the type system enforces that the C classical variables can be evaluated to values in the compilation time. [23], while tracks M variables which represent the measurement results of quantum sessions. Rule TEXP enforces that a classical variable x is replaced with its assignment value n in s. The substitution statement $s[n/x]$ also evaluates classical expressions in s, which is described in Appendix B. In measurement rules (TMEA and TMEA-N), we apply some gradual typing techniques. There is an ghost expression ret generated from one step evaluation of the measurement. Before the step evaluation, rule TMEA types the partial measurement results as a classical M mode variable x and a possible quantum leftover λ as $\mathtt{CH}\ \infty$. After the step is transitioned, we know the exact value for x as (r, n), so that we carry the result to type λ as $\mathtt{CH}\ \overline{c'}(|\overline{c'}|)$. This does not violate type preservation because we have the subtyping relation $\mathtt{CH}\ \overline{c'}(|\overline{c'}|) \sqsubseteq_{|\lambda|} \mathtt{CH}\ \infty$.

Finally, the type system extracts the result type environment of a for-loop as $\sigma'[n_2/x]$ based on the extraction of a type environment invariant on the i-th loop step of executing a conditional if (b) s in rule TLOOP, regardless if the conditional is classical or quantum.

C.2 QAFNY Semantics and Type Soundness

We define the semantics of an QASM program as a partial function $[\![\ ]\!]$ from an instruction ι and input state φ to an output state φ', written $[\![\iota]\!]\varphi = \varphi'$, shown in Figure 26.

Recall that a state φ is a tuple of d qubit values, modeling the tensor product $q_1 \otimes \cdots \otimes q_d$. The rules implicitly map each variable x to a range of qubits in the state, e.g., $\varphi(x)$ corresponds to some sub-state $q_k \otimes \cdots \otimes q_{k+n-1}$ where $\Sigma(x) = n$. Many of the rules in Figure 26 update a *portion* of a state. We write $\varphi[(x, i) \mapsto q_{(x,i)}]$ to update the i-th qubit of variable x to be the (single-qubit) state $q_{(x,i)}$, and $\varphi[x \mapsto q_x]$ to update variable x according to the qubit *tuple* q_x. $\varphi[(x, i) \mapsto \uparrow q_{(x,i)}]$ and

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

$$\textsc{SPar}\ \frac{\varphi \equiv \varphi'}{(\varphi, s) \longrightarrow (\varphi', s)} \qquad \textsc{SMEA}\ \frac{\sigma(y) = y[0..k] \uplus \lambda \mapsto \sum_{j=0}^{m} z_j\,|c_j\rangle \qquad r = \forall j \in [0,m).\ c_j = (|n|).c \Rightarrow \sum |z_j|^2}{(\varphi, \texttt{let}\ x\ =\ \texttt{measure}(y)\ \texttt{in}\ s \longrightarrow (\varphi, \texttt{let}\ x\ =\ \texttt{ret}((y,(r,n)))\ \texttt{in}\ s)}$$

$$\textsc{SSEQ-1}\ \frac{(\varphi, s_1) \longrightarrow (\varphi', s_1')}{(\varphi, s_1\ ;\ s_2) \longrightarrow (\varphi', s_1'\ ;\ s_2)} \qquad \textsc{SMEA-N}\ \frac{\varphi(y) = \{y[0..k] \uplus \lambda : \sum_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\rangle\} \qquad \overline{c} = \{c_{j2} | c_{j1} = (|n|)\} \qquad c_j \in \overline{c}}{(\varphi, \texttt{let}\ x\ =\ \texttt{ret}((y,(r,n)))\ \texttt{in}\ s \longrightarrow (\varphi[x \mapsto (r,n), \lambda \mapsto \sum_{j=0}^{|\overline{c}|} \frac{1}{\sqrt{r}} z_j\,|c_j\rangle], s)}$$

$$\textsc{SSEQ-2}\ (\varphi, \{\}\ ;\ s_2) \longrightarrow (\varphi, s_2) \qquad \textsc{SA-CH}\ \frac{\varphi(\lambda) = \{\lambda \uplus \lambda' \mapsto \sum_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\rangle\} \quad |c_{j1}| = |\lambda| \quad [\![\mu]\!] c_{j1} = z_j'\,|c_{j1}'\rangle}{(\varphi, \lambda \leftarrow \mu) \longrightarrow (\varphi[\lambda \uplus \lambda' \mapsto \sum_{j=0}^{m} z_j' \cdot z_j\,|c_{j1}'.c_{j2}\rangle], \{\})}$$

$$\textsc{SEXP}\ \frac{}{(\varphi, \texttt{let}\ x\ =\ n\ \texttt{in}\ s) \longrightarrow (\varphi, s[n/x])}$$

$$\textsc{SIF}\ \frac{\lambda = \lambda_1 \uplus x[j, \texttt{S}\ j] \uplus \lambda_2 \qquad FV(b@x[j]) = \lambda \uplus x[j, \texttt{S}\ j] \quad \varphi(\lambda) = \sum_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\rangle] \qquad (\varphi, s) \longrightarrow^* (\varphi[\lambda \mapsto \sum_{j=0}^{m} z_j'\,|c_{j1}.c_{j2}'\rangle], \{\}) \qquad |c_{j1}| = |\lambda|}{(\varphi, \texttt{if}\ (b@x[j])\ s) \longrightarrow (\varphi[\lambda \mapsto \texttt{pmap}(m, z_j, z_j', c_{j1}, c_{j1}', c_{j2})], \{\})}$$

Fig. 36. QAFNY small step semantics. $[\![\mu]\!]c$ is the QASM semantics of interpreting reversible expression μ in Figure 26. Boolean expression b can be $a_1 = a_2$, $a_1 < a_2$ or `true`. $\varphi(y) = \{\lambda \mapsto q\}$ produces the map entry $\lambda \mapsto q$ and the range $y[0..|y|]$ is in λ. $\varphi(\lambda) = q$ is an abbreviation of $\varphi(\lambda) = \{\lambda \mapsto q\}$.

$\varphi[x \mapsto \uparrow q_x]$ are similar, except that they also accumulate the previous global phase of $\varphi(x, i)$ (or $\varphi(x)$). We use $\downarrow$ to convert a qubit $\alpha(b)\overline{q}$ to an unphased qubit $\overline{q}$.

Function `xg` updates the state of a single qubit according to the rules for the standard quantum gate X. `cu` is a conditional operation depending on the `Nor`-basis qubit (x, i). `SR` (or $\texttt{SR}^{-1}$) applies an $m+1$ series of `RZ` (or $\texttt{RZ}^{-1}$) rotations where the i-th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$ (or $\alpha(-\frac{1}{2^{m-i+1}})$). `qt` applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes \cdots \otimes |b_i\rangle$ (assuming $\Sigma(y) = i$) and n is the degree of approximation. If $n = i$, then the operation is the standard QFT. Otherwise, each qubit in the state is mapped to $|\Phi(\frac{y}{2^{n-k}})\rangle$, which is equal to $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(\frac{y}{2^{n-k}})\,|1\rangle)$ when $k < n$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ when $n \leq k$ (since $\alpha(n) = 1$ for any natural number n). $\texttt{qt}^{-1}$ is the inverse function of `qt`. Note that the input state to $\texttt{qt}^{-1}$ is guaranteed to have the form $\bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{n-k}})\rangle)$ because it has type `Phi` n. $\texttt{pm}_l$, $\texttt{pm}_r$, and $\texttt{pm}_a$ are the semantics for `Lshift`, `Rshift`, and `Rev`, respectively.

C.3 Logic Proof System

The reason of having the session type system in Figure 35 is to enable the proof system given in **??**. Every proof rule is a structure as $\Omega \vdash_g T; P \vdash_s \{T'\}\ Q\ \{\}$, where g and Ω are the type entities mentioned in **??**. T and T' are the pre- and post- type predicates for the statement s, meaning that there is type environments $\mathcal{T}$ and $\mathcal{T}'$, such that $\mathcal{T} \models T$, $\mathcal{T}' \models T'$, $g, \Omega, \mathcal{T} \vdash s : \zeta \triangleright \tau$, and

$(\zeta \mapsto \tau) \in \mathcal{T}'$. We denote $(\mathcal{T}, \mathcal{T}') \models (T, s, T') : \zeta \triangleright \tau$ as the property described above. P and Q are the pre- and post- Hoare conditions for statement s.

The proof system is an imitation of the classical Hoare Logic array theory. We view the three different quantum state forms in Figure 21 as arrays with elements in different forms, and use the session types to guide the occurrence of a specific form at a time. Sessions, like the array variables in the classical Hoare Logic theory, represent the stores of quantum states. The state changes are implemented by the substitutions of sessions with expressions containing operation's semantic transitions. The substitutions can happen for a single index session element or the whole session.

Rule PA-NOR and PA-CH specify the assignment rules. If a session ζ has type `Nor`, it is a singleton array, so the substitution $[\![a]\!]\zeta/\zeta$ means that we substitute the singleton array by a term with the a's application. When ζ has type `CH`, term $\zeta[k]$ refers to each basis state in the entanglement. The assignment is an array map operation that applies a to every element in the array. For example, in Figure 19 line 12, we apply a series of `H` gates to array x. Its post-condition is $[(x, 0, n)] = \bigotimes_{k=0}^{n} |\Phi(0)\rangle$, where $[(x, 0, n)]$ is the session representing register variable x. Thus, replacing the session $[(x, 0, n)]$ with the `H` application results in a pre-condition as $\mathtt{H}[(x, 0, n)] = \bigotimes_{k=0}^{n} |\Phi(0)\rangle$, which means that $[(x, 0, n)]$ has the state $|0\rangle^n$.

Rule P-MEA is the rule for partial/complete measurement. y's session is ζ, but it might be a part of an entangled session $\zeta \uplus \zeta'$. After the measurement, M-mode x has the measurement result $(\mathtt{as}(\zeta[v])^2, \mathtt{bs}(\zeta[v]))$ coming from one possible basis state of y (picking a random index v in ζ), $\mathtt{as}(\zeta[v])$ is the amplitude and $\mathtt{bs}(\zeta[v])$ is the base. We also remove y and its session ζ $(\perp/\zeta)$ in the new pre-condition because it is measured away. The removal means that the entangled session $\zeta \uplus \zeta'$ is replaced by ζ' with the re-computation of the amplitudes and bases for each term.

Rule P-IF deals with a quantum conditional where the Boolean guard $b(@x[v])$ is of type $\bigotimes_n$ `CH` $2m(\beta_1 \cdot 0 \cup \beta_2 \cdot 1)$. The bases are split into two sets $\beta_1 \cdot 0$ and $\beta_2 \cdot 1$, where the last bit represents the base state for the $x[v]$ position. In quantum computing, a conditional is more similar to an assignment, where we create a new array to substitute the current state represented by the session $\zeta \uplus [(x, v, v+1)] \uplus \zeta'$. Here, the new array is given as $(\zeta \uplus 0 \uplus \zeta')$++$(\zeta \uplus 1 \uplus [\![s]\!]\zeta')$, where we double the array: if the $x[v]$ position is 0, we concatenate the current session ζ' for the conditional body, if $x[v] = 1$, we apply $[\![s]\!]$ on the array ζ' and concatenate it to $(\zeta \uplus 1)$.

Rule P-Loop is an initiation of the classical while rule in Hoare Logic with the loop guard possibly having quantum variables. In QWhile, we only has `for-loop` structure and we believe it is enough to specify any current quantum algorithms. For any i, if we can maintain the loop invariant $P(i)$ and $T(i)$ with the post-state $P(f(i))$ and $T(f(i))$ for a single conditional `if` $(x[i])$ s, the invariant is maintained for multiple steps for i from the lower-bound a_1 to the upper bound a_2.

Rule P-DIS proves a diffusion operator `diffuse`(x). The quantum semantics for `diffuse`(x) is $\frac{1}{2^n}(2\sum_{i=0}^{2^n}(\sum_{j=0}\alpha_j)\,|i\rangle - \sum_{j=0}\alpha_j\,|x_j\rangle)$. As an array operation, `diffuse`(x) with the session ζ is an array operation as follows: assume that $\zeta = (x, 0, \Sigma(x)) \uplus \zeta_1$, for every k, if $\zeta[k]$'s value is $\theta_k(\overline{d_x} \cdot \overline{d_1})$, for any bitstring z in $\mathcal{P}(\Sigma(x))$, if $z \cdot \overline{d_1}$ is not a base for $\zeta[j]$ for any j, then the state is $\frac{1}{2^{n-1}}\sum_{k=0}\theta_k(z \cdot \overline{d_1})$; if the base of $\zeta[j]$ is $z \cdot \overline{d_1}$, then the state for $\zeta[j]$ is $\frac{1}{2^{n-1}}(\sum_{k=0}\theta_k) - \theta_j(z \cdot \overline{d_1})$.

We evaluate VQO by (1) demonstrating how it can be used for validation, both by verification and random testing, and (2) by showing that it gets good performance in terms of resource usage compared to Quipper, a state-of-the-art quantum programming framework [14]. This section presents the arithmetic operators we have implemented in ⓄQASM, while the next section discusses the geometric operators and expressions implemented in ⓄQIMP. The following section presents an end-to-end case study applying Grover's search.

1		$\{A(x) * A(y)\}$ where $A(\beta) = \beta[0..n] \mapsto \|\bar{0}\rangle$ $B = 1 < a < N \wedge n > 0 \wedge N < 2^n \wedge \gcd(a, N) = 1$
2		$\Rightarrow \{\|\mathsf{H}\|(x[0..n]) \mapsto C * A(y) * B\}$ where $C = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (\|0\rangle + \|1\rangle)$
3	$x \leftarrow \mathsf{H}$;	$\{x[0..n] \mapsto C * A(y) * B\}$
4		$\Rightarrow \{x[0..n] \mapsto C * \|y{+}1\|(y[0..n]) \mapsto \|\bar{0}.1\rangle * B\}$
5	$y \leftarrow y{+}1$;	$\{x[0..n] \mapsto C * y[0..n] \mapsto \|\bar{0}.1\rangle * B\}$
6		$\Rightarrow \{E(0) * B\}$ where $E(k) =$ $x[0..n-k] \mapsto \frac{1}{\sqrt{2^{n-k}}} \bigotimes_{j=0}^{n-k} (\|0\rangle + \|1\rangle) *$ $\{x[0..k], y[0..n]\} \mapsto \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}} \|(\|j\|).(\|a^j \% N\|)\rangle$
7	for (int $j := 0$; $j < n$ && $x[j]$; ++j)	$\{E(j) * B\}$
8	{ $y \leftarrow a^{2^j} y \% N$	$\{E(j+1) * B\}$
9	}	$\{E(n) * B\}$
10		$\Rightarrow \{\{x[0..n], y[0..n]\} \mapsto \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} \|(\|j\|).(\|a^j \% N\|)\rangle * B\}$
11	let u = measure(y) in ...	$\{ x[0..n] \mapsto \frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} \|t + kp\rangle \wedge p = \mathsf{ord}(a, N) \wedge \mathsf{nat}(u) = a^t \% N \wedge s = \mathsf{rnd}(\frac{2^n}{p}) \wedge B \}$

Fig. 37. Pre-measurement quantum steps of the Shor's algorithm. Second half in Figure 38. $\mathsf{nat}(u)$ gets the integer number part of u (mode M). $\mathsf{ord}(a, N)$ gets the order of a and N. $\mathsf{rnd}(r)$ rounds r to the nearest integer.

11	let z = measure(y) in	$\{ x[0..n] \mapsto \frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} \|t + jr\rangle \wedge \mathsf{nat}(z) = a^n \% N \wedge s = \mathsf{rnd}(\frac{2^n}{r}) \wedge B \}$
12	$x \leftarrow \mathsf{QFT}^{-1}$;	$\{x[0..n] \mapsto \frac{1}{\sqrt{s2^n}} \Sigma_{k=0}^{2^n} (\omega^{tk} \Sigma_{j=0}^{s} \omega^{tkj}) \|k\rangle \wedge s = \mathsf{rnd}(\frac{2^n}{r}) \wedge B\}$
13	let u = measure(x) in	$\{\mathsf{nat}(u) = r \wedge \mathsf{pos}(u) = \frac{4}{\pi^{2r}} \wedge s = \mathsf{rnd}(\frac{2^n}{r}) \wedge r = \mathsf{ord}(a, N) \wedge B\}$
14	post(u)	$\{\mathsf{nat}(\mathsf{post}(u)) = r \wedge r = \mathsf{ord}(a, N) \wedge \mathsf{pos}(u) = \frac{4e^{-2}}{\pi^2 \log_2^4 N \wedge B}\}$

$$B = 1 < a < N \wedge n > 0 \wedge N < 2^n \wedge \gcd(a, N) = 1 \qquad \omega = e^{\frac{2\pi i}{2^n}}$$

Fig. 38. Second half of the Shor's algorithm quantum part in Qafny.

C.4 Implemented Operators

Figure 16 and **??** tabulate the arithmetic operators we have implemented in ℚASM.

The addition and modular multiplication circuits (parts (a) and (d) of Figure 16) are components of the oracle used in Shor's factoring algorithm [48], which accounts for most of the algorithm's cost [12]. The oracle performs modular exponentiation on natural numbers via modular multiplication, which takes a quantum variable x and two co-prime constants $M, N \in \mathbb{N}$ and produces $(x * M)\%N$. We have implemented two modular multipliers—inspired by Beauregard [1] and Markov and Saeedi [33]—in ℚASM. Both modular multipliers are constructed using controlled modular addition by a constant, which is implemented in terms of controlled addition and subtraction by a constant, as shown in **??**. The two implementations differ in their underlying adder and subtractor circuits: the first (QFT) uses a quantum Fourier transform-based circuit for addition and subtraction [10],

$$\begin{array}{lll} \tau & \sqsubseteq & \tau \\ \mathsf{Nor} & \sqsubseteq & \mathsf{CH} \\ \mathsf{Had} & \sqsubseteq & \mathsf{CH} \end{array}$$

(a) Subtyping

$$\begin{array}{lll} q & \equiv_{|q|} & q \\ |c\rangle & \equiv_n & \Sigma_{j=0}^{1} |c\rangle \\ \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle) & \equiv_n & \Sigma_{j=0}^{2^n} \frac{\alpha(\sum_{k=0}^{n} r_k \cdot (\!|j|\!)[k])}{\sqrt{2^n}} |j\rangle \end{array}$$

(b) Quantum Value Equivalence

$$\lambda \equiv \lambda \qquad x[n,n] \equiv \bot \qquad \bot \uplus \lambda \equiv \lambda \qquad x[n,m] \uplus \lambda \equiv x[n,j] \uplus x[j,m] \uplus \lambda \;\; \mathtt{where}\;\; n \le j < m$$

(c) Session Equivalence

$$\begin{array}{llll} \sigma & \le \sigma & \varphi & \equiv \varphi \\ \{\bot : \tau\} \cup \sigma & \le \sigma & \{\bot : q\} \cup \varphi & \equiv \varphi \\ \{\lambda : \tau\} \cup \sigma & \le \{\lambda : \tau'\} \cup \sigma \;\; \mathtt{where}\;\; \tau \sqsubseteq_{|\lambda|} \tau' & \{\lambda : q\} \cup \varphi & \equiv \{\lambda : q'\} \cup \varphi \;\; \mathtt{where}\;\; q \equiv_{|\lambda|} q' \\ \{\lambda_1 \uplus l_1 \uplus l_2 \uplus \lambda_2 : \tau\} \cup \sigma & \le \{\lambda_1 \uplus l_2 \uplus l_1 \uplus \lambda_2 : \tau\} \cup \sigma & \{\lambda_1 \uplus l_1 \uplus l_2 \uplus \lambda_2 : q\} \cup \varphi & \equiv \{\lambda_1 \uplus l_2 \uplus l_1 \uplus \lambda_2 : \mathtt{mut}(q, |\lambda_1|)\} \cup \varphi \\ \{\lambda_1 : \tau\} \cup \{\lambda_2 : \tau\} \cup \sigma & \le \{\lambda_1 \uplus \lambda_2 : \tau\} \cup \sigma & \{\lambda_1 : q_1\} \cup \{\lambda_2 : q_2\} \cup \varphi & \equiv \{\lambda_1 \uplus \lambda_2 : \mathtt{mer}(q_1, q_2)\} \cup \varphi \\ \{\lambda_1 \uplus \lambda_2 : \tau\} \cup \sigma & \le \{\lambda_1 : \tau\} \cup \{\lambda_2 : \tau\} \cup \sigma \;\; \mathtt{where}\;\; \tau \neq \mathsf{CH} & \{\lambda_1 \uplus \lambda_2 : \varphi\} \cup \sigma & \equiv \{\lambda_1 : \varphi_1\} \cup \{\lambda_2 : \varphi_2\} \cup \sigma \;\; \mathtt{where}\;\; \mathtt{spt}(\tau, |\lambda_1|) = (\varphi_1, \varphi_2) \end{array}$$

(d) Environment Equivalence

(e) State Equivalence

$$\begin{array}{l} \mathtt{pmut}((c_1.i_1.i_2.c_2), n) = (c_1.i_2.i_1.c_2) \;\; \mathtt{when}\; |c_1| = n \\ \mathtt{mut}(|c\rangle, n) = |\mathtt{pmut}(c, n)\rangle \\ \mathtt{mut}(\frac{1}{\sqrt{2^m}} (q_1 \bigotimes (|0\rangle + \alpha(r_n)\,|1\rangle) \bigotimes (|0\rangle + \alpha(r_{n+1})\,|1\rangle) \bigotimes q_2), n) \\ \quad = \frac{1}{\sqrt{2^m}} (q_1 \bigotimes (|0\rangle + \alpha(r_{n+1})\,|1\rangle) \bigotimes (|0\rangle + \alpha(r_n)\,|1\rangle) \bigotimes q_2) \quad \mathtt{when}\; |q_1| = n \\ \mathtt{mut}(\Sigma_{j=0}^{m} z_j\,|c_j\rangle, n) = \Sigma_{j=0}^{m} z_j\,|\mathtt{pmut}(c_j, n)\rangle \\ \mathtt{mer}(|c_1\rangle, |c_2\rangle) = |c_1.c_2\rangle \\ \mathtt{mer}(\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle), \frac{1}{\sqrt{2^m}} \bigotimes_{j=0}^{m} (|0\rangle + \alpha(r_j)\,|1\rangle)) = \frac{1}{\sqrt{2^{n+m}}} \bigotimes_{j=0}^{n+m} (|0\rangle + \alpha(r_j)\,|1\rangle) \\ \mathtt{mer}(\Sigma_{j=0}^{n} z_j\,|c_j\rangle, \Sigma_{k=0}^{m} z_k\,|c_k\rangle) = \Sigma^{n*m} z_j \cdot z_k\,|c_j.c_k\rangle \\ \mathtt{spt}(|c_1.c_2\rangle, n) = (|c_1\rangle, |c_2\rangle) \;\; \mathtt{when}\; |c_1| = n \\ \mathtt{spt}(q_1 \bigotimes q_2, n) = (q_1, q_2) \;\; \mathtt{when}\; |q_1| = n \end{array}$$

Fig. 39. Qafny type/state relations. $\{(\!|j|\!)|j \in [0, 2^n)\}(2^n)$ defines a set $\{(\!|j|\!)|j \in [0, 2^n)\}$ with the emphasis that it has 2^n elements. $\{0, 1\}$ is a set of two single element bitstrings 0 and 1. $\cdot$ is the multiplication operation, $(\!|j|\!)$ turns a number j to a bitstring, $(\!|j|\!)[k]$ takes the k-th element in the bitstring $(\!|j|\!)$, and $|j\rangle$ is an abbreviation of $|(\!|j|\!)\rangle$. We use set union ($\cup$) to describe the state concatenation with the empty set operation $\emptyset$. i is a single bit either 0 or 1. The . operation is bitstring concatenation. Term $\Sigma^{n*m}\,P$ is a summation formula that omits the indexing details. Term $(\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} q_j) \bigotimes (\frac{1}{\sqrt{2^m}} \bigotimes_{j=0}^{m} q_j)$ is equivalent to $\frac{1}{\sqrt{2^{n+m}}} \bigotimes_{j=0}^{n+m} q_j$.

while the second (TOFF) uses a ripple-carry adder [33], which makes use of classical controlled-controlled-not (Toffoli) gates.

C.5 State Equivalence

As we suggested in **??**, quantum states have certain level of permutation symmetries. Essentially, quantum computation is implemented as circuits. In Figure 2, if the first and second circuit lines and qubits are permuted, it is intuitive that the two circuit results are equivalence up to the permutation. Additionally, as indicated in **??**, we need quantum sessions to be split and regrouped sometimes. All these properties are formulated in Qafny as equational properties in Figure 39 that rely on session rewrites, which can then be used as builtin libraries in the proof system. As one can imagine, the equational properties might bring nondeterminism in the Qafny implementation, such that

Predicate modeling:

$$\frac{\Omega;\sigma \vdash \kappa \qquad \models \varphi(\kappa) \mapsto q}{\Omega;\sigma,\varphi \models \kappa \mapsto q} \qquad \frac{q \equiv_{|q|} q'}{\models q \mapsto q'} \qquad \frac{\sum_{j=0}^{m} z_j |c_j\rangle \subseteq \sum_{j=0}^{m} z'_j |c'_j\rangle \qquad \sum_{j=0}^{m} z'_j |c'_j\rangle \subseteq \sum_{j=0}^{m} z_j |c_j\rangle}{\models \sum_{j=0}^{m} z_j |c_j\rangle \mapsto \sum_{j=0}^{m} z'_j |c'_j\rangle}$$

$$\frac{\sigma \perp \sigma' \qquad \varphi \perp \varphi' \qquad \Omega;\sigma,\varphi \models P \qquad \Omega;\sigma',\varphi' \models Q}{\Omega;\sigma \cup \sigma', \varphi \cup \varphi' \models P * Q}$$

Sequence Semantic and Proof Rules:

SSeq-1
$$\frac{(\varphi, s_1) \longrightarrow (\varphi', s'_1)}{(\varphi, s_1 \,;\, s_2) \longrightarrow (\varphi', s'_1 \,;\, s_2)}$$

SSeq-2
$$(\varphi, \{\} \,;\, s_2) \longrightarrow (\varphi, s_2)$$

PSeq
$$\frac{\Omega;\sigma \vdash_g s_1 \triangleright \sigma' \qquad \Omega;\sigma \vdash_g \{P\}\, s_1\, \{R\} \qquad \Omega;\sigma[\uparrow \sigma'] \vdash_g \{R\}\, s_2\, \{Q\}}{\Omega;\sigma \vdash_g \{P\}\, s_1 \,;\, s_2\, \{Q\}}$$

Pre-condition strengthening and Post-condition weakening Proof Rules:

PConL
$$\frac{(\Omega,\sigma,P) \Rightarrow (\Omega,\sigma',P') \qquad \Omega;\sigma' \vdash_g \{P'\}\, s\, \{Q\}}{\Omega;\sigma \vdash_g \{P\}\, s\, \{Q\}}$$

PConR
$$\frac{\Omega,\sigma \vdash_g s_1 \triangleright \sigma' \qquad \Omega;\sigma' \vdash_g \{P\}\, s\, \{Q'\} \qquad (\Omega,\sigma'',Q') \twoheadrightarrow (\Omega,\sigma[\uparrow \sigma'],Q)}{\Omega;\sigma \vdash_g \{P\}\, s\, \{Q\}}$$

$$(\Omega,\sigma,P) \Rightarrow (\Omega,\sigma',P') \triangleq \Omega,\sigma \vdash P \wedge \Omega,\sigma' \vdash P' \wedge \sigma \preceq \sigma' \wedge P \Rightarrow P'$$
$$(\Omega,\sigma,Q) \twoheadrightarrow (\Omega,\sigma',Q') \triangleq \Omega,\sigma \vdash Q \wedge \Omega,\sigma' \vdash Q' \wedge \sigma' \preceq \sigma \wedge Q \Rightarrow Q'$$

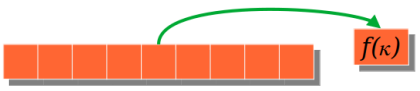
Fig. 40. Sequence and Consequence Rules

the automated system does not know which equations to apply in a step. In dealing with the nondeterminism, we design a type system for Qafny to track the uses, split, and join of sessions, as well as the three state types in every transition step, so that the system knows exactly how to apply an equation.

Figure 39 shows the equivalence relations on types and states. Figure 39a shows the subtyping relation such that Nor and Had subtype to CH. Correspondingly, the subtype of Nor to CH represents the first line equation in Figure 39b, where a Nor state is converted to a CH form. Similarly, a Had state can also be converted to a CH state in the second line. Additionally, Figure 39c defines the equivalence relations for the session concatenation operation $\uplus$: it is associative, identitive with the identity empty session element $\perp$. We also view a range $x[n,n]$ to be empty ($\perp$), and a range $x[n,m]$ can be split into a two ranges in the session as $x[n,j] \uplus x[j,m]$.

The main result to define state equivalence is to capture the permutation symmetry, split, and join of sessions introduced in **??**. The first rule describes the case for empty session, while the second rule in Figure 39e connects the quantum value equivalence to the state equivalence. The third rule describes the qubit permutation equivalence by the mut function. The fourth rule describes the join of two sessions in a state. For the two sessions are of the type Nor and Had, a join means an array concatenation. If the two sessions have CH types, a join means a Cartesian product of the two basis states. The final rule is to split a session, where we only allows the split of a Nor and Had type state and their splits are simply array splits. Splitting a CH type state is equivalent to qubit disentanglement, which is a hard problem and we need to upgrade the type system to permit certain types of such disentanglement. In Appendix C, we upgrade the Qafny type system to a dependent type system to track the disentanglement of CH type state.

(a) Application Analogy



(b) App Function Modeling

$$\frac{\forall j.\ |c_{j1}| = n \qquad \Omega;\sigma;\varphi \models \sum_{j=0}^{m} z_j\,|[\![\mu]\!](c_{j1}).c_{j2}\ \beta_j\rangle \mapsto q}{\Omega;\sigma;\varphi \models \eth\ n.[\![\mu]\!](\sum_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\ \beta_j\rangle) \mapsto q}$$

(c) Semantic/Proof Rules

SH-N

$$\frac{FV(\emptyset, l) = \kappa \qquad \varphi(\kappa) = |c\rangle}{(\varphi, l \leftarrow \mathsf{H}) \longrightarrow (\varphi[\kappa \mapsto \frac{1}{\sqrt{2^{|c|}}} \bigotimes_{j=0}^{|c|} (|0\rangle + \alpha(\frac{1}{2^{c[j]}})\,|1\rangle)], \{\})}$$

PH-N

$$\frac{FV(\Omega, l) = \kappa \qquad \sigma(\kappa) = \tau}{\Omega;\sigma \vdash_g \{P[\eth\ \kappa.[\![\mathsf{H}]\!](\kappa)/\kappa]\}\ l \leftarrow \mathsf{H}\ \{P\}}$$

Fig. 41. Oracle application and state preparation rules. $\eth$ is an array map operation, where $\eth\ \kappa.[\![\mu]\!](\kappa \uplus \kappa')$ means that for every basis state in the state of $\kappa \uplus \kappa'$, we apply $[\![\mu]\!]$ to κ part of session.

$$\frac{\Omega;\sigma_2 \vdash_{\mathsf{M}} \{X(\mathsf{S}\ j) * \{y[0..n]\} \mapsto C(j).1\}\ s\ \{X(\mathsf{S}\ j) * \{y[0..n]\} \mapsto C'(j).1\}}{\Omega;\sigma_2 \vdash_{\mathsf{M}} \{X(\mathsf{S}\ j) * \mathcal{M}(b, \{y[0..n]\}) \mapsto 0.C(j) + 1.C(j)\}\ s\ \{X(\mathsf{S}\ j) * \{y[0..n]\} \mapsto C'(j).1\}}$$

$$\frac{\Omega, \sigma_1 \vdash_{\mathsf{M}} \{X(\mathsf{S}\ j) * \{x[0..\mathsf{S}\ j], y[0..n]\} \mapsto 0.C(j) + 1.C(j)\}\mathtt{if}\ (x[j])\ s \quad \{X(\mathsf{S}\ j) * \mathcal{U}(\neg x[j], \{x[0..\mathsf{S}\ j], y[0..n]\}) \mapsto 0.C(j) * \mathcal{U}(x[j], \{x[0..\mathsf{S}\ j], y[0..n]\}) \mapsto C'(j).1\}}{\Omega;\sigma \vdash_{\mathsf{M}} \{X(j) * \{x[0..j], y[0..n]\} \mapsto C(j)\}\ \mathtt{if}\ (x[j])\ s\ \{X(j-1) * \{x[0..\mathsf{S}\ j], y[0..n]\} \mapsto 0.C(j) + 1.C'(j)\}}$$

$X(j) = \frac{1}{\sqrt{2^{n-j}}} \bigotimes_{j=0}^{n-j} (|0\rangle + |1\rangle)$ $\quad C(j) = \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}}\,|(|j|)^k.(|a^{(|j|)^k}\ \%\ N|)\rangle$ $\quad i.C(j) = \Sigma_{j=0}^{2^{\mathsf{S}\ k}} \frac{1}{\sqrt{2^{\mathsf{S}\ k}}}\,|(|j|)^k.i.(|a^{(|j|)^k}\ \%\ N|)\rangle$

$C'(j).i = \Sigma_{j=0}^{2^{\mathsf{S}\ k}} \frac{1}{\sqrt{2^{\mathsf{S}\ k}}}\,|(|a^{(|j|)^k.1}\ \%\ N|)\,|\,(|j|)^k.i\rangle$ $\quad i.C'(j) = \Sigma_{j=0}^{2^{\mathsf{S}\ k}} \frac{1}{\sqrt{2^{\mathsf{S}\ k}}}\,|(|j|)^k.i.(|a^{(|j|)^k.1}\ \%\ N|)\rangle$

$\sigma_1 = \{x[0..n-\mathsf{S}\ j] \mapsto \mathsf{Had}, \{x[0..\mathsf{S}\ j], y[0..n]\} \mapsto \mathsf{CH}\}$ $\quad \sigma = \{x[0..n-\mathsf{S}\ j] \mapsto \mathsf{Had}, y[0..n] \mapsto \mathsf{CH}\}$ $\quad s = y \leftarrow a^{2^j}\ y\ \%\ N$

C.6 Additional Proof Rules

The proof is built from bottom up. We first cut the Had type state into two sessions ($x[0..n-\mathsf{S}\ j]$ and $x[j]$), join $x[j]$ with session $\{x[0..j], y[0..n]\}$, and double the state elements to be $0.C(j) + 1.C(j)$, which is proved by applying the consequence rules. Notice that the type environment is also transitioned from σ to σ_1. By the same strategy of the $\mathcal{U}$ rule in Figure 41b, we combine the two $\mathcal{U}$ terms into the final result. The second step applies rule PIF to substitute session $\{x[0..\mathsf{S}\ j], y[0..n]\}$ with the mask construct $\mathcal{M}(b, \{y[0..n]\})$ in the pre-condition and create two $\mathcal{U}$ terms in the post-condition. The step on the top applies the modulo multiplication on every element in the masked state $\mapsto C(j).1$ by rule PA-CH.

C.7 Well-formed Sessions and Predicates

Definition C.1 (Well-formed session domain). The domain of a environment σ (or state φ) is *well-formed*, written as $\Omega \vdash \mathsf{dom}(\sigma)$ (or $\mathsf{dom}(\varphi)$), iff for every session $\kappa \in \mathsf{dom}(\sigma)$ (or $\mathsf{dom}(\varphi)$):

$$\textsc{Frame}\ \frac{FV(s)\cap FV(R)=\emptyset \quad FV(s)\subseteq \mathrm{dom}(\sigma) \quad \sigma\perp\sigma' \quad \Omega;\sigma\vdash_g\{P\}\,s\,\{Q\}}{\Omega;\sigma\cup\sigma'\vdash_g\{P*R\}\,s\,\{Q*R\}} \qquad \frac{\sigma\perp\sigma' \quad \varphi\perp\varphi' \quad \Omega;\sigma;\psi;\varphi\models_g P \quad \Omega;\sigma';\psi;\varphi'\models_g Q}{\Omega;\sigma\cup\sigma';\psi;\varphi\cup\varphi'\models_g P*Q}$$

$$\textsc{SSeq-1}\ \frac{(\psi,\varphi,s_1)\longrightarrow(\psi',\varphi',s_1')}{(\psi,\varphi,s_1\ ;\ s_2)\longrightarrow(\psi',\varphi',s_1'\ ;\ s_2)} \qquad \textsc{SSeq-2}\ (\psi,\varphi,\{\}\ ;\ s_2)\longrightarrow(\psi,\varphi,s_2)$$

$$\textsc{PSeq}\ \frac{\Omega;\sigma\vdash_g s_1\triangleright\sigma' \quad \Omega;\sigma\vdash_g\{P\}\,s_1\,\{R\} \quad \Omega;\sigma[\uparrow\sigma']\vdash_g\{R\}\,s_2\,\{Q\}}{\Omega;\sigma\vdash_g\{P\}\,s_1\ ;\ s_2\,\{Q\}}$$

$$\textsc{PConL}\ \frac{(\Omega,\sigma,P)\Rightarrow(\Omega,\sigma',P') \quad \Omega;\sigma'\vdash_g\{P'\}\,s\,\{Q\}}{\Omega;\sigma\vdash_g\{P\}\,s\,\{Q\}} \qquad \textsc{PConR}\ \frac{\Omega,\sigma\vdash_g s_1\triangleright\sigma' \quad \Omega;\sigma'\vdash_g\{P\}\,s\,\{Q'\} \quad (\Omega,\sigma'',Q')\twoheadrightarrow(\Omega,\sigma[\uparrow\sigma'],Q)}{\Omega;\sigma\vdash_g\{P\}\,s\,\{Q\}}$$

$$(\Omega,\sigma,P)\Rightarrow(\Omega,\sigma',P')\triangleq\Omega,\sigma\vdash P\wedge\Omega,\sigma'\vdash P'\wedge\sigma\preceq\sigma'\wedge P\Rightarrow P'$$
$$(\Omega,\sigma,Q)\twoheadrightarrow(\Omega,\sigma',Q')\triangleq\Omega,\sigma\vdash Q\wedge\Omega,\sigma'\vdash Q'\wedge\sigma'\preceq\sigma\wedge Q\Rightarrow Q'$$

Fig. 42. Sequence and Consequence Rules

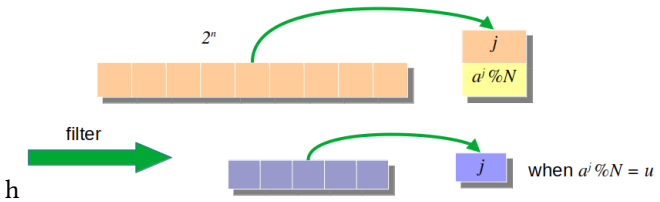


Fig. 43. The array analogy of Shor's first half in Figure 3.

- κ is disjoint unioned, i.e., for every two ranges $x[i..j]$ and $y[i'..j']$, $x[i..j]\cap y[i'..j']=\emptyset$.
- For every range $x[i..j]\in\kappa$, $\Omega(x)=\mathtt{Q}\ n$ and $[i,j)\subseteq[0,n)$.

Definition C.2 (Well-formed state predicate). A predicate P is well-formed, written as $\Omega,\sigma\vdash P$, iff every variable and session appearing in P is defined in Ω and σ, respectively; particularly, if $P=\kappa\mapsto q*P'$, $\kappa\in\mathrm{dom}(\sigma)$.

C.8 Translation from QAFNY to SQIR Full Rules

Figure 44 depicts compilation rules. The first two rules in the first line show how a `let`-binding is handled for a `C` and `M` kind variable. Similar to the type system, we assume that `let`-binding introduces new variables probably with proper variable renaming. The last rule in the first line describes an oracle application compilation, which is handled by the ⟦⟧QASM compiler [28]. The QAFNY type system ensures that the qubits mentioned in μ are the same as qubits in session κ, so that the translation does not rely on κ itself.

$$\frac{x \notin \mathtt{dom}(\Omega) \quad \Omega[x \mapsto \mathtt{C}]; \gamma; n \vdash s[m/x] \to \epsilon}{\Omega; \gamma; n \vdash \mathtt{let}\ x\ =\ m\ \mathtt{in}\ s \to \epsilon} \qquad \frac{x \notin \mathtt{dom}(\Omega) \quad \Omega[x \mapsto \mathtt{M}]; \gamma; n \vdash s[(r,n)/x] \to \epsilon}{\Omega; \gamma; n \vdash \mathtt{let}\ x\ =\ (r,n)\ \mathtt{in}\ s \to \epsilon} \qquad \frac{\Omega; \gamma; n \vdash \mu \to \epsilon}{\Omega; \gamma; n \vdash \kappa \leftarrow \mu \to \epsilon}$$

$$\frac{\Omega; \gamma; n \vdash b@x[i] \to \epsilon \qquad \Omega; \gamma; n \vdash s \to \epsilon'}{\Omega; \gamma; n \vdash \mathtt{if}\ (b@x[i])\ s \to \epsilon\ ;\ \mathtt{ctrl}(\gamma(x[i]), \epsilon')} \qquad \frac{\forall t \in [i,j).\, \Omega; \gamma; n \vdash \mathtt{if}\ (b[t/x])\ s[t/x] \to \epsilon_t}{\Omega; \gamma; n \vdash \mathtt{for}\ (\mathtt{int}\ x \in [i,j)\ \&\&\ b)\ s \to \epsilon_i\ ;\ldots;\ \epsilon_{j-1}}$$

Fig. 44. Select QAFNY to SQIR translation rules (SQIR circuits are marked blue)