Quantum Natural Proof

ANONYMOUS AUTHOR(S)

Q-Dafny.

1 BACKGROUND

Here, we provide background information for quantum computing and an example explaining QNP and QAFNY .

1.1 Preliminaries

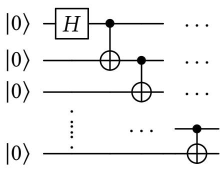
We begin with some background on quantum computing.

Quantum States. A quantum state consists of one or more quantum bits (*qubits*). A qubit can be expressed as a two dimensional vector $\left(\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right)$ 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 $|0\rangle = \left(\begin{smallmatrix}1\\ 0\end{smallmatrix}\right)$ and $|1\rangle = \left(\begin{smallmatrix}0\\ 1\end{smallmatrix}\right)$ 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*.

Alternative State Representation. n-qubit quantum states are typically represented as 2^n dimensional vectors above. Alternatively, the state 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. Qubits that are in superposition but not entangled, such as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, 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 H*adamard typed state* (Had) in QNP. The above two qubit superposition can be expressed as $\frac{1}{\sqrt{4}} \bigotimes_{j=0}^{2} (|0\rangle + |1\rangle)$. $\alpha(r_j)$ is named the *local phase* of the state, which are special quantum amplitudes (see below) such that the norm is 1, i.e., $|\alpha(r_j)|^2 = 1$. In the above state, we can view the local phase 1 as e^0, and $\frac{1}{\sqrt{4}}e^0$ 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 named *amplitude*, c_j is an n-length bitstring named *basis*, and $m \le 2^n$. Each j-th term $z_j\,|c_j\rangle$ in the formula represents a *basis state* in a superposition state as $z_0\,|c_0\rangle + \ldots + z_{m-1}\,|c_{m-1}\rangle$. 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 $\sum_0^m |z_j|^2 \le 1$, because the 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 $\Sigma_{j=0}^{m_1} z_j\,|c_j\rangle$ locally, so that $\sum_0^{m_1} |z_i|^2 < 1$. Obviously, in the top-most program evaluation level, every state satisfies the restriction that $\sum_0^m |z_j|^2 = 1$.

Quantum Computations. In the *QRAM model* [Knill 1996] 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. High-level quantum



(a) Quantum circuit

1 $\{x[0..n] \mapsto |\overline{0}\rangle\}$

2 $x[0] \leftarrow \mathsf{H};$

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

4 $\mathtt{for}\ (\mathtt{int}\ j \in [0, n-1)\ \&\&\ x[j])$

5 $\quad \{x[0..\mathsf{S}\ j] \mapsto \Sigma_{i=0}^{2} \frac{1}{\sqrt{2}} |\overline{i}\rangle * x[\mathsf{S}\ j..n] \mapsto |\overline{0}\rangle * \mathsf{S}\ j < n\}$

6 $\quad x[\mathsf{S}\ j] \leftarrow x[\mathsf{S}\ j] + 1;$

7 $\{x[0..n] \mapsto \Sigma_{i=0}^{2} \frac{1}{\sqrt{2}} |\overline{i}\rangle * x[n..n] \mapsto |\overline{0}\rangle\}$

8 $\{x[0..n] \mapsto \Sigma_{i=0}^{2} \frac{1}{\sqrt{2}} |\overline{i}\rangle\}$

(b) QAFNY Program and Proof

Fig. 1. Example quantum program: GHZ state preparation. $\mathsf{S}\ j = j + 1$. $\overline{i}$ refers to m number of $i \in [0, 1]$ bits, where $m = |\kappa|$, if $\kappa \mapsto \Sigma_{i=0}^{2} |\overline{i}\rangle$ and $|\kappa|$ is the length of κ.

programming languages are designed to follow this model. 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 1a, which constructs a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [Greenberger et al. 1989], 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 1a 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 semantics of doing so is expressed by multiplying the state vector by the gate's corresponding matrix representation; single-qubit gates are 2-by-2 matrices, and two-qubit gates are 4-by-4 matrices. A gate's matrix must be *unitary*, ensuring that it preserves the unitarity invariant of quantum states' amplitudes. An entire circuit can be expressed as a matrix by composing its constituent gates.

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.

1.2 Quantum Algorithms and Programs and Proofs in QNP

QNP has two levels of advantanges: the programming language and the automated proof system levels. The QNP programming language, QAFNY , permits the operation functionality based quantum programs that can be compiled to quantum circuits via the QAFNY to SQIR compiler. As a contrast, most quantum programming languages are either built by meta-programs embedded in a host language, such as Python (for Qiskit [Cross 2018], Cirq [Google Quantum AI 2019], PyQuil [Rigetti Computing 2021], and others), Haskell (for Quipper [Green et al. 2013]), or Coq (for SQIR and VOQC [Hietala et al. 2021b]), or contain some high level opertions with the mix of some circuit gates without a compiler, like [Bichsel et al. 2020] and [Microsoft 2017]. In QAFNY , we think about program opertions as their functionality such as preparing superposition states by using H and QFT gates. Figure 1b Line 2 applies a H gate on $x[0]$, the 0-th position of the array x, but the gate application can also be applied on the whole array, to prepare superposition of n-qubits, as in Figure 2. Line 4-6 applies $n-1$ `CNOT` gates as shown in Figure 1a. However, in Qafny , we think of such behavior as an application of $n-1$ loop steps of quantum conditionals. In the j-th step, assume that we have a possibly entangled quantum state $\Sigma_{i=0}^{m} z_i \left|c_i\right\rangle$; for every basis state $\left|c_i\right\rangle$, we look at the j-th and $(\mathsf{S}\ j)$-th position as $c_i[j]$ and $c_i[\mathsf{S}\ j]$, if $c_i[j]=0$, we do nothing, if $c_i[j]=1$, we apply the loop body $x[\mathsf{S}\ j] \leftarrow x[\mathsf{S}\ j]+1$ to $c_i[\mathsf{S}\ j]$ [1]. We believe that this viewpoint can save quantum programmers' effort in writing quantum algorithms.

The second level is the proof system. While most quantum proof systems, such as QHL [Liu et al. 2019], QBricks [Chareton et al. 2021], QSL/BI [Zhou et al. 2021], and QSL [Le et al. 2022], built proof systems based on quantum computation theories, QNP tries to connect the Qafny proof system to traditional Hoare/Separation logic systems; 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 3.1. The way is to think of Qafny quantum operations as classical array aggregate operations. Indeed, quantum gate applications are linear in the sense that the whole state effect can be viewed as a synthesized effect of applications on individual basis states. For example, applying a `X` gate on the first qubit in a Bell pair $\frac{1}{\sqrt{2}}(\left|00\right\rangle+\left|11\right\rangle)$ is equal to $\frac{1}{\sqrt{2}}(\mathsf{X}\left|00\right\rangle+\mathsf{X}\left|11\right\rangle)$, the sum of applying the gate on the second elment of the individual basis states.

If we analogize a quantum state $\Sigma_{j=0}^{m} z_j \left|c_j\right\rangle$ as an m-length array and each basis state $z_j\left|c_j\right\rangle$ as the j-th element in the array. Figure 1b line 1, we have a singleton array $x[0..n]$, each element of which is n-length. In QNP, we named the name structure referring to an array representing a quantum state as a *session*. Here $x[0..n]$ is a session indicating that the location of the array is in a region named x, and the range is from 0 to n, which means that the length of each element is n. After the application of $x[0] \leftarrow \mathsf{H}$, we split the session into two parts: $x[0..1]$ and $x[1..n]$ having a `Had` type ($\frac{1}{\sqrt{2}}(\left|0\right\rangle+\left|1\right\rangle)$) and `Nor` ($\left|\overline{0}\right\rangle$) type states, respectively. Here, the singleton element in $x[1..n]$ is now $n-1$ length. State $\frac{1}{\sqrt{2}}(\left|0\right\rangle+\left|1\right\rangle)$ can also be viewed as $\Sigma_{j=0}^{2}\frac{1}{\sqrt{2}}\left|\overline{i}\right\rangle$ with $i\in[0,1]$. Here, $\overline{i}$ is a singleton bitstring. In each step of the loop in line 4-6, we cut off the j-th bit in the singleton element array, so the array session becomes $x[\mathsf{S}\ j..n]$, then we put the bit in the original $x[0..1]$ session which is now structured as $x[0..j]$. The state becomes $\Sigma_{j=0}^{2}\frac{1}{\sqrt{2}}\left|\overline{i}\right\rangle\left|0\right\rangle$. Notice that the quantum conditional in each loop step applies on the j-th and $(\mathsf{S}\ j)$-th bits of the two basis states ($\overline{0}$ and $\overline{1}$). When $i=0$, we do nothing; if $i=1$, we flip the bit, which means that we turn the state to be $\Sigma_{j=0}^{2}\frac{1}{\sqrt{2}}\left|\overline{i}\right\rangle$, with the $\overline{i}$ length increases by one and it is the same as the length the session $x[0..\mathsf{S}\ j]$. In the end, we can remove the $x[n..n]\mapsto\left|\overline{0}\right\rangle$ in the final state, because session $x[n..n]$ means that each element in the state array is 0, so we claim such session as empty.

One benefit of QNP is that we can achieve the level of automation that was never seen previously. The Figure 1b verification in our Qafny implementation in Dafny has exactly eight lines of code. In fact, programmers only need to input the marked black parts in Figure 1b, while the marked teal parts are automatically inferred by Qafny . To make such quantum to classical agreegate operation mapping happen, in verifying a program in Qafny , we utilize a type system to track the sessions in the program, as well as pre- and post-conditions in Section 2.

2 QAFNY: A HIGH-LEVEL QUANTUM LANGUAGE ADMITTED A PROOF SYSTEM

We designed Qafny , the core language of QNP, to be able to express quantum programs in terms of 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, clusters of possibly entangled

[1] The addition $x[\mathsf{S}\ j]+1$ is a $|x[\mathsf{S}\ j]|$ bit addition. Here, it is single bit one.

$$\{A(x) * A(y)\} \;\text{where}\; A(\beta) = \beta[0..n] \mapsto |\overline{0}\rangle \qquad \{x[0..n] : \mathsf{Nor}, y[0..n] : \mathsf{Nor}\}$$

$$x \leftarrow \mathsf{H};$$

$$\{x[0..n] \mapsto \|\mathsf{H}\|(|\overline{0}\rangle) * A(y)\} \qquad \{x[0..n] : \mathsf{Had}, y[0..n] : \mathsf{Nor}\}$$

$$\Rightarrow \{x[0..n] \mapsto C * A(y)\} \;\text{where}\; C = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + |1\rangle) \qquad \{x[0..n] : \mathsf{Had}, y[0..n] : \mathsf{Nor}\}$$

$$y \leftarrow y{+}1;$$

$$\{x[0..n] \mapsto C * y[0..n] \mapsto \|y{+}1\|(|\overline{0}\rangle)\} \qquad \{x[0..n] : \mathsf{Had}, y[0..n] : \mathsf{Nor}\}$$

$$\Rightarrow \{x[0..n] \mapsto C * y[0..n] \mapsto |\overline{0}.1\rangle\} \qquad \{x[0..n] : \mathsf{Had}, y[0..n] : \mathsf{Nor}\}$$

$$\Rightarrow \{E(0)\} \;\texttt{where}\; E(k) = \qquad \{x[0..n] : \mathsf{Had}, \{x[0..0], y[0..n]\} : \mathsf{CH}\}$$
$$x[k..n] \mapsto \frac{1}{\sqrt{2^{n-k}}} \bigotimes_{j=0}^{n-k} (|0\rangle + |1\rangle) *$$
$$\{x[0..k], y[0..n]\} \mapsto \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}} |j\rangle\, a^j \,\%\, N$$

$$\texttt{for (int } j := 0;\; j < n \;\&\&\; x[j] \;;\; {+}{+}j\texttt{)}$$

$$\{E(j)\} \qquad \{x[j..n] : \mathsf{Had}, \{x[0..j], y[0..n]\} : \mathsf{CH}\}$$

$$y \leftarrow a^{2^j} y \,\%\, N;$$

$$\{E(n)\} \qquad \{x[0..0] : \mathsf{Had}, \{x[0..n], y[0..n]\} : \mathsf{CH}\}$$

$$\Rightarrow \{\{x[0..n], y[0..n]\} \mapsto \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} |j\rangle\, a^j \,\%\, N\} \qquad \{\{x[0..n], y[0..n]\} : \mathsf{CH}\}$$

$$\texttt{let } u \;=\; \texttt{measure}(y) \texttt{ in} \ldots$$

$$\left\{ \begin{array}{l} x[0..n] \mapsto \frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} |t + kp\rangle \wedge p = \mathsf{ord}(a, N) \\ \wedge\, u = (\frac{p}{2^n}, a^t \,\%\, N) \wedge s = \mathsf{rnd}(\frac{2^n}{p}) \end{array} \right\} \qquad \{\{x[0..n]\} : \mathsf{CH}\}$$

Fig. 2. Pre-measurement quantum steps of the Shor's algorithm. $\mathsf{ord}(a, N)$ gets the order of a and N. $\mathsf{rnd}(r)$ rounds r to the nearest integer. The right-hand-side contains the types for the sessions involved. $|j\rangle$ is an abbreviation of $|(\!|j|\!)\rangle$. $(\!|j|\!)$ turns a number j to a bitstring. $\overline{0}.1$ is a bitstring concatenation operation.

qubits and the state unit in Qafny programming, with three types indicating the qubit clusters' state representations. The Qafny proof system is designed to capture the quantum to classical array aggregate operation analogies by utilizing the type system to ensure the session formats in programs and predicates. 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 [Shor 1994] shown in Figure 2. 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 $k = \mathsf{gcd}(a, N)$ [2]; (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 [3]; (4) if p is even and $a^{\frac{p}{2}} \neq -1\%N$, $\mathsf{gcd}(a^{\frac{p}{2}} \pm 1, N)$ are the factors, otherwise, we repeat the process. Step (2) is

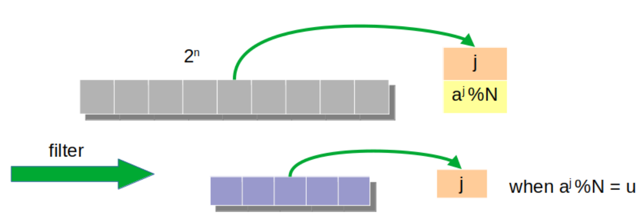


Fig. 3. The array analogy of Shor's first half in Figure 2.

[2] compute the greatest common divisor of a and N

[3] the order p is the smallest number such that $a^p \% N = 1$

Basic Terms:

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

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

Mode	g	$::=$	$\mathtt{C} \mid \mathtt{M}$		
Classical Value	v	$::=$	$n \mid (r, n)$		
Kind	$\overline{g}$	$::=$	$g \mid \mathtt{Q}\ n$		
Basis	β	$::=$	$(\lvert c \rangle)^+$		
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	l	$::=$	$x[n..m]$		
Session	κ	$::=$	$\overline{l}$	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. $(\lvert c \rangle)^+$ is a non-empty list of bases $\lvert c_i \rangle$, and refers to $\lvert c_0 \rangle \otimes ... \otimes \lvert c_n \rangle$.

the quantum part of Shor's algorithm and Figure 2 and Figure 33 show its automated proof in QNP. In Figure 15, we show the actual implementation and proof in the Qafny tool.

The Shor's pre-measurement quantum steps in Figure 2 can be analogized as an efficient array filter operation. 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 $(j, a^j \% N)$ where $j \in [0, 2^n)$. The measurement in line 14 filters the array as a new one ($x[0..n]$) with all elements j satisfying $a^j \% N = u$ where u is a randomly picked number. Notice that modulo multiplication $f(j) = a^j \% N$ is a periodic function. All elements in $x[0..n]$ satisfy $a^j \% N = u$, which means that 1) there is a smallest t such that $a^t \% N = u$, and 2) all elements can be rewritten as $j = t + kp$ and p is the period of the modulo multiplication function, which is given as the post-condition on the right of line 15. The implementation and correctness proof in Figure 2 exactly reflects the array analogy aspect. Notice that only the marked black and purple parts in Figure 2 are required to input in the Qafny implementation, and the marked teal parts can be inferred by the Qafny proof system. [4]

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

2.1 Classical and Quantum States

Qafny has three *kinds* of parameters in Figure 4: a C-kind classical integer parameter [5], a M-kind classical integer parameter (r, n) with a probability characteristic r representing the theoretical probability of the measurement resulting in the natural number value n, and a $\mathtt{Q}\ n$ kind quantum parameter, where n represents the number of bits in a qubit array piece. Quantum parameters are classified as three types: Nor, Had, or CH, representing the three types of quantum values in

[4]The purple is also not needed if we verify the whole Shor's algorithm in Figure 15.

[5]In the Qafny implementation one can utilize any classical typed parameters allowed in Dafny . For simplicity, we only allow integers in this paper.

Section 1. We have subtyping relations over quantum types, such that Nor and Had are subtypes of CH, representing the fact that Nor and Had quantum values can be rewritten as CH-forms.

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 variable 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 array piece variables in this paper, i.e., two distinct variables represent disjoint array pieces. For example, $\{\{x[0..n], y[0..n]\}$ in Figure 2 line 12 represents a $2n$ qubit array containing two disjoint pieces $x[0..n]$ and $y[0..n]$ referring to the ranges $[0, n)$ in groups x and y, respectively. We also abbreviate a singleton session $\{x[n..m]\}$ as a range $x[n..m]$. In Qafny quantum type environments and states, qubit values are always associated with sessions, where a length-n session is associated with a Nor-type state $z\beta$, Had-type state $\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$, or CH-type state $\Sigma_{j=0}^{m} z_j \beta_j$. In the Nor-type or Nor-type state, the lengths $|\beta|$ (or $|\beta_j|$) of every basis β (or $|\beta_j|$) [6] are the same and $|\beta| \geq n$ (or $|\beta_j| \geq n$); in Had-type state, the session length is the same as the qubit array length n. Qafny type environments and states are finite, and the domain sessions do not overlap, i.e., for all $\kappa, \kappa' \in \mathrm{dom}(\sigma)$ (or φ), $\kappa \neq \kappa' \implies \kappa \cap \kappa' = \emptyset$.

Qafny utilize *equivalence relations* over quantum sessions, quantum values and states (as shown in Figure 34) to facilitate automated program verification, written as $\equiv$ for state equivalence and $\leq$ for environment partial order. One example is the rewrite from line 12 to line 13 in Figure 4, where the state of session $x[0..0]$ is rewritten to `true`, because the session is essentially empty. The common equivalence relations are state form rewrites, permutations, split and joins. State rewrites are to 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 4. 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 $\{y[0..n], x[0..n]\} \mapsto \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}}\, |a^j \,\%\, N\rangle\, j$ [7]. State joins merges two sessions together. Merging a Nor-type and CH-type state is analogized to add the Nor-type state's basis string to every basis states in the CH-type one. An example and its explanation are given in Figure 1 line 4-6 and Section 1.2, where the Nor-type qubit $x[j]$ is merge to CH-type session $x[0..j]$. 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 $\{x[0..j], y[0..n]\}$, and the state becomes $\Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}}\, |(\!|j|\!).0\rangle\, a^j \,\%\, N + \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}}\, |(\!|j|\!).1\rangle\, a^j \,\%\, N$ [8]. Merging two CH-type states computes the Cartesian product of basis states in the two groups (Figure 34). 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 of a CH-type is equal to disentanglement, a very 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 invented an upgraded type system in Appendix C to permit few cases, while the normal Qafny type system in Section 2.2 does not permit such behavior.

2.2 Syntax and Type Sysem

One of the key Qafny design principles is to allow programmers think of quantum programs as sequences of functional operations that are analogized to array aggregate operations, instead of dealing with quantum circuit gates in many other languages. Figure 5 shows the Qafny syntax. A program consists of a sequence of C-like statements s that end at a SKIP operation $\{\}$. The let operation (`let` x `=` e `in` s) in the first row introduces a new variable x with its initial value defined

[6]The length of β is defined as the sum of all length of basis bitstrings in β.

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

[8]$|j\rangle$ is an abbreviation of $|(\!|j|\!)\rangle$. $(\!|j|\!)$ turns a number j to a bitstring. $c_1.c_2$ is a bitstring concatenation.

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

Fig. 5. Core Qafny syntax. ⓄQASM is in Section 2. 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 array piece $x[0..n]$ and n is the length of x.

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

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

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

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

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

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

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

Fig. 6. Qafny type system. $\sigma(y) = \{\kappa \mapsto \tau\}$ produces the map entry $\kappa \mapsto \tau$ and the range $y[0..|y|]$ is in κ. $\sigma(\kappa) = \tau$ is an abbreviation of $\sigma(\kappa) = \{\kappa \mapsto \tau\}$. $FV(\Omega, -)$ produces a session by union all qubits appearing in $-$ with the qubit piece info in Ω; see Appendix B.1.

e and used in s. If e is an arithmetic expression (a), it introduces a C or M kind classical variable. [9] $\mathtt{let}\ x\ =\ \mathtt{measure}(y)\ \mathtt{in}\ s$ measures qubit group y, stores the result in M-kind variable x, and is used in s. The last three operations in first row are the quantum data-flow operations. $l \leftarrow op$ prepares a quantum superposition state of quantum qubits l through Hadamard gates H or QFT gates. It is also used to Fourier transform quantum qubit states by a $\mathtt{QFT}^{-1}$ gate in the end of the quantum phase estimation algorithm. We only permit op to be state preparation gates such as H and $\mathtt{QFT}^{[-1]}$ gates. The other gate applications are done through $\kappa \leftarrow \mu$ that performs ⓄQASM quantum oracle computation μ ([Li et al. 2022]) on each basis state of session κ's state. Almost all quantum reversible arithmetic operations are defined in ⓄQIMP, an C-like oracle language based on ⓄQASM; hence, we permit μ's description in Qafny to be arithmetic operations as the expression a

[9]For simplicity, we assume that M-kind arithmetic operations manipulates the nat number parts, so that $(r, n_1) + n_2 = (r, n_1 + n_2)$; and we only interacts a M kind with a C one in an arithmetic operation, i.e., the $(r_1, n_1) + (r_2, n_2)$ is disallowed in Qafny .

in Figure 5, such as fig. 2 line 5 and 11. $l \leftarrow \texttt{dis}$ is a quantum diffusion operation applying on the parameter l, where l may be part of a 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 $\mathbb{O}$QASM oracle operations, and 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 $\texttt{true}$ [10], and store the value in the qubit bit $x[a]$ as $b \oplus x[a]$. $\texttt{for}\ (\texttt{int}\ j \in [a_1, a_2)\ \&\&\ b)\ s$ is a possibly quantum for-loop depending on Boolean guard b. A classical variable j is introduced and it is initialized as the lower bound a_1, increments in each loop step by $++j$, and ends at the upper bound a_2. For example, line 9-11 in Figure 2 uses a for-loop to repeatably entangle the $\texttt{Had}$-type qubit $x[j]$ with the $\texttt{CH}$-type session $\{x[0..j], y[0..n]\}$ by the modulo multiplication at line 11. [11]

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' ($\texttt{dom}(\sigma')$), 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 QAFNY type system enforces variable scope; such enforcement is neglected in Figure 30 for simplicity. [12] We assume that variables introduced in $\texttt{let}$ expressions are all distinct through proper alpha conversions, as the cases in TEXP and TMEA. $\texttt{dom}(\sigma)$ is large enough to describe all sessions pointed to by quantum variables in s, while $\texttt{dom}(\sigma')$ should contain the exact sessions describing quantum qubits in s. Selected type rules are given in Figure 30; the rules not mentioned are similar and listed in Appendix B. g reused as context modes ($\texttt{C}$ and $\texttt{M}$) for enforcing no quantum information leak in a quantum conditional.

The type system enforces four invariants. First, we place well-formed and context restrictions for 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 the quantum conditional body 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 $\texttt{C}$ permits most QAFNY operations. Once a type rule turns a mode to $\texttt{M}$, as in TIF, we disallow $\texttt{measure}$ operations, such that rules TMEA is valid only if the input context mode is $\texttt{C}$. 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 sessions appearing in μ might be 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 change the session structure. For example, in Figure 1 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 the $\texttt{C}$ classical variables can be evaluated to values in the compilation time [13], while tracks

---

[10] a_1 and a_2 are possibly quantum array piece variable x whose state contains basis states.
[11] In 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.
[12] In the QAFNY implementation, we have an $\texttt{init}\ n$ operation to allocate n-number of $|0\rangle$ qubits, which create $\texttt{Q}\ n$ variable in Ω; here, we assume that $\texttt{Q}\ n$ array piece variables are pre-allocated in Ω.
[13] We consider all computation that only needs classical computer is done in the compilation time.

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

$$\text{SSEQ-1}\ \frac{(\psi, \varphi, s_1) \longrightarrow (\psi', \varphi', s_1')}{(\psi, \varphi, s_1 \ ;\ s_2) \longrightarrow (\psi', \varphi', s_1' \ ;\ s_2)} \qquad \text{SSEQ-2}\ (\psi, \varphi, \{\} \ ;\ s_2) \longrightarrow (\psi, \varphi, s_2) \qquad \text{PSEQ}\ \frac{\Omega; \sigma \vdash_g s_1 \triangleright \sigma' \quad \Omega; \sigma \vdash_g \{P\}\, s_1\, \{R\} \quad \Omega; \sigma[\uparrow \sigma'] \vdash_g \{R\}\, s_2\, \{Q\}}{\Omega; \sigma \vdash_g \{P\}\, s_1 \ ;\ s_2\, \{Q\}}$$

$$\text{PCONL}\ \frac{(\Omega, \sigma, P) \Rightarrow (\Omega, \sigma', P') \quad \Omega; \sigma' \vdash_g \{P'\}\, s\, \{Q\}}{\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}} \qquad \text{PCONR}\ \frac{\Omega, \sigma \vdash_g s_1 \triangleright \sigma' \quad \Omega; \sigma' \vdash_g \{P\}\, s\, \{Q'\} \quad (\Omega, \sigma'', Q') \nRightarrow (\Omega, \sigma[\uparrow \sigma'], Q)}{\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}}$$

$$(\Omega, \sigma, P) \Rightarrow (\Omega, \sigma', P') \triangleq \Omega, \sigma \vdash P \wedge \Omega, \sigma' \vdash P' \wedge \sigma \leq \sigma' \wedge P \Rightarrow P'$$
$$(\Omega, \sigma, Q) \nRightarrow (\Omega, \sigma', Q') \triangleq \Omega, \sigma \vdash Q \wedge \Omega, \sigma' \vdash Q' \wedge \sigma' \leq \sigma \wedge Q \Rightarrow Q'$$

Fig. 7. Sequence and Consequence Rules

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, where classical expressions containing x are evaluated. See Appendix B. Finally, in rule TLoop, the type system automatically infers the result type $\sigma'[n_2/x]$ based on the type environment invariant for a single loop step that executes the conditional $\texttt{if}\ (b[j/x])\ s[j/x]$.

2.3 The Qafny Semantics and Proof System

QNP intends to create a proof system that utilizes classical automated reasoning infrastructure in analyzing quantum programs. As we introduced above, quantum operations can be analogized to array aggregate operations. The prototype of designing the QNP proof system is the classical Separation Logic [Reynolds 2002] without stack pointer variables since we assume no aliasing. To materialize such analogy, Qafny sessions need to play two roles, both as variable names representing quantum arrays and as qubit array structure indicators, which means that we enforce well-formedness and types on states and proof system predicates.

Definition 2.1 (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 $\mathsf{dom}(\varphi)$):

- 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 $x[i..j] \in \kappa$, $\Omega(x) = \mathsf{Q}\ n$ and $[i, j) \subseteq [0, n)$.

Definition 2.2 (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 $P = \kappa \mapsto q * P'$, $\kappa \in \mathsf{dom}(\sigma)$.

Qafny semantics is a small-step transition relation $(\psi, \varphi, s) \longrightarrow (\psi', \varphi', s')$, 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. Basically, any QNP proof judgment has a hidden type restriction as follows:

$$\Omega; \sigma \vdash_g s \triangleright \sigma' \wedge \Omega; \sigma \vdash P \wedge \Omega; \sigma[\uparrow \sigma'] \vdash Q$$

The restriction is required not only at the bottom proof tree, but also at all levels of the proof tree. For example, rule PSeq in Figure 35 is no more than a sequence rule but having additional type restrictions, such that every step transition requires a computation of a new type environment $\sigma[\uparrow \sigma']$. Similarly, the Frame rule is no more than a frame rule in a separation logic [14], 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 above with respect to Ω and σ.

The rule besides rule Frame in Figure 35 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 similar type restrictions as: $\mathtt{dom}(\psi) \subseteq \Omega$ and $\Omega; \sigma \vdash_g \varphi$, such that $\Omega; \sigma \vdash_g \varphi$ is a well-formed state restriction defined as follows:

Definition 2.3 (Well-formed Qafny state). A state φ is *well-formed*, written $\Omega; \sigma \vdash_g \varphi$, iff $\mathtt{dom}(\sigma) = \mathtt{dom}(\varphi)$, $\Omega \vdash \mathtt{dom}(\sigma)$, and:

- For every $\kappa \in \sigma$ such that $\sigma(\kappa) = \mathtt{Nor}$, $\varphi(\kappa) = z\,|c\rangle$ [15] and $|\kappa| \leq |c|$ and $|z|^2 \leq 1$ [16]; specifically, if $g = \mathtt{C}$, $|\kappa| = |c|$ and $|z|^2 = 1$.
- For every $\kappa \in \sigma$ such that $\sigma(\kappa) = \mathtt{Had}$, $\varphi(\kappa) = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$ and $|\kappa| = n$.
- For every $\kappa \in \sigma$ such that $\sigma(\kappa) = \mathtt{CH}$, $\varphi(\kappa) = \sum_{j=0}^{m} z_j\,|c_j\rangle$ 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 $g = \mathtt{C}$, $|\kappa| = |c_j|$ and $\sum_{j=0}^{m} |z|^2 = 1$.

The reason that we place a relaxed well-formed state definition for a $\mathtt{M}$-mode state is because such state lives inside a quantum conditional where parts of the sessions and states are hidden, whereas once the quantum conditional finishes execution and the program counter is transitioned to the top-most location, which is in $\mathtt{C}$-mode, the hidden parts are added back to the state and the united state is required to satisfy the principles of quantum states, which are described as the $\mathtt{C}$-mode restrictions in Definition A.1. See Section 2.3.

$$\frac{\forall j.\ |\kappa| = |c_j|}{\Omega; \sigma; \psi; \varphi[\kappa \mapsto \sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta'_j] \models_g \kappa \mapsto \sum_{j=0}^{m} z_j\,|c_j\rangle\,\beta_j}$$

Other than the type restrictions, a modeling relation in QNP is similar to the ones appearing in a separation logic exact the rule for the session mapping as above, where the session state in φ and in the predicate are the same except that the tail locations (β'_j and β_j) can be different. Rules PConL and PConR are the consequence rules, where the implications also have well-formed restrictions as all other entities above. We define a special $\Rightarrow$ and $\not\Rightarrow$ implications for rules PConL and PConR, respectively. We now discuss few interesting cases.

State Preparation and Oracle Application Rules. The Qafny state preparation $l \leftarrow op$ and oracle application $\kappa \leftarrow \mu$ operations are analogized to classical array map operation as discussed in Section 1. We have a session $\kappa \uplus \kappa'$, for each element $z_j\,|c_j\rangle\,\beta_j$ in the $\mathtt{CH}$ type state, we first find c_j as the corresponding basis state for the κ positions because they have the same length and c_j is the prefix basis state, then we apply μ on the c_j part, which is described in the semantic rule SA-CH. Rule PA-CH describes the proof rule for capturing the array map analogy, which describes the exact behavior as the semantic rule. For example, in the loop body in Figure 2 line

[14]We use $FV(s) \cap FV(R) = \emptyset$ here because $FV(s)$ is the exact session set where s modifies in Qafny .
[15]every β can be viewed as $|c_0\rangle \otimes ... \otimes |c_n\rangle$; thus, it is equal to $|c_0.c_1. ... .c_n\rangle$, where $c_i.c_{i+1}$ is bitstring concatenation.
[16]$|z|^2$ is the norm of z.

SA-CH

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

PA-CH

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

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

Fig. 8. Oracle application and state preparation rules

8, we apply $y \leftarrow a^{2^j} y \% N$ to a state $y[0..n] \mapsto \Sigma_{j=0}^{2^{k-1}} \frac{1}{\sqrt{2^{k-1}}} |(\!| a^{(\!|j|\!)} \% N |\!) \mid (\!| j |\!).1\rangle$ [17]. The result is $\Sigma_{j=0}^{2^{k-1}} \frac{1}{\sqrt{2^{k-1}}} |(\!| a^{(\!|j|\!).1} \% N |\!) \mid (\!| j |\!).1\rangle$ [18]. The other similar rules, such as state preparation rules, can be found in Appendix B.

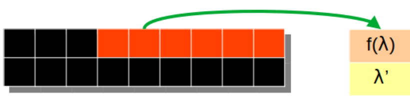
Rules for Conditionals and For-Loops. Figure 9a describes the analogy of quantum conditionals in Qafny , which are partial map functions that only apply applications on the marked red parts and mask the marked black parts. It contains two level of masking. For each basis state element in a state with session $\kappa_b \uplus \kappa_a$, it masks the κ_b part of the state, i.e., if the state has the form: $\Sigma_{j=0}^{m} z_j \, |c_{j1}.c_{j2}\beta_j\rangle$ with $|c_{j1}| = |\kappa_b|$, we mask the c_{j1} part by pushing it to a masked stack as $\Sigma_{j=0}^{m} z_j \, |c_{j2} \mid c_{j1} \, \beta_j\rangle$, which is described in preparing the pre-state of the upper-level transition in rule SIF (Figure 9c). The second level of masking happens in selecting elements in the state by checking the Boolean condition b on them. The R condition in ruleSIF finishes such task for us. Here, we split the state into two parts as $\Sigma_{j=0}^{m} z_j \, |c_{j1}.c_{j2} \, \beta_j\rangle + q(\kappa, \neg b)$, where the first part are all basis states satisfying b since if we replace the variables in b with c_{j1}, the evaluation $\llbracket b[c_{j1}/\kappa] \rrbracket$ returns true, while the second part $q(\kappa, \neg b)$ contains all basis states evaluated to false. After the masking, we apply the application s on each element in the selected states, install the results (z'_j and c'_{j2} back to the unmasked state, and result in $\Sigma_{j=0}^{m'} z'_j \, |c_{j1}.c'_{j2} \, \beta_j\rangle + q(\kappa, \neg b)$. The result state number m' might be different from the pre-state number 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 mask ($\mathcal{M}$) and unmask ($\mathcal{U}$) operations (Figure 36b), both take a Boolean expression b and a quantum state as the argument. $\mathcal{M}$'s modeling materializes the masking mechanism above. For a state $\Sigma_{j=0}^{m} z_j \, |c_{j1}.c_{j2} \, \beta_j\rangle + q(\kappa, \neg b)$, we preserve the basis states satisfying b, as shown in the predicate R, remove the unsatisfied basis states $q(\kappa, \neg b)$, and push the bases c_{j1} to state stacks. Here, we do not need to input $\mathcal{M}$ the session associated with c_{j1}, because we can learn it through b. In the pre-condition manipulation of rule PIF (Figure 9c), we substitute $\kappa \uplus \kappa'$ with $\mathcal{M}(b, \kappa')$. During the process, the type for the session in σ is changed from $\kappa \uplus \kappa'$ in the bottom to κ' in the upper level. The unmask function $\mathcal{U}$ assembles the result state of applying s to the masked κ state with the other parts hidden in the unmasked state (the part marked black in Figure 9a). Function $\mathcal{U}$ is usually appeared to be a pair with both b and $\neg b$. In the post-condition manipulation, we substitute $\kappa \uplus \kappa'$ with $\mathcal{U}(\neg b, \kappa \uplus \kappa')$ in P, representing the unchanged and masked state, substitute κ' with $\mathcal{U}(b, \kappa \uplus \kappa')$ in Q', representing the result of

[17]The state is a equivalence state $(\{x[0..k], y[0..n]\} \mapsto \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}} \, |(\!| j |\!).(\!| a^j \% N |\!)\rangle)$ of the masking session $x[0..j+1]$.

[18]We take the bitstring exponent formula as: $a^c = a^{\Sigma_j^{2^{c[j]}}}$.

(a) Conditional Analogy



(b) Mask/Unmask Function Modeling

$$\frac{R \qquad \Omega;\sigma;\varphi \models \theta \mapsto \sum_{j=0}^{m} z_j \,|c_{j2}\rangle |c_{j1}\,\beta_j\rangle}{\Omega;\sigma;\varphi \models \mathcal{M}(b,\theta) \mapsto \sum_{j=0}^{m} z_j \,|c_{j1}.c_{j2}\,\beta_j\rangle + q(\kappa,\neg b)}$$

$$\frac{R \qquad \Omega;\sigma,\varphi \models \theta \mapsto \sum_{j=0}^{m'} z'_j \,|c_{j1}.c'_{j2}\,\beta_j\rangle + q(\kappa,\neg b)}{\begin{array}{c}\Omega;\sigma,\varphi \models \mathcal{U}(\neg b,\theta) \mapsto \sum_{j=0}^{m} z_j \,|c_{j1}.c_{j2}\,\beta_j\rangle + q(\kappa,\neg b)\\ *\,\mathcal{U}(b,\theta) \mapsto \sum_{j=0}^{m'} z'_j \,|c'_{j2}\rangle |c_{j1}\,\beta_j\rangle\end{array}}$$

(c) Semantic/Proof Rules

SIF

$$\frac{R \qquad FV(\emptyset,s)\subseteq\kappa' \qquad (\varphi[\kappa' \mapsto \sum_{j=0}^{m} z_j\,|c_{j2}\rangle|c_{j1}\,\beta_j\rangle],s) \longrightarrow (\varphi[\kappa' \mapsto \sum_{j=0}^{m'} z'_j\,|c'_{j2}\rangle|c_{j1}\,\beta_j\rangle],s')}{(\varphi[\kappa\uplus\kappa' \mapsto \sum_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\,\beta_j\rangle + q(\kappa,\neg b)],\texttt{if }(b)\ s) \longrightarrow (\varphi[\kappa\uplus\kappa' \mapsto \sum_{j=0}^{m'} z'_j\,|c_{j1}.c'_{j2}\,\beta_j\rangle + q(\kappa,\neg b)],\texttt{if }(b)\ s')}$$

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

$$\frac{n_1 < n_2}{\begin{array}{l}(\varphi,\texttt{for (int } j\in[n_1,n_2)\ \texttt{\&\&}\ b)\ s) \longrightarrow\\ \quad(\varphi,\texttt{if }(b)\ s\ ;\ \texttt{for (int } j\in[\mathsf{S}\,n_1,n_2)\ \texttt{\&\&}\ b)\ s)\end{array}}$$

PLOOP

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

$$\theta := q \mid \kappa$$
$$q(\kappa,\neg b) = \Sigma_{k=0}^{m} z_k\,|c_{k1}.c_{k2}\,\beta_k\rangle \ \text{ where } \forall k.\ |c_{k1}| = |\kappa| \wedge [\![\neg b[c_{k1}/\kappa]]\!]$$
$$R = FV(\emptyset,b) = \kappa \wedge \forall j.\ |c_{j1}| = |\kappa| \wedge [\![b[c_{j1}/\kappa]]\!]$$

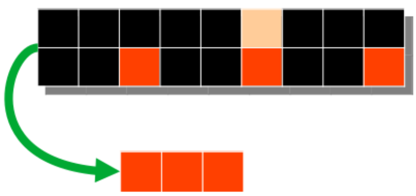
Fig. 9. Semantic and Proof Rules for Conditionals and For-loops. θ is either a session or a quantum state. $\mathcal{M}$ is the mask function and $\mathcal{U}$ is the unmask function.

applying s on the unmasked part, and assemble them together through the separation operation $*$. $\mathcal{U}$'s modeling in Figure 36b captures the assemble procedure by merging two $\mathcal{U}$ constructs together.

Rules SLOOP and PLOOP are the semantic and proof rules for a for-loop, which is a repeat operation of conditionals in QAFNY . The $P(j)$ is a loop invariant with j being a variable. The case for $n_1 \geq n_2$ is given in Appendix C. As an example, we show the proof for a loop-step in Figure 2.

$$\frac{\dfrac{\Omega;\sigma_2\vdash_{\mathtt{M}}\{X(\mathsf{S}\,j) * \{y[0..n]\}\mapsto C(j).1\}\ s\ \{X(\mathsf{S}\,j) * \{y[0..n]\}\mapsto C'(j).1\}}{\Omega;\sigma_2\vdash_{\mathtt{M}}\{X(\mathsf{S}\,j) * \mathcal{M}(b,\{y[0..n]\})\mapsto 0.C(j)+1.C(j)\}\ s\ \{X(\mathsf{S}\,j) * \{y[0..n]\}\mapsto C'(j).1\}}}{\dfrac{\begin{array}{l}\Omega,\sigma_1\vdash_{\mathtt{M}}\{X(\mathsf{S}\,j) * \{x[0..\mathsf{S}\,j],y[0..n]\}\mapsto 0.C(j)+1.C(j)\}\texttt{if }(x[j])\ s\\ \qquad\{X(\mathsf{S}\,j) * \mathcal{U}(\neg x[j],\{x[0..\mathsf{S}\,j],y[0..n]\})\mapsto 0.C(j) * \mathcal{U}(x[j],\{x[0..\mathsf{S}\,j],y[0..n]\})\mapsto C'(j).1\}\end{array}}{\Omega;\sigma\vdash_{\mathtt{M}}\{X(j) * \{x[0..j],y[0..n]\}\mapsto C(j)\}\ \texttt{if }(x[j])\ s\ \{X(j-1) * \{x[0..\mathsf{S}\,j],y[0..n]\}\mapsto 0.C(j)+1.C'(j)\}}}$$

(a) Measurement Analogy



(b) Measurement Modeling

$$\frac{\Omega;\sigma;\varphi \models \theta \mapsto \theta'}{\Omega;\sigma;\varphi \models \mathcal{F}(x,n,\theta) \mapsto \mathcal{F}(x,n,\theta')}$$

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

(c) Semantic/Proof Rules

SMea

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

PMea

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

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

Fig. 10. 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.

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

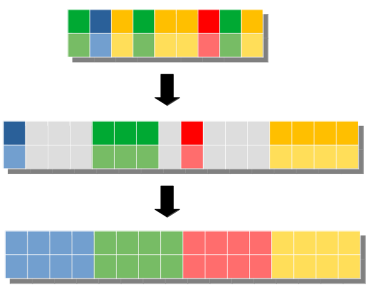
$$C'(j).i = \Sigma_{j=0}^{2^{\texttt{S}\,k}} \tfrac{1}{\sqrt{2^{\texttt{S}\,k}}} |(\!|a^{(\!|j|\!)^k.1} \ \%\ N|\!)\ |\ (\!|j|\!)^k.i\rangle \qquad i.C'(j) = \Sigma_{j=0}^{2^{\texttt{S}\,k}} \tfrac{1}{\sqrt{2^{\texttt{S}\,k}}} |(\!|j|\!)^k.i.(\!|a^{(\!|j|\!)^k.1} \ \%\ N|\!)\rangle$$

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

The proof is built from bottom up. We first cut the Had type state into two sessions ($x[0..n - \texttt{S}\ j]$ 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 36b, we combine the two $\mathcal{U}$ terms into the final result. The second step applies rule PIF to substitute session $\{x[0..\texttt{S}\ j], y[0..n]\}$ 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.

Rules for Measurement. As Figure 10a describes, quantum measurement is a two-step array filter: 1) each element in the state is partitioned into two parts, and we select a first part of an element as a key, as shown in the marked pink part; and 2) we create a new array state by removing all the first parts in the old state and collecting the elements whose first part is equal to the key. The second step actually collects elements in a periodical manner as shown in the analogy in Figure 10a, where the marked red basis states appear in a periodical pattern in the whole state. This behavior is universally true for quantum operations, and many quantum algorithms utilize the periodical pattern of quantum computation.

(a) Diffusion Analogy



(b) Diffusion Modeling

$$\frac{\Omega;\sigma;\varphi \models \theta \mapsto \theta'}{\Omega;\sigma;\varphi \models \mathcal{D}(n,\theta) \mapsto \mathcal{D}(n,\theta')} \qquad \frac{\Omega;\sigma;\varphi \models \theta \mapsto \mathcal{D}'(n, \sum_{j=0}^{n'} z_i \,|c_i\rangle)}{\Omega;\sigma;\varphi \models \mathcal{D}(n,\theta) \mapsto \sum_{i=0}^{n'} z_i \,|c_i\rangle}$$

$$\frac{|(\!|j|\!)| = n \qquad \Omega;\sigma;\varphi \models \theta \mapsto \sum_{j=0}^{m}\sum_{k=0}^{2^n}(2z_{jk}\sum_{t=0}^{2^n} z_{jt} - z_{jk})\,|(\!|k|\!).c_{jk}\rangle}{\Omega;\sigma;\varphi \models \theta \mapsto \mathcal{D}'(n, \sum_{j=0}^{m}\sum_{k=0}^{2^n} z_{jk}\,|(\!|k|\!).c_{jk}\rangle)}$$

(c) Semantic/Proof Rules

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

Fig. 11. Semantic and Proof Rules for Diffusion Operations. $\mathcal{D}$ models the diffusion operations.

In rule SMea, we pick an n-length bitstring c as the marked pink key, and elect m basis states $\sum_{j=0}^{m} z_j \,|c.c_j\rangle$ that has the key c. In the post-state, we update the remaining session κ to $\sum_{j=0}^{m} \frac{z_j}{\sqrt{r}} \,|c_j\rangle$ 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, the operation $\mathcal{F}(x,n,\kappa)$ is invented, whose modeling is in Figure 10b, to do exactly the two steps above by selecting an n-length prefix bitstring c in a basis state, computing the probability r, and assigning $(r, \{\!|c|\!\})$ to variable x. Rule PMea in Figure 10c replaces the 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.

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

$C = \sum_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} \,|(\!|j|\!).(\!|a^j \ \%\ N|\!)\rangle \quad C' = \sum_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} \,|(\!|a^j \ \%\ N|\!).(\!|j|\!)\rangle \quad \sigma = \{\{x[0..S\ j], y[0..n]\} : \texttt{CH}\}$

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

For an instance, we show a proof fragment above for the partial measurement in line 11 in Figure 2. The bottom two lines are to modify the array group order of the session from $\{y[0..n], x[0..n]\}$ to $\{x[0..n], y[0..n]\}$ through the consequence rule of state equivalence. The second line proof, applies rule PMea by replacing session $\{x[0..n], y[0..n]\}$ with $\mathcal{F}(u,n,x[0..n])$. On the top statement, the pre- and post-conditions are equivalent, because of the periodical aspects in quantum computing. In session $\{y[0..n], x[0..n]\}$, group $y[0..n]$ stores the basis state $(\!|a^j \ \%\ N|\!)$, which contains value j that represents the basis states for group $x[0..n]$. Selecting a basis state $a^t \ \%\ N$ also filters the j in $x[0..n]$, which refers to any values 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 $a^{t+kp} \ \%\ N = a^t \ \%\ N$, such that p is the period order. Thus, the $x[0..n]$ state is rewritten as a summation of k: $\frac{1}{\sqrt{s}} \sum_{k=0}^{s} |t+kp\rangle$. The probability of selecting $(\!|a^j \ \%\ N|\!)$ is $\frac{s}{2^n}$. In Qafny , we set up additional axioms for representing these periodical manners, which is why the pre- and post-condition equivalence is granted.

Rules for Diffusion. Quantum diffusion operations ($l \leftarrow \mathtt{dis}$) 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 many programming languages [19]. The aggregate operation first applies a reshape, where elements are regrouped into a normal form, as the first agree of Figure 11a. More specifically, the diffusion modeling function $\mathcal{D}(n, \theta)$ takes an n'-element state $\sum_{j=0}^{n'} z_j \,|c_j\rangle$ as the second rule in Figure 11b. The n number corresponds to the number of bits in the session potion matching l. Then, we rearrange the state by a helper function $\mathcal{D}'$ by extending the element number from n' to $m * n$ with probably adding new elements that originally have zero amplitude (the marked white elements in Figure 11a), as the third rule in Figure 11b. Here, let's view a basis c_i as a small-endian (LSB) number $\{\!|c_i|\!\}$. The rearrangement of changing bases c_i (for all i) to $(\!|k|\!).c_{jk}$ is analogized to rearranging a number $\{\!|c_i|\!\}$ to become the form $2^n j + k$, with $k \in [0, 2^n)$. Basically, the reshape step rearranges the basis states to be placed in a periodical counting sequence, with 2^n being the order.

The mean computation analogy (the second arrow in Figure 11a) takes every period in the reshaped state, and for each basis state, we redistribute the amplitude by the formula $(2z_{jk} \sum_{t=0}^{2^n} z_{jt} - z_{jk})$. In another word, for each period, given a basis state k, we sum all the amplitudes from 0 to 2^n in the period as z_{sum}, then the redistributed amplitude for k is $2z_k z_{sum} - z_k$.

Rule SDis is the semantics for diffusion $l \leftarrow \mathtt{dis}$, which applies the $\mathcal{D}$ function to the session $\kappa \uplus \kappa'$, where κ corresponds to the l's session. Proof rule PDis replaces session $\kappa \uplus \kappa'$ with the application $\mathcal{D}$ on the session. Quantum diffusion operations are used in many algorithms, such as amplifying a basis state's amplitude value in Grover's search algorithm, or redistributing a possible path direction in quantum walk algorithm. In these algorithms, the session piece that is diffused has either a small constant number of qubits or the whole session, meaning that the n number in the $\mathcal{D}$ function is either very small or equal to $|\kappa \uplus \kappa'|$, as the whole session. In either case, the summation formula in $\mathcal{D}$'s modeling (Figure 11b) can be rewritten as very few terms that facilitate the automated verification, which is exactly how we handle the diffusion operations in Qafny . An example of Grover's search and quantum walk algorithm is given in Section 4.

2.4 Qafny Metatheory

The type system is sound and the proof system is proved to be sound and complete.

Type Soundness. We prove that well-typed Qafny programs are well defined; i.e., the type system is sound with respect to the semantics. We begin by defining the session domain and state well-formedness.

The Qafny type soundness is stated as two theorems, type progress and preservation theorems. The proofs are done by induction on Qafny statements s and mechanized in Coq. Type progress states that any well-typed Qafny program can take a move, while type preservation states that for any such move, the transitioned type and state are preserved and well-typed, respectively.

Theorem 2.4 (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 2.5 (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 well defined; i.e., any properties derived in the Qafny proof system for well-typed Qafny programs can be interpreted by the state transitions in the Qafny semantics. In Qafny , there are three different

[19] Such as Python

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 φ, the semantic transition $(\varphi, s) \longrightarrow^* (\varphi', \{\})$ might not be unique, in the sense that there might be different representations of φ', due to the different state 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 a *most general state representation* of evaluating a statement s in an initial state φ below.

Definition 2.6 (Most general QAFNY *state).* Given a statement s, an initial state φ, kind environment Ω, type environment σ, and context mode g, such that $\Omega; \sigma \vdash \varphi$, $\vdash_g s \triangleright \sigma^*$, $\Omega; \sigma[\uparrow \sigma^*] \vdash \varphi^*$, and $(\varphi, s) \longrightarrow^* (\varphi^*, \{\})$, φ^* is the most general state representation of evaluating (φ, s), iff for all σ' and φ', such that $\vdash_g s \triangleright \sigma'$, $\Omega; \sigma[\uparrow \sigma'] \vdash \varphi'$ and $(\varphi, s) \longrightarrow^* (\varphi', \{\})$, $\sigma' \leq \sigma^*$ 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 of the whole Qafny+Dafny system, and the quantum portion contains non-terminated programs. Hence, the soundness and completeness essentially refers to the partial correctness of the QAFNY proof system and the total correctness is achieved by the Qafny+Dafny system through mapping QAFNY to Dafny, i.e., a separation logic proof system. Essentially, 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, we can conclude that the pre- and post- conditions of a program are modeled properly through the above modeling rules that rely on well-typed transition states.

THEOREM 2.7 (PROOF SYSTEM SOUNDNESS). For a well-typed program s, such that $\Omega; \sigma \vdash_g s \triangleright \sigma'$, $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$, $\Omega; \sigma; \varphi \models P$, then there exists a state representation φ', such that $(\varphi, s) \longrightarrow (\varphi', \{\})$ and $\Omega; \sigma[\uparrow \sigma']; \varphi' \models Q$, and there is a most general state representation φ^* of evaluating (φ, s) as $(\varphi, s) \longrightarrow (\varphi^*, \{\})$ and $\varphi' \equiv \varphi*$.

THEOREM 2.8 (PROOF SYSTEM RELATIVE COMPLETENESS). For a well-typed program s, such that $\Omega; \sigma \vdash_g s \triangleright \sigma'$, $(\varphi, s) \longrightarrow (\varphi', \{\})$ and $\Omega; \sigma \vdash \varphi$, there is most general state representation φ^*, such that $(\varphi, s) \longrightarrow (\varphi', \{\})$ and $\varphi' \equiv \varphi^*$ and $\Omega; \sigma \vdash_g s \triangleright \sigma^*$ and $\Omega; \sigma[\uparrow \sigma^*] \vdash \varphi^*$, and there are predicates P and Q, such that $\Omega; \sigma; \varphi \models P$ and $\Omega; \sigma[\uparrow \sigma^*]; \varphi^* \models Q$ and $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$.

3 QAFNY PROOF SYSTEM AND PROGRAM COMPILATION

We discuss the two dimensions of compilation 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 that QAFNY programs can be executed in a quantum machine.

3.1 Translation from QAFNY to Dafny

The implementation of QAFNY on Dafny in QNP is a compilation process from the QAFNY proof system to the Dafny proof system, a separation logic proof system. The QAFNY proof rules utilize extra session types to track the qubits in terms of sessions as well as state representation formats. However, there is no different kinds of state representations in Dafny, neither does Dafny support automatic equational rewrites of state forms. Additionally, Dafny predicate variables do not permit structures as sessions. All these entities require additional constructs and annotations to be inserted to the compiled predicates and programs when translating QAFNY programs and specifications to Dafny.

$$\{x : \mathsf{Q}\ n, y : \mathsf{Q}\ 1\}; \{x[0..n] : \mathsf{Had}, y[0..1] : \mathsf{Nor}\} \vdash_g$$
$$\{x[0..n] \mapsto \tfrac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + |1\rangle) * y[0..1] \mapsto |0\rangle\}$$
$$\{x[0..n], y[0..1]\} \leftarrow x < 5 @ y[0]$$
$$\{\{x[0..n], y[0..1]\} \mapsto \Sigma_{j=0}^{2^n} |(|j|).(|j < 5|)\rangle\}$$

.

$$\{\mathsf{name}(u_1) = x[0..n] \wedge \mathsf{name}(u_2) = y[0..1] \wedge \mathsf{type}(u_1) = \mathsf{Had} \wedge \mathsf{type}(u_2) = \mathsf{Nor}$$
$$* u_1 \mapsto \tfrac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + |1\rangle) * u_2 \mapsto |0\rangle\}$$
$$\mathtt{lift}(y[0..1]) \ ; \mathtt{lift}(x[0..n) \ ; \mathtt{join}(x[0..n], y[0..1]) \ ; \{x[0..n], y[0..1]\} \leftarrow x < 5 @ y[0]$$
$$\{\mathsf{name}(u_1) = x[0..n] \wedge \mathsf{name}(u_2) = y[0..1] \wedge \mathsf{name}(u_3) = \{x[0..n], y[0..1]\} \wedge \mathsf{type}(u_1) = \mathsf{CH}$$
$$\wedge \mathsf{type}(u_2) = \mathsf{CH} \wedge \mathsf{type}(u_3) = \mathsf{CH} * u_1 \mapsto \mathsf{fst}(u_3) * u_2 \mapsto \mathsf{snd}(u_3) * u_3 \mapsto \Sigma_{j=0}^{2^n} |(|j|).(|j < 5|)\rangle\}$$

Fig. 12. Qafny to Dafny Compilation Example

This section shows how additional constructs and annotations are inserted in compiling Qafny programs and specifications to Dafny ones, with no loss of expressiveness. We present a compilation algorithm that converts from Qafny to Dafny. Our compilation algorithm is evidence that proofs in Qafny essentially utilize the classical Hoare logic style automated system and we build the connection between quantum computation and classical ones through the view of quantum computation as some classical aggregate operations that can be much more efficiently done in a quantum computer.

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 input of pre-condition P and post-condition Q, as well as the compilation result of the Dafny pre-condition P', post-condition Q' and program s', such that the proof $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$ is valid in Qafny , we have $\{P'\}s'\{Q'\}$ being valid in Dafny. Ξ is an additional map from sessions to predicate variables in Dafny, such that we use variables in P' and Q' to represent sessions in P and Q. 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 4.

Conceptually, the compilation procedure does three items. First, every time there is a change in a session, such as qubit position permutation and split/join of sessions, we generate additional variables representing sessions, which reflect such change. Second, for a program s, if there is a change of state forms, we insert an additional construct to reflect the change so that the Dafny proof system can capture the state form transformation. Third, we generate additional Dafny axioms and inference rules for types and state transitions. Essentially, for a Qafny proof $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$ with the type judgment $\Omega; \sigma \vdash_g \sigma'$, a compiled Dafny proof has the form:

$$\{A \wedge T \wedge \overline{P}\}s'\{A \wedge T' \wedge \overline{Q}\}$$

where s' is the compiled Dafny program with the additional construct insertions; $\overline{P}$ 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 quantum semantic rewrite rules. 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 $y[0]$ in Figure 12 [20], where x and y are of type Had and Nor, respectively. The result of the application is an entanglement state having 2^n basis states, where

[20]The Boolean equation is a Qafny Boolean guard which unveils that Qafny Boolean guards for conditionals are compiled to oracle applications.

$$\frac{x \notin \mathrm{dom}(\Omega) \qquad \Omega[x \mapsto \mathtt{Q}\ m]; \gamma[x[0..n] \mapsto [n, n+m)]; n+m \vdash s \to \epsilon}{\Omega; \gamma; n \vdash \mathtt{let}\ x\ =\ \mathtt{init}\ m\ \mathtt{in}\ s \to \epsilon} \qquad \frac{x \notin \mathrm{dom}(\Omega) \qquad \Omega[x \mapsto \mathtt{M}]; \gamma; n \vdash s[(r,n)/x] \to \epsilon}{\Omega; \gamma; n \vdash \mathtt{let}\ x\ =\ (r,n)\ \mathtt{in}\ s \to \epsilon}$$

$$\frac{\Omega; \gamma; n \vdash \mu \to \epsilon}{\Omega; \gamma; n \vdash \lambda \leftarrow \mu \to \epsilon} \qquad \frac{\Omega; \gamma; n \vdash b@x[i] \to \epsilon \qquad \Omega; \gamma; n \vdash s \to \epsilon'}{\Omega; \gamma; n \vdash \mathtt{if}\ (b@x[i])\ s \to \epsilon\ ;\ \mathtt{ctrl}(\gamma(x[i]), \epsilon')}$$

$$\frac{\Omega; \gamma; n \vdash \mathtt{if}\ (b[i/x])\ s[i/x] \to \epsilon_i \quad \dots \quad \Omega; \gamma; n \vdash \mathtt{if}\ (b[j-1/x])\ s[j-1/x] \to \epsilon_{j-1}}{\Omega; \gamma; n \vdash \mathtt{for}\ (\mathtt{int}\ x \in [i, j)\ \&\&\ b)\ s \to \epsilon_i\ ; ...;\ \epsilon_{j-1}}$$

Fig. 13. Select QAFNY to SQIR translation rules (SQIR circuits are marked blue)

for any basis state, the $y[0]$ bit position stores the the result of computing $x < 5$. Since variables x and y initially are of type Had and Nor, their types are turned to CH, and the two sessions join together after the application, the compiler notices this and inserts two `lift` statements to turn x and y to CH type and then insert a `join` statement to join the two CH type 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 name function that takes a variable and outputs its pointed-to session. In addition, the compiled result has no type environment, therefore, we use the `type` function to track the session types.

In compiling quantum conditionals, not only we need to do the above type conversion, will we also explicitly insert mask ($\mathcal{M}$) and unmask ($\mathcal{U}$) functions. For example, 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 need to add the $\mathcal{M}$ and $\mathcal{U}$ before and after the conditional as:

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

The mask ($\mathcal{M}$) and unmask ($\mathcal{U}$) functions will mask the qubits in the Boolean predicate, e.g. $x[0]$, in the conditional body and assemble the conditional result back to the unmasked state after the conditional computation. We implemented the compiler in Dafny as well as formalize it in Coq and prove the correctness theorem below. The target Dafny formalism is a classical separation logic framework [Reynolds 2002] with the QAFNY programs syntax and predicate functions mentioned in Section 2 and Section 3.1.

THEOREM 3.1 (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.*

3.2 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 $i < \mathtt{nat}(\Omega(x))$ ($\Omega(x) = \mathtt{Q}\ m$ and $\mathtt{nat}(\mathtt{Q}\ m = m)$, γ maps $x[i]$ to a unique concrete index chosen from 0 to n. Ω is populated through the QAFNY type checking in Figure 30, while γ and n are populated when hitting an `init` operation as shown in Figure 13.

Algorithm	Running Time (seconds)	Human Resources (days)	Number of Lines
GHZ	4.2	< 1	13
Controlled GHZ	6.4	< 1	15
Deutsch–Jozsa	3.3	< 1	11
Grover's search	26.7	2	19
Shor's algorithm	36.3	30	28
Quantum Walk	43.1	2	35

Fig. 14. Computer running time and human labor time for verifying algorithms in Qafny . Running time is measured in a i7 windows computer. Every algorithm is verified by a single person, thus the human resources measure the time for a person to finish programming and verifying an algorithm. The quantum walk algorithm is the core of the Childs' Boolean equation algorithm [Childs et al. 2007].

Figure 13 depicts a selection of translation rules.[21] The two rules in the first line show how a `let`-binding is handled for a `Q` and `M` mode variable, while `C`-mode `let`-binding is handled by a similar manner. Similar to the type system, we assume that `let`-binding introduces new variables probably with proper variable renaming. The first rule in the second line describes an oracle application compilation, which is handled by the OQASM compiler [Li et al. 2022]. The Qafny type system ensures that the qubits mentioned in μ are the same as qubits in session λ, so that the translation does not rely on λ itself. The next rule describes the translation of a conditional to a controlled operation in sqir. In the conditional translation, the rule assumes that ι's translation does not affect the γ position map. This requirement is assured for well-typed programs per rule TIF in Figure 25. Here, we first translate the Boolean guard $b@x[i]$, 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 [Nielsen and Chuang 2011, Chapter 4.3]. The last rule translate Qafny for-loops. Essentially, a for-loop is compiled to a series of conditionals with each step differs in the loop step value for x.

The compiler is implemented in Coq and validated through testing. We extract both the Qafny semantics and the Qafny to sqir compiler to Ocaml and compile compiled sqir programs to OpenQASM [Cross et al. 2021] 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.

4 QAFNY 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, 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, Grover's search, and Shor's algorithm, as a demonstration of verifying quantum algorithms in Qafny .

Figure 14 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

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

framework [Leino and Moskal 2014; Qiu et al. 2013], and better than the existing quantum automated frameworks, such as QBricks [Chareton et al. 2021].

The computer running time is usually a less important factor in verifying programs, while the human effort is the more important issues. 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 14, all the algorithms are written with less than 35 lines of code. One of the examples is the Shor's algorithm specification written in the Qafny interface in Figure 15. In contrast, most algorithms written in other quantum automated verification frameworks require more than 1000 lines of code. For example, the Grover's search specification in quantum Haore Logic [Liu et al. 2019] has 3184 lines of code, and the Shor's algorithm specification in QBricks [Chareton et al. 2021] has 1163 lines of code [22]. In terms of human resources, other than Shor's algorithm, which was done before the Qafny compiler was fully constructed, most algorithms were written and verified in Qafny (Figure 14) in two days by a single researcher. As a comparison, the complete Shor's algorithm correctness proof [Peng et al. 2022] 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. Therefore, QNP help relieve programmers' pain in reasoning about quantum programs.

As a consequence, QNP also helps programmers in discovering new ways of utilizing quantum algorithms. The controlled GHZ algorithm does not typically appear in many quantum information books [Nielsen and Chuang 2011], but it is a nice usage of the GHZ algorithm to prepare different entanglement structures. The implementation and verification of the controlled GHZ algorithm finishes momentarily (Figure 14).

REFERENCES

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

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

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

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).

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

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]

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

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

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

Mike Gordon. 2012. Background reading on Hoare Logic. https://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf

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

[22] 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 .

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

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

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

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

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

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.

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

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.

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.

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

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

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

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

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

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

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

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

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. 15. Shor's Algorithm in Q-Dafny

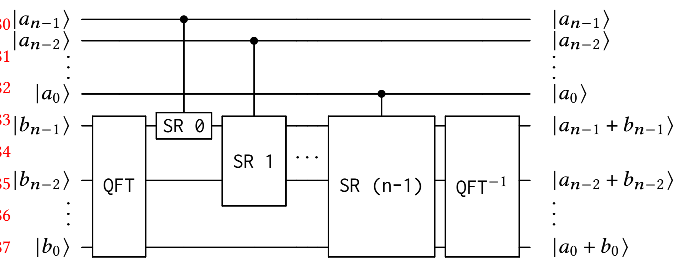
A OQASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

We designed OQASM to be able to express efficient quantum oracles that can be easily tested and, if desired, proved correct. OQASM operations leverage both the standard computational basis and an alternative basis connected by the quantum Fourier transform (QFT). OQASM's type system tracks the bases of variables in OQASM programs, forbidding operations that would introduce entanglement. OQASM states are therefore efficiently represented, so programs can be effectively tested and are simpler to verify and analyze. In addition, OQASM 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 OQASM states and the language's syntax, semantics, typing, and soundness results. As a running example, we use the QFT adder [Beauregard 2003] shown in Figure 16. The Coq function `rz_adder` generates an OQASM program that adds two natural numbers `a` and `b`, each of length `n` qubits.

A.1 OQASM States

An OQASM program state is represented according to the grammar in Figure 17. 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 OQASM program on any well-formed OQASM 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. 16. 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	::=	$\mathtt{Nor} \mid \mathtt{Phi}\ n$
Unphased qubit	$\overline{q}$	::=	$\lvert b\rangle \mid \lvert\Phi(r)\rangle$
Qubit	q	::=	$\alpha(r)\overline{q}$
State (length d)	φ	::=	$q_1 \otimes q_2 \otimes \cdots \otimes q_d$

Fig. 17. OQASM state syntax

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

Fig. 18. OQASM syntax. For an operator OP, $\mathtt{OP}^{[-1]}$ indicates that the operator has a built-in inverse available.

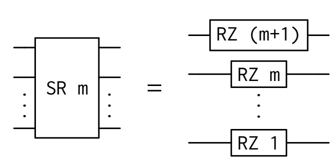


Fig. 19. SR unfolds to a series of RZ instructions

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 [2003], 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 18 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 $\mathtt{RZ}^{[-1]}\ n\ p$ —and instruction sequencing. The instructions in the next row apply to whole

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

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

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

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

Fig. 20. Select 𝒪QASM typing rules

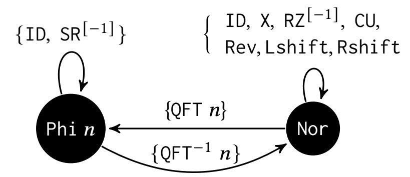


Fig. 21. Type rules' state machine

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 19). 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 16(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 18, 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 25; 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))] & \texttt{where}\ \ \texttt{xg}(|0\rangle) = |1\rangle \qquad \texttt{xg}(|1\rangle) = |0\rangle \\
[\![\texttt{CU}\ (x,i)\ \iota]\!]\varphi & = \texttt{cu}(\downarrow \varphi(x,i), \iota, \varphi) & \texttt{where}\ \ \texttt{cu}(|0\rangle, \iota, \varphi) = \varphi \qquad \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))] & \texttt{where}\ \ \texttt{rz}(m, |0\rangle) = |0\rangle \qquad \texttt{rz}(m, |1\rangle) = \alpha(\frac{1}{2^m})\,|1\rangle \\
[\![\texttt{RZ}^{-1}\ m\ (x,i)]\!]\varphi & = \varphi[(x,i) \mapsto \uparrow \texttt{rrz}(m, \downarrow \varphi(x,i))] & \texttt{where}\ \ \texttt{rrz}(m, |0\rangle) = |0\rangle \qquad \texttt{rrz}(m, |1\rangle) = \alpha(-\frac{1}{2^m})\,|1\rangle \\
[\![\texttt{SR}\ m\ x]\!]\varphi & = \varphi[\forall i \le m.\ (x,i) \mapsto \uparrow |\Phi(r_i + \frac{1}{2^{m-i+1}})\rangle] & \texttt{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 - \frac{1}{2^{m-i+1}})\rangle] & \texttt{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)] & \texttt{where}\ \ \texttt{qt}(i, |y\rangle, n) = \bigotimes_{k=0}^{i-1}(|\Phi(\frac{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))] & \texttt{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))] & \texttt{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))] & \texttt{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}
$$

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

Fig. 22. $\mathcal{O}$QASM semantics

the quantum *no-cloning rule*. For example, we can apply the CU in `rz_adder'` (Figure 16) 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 21. 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 Definition A.1) 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 $\mathcal{O}$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 22.

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 22 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 $\sum_{j=0}^{2^n}(-i)^x\,|x\rangle$ [Childs et al. 2007], 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 $n \le k$ (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 $n \le \Sigma(x)$, there exists a value v such that for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|\Phi(\frac{v}{2^{n-k}})\rangle$.[23]

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 $\mathcal{H}^d$,[24] 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 [Hietala et al. 2021a]. 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 Definition A.1. 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.

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

[24] 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. 23. Select $\mathcal{O}$QASM inversion rules

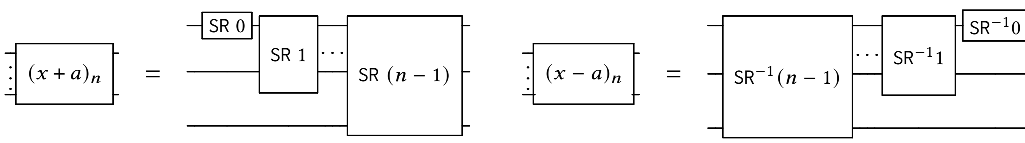


Fig. 24. Addition/subtraction circuits are inverses

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 $\mathcal{O}$QASM operation is valid in a traditional quantum language like SQIR. We then have the following two two theorems to connect $\mathcal{O}$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 $\mathcal{O}$QASM to SQIR (**??**). Theorem A.3 suggests that the space $\mathcal{S}^d$ is closed under the application of any well-typed $\mathcal{O}$QASM operation. Theorem A.4 says that $\mathcal{O}$QASM oracles can be safely applied to superpositions over classical states.[25]

$\mathcal{O}$QASM programs are easily invertible, as shown by the rules in Figure 23. 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 23). 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.

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

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

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

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

Fig. 25. Arith, Bool, Gate Mode Checking

$\bigotimes_n$ $\mathtt{Had}$ 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. $\bigcirc$ 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 $\bigotimes_n$ $\mathtt{CH}$ $m\beta$ 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 $\bigotimes_n$ $\mathtt{CH}$ $m\infty$. In such case, the only quantum operation to apply is a measurement. If a session has type $\bigotimes_n$ $\mathtt{CH}$ $m\beta$ 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 $\bigotimes_n$ $\mathtt{CH}$ $2\{\overline{0}^n, \overline{1}^n\}$, where $\overline{d}^n$ is a n-bit string having bit d.

The type $\bigotimes_n$ $\mathtt{CH}$ $m\beta$ 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 $\mathtt{CH}$, 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 30 has several tasks. First, it enforces context mode restrictions. Context mode g is either Cor $\mathtt{Q}$. Q represents the current expression lives inside a quantum conditional or loop, while Crefers 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 25. In QWhile, $\mathtt{C}$-mode variables are evaluated to values during type checking. In a `let` statement (Figure 30), $\mathtt{C}$-mode expression is evaluated to a value n, and the variable x is replaced by n in s. The expression mode checking (Figure 25) 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 $\mathtt{C}$-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
$x[i]$ y	Nor any	Had Nor	Had Nor	Had Had	Had Had	Had CH	Had CH	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. 26. Control Gate Entanglement Situation

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

Fig. 27. Session Type Subtyping

such as the index value $x[n]$ in difference expressions in Figure 25. It is worth noting that the session computation ($\uplus$) is also commutative as the last rule in Figure 25.

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 17) 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 [Shor 1994] and Childs' Boolean equation algorithm (BEA) [Childs et al. 2007], programmers care about if qubits eventually become entangled during a quantum loop execution. This is why the normal basis type ($\bigotimes_n$ Nor $\overline{d}$) can also be a subtype of a entanglement type ($\bigotimes_n$ CH $1\{\overline{d}\}$) in our system (Figure 27).

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 26.

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 [Childs et al. 2007]. 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

Nor ∞	$\sqsubseteq_n$	CH ∞
Nor c	$\sqsubseteq_n$	CH $\{c\}$
CH $\bar{c}(1)$	$\sqsubseteq_n$	Nor $\bar{c}[0]$
Had p	$\sqsubseteq_n$	CH $\{(\!|j|\!)\|j \in [0, 2^n)\}(2^n)$
CH $\{0, 1\}$	$\sqsubseteq_1$	Had ∞
CH p	$\sqsubseteq_n$	CH ∞

(a) Subtyping

$\|c\rangle$	$\equiv_n$	$\Sigma_{j=0}^{1} \|c\rangle$
$\Sigma_{j=0}^{1} z_j \|c_j\rangle$	$\equiv_n$	$\|c_0\rangle$
$\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (\|0\rangle + \alpha(r_j) \|1\rangle)$	$\equiv_n$	$\Sigma_{j=0}^{2^n} \frac{\alpha(\Sigma_{k=0}^{n} r_k \cdot (\!|j|\!)[k])}{\sqrt{2^n}} \|j\rangle$
$\Sigma_{j=0}^{2} z_j \|c_j\rangle$	$\equiv_1$	$\frac{1}{\sqrt{2}} \bigotimes_{j=0}^{1} (\|0\rangle + \frac{\sqrt{2}z_1}{z_0} \|1\rangle)$ when $c_0 = 0$ $c_1 = 1$

(b) State Equivalence

Fig. 28. Qafny type/state relations. $\bar{c}[n]$ produces the n-th element in set $\bar{c}$. $\{(\!|j|\!)|j \in [0, 2^n)\}(2^n)$ 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 $\sum_{i=0}^{n} |x_l, y, x_r\rangle$, 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: $\sum_{k=0}^{2^n} |k\rangle$) and a Nor type factor $x[i]$ (state: $|0\rangle$). The result is an entanglement $\sum_{k=0}^{2^n} |k, k < i\rangle$, where the $x[i]$ position stores the Boolean bit result $k < i$. [26] 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

[26]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 34. 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 34a 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 $\{c\}$ [27]. 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 $\mathsf{dom}(\sigma')$, 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 30 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

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

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

$$
\begin{array}{l}
\mathtt{pmut}((c_1.i_1.i_2.c_2), n) = (c_1.i_2.i_1.c_2) \ \text{ when } |c_1| = n \\
\mathtt{mut}(\mathtt{Nor}\ c, n) = \mathtt{Nor}\ \mathtt{pmut}(c, n) \qquad \mathtt{mut}(\mathtt{CH}\ \overline{c}(m), n) = \mathtt{CH}\ \{\mathtt{pmut}(c, n) | c \in \overline{c}(m)\}(m) \qquad \mathtt{mut}(\tau, n) = \tau \ \ [\mathtt{owise}] \\
\mathtt{mer}(\mathtt{Nor}\ c_1, \mathtt{Nor}\ c_2) = \mathtt{Nor}\ (c_1.c_2) \qquad \mathtt{mer}(\mathtt{Had} \bigcirc, \mathtt{Had} \bigcirc) = \mathtt{Had} \bigcirc \qquad \mathtt{mer}(T\ \infty, T\ t) = T\ \infty \\
\mathtt{mer}(\mathtt{CH}\ \overline{c_1}(m_1), \mathtt{CH}\ \overline{c_2}(m_2)) = \mathtt{CH}\ (\overline{c_1} \times \overline{c_2})(m_1 * m_2) \\
\mathtt{spt}(\mathtt{Nor}\ c_1.c_2, n) = (\mathtt{Nor}\ c_1, \mathtt{Nor}\ c_2) \ \text{ when } |c_1| = n \qquad \mathtt{spt}(\mathtt{Had}\ t, n) = (\mathtt{Had}\ t, \mathtt{Had}\ t) \\
\mathtt{spt}(\mathtt{CH}\ \{c_j.c | j \in [0, m) \wedge |c_j| = n\}(m), n) = (\mathtt{CH}\ \{c_j | j \in [0, m) \wedge |c_j| = n\}(m), \mathtt{Nor}\ c)
\end{array}
$$

Fig. 29. 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.

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

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

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

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

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

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

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

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

Fig. 30. Qafny type system. $\llbracket \mu \rrbracket c$ is the $\mathcal{O}$qasm semantics of interpreting reversible expression μ in Figure 22. Boolean expression b can be $a_1 = a_2$, $a_1 < a_2$ or true. $b[(|n|)/\lambda]$ means that we treat b as a $\mathcal{O}$qasm μ expression, replace qubits in array λ with bits in bitstring $(|n|)$, 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 $-$.

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 34d, which will be explained shortly. Selected type rules are given in Figure 30; 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 $\mathsf{CH}\ \overline{c}(m)$, 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 unchanged. Here, we utilize the $\mathbb{O}$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 $\mathsf{CH}\ \overline{c}(m)$, with $|c_1| = j$, we keep only the ones with $c_1 = (\!|n|\!)$ (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 34d, 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 34d 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 34d. Merging two sessions is valuable for analyzing the behavior of quantum conditionals. In rule TIF, the session $(\lambda_1 \uplus x[j, \mathsf{S}\ j])$ 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 34d by computing the Cartesian product of the two type bases, such that if the two sessions are both CH types $\lambda_1 \uplus x[j, \mathsf{S}\ j] \mapsto \mathsf{CH}\ \overline{c_1}(m_1)$ and $\lambda_2 \mapsto \mathsf{CH}\ \overline{c_2}(m_2)$, the result is of type $\mathsf{CH}\ (\overline{c_1} \times \overline{c_2})(m_1 * m_2)$. 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. [28], 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 $\mathtt{if}\ (b)\ s$ 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 22.

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 22 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). `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 $n \le k$ (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.

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

$$\text{SPAR}\quad \frac{\varphi \equiv \varphi'}{(\varphi, s) \longrightarrow (\varphi', s)}$$

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

$$\text{SSEQ-1}\quad \frac{(\varphi, s_1) \longrightarrow (\varphi', s_1')}{(\varphi, s_1\ ;\ s_2) \longrightarrow (\varphi', s_1'\ ;\ s_2)}$$

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

$$\text{SSEQ-2}\quad (\varphi, \{\}\ ;\ s_2) \longrightarrow (\varphi, s_2)$$

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

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

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

Fig. 31. QAFNY small step semantics. $[\![\mu]\!]c$ is the OQASM semantics of interpreting reversible expression μ in Figure 22. 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 $\varphi(\lambda) = \{\lambda \mapsto q\}$.

C.3 Logic Proof System

The reason of having the session type system in Figure 30 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 Appendix C.1. 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 17 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 $\texttt{Nor}$, 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 $\texttt{CH}$, 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 15 line 12, we apply a series of $\texttt{H}$ 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 $\texttt{H}$ application results in a pre-condition as $\texttt{H}[(x, 0, n)] = \bigotimes_{k=0}^{n} |\Phi(0)\rangle$, 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 $(\texttt{as}(\zeta[v])^2, \texttt{bs}(\zeta[v]))$ coming from one possible basis state of y (picking a random index v in ζ), $\texttt{as}(\zeta[v])$ is the amplitude and $\texttt{bs}(\zeta[v])$ 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 $\bigotimes_n$ $\texttt{CH}$ $2m(\beta_1 \cdot 0 \cup \beta_2 \cdot 1)$. 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 $\texttt{if}$ $(x[i])$ 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 $\texttt{diffuse}(x)$. The quantum semantics for $\texttt{diffuse}(x)$ 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, $\texttt{diffuse}(x)$ 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 $\mathscr{P}(\Sigma(x))$, 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 [Green et al. 2013]. This section presents the arithmetic operators we have implemented in 𝔒QASM, while the next section discusses the geometric operators and expressions implemented in 𝔒QIMP. The following section presents an end-to-end case study applying Grover's search.

C.4 Implemented Operators

Figure 14 and ?? tabulate the arithmetic operators we have implemented in 𝔒QASM.

The addition and modular multiplication circuits (parts (a) and (d) of Figure 14) are components of the oracle used in Shor's factoring algorithm [Shor 1994], which accounts for most of the algorithm's cost [Gidney and Ekerå 2021]. 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 [2003] and Markov and Saeedi [2012]—in 𝔒QASM. 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 [Draper 2000], while the second (TOFF) uses a ripple-carry adder [Markov and Saeedi 2012], which makes use of classical controlled-controlled-not (Toffoli) gates.

Code	Assertion
	$\{A(x) * A(y)\}$ where $A(\beta) = \beta[0..n] \mapsto \lvert\overline{0}\rangle$
	$B = 1 < a < N \wedge n > 0 \wedge$
	$N < 2^n \wedge \text{gcd}(a, N) = 1$
	$\Rightarrow \{\lVert \mathsf{H} \rVert(x[0..n]) \mapsto C * A(y) * B\}$
	where $C = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (\lvert 0\rangle + \lvert 1\rangle)$
$x \leftarrow \mathsf{H};$	$\{x[0..n] \mapsto C * A(y) * B\}$
	$\Rightarrow \{x[0..n] \mapsto C * \lVert y{+}1 \rVert(y[0..n]) \mapsto \lvert\overline{0}.1\rangle * B\}$
$y \leftarrow y{+}1;$	$\{x[0..n] \mapsto C * y[0..n] \mapsto \lvert\overline{0}.1\rangle * B\}$
	$\Rightarrow \{E(0) * B\}$ where $E(k) =$
	$x[0..n-k] \mapsto \frac{1}{\sqrt{2^{n-k}}} \bigotimes_{j=0}^{n-k} (\lvert 0\rangle + \lvert 1\rangle) *$
	$\{x[0..k], y[0..n]\} \mapsto \Sigma_{j=0}^{2^k} \frac{1}{\sqrt{2^k}} \lvert (\lvert j \rvert).(\lvert a^j \% N \rvert)\rangle$
`for` $(\texttt{int}\ j{:=}0;\ j{<}n\ \&\&\ x[j]\ ;\ {+}{+}j)$	$\{E(j) * B\}$
$\{\quad y \leftarrow a^{2^j} y \% N$	$\{E(j+1) * B\}$
$\}$	$\{E(n) * B\}$
	$\Rightarrow \{\{x[0..n], y[0..n]\} \mapsto \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} \lvert (\lvert j \rvert).(\lvert a^j \% N \rvert)\rangle * B\}$
`let` u `=` `measure`(y) `in` ...	$\left\{ \begin{array}{l} x[0..n] \mapsto \frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} \lvert t+kp\rangle \wedge p = \texttt{ord}(a, N) \\ \wedge\ \texttt{nat}(u) = a^t \% N \wedge s = \texttt{rnd}(\frac{2^n}{p}) \wedge B \end{array} \right\}$

Fig. 32. Pre-measurement quantum steps of the Shor's algorithm. Second half in Figure 33. $\texttt{nat}(u)$ gets the integer number part of u (mode $\texttt{M}$). $\texttt{ord}(a, N)$ gets the order of a and N. $\texttt{rnd}(r)$ rounds r to the nearest integer.

Code	Assertion
`let` z `=` `measure`(y) `in`	$\left\{ \begin{array}{l} x[0..n] \mapsto \frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} \lvert t+jr\rangle \wedge \\ \texttt{nat}(z) = a^n \% N \wedge s = \texttt{rnd}(\frac{2^n}{r}) \wedge B \end{array} \right\}$
$\quad x \leftarrow \texttt{QFT}^{-1}$;	$\{x[0..n] \mapsto \frac{1}{\sqrt{s2^n}} \Sigma_{k=0}^{2^n} (\omega^{tk} \Sigma_{j=0}^{s} \omega^{tkj}) \lvert k\rangle \wedge s = \texttt{rnd}(\frac{2^n}{r}) \wedge B\}$
$\quad$ `let` u `=` `measure`(x) `in`	$\{\texttt{nat}(u) = r \wedge \texttt{pos}(u) = \frac{4}{\pi^2 r} \wedge s = \texttt{rnd}(\frac{2^n}{r}) \wedge r = \texttt{ord}(a, N) \wedge B\}$
$\qquad$ `post`(u)	$\{\texttt{nat}(\texttt{post}(u)) = r \wedge r = \texttt{ord}(a, N) \wedge \texttt{pos}(u) = \frac{4e^{-2}}{\pi^2 \log_2^4 N \wedge B}\}$

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

Fig. 33. Second half of the Shor's algorithm quantum part in Qafny.

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 1, 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 34 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 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,

$$
\begin{array}{lcl}
\tau & \sqsubseteq & \tau \\
\texttt{Nor} & \sqsubseteq & \texttt{CH} \\
\texttt{Had} & \sqsubseteq & \texttt{CH}
\end{array}
$$

(a) Subtyping

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

(b) Quantum Value Equivalence

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

(c) Session Equivalence

$$
\begin{array}{ll}
\sigma & \preceq \sigma \\
\{\bot : \tau\} \cup \sigma & \preceq \sigma \\
\{\lambda : \tau\} \cup \sigma & \preceq \{\lambda : \tau'\} \cup \sigma \\
 & \quad \texttt{where} \;\; \tau \sqsubseteq_{|\lambda|} \tau' \\
\{\lambda_1 \uplus l_1 \uplus l_2 \uplus \lambda_2 : \tau\} \cup \sigma & \preceq \{\lambda_1 \uplus l_2 \uplus l_1 \uplus \lambda_2 : \tau\} \cup \sigma \\
\{\lambda_1 : \tau\} \cup \{\lambda_2 : \tau\} \cup \sigma & \preceq \{\lambda_1 \uplus \lambda_2 : \tau\} \cup \sigma \\
\{\lambda_1 \uplus \lambda_2 : \tau\} \cup \sigma & \preceq \{\lambda_1 : \tau\} \cup \{\lambda_2 : \tau\} \cup \sigma \\
 & \quad \texttt{where} \;\; \tau \neq \texttt{CH}
\end{array}
$$

(d) Environment Equivalence

$$
\begin{array}{ll}
\varphi & \equiv \varphi \\
\{\bot : q\} \cup \varphi & \equiv \varphi \\
\{\lambda : q\} \cup \varphi & \equiv \{\lambda : q'\} \cup \varphi \\
 & \quad \texttt{where} \;\; q \equiv_{|\lambda|} q' \\
\{\lambda_1 \uplus l_1 \uplus l_2 \uplus \lambda_2 : q\} \cup \varphi & \equiv \{\lambda_1 \uplus l_2 \uplus l_1 \uplus \lambda_2 : \texttt{mut}(q, |\lambda_1|)\} \cup \varphi \\
\{\lambda_1 : q_1\} \cup \{\lambda_2 : q_2\} \cup \varphi & \equiv \{\lambda_1 \uplus \lambda_2 : \texttt{mer}(q_1, q_2)\} \cup \varphi \\
\{\lambda_1 \uplus \lambda_2 : \varphi\} \cup \sigma & \equiv \{\lambda_1 : \varphi_1\} \cup \{\lambda_2 : \varphi_2\} \cup \sigma \\
 & \quad \texttt{where} \;\; \texttt{spt}(\tau, |\lambda_1|) = (\varphi_1, \varphi_2)
\end{array}
$$

(e) State Equivalence

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

Fig. 34. 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 $\frac{1}{\sqrt{2^{n+m}}} \bigotimes_{j=0}^{n+m} q_j$.

as well as the three state types in every transition step, so that the system knows exactly how to apply an equation.

Figure 34 shows the equivalence relations on types and states. Figure 34a 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 34b, 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 34c defines the equivalence relations for the session concatenation operation $\uplus$: it is associative, identitive with the identity empty session element $\bot$. We also view a range $x[n,n]$ to be empty ($\bot$), 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

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:

$$\textsc{SSeq-1}\ \frac{(\varphi,s_1)\longrightarrow(\varphi',s'_1)}{(\varphi,s_1\ ;\ s_2)\longrightarrow(\varphi',s'_1\ ;\ s_2)} \qquad \textsc{SSeq-2}\ (\varphi,\{\}\ ;\ s_2)\longrightarrow(\varphi,s_2) \qquad \textsc{PSeq}\ \frac{\begin{array}{c}\Omega;\sigma\vdash_g s_1\triangleright\sigma'\\ \Omega;\sigma\vdash_g\{P\}\,s_1\,\{R\} \qquad \Omega;\sigma[\uparrow\sigma']\vdash_g\{R\}\,s_2\,\{Q\}\end{array}}{\Omega;\sigma\vdash_g\{P\}\,s_1\ ;\ s_2\,\{Q\}}$$

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

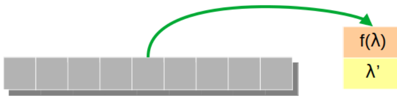
$$\textsc{PConL}\ \frac{(\Omega,\sigma,P)\Rightarrow(\Omega,\sigma',P') \qquad \Omega;\sigma'\vdash_g\{P'\}\,s\,\{Q\}}{\Omega;\sigma\vdash_g\{P\}\,s\,\{Q\}} \qquad \textsc{PConR}\ \frac{\begin{array}{c}\Omega,\sigma\vdash_g s_1\triangleright\sigma'\\ \Omega;\sigma'\vdash_g\{P\}\,s\,\{Q'\} \qquad (\Omega,\sigma'',Q')\Rrightarrow(\Omega,\sigma[\uparrow\sigma'],Q)\end{array}}{\Omega;\sigma\vdash_g\{P\}\,s\,\{Q\}}$$

$$(\Omega,\sigma,P)\Rightarrow(\Omega,\sigma',P')\triangleq\Omega,\sigma\vdash P\wedge\Omega,\sigma'\vdash P'\wedge\sigma\leq\sigma'\wedge P\Rightarrow P'$$
$$(\Omega,\sigma,Q)\Rrightarrow(\Omega,\sigma',Q')\triangleq\Omega,\sigma\vdash Q\wedge\Omega,\sigma'\vdash Q'\wedge\sigma'\leq\sigma\wedge Q\Rightarrow Q'$$

Fig. 35. Sequence and Consequence Rules

rule in Figure 34e 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. 36. 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.