Quantum Natural Proof

ANONYMOUS AUTHOR(S)

Q-Dafny.

1 BACKGROUND

We begin with some background on quantum computing and quantum algorithms.

Quantum States. A quantum state consists of one or more quantum bits (*qubits*). A qubit can be expressed as a two dimensional vector $\binom{\alpha}{\beta}$ where α, β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write the qubit vector as $\alpha\,|0\rangle + \beta\,|1\rangle$ where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are *computational basis states*. When both α and β are non-zero, we can think of the qubit as being "both 0 and 1 at once," a.k.a. a *superposition*. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is an equal superposition of $|0\rangle$ and $|1\rangle$.

We can join multiple qubits together to form a larger quantum state with the *tensor product* ($\otimes$) from linear algebra. For example, the two-qubit state $|0\rangle \otimes |1\rangle$ (also written as $|01\rangle$) corresponds to vector $[\ 0\ 1\ 0\ 0\ ]^T$. Sometimes a multi-qubit state cannot be expressed as the tensor of individual states; such states are called *entangled*. One example is the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*. Entangled states lead to exponential blowup: A general n-qubit state must be described with a 2^n-length vector, rather than n vectors of length two. The latter is possible for unentangled states like $|0\rangle \otimes |1\rangle$; ⟦QASM's type system guarantees that qubits remain unentangled.

Quantum Circuits. Quantum programs are commonly expressed as *circuits*, like those shown in Figure 1. In these circuits, each horizontal wire represents a qubit, and boxes on these wires indicate quantum operations, or *gates*. Gates may be *controlled* by a particular qubit, as indicated by a filled circle and connecting vertical line. The circuits in Figure 1 use four qubits and apply 10 (left) or 7 (right) gates: four *Hadamard* (H) gates and several controlled z-axis rotation ("phase") gates. When programming, circuits are often built by meta-programs embedded in a host language, e.g., 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 our work).

Quantum Fourier Transform. The quantum Fourier transform (QFT) is the quantum analogue of the discrete Fourier transform. It is used in many quantum algorithms, including the phase estimation portion of Shor's factoring algorithm [Shor 1994]. The standard implementation of a QFT circuit (for 4 qubits) is shown on the left of Figure 1; an *approximate QFT* (AQFT) circuit can be constructed by removing select controlled phase gates [Barenco et al. 1996; Hales and Hallgren 2000; Nam et al. 2020]. This produces a cheaper circuit that implements an operation mathematically similar to the QFT. The AQFT circuit we use in VQO (for 4 qubits) is shown on the right of Figure 1. When it is appropriate to use AQFT in place of QFT is an open research problem, and one that is partially addressed by our work on ⟦QASM, which allows efficient testing of the effect of AQFT inside of oracles.

Computational and QFT Bases. The computational basis is just one possible basis for the underlying vector space. Another basis is the *Hadamard basis*, written as a tensor product of $\{|+\rangle, |-\rangle\}$, obtained by applying a *Hadamard transform* to elements of the computational basis, where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. A third useful basis is the *Fourier (or QFT) basis*, obtained by applying a *quantum Fourier transform* (QFT) to elements of the computational basis.

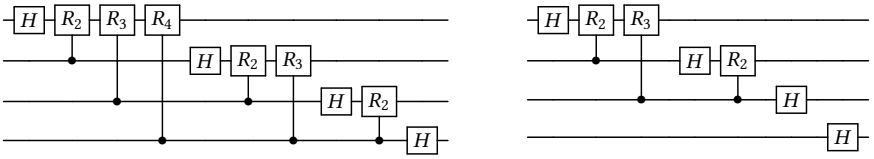


Fig. 1. Example quantum circuits: QFT over 4 qubits (left) and approximate QFT with 3 qubit precision (right). R_m is a z-axis rotation by $2\pi/2^m$.

Measurement. A special, non-unitary *measurement* operation extracts classical information from a quantum state, typically when a computation completes. Measurement collapses the state to a basis states with a probability related to the state's amplitudes. For example, measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ in the computational basis will collapse the state to $|0\rangle$ with probability $\frac{1}{2}$ and likewise for $|1\rangle$, returning classical value 0 or 1, respectively. In all the programs discussed in this paper, we leave the final measurement operation implicit.

Quantum Algorithms and Oracles. Quantum algorithms manipulate input information encoded in "oracles," which are callable black box circuits. For example, Grover's algorithm for unstructured quantum search [Grover 1996, 1997] is a general approach for searching a quantum "database," which is encoded in an oracle for a function $f : \{0,1\}^n \to \{0,1\}$. Grover's finds an element $x \in \{0,1\}^n$ such that $f(x) = 1$ using $O(2^{n/2})$ queries, a quadratic speedup over the best possible classical algorithm, which requires $\Omega(2^n)$ queries. An oracle can be constructed for an arbitrary function f simply by constructing a reversible classical logic circuit implementing f and then replacing classical logic gates with corresponding quantum gates, e.g., `X` for "not," `CNOT` for "xor," and `CCNOT` (aka *Toffoli*) for "and." However, this approach does not always produce the most efficient circuits; for example, quantum circuits for arithmetic can be made more space-efficient using the quantum Fourier transform [Draper 2000].

Transforming an irreversible computation into a quantum circuit often requires introducing ancillary qubits, or *ancillae*, to store intermediate information [Nielsen and Chuang 2011, Chapter 3.2]. Oracle algorithms typically assume that the oracle circuit is reversible, so any data in ancillae must be *uncomputed* by inverting the circuit that produced it. Failing to uncompute this information leaves it entangled with the rest of the state, potentially leading to incorrect program behavior. To make this uncomputation more efficient and less error-prone, recent programming languages such as Silq [Bichsel et al. 2020] have developed notions of *implicit* uncomputation. We have similar motivations in developing VQO: we aim to make it easier for programmers to write efficient quantum oracles, and to assure, through verification and randomized testing, that they are correct.

2 KEY FEATURES THROUGH A RUNNING EXAMPLE

As an illustration of QNP, we implement and prove part of the Shor's algorithm 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 $k = \gcd(a, N)$ [1]; (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 [2]; (4) if p is even and $a^{\frac{p}{2}} \neq -1 \% N$, $\gcd(a^{\frac{p}{2}} \pm 1, N)$ are the factors, otherwise, we repeat the process. Step (2) is the quantum part of Shor's algorithm and Figure 2 and Figure 14 show its automated proof in QNP. In Figure 22, we show the actual implementation and proof in the Qafny tool.

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

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

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

Fig. 2. First half of the Shor's algorithm quantum part. Second half in Figure 14. $\mathsf{nat}(u)$ gets the integer number part of u (mode M). $\mathsf{ord}(a, N)$ gets the order of a and N. $\mathsf{rnd}(r)$ rounds r to the nearest integer.

The Shor's quantum first half in Figure 2 can be analogized as an efficient array filter operation. The steps before line 11 (steps at line 3, 4 and 7-9) 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 11 filters the array as a new one (x) with all elements j satisfying $a^j \% N = z$ where z is a randomly picked number. Notice that modulo multiplication $f(j) = a^j \% N$ is a periodic function. All elements in x satisfy $a^j \% N = z$, which means that 1) there will be a smallest t such that $a^t \% N = z$, and 2) all elements can be rewritten as $j = t + kp$ and p is the period of the modulo multiplication function because $a^{t+kp} \% N = z$, which is given as the post-condition on the right of line 11. The first half implementation and correctness proof in Figure 2 exactly reflects the array analogy aspect. One of the biggest advantage of using Qafny is that we can analogize quantum operations to classical array operations, so that we can utilize the existing automated proof infrastructure in many tools, such as Dafny. In the Qafny implementation, verifying a quantum program usually requires only the user input of the pre-, post-, and loop invariant. For example, the proof of Figure 2 in Qafny requires none of the states marked teal, but only the conditions marked black and purple. [3] Here, we step by step discuss the states, program syntax, and proofs of the aspect.

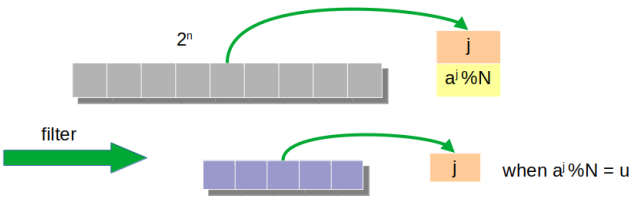


Fig. 3. The array analogy of Shor's first half in Figure 2.

[3] The purple is also not needed if we combine the Shor's first and second half algorithms together.

Basic Terms:

Nat. Num	m, n	$\in$	$\mathbb{N}$	Real	r	$\in$	$\mathbb{R}$	Complex Number	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	::=	C | M	
Classical Value	v	::=	$n \mid (r, n)$	
Full Mode (Kind)	$\overline{g}$	::=	$g \mid \mathtt{Q}\ n$	
Quantum Type	τ	::=	Nor	$\mid$ Had $\mid$ CH
Quantum Value	q	::=	$\lvert c\rangle$	$\mid \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (\lvert 0\rangle + \alpha(r_j) \lvert 1\rangle) \mid \Sigma_{j=0}^{m} z_j \lvert c_j\ \beta_j\rangle$

Quantum Sessions, Environment, and States

Range	l	::=	$x[n..m]$		
Session	λ	::=	$\overline{x[n..m]}$	by	$\uplus$
Type Environment	σ	::=	$\overline{\lambda : \tau}$	by	$\cup$
Quantum State	φ	::=	$\overline{\lambda : q}$	by	$\cup$

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

Classical and Quantum States. On the right of Figure 2 line 1, it is the the pre-condition of the program, which describes two quantum arrays; both are initialized to be n-qubit of bits 0 (as $\overline{0}$), represented as predicates $A(x)$ and $A(y)$, and other restrictions for integers a and N represented as predicate B. In Qafny, there are three kinds of values, two of which are classical ones represented by the two modes: C and M in Figure 4. 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 natural number value n. In Figure 2, a and N are C-mode values and u is a M value so that we can use function nat to produce its natural number part. Quantum values are defined in the units of qubit arrays that represent potentially entangled qubit clusters and are represented as kind Q n, where n is the array size. Qafny represents qubit arrays as *sessions* (λ), 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, named a qubit group, initialized by an `init` statement, and $[n..m]$ represents the array fragment from position n to m (exclusive) in group 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 10 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 $x[n..m]$.

Each length-n session is associated to a quantum state with the same length that can be one of the three forms (q in Figure 4) that are corresponding to three different types (τ in Figure 4). A CH-type session has state form $\Sigma_{j=0}^{m} z_j \lvert c_j\ \beta_j\rangle$, which is the most general quantum state representation and is designed as an analogy to classical arrays in the sense that each quantum state basis is viewed as an individual array element that does not Intervene with other elements. For example, In Figure 2 line 10, session $\{\{x[0..n], y[0..n]\}$ has state $\Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} \lvert(\lfloor j \rfloor).(\lfloor a^j \ \% \ N \rfloor)\rangle$, represented as a 2^n

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

Fig. 5. Core QAFNY syntax. ©QASM is in Appendix A. For an operator OP, $\texttt{OP}^{[-1]}$ indicates that the operator has a built-in inverse available. Arithmetic expressions in e are only used for classical operations, while Boolean expressions are used for both classical and quantum operations. $x[a]$ represents the a-th element in the qubit array x, while a quantum variable x represents the qubit group $x[0..n]$ and n is the length of x.

entanglement array, whose element is a triple of an amplitude $\frac{1}{\sqrt{2^n}}$, a basis $(\!|j|\!).(\!|a^j \% N|\!)$ [4], and an empty β_j term. The state is represented in QAFNY as an array of the form: $\frac{1}{\sqrt{2^n}} |(\!|0|\!).(\!|a^0 \% N|\!)\rangle + ... + \frac{1}{\sqrt{2^n}} |(\!|n-1|\!).(\!|a^{n-1} \% N|\!)\rangle$; thus, applying any quantum unitary operation on the state results in applying the operation on individual basis term $\frac{1}{\sqrt{2^j}} |(\!|j|\!).(\!|a^j \% N|\!)\rangle$ in the state, which indicates that a unitary operation on a quantum state is analogized to an array map function.

The state representation of the other two typesNor and Had is an array of qubits rather than a basis state in the CH type state. A Nor type state has the form $|c\rangle$ which is a qubit array, each element of which is 0 or 1. The $A(x)$ predicate in Figure 2 line 1 means that an n-size session $x[0..n]$ has Nor type state $|\bar{0}\rangle$ with n qubits and all are bit 0. A Had type state has the form $\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$, which represents an n-length array of qubits that are in superposition but not entangled. We represent the state based on qubit elements for these two types because the split and join of a session is as simple as the split and join of an array, while the join of two CH type sessions are computing the Cartesian product and the split of a CH type state is disentanglement, a very hard problem. In many quantum algorithms, the entangled qubit establishment is to add a single superposition qubit to an existing CH type session. For example, in the loop body in line 8, each iteration grows the entanglement session $\{x[0..j], y[0..n]\}$ to $\{x[0..j+1], y[0..n]\}$ by adding a Had type qubit $\{x[j-1..j]\}$ that is split from the Hadtype session $\{x[0..j]\}$. In this procedure, the elements CH state are doubled, which will be explained shortly.

Language Syntax. 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 map functions, instead of dealing with quantum circuit gates in many other languages. For example, line 3 in Figure 2, we prepare a state by applying a series of H to each qubit in group x. In an iteration in line 8, for the session $\{\{x[0..n], y[0..n]\}$ having CH type, the operation first selects the basis states where the $x[j]$ qubit basis being 1, then applies the modulo multiplication $a^{2^j} y \% N$ only to these basis states. 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 ($\texttt{let}\ x\ =\ e\ \texttt{in}\ s$) in the first row introduces a new variable x with its initial value defined e and used in s. If e is an arithmetic expression (a), it introduces a C or M classical variable. For simplicity, we assume that M arithmetic operations manipulates the nat number parts, so that $(r, n_1) + n_2 = (r, n_1 + n_2)$; and

[4] $(\!|j|\!).(\!|a^j \% N|\!)$ can be split into $(\!|j|\!)$ and $(\!|a^j \% N|\!)$ – both are n-length bitstrings and the the basis states for the n-length group $x[0..n]$ and $y[0..n]$, respectively.

we only interacts a M value with a C one in an arithmetic operation, i.e., the $(r_1, n_1) + (r_2, n_2)$ is disallowed in Qafny . let x = init a in s initializes an a-length qubit group named x with the value $|\overline{0}\rangle$ (a number of bit 0) and is used in statement s, while let x = measure(y) in s measures qubit group y, stores the result in M-mode variable x, and is used in s. The measurement semantic definition relies on a ghost expression ret, and it turns measure(y) to ret(y, (r, n)), which does not appear in a Qafny source program but appears during semantic evaluation, and it records the intermediate measurement result of group y as (r, n).

The last three operations in first row are the quantum data-flow operations. $l \leftarrow op$ is a quantum state preparation operation that prepares superposition of quantum qubits l through Hadamard gates H or QFT gates. It is also used to transform quantum qubit states by a $\texttt{QFT}^{-1}$ gate in the end of the quantum phase estimation algorithm. We only permit op to be state preparation gates such as H and $\texttt{QFT}^{[-1]}$ gates. The other gate applications are done through $\lambda \leftarrow \mu$ that performs quantum oracle computation μ on each basis state of a CH type session λ, such as quantum arithmetic operation. While let operation only performs classical arithmetic computation, quantum oracle arithmetic operation is performed through QASM expressions [Li et al. 2021] (also in Appendix A), which can be used to define almost all reversible arithmetic operations such as the ones in Figure 5. For example, the Shor's algorithm implementation in Figure 2 utilizes the addition (line 5) and modulo multiplication (line 8) on qubit group y, which can be expressed as an QASM circuit. $l \leftarrow$ 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. Details are in Appendix A.

The second row of statements in Figure 5 are control-flow operations. s_1 ; s_2 is a sequential operation. if (b) s is a classical or quantum conditional depending on if b contains quantum parameter. If b is quantum, it must be reversible as quantum gate applications are essentially unitary matrix operations. The reversibility requires that a Boolean equality and inequality expression to be written as $(a_1 = a_2)@x[a]$ and $(a_1 < a_2)@x[a]$, respectively; where an additional bit $x[a]$ is required to hold the result of computing $a_1 = a_2$ or $a_1 < a_2$. for (int $j \in [a_1, a_2)$ && b) s is a possibly quantum for-loop depending on if b contains quantum parameters. A classical variable j is introduced and it is initialized as the lower bound a_1, increments in each loop step defined by $++j$, and ends at the upper bound a_2. For example, line 7-9 in Figure 2 uses a for-loop to repeatably entangle the $x[j]$ qubit with the entangled qubit session $\{x[0..j], y[0..n]\}$ with the application of the modulo multiplication at line 8. 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.

Proofs in Qafny . 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 classical array map functions. Thus, the prototype of designing the QNP proof system is the array application rule in classical Hoare Logic [Gordon 2012], written as:

$$\{P[\wp(\lambda)/\lambda]\}\lambda \leftarrow \wp\{P\}$$

We analogize array variable as the quantum state sessions that are represented arrays here. $\wp(\lambda)$ is the result of applying an operation $\wp$ to every element in λ. For example, in Figure 2 line 3, the post-condition of apply the H gate is $x[0..n] \mapsto C * A(y) * B$, so that the pre-condition in line 2 is $\|\mathsf{H}\|(x[0..n]) \mapsto C * A(y) * B$, by replacing $x[0..n]$ with $\|\mathsf{H}\|(x[0..n])$. The state $A(x) \equiv x[0..n] \mapsto |\overline{0}\rangle$ in line 1 can be rewritten to $\|\mathsf{H}\|(x[0..n]) \mapsto C$, because if we apply $\|\mathsf{H}\|$ on the both side, it becomes:

$$\|\mathsf{H}\|(x[0..n]) \mapsto \|\mathsf{H}\|(|\overline{0}\rangle) = \tfrac{1}{\sqrt{2^n}} \otimes_{j=0}^{n} (|0\rangle + |1\rangle) = C \qquad \texttt{where } \|\mathsf{H}\|(|0\rangle) = \tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

The right hand side formula is a typical superposition state preparation by a Hadamard gate in quantum computation. In Qafny , we are using the separating conjunction operation $*$ from separation logic [Reynolds 2002], where $\varphi \cup \varphi' \models P * Q$ iff $\varphi \perp \varphi'$ and $\varphi \models P$ and $\varphi' \models Q$. Here, φ and φ' are finite quantum states mapping from sessions to the three forms above.

In mapping Qafny rules to classical array proof rules, there are three subtleties. First, sessions are both indicators, which are viewed as array structures that record qubit positions, for entangled qubit groups; as well as variables, which are viewed as names, representing qubit states in pre- and post-conditions. In preserving the two session roles, we design a type system in Appendix A to track transitions of sessions and require that a well-formed pre-condition only mentions the sessions defined in a given type environment. Second, we also define equational properties, representing the permutation symmetries of quantum states, for sessions and quantum states mapping from sessions to the three forms above. This permits the use of ideal quantum state form in proving a statement. For example, in Figure 2 line 10, the session $\{\{x[0..n], y[0..n]\}$ has the state $\Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} |(|j|).(|a^j \% N|)\rangle$, which can also be equivalently formulated as a session $\{\{y[0..n], x[0..n]\}$ having the state $\Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} |(|a^j \% N|).(|j|)\rangle$ by switching the x and y positions. Details are in Section 3.1.

Third, line 7-9 describes the transition for a for-loop based on a series of quantum conditionals. In each iteration, a Had type qubit $x[j]$ is joined into the entangled session $\{x[0..j], y[0..n]\}$, so that the session is extended to $\{x[0..\mathtt{S}\ j], y[0..n]\}$ and the basis state number is doubled. Then, the conditional body modulo multiplication in line 8 is applied to exactly half of the basis states whose $x[j]$ position's bit value is 1, while the others are preserved to be the same. There are two mysterious points in the above scenario. The first one is the quantum state split and join operations that can be captured by additional equational properties in Section 3.1. The second point is the proof rule for a quantum conditional that represents the partial map behavior where operations in the conditional body are only applied to some of the basis states. Details are in Appendix C.1.

Fourth, line 11 is a partial measurement operation. Given a quantum state φ, a measurement is the only operation that modifies φ's domain and it cannot be placed inside a quantum conditional; otherwise, the system leaks quantum information during a quantum computation, a violation of a quantum information principle. More details are in Appendix C.2.

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

Here, we first introduce Qafny state equational properties and type system. Then, we discuss its semantics and proof system.

3.1 State Equivalence, Type, and Sequence Rules

As we suggested in Section 2, 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 Section 2, we need quantum sessions to be split and regrouped sometimes. All these properties are formulated in Qafny as equational properties in Figure 35 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, as well as the three state types in every transition step, so that the system knows exactly how to apply an equation.

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

(a) Subtyping

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

(b) Quantum Value Equivalence

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

(c) Session Equivalence

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

(d) Environment Equivalence

(e) State Equivalence

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

Fig. 6. 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 $\frac{1}{\sqrt{2^{n+m}}} \bigotimes_{j=0}^{n+m} q_j$.

State Equivalence. Figure 35 shows the equivalence relations on types and states. Figure 35a 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 35b, 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 6c 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 Section 2. The first rule describes the case for empty session, while the second rule in Figure 6e connects the quantum value equivalence to the state equivalence. The third rule describes the qubit permutation equivalence by the mut function. The fourth rule

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

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

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

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

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

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

$$\sigma[\uparrow \sigma'] = \sigma[\forall \lambda : \tau \in \sigma'\ .\ \tau/\lambda] \qquad \sigma|_{\notin \mathtt{dom}(\sigma')} = \{\lambda : \tau | \lambda \notin \mathtt{dom}(\sigma')\}$$

Fig. 7. Qafny type system. $\sigma(y) = \{\lambda \mapsto \tau\}$ produces the map entry $\lambda \mapsto \tau$ and the range $y[0..|y|]$ is in λ. $\sigma(\lambda) = \tau$ is an abbreviation of $\sigma(\lambda) = \{\lambda \mapsto \tau\}$. $FV(\Omega, -)$ produces a session by union all qubits appearing in $-$ with the qubit group info in Ω; see Appendix B.1.

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.

Type Checking: A Quantum Session Type System. In Qafny , typing is with respect to a *kind environment* Ω and a *finite type environment* σ, which map Qafny variables to kinds and map sessions to types, respectively. The typing judgment is written as $\Omega; \sigma \vdash_g s \triangleright \sigma'$, which states that statements s is well-typed under the context mode g and environments Ω and σ, the sessions representing s is exactly the domain of σ' as $\mathtt{dom}(\sigma')$, and s transforms types for the sessions in σ to types in σ'. Ω describes the kinds for all program variables. Ω is populated through let expressions that introduce variables, and the Qafny type system enforces variable scope; such enforcement is neglected in Figure 37 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 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. Selected type rules are given in Figure 37; 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 is 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 $\lambda \uplus \lambda'$. The μ application preserves the session in CH provided that the input type is turned to CH type through rule TPAR. Third, the type system enforces equational properties of quantum qubit sessions through a partial order relation over type environments, including subtyping, qubit position permutation, merge and split quantum sessions through rule TPAR. 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 36, which casts the left type environment to the format on the right through rule TPAR.

Fourth, the type system enforces that the C classical variables can be evaluated to values in the compilation time. [5], 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. Finally, the type system extracts the result type environment of a for-loop as $\sigma'[n_2/x]$ based on the extraction of a type environment invariant on the i-th loop step of executing a conditional `if` (b) s in rule TLOOP, regardless if the conditional is classical or quantum.

Taste of Semantic and Proof Rules: Sequence and Consequence. We define the small step operational semantics of an QAFNY program as a relation $(\Phi, s) \longrightarrow (\Phi', s')$, where Φ and Φ' are the pre- and post- states.

partial function $[\![\ ]\!]$ from an instruction ι and input state φ to an output state φ', written $[\![\iota]\!]\varphi = \varphi'$, shown in Figure 29.

State Preparation and Oracle Application Rules. The QAFNY state preparation $l \leftarrow op$ and oracle application $\lambda \leftarrow \mu$ operations are analogized to classical array map operation as indicated by the image on top of Figure 9, in which we have a session $\lambda \uplus \lambda'$, for each element $z_j\ |c_{j1}.c_{j2}\ \beta_j\rangle$ in the CH type state, managed as an array having the form $\Sigma_{j=0}^{m}\ z_j\ |c_{j1}.c_{j2}\ \beta_j\rangle$ and c_{j1} are the basis state for the λ positions, we apply μ on the c_{j1} part, which is described in the semantic rule SA-CH. Rule PA-CH describes the proof rule for capturing the array map analogy, where we substitute session $\lambda \uplus \lambda'$, representing the state before applying μ, with the application of $\eth\ \lambda.[\![\mu]\!]$ on every element of $\lambda \uplus \lambda'$. For example, in the loop body in Figure 2 line 8, we apply $y \leftarrow a^{2^j}\ y\ \%\ N$ to a state $\{y[0..n]\} \mapsto \Sigma_{j=0}^{2^{k-1}}\ \frac{1}{\sqrt{2^{k-1}}}\ |(|a^{(|j|)}\ \%\ N|)\ |\ (|j|).1\rangle$ [6]. The result is $\Sigma_{j=0}^{2^{k-1}}\ \frac{1}{\sqrt{2^{k-1}}}\ |(|a^{(|j|).1}\ \%\ N|)\ |\ (|j|).1\rangle$ [7].

Apparently, state preparation operations $l \leftarrow op$ can also be described by an array map operation. The only tricky case is that applying an H or QFT gate in a basis state might result in generating more elements because of state preparations essentially turns a Nor state to become superposition. Here, we list the semantics and proof rules SH-N and PH-N for the case when dealing with Nor states. The other cases are listed in Appendix B.

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

[6] 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 $x[0..j+1]$.

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

Predicate modeling:

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

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

Sequence Semantic and Proof Rules:

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

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

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

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

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

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

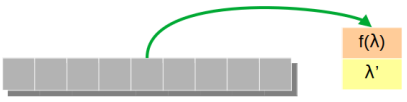
$$(\Omega,\sigma,P) \Rightarrow (\Omega,\sigma',P') \triangleq \Omega,\sigma \vdash P \wedge \Omega,\sigma' \vdash P' \wedge \sigma \preceq \sigma' \wedge P \Rightarrow P'$$
$$(\Omega,\sigma,Q) \twoheadrightarrow (\Omega,\sigma',Q') \triangleq \Omega,\sigma \vdash Q \wedge \Omega,\sigma' \vdash Q' \wedge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

Fig. 8. Sequence and Consequence Rules

Rules for Conditionals and For-Loops. Figure 10a 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 $\lambda_b \uplus \lambda_a$, it masks the λ_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 $|c_{j1}| = |\lambda_b|$, we mask the c_{j1} part by pushing it to a masked stack as $\Sigma_{j=0}^{m} z_j\,|c_{j2}\,|\,c_{j1}\,\beta_j\rangle$, which is described in preparing the pre-state of the upper-level transition in rule SIF (Figure 10c). 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 $\Sigma_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\,\beta_j\rangle + q(\lambda, \neg b)$, where the first part are all basis states satisfying b since if we replace the variables in b with c_{j1}, the evaluation $[\![b[c_{j1}/\lambda]]\!]$ returns true, while the second part $q(\lambda, \neg b)$ 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 $\Sigma_{j=0}^{m'} z'_j\,|c_{j1}.c'_{j2}\,\beta_j\rangle + q(\lambda, \neg b)$. 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 10b), 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 $\Sigma_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\,\beta_j\rangle + q(\lambda, \neg b)$, we preserve the basis states satisfying b, as shown in the predicate R, remove the unsatisfied basis states $q(\lambda, \neg b)$, 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 10c), we substitute $\lambda \uplus \lambda'$ with $\mathcal{M}(b, \lambda')$. During the process, the type for the session in σ is changed

(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

SA-CH

$$\frac{\varphi(\lambda) = \{\lambda \uplus \lambda' \mapsto \sum_{j=0}^{m} z_j\,|c_{j1}.c_{j2}\,\beta_j\rangle\} \qquad \forall j.\ |c_{j1}| = |\lambda| \wedge [\![\mu]\!]c_{j1} = z'_j\,|c'_{j1}\rangle}{(\varphi, \lambda \leftarrow \mu) \longrightarrow (\varphi[\lambda \uplus \lambda' \mapsto \sum_{j=0}^{m} z'_j \cdot z_j\,|c'_{j1}.c_{j2}\,\beta_j\rangle], \{\})}$$

PA-CH

$$\frac{\sigma(\lambda) = \{\lambda \uplus \lambda' \mapsto \mathsf{CH}\}}{\Omega;\sigma \vdash_g \{P[\eth\ \lambda.[\![\mu]\!](\lambda \uplus \lambda')/\lambda \uplus \lambda']\}\ \lambda \leftarrow \mu\ \{P\}}$$

SH-N

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

PH-N

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

Fig. 9. Oracle application and state preparation rules. $\eth$ is an array map operation, where $\eth\ \lambda.[\![\mu]\!](\lambda \uplus \lambda')$ means that for every basis state in the state of $\lambda \uplus \lambda'$, we apply $[\![\mu]\!]$ to λ part of session.

from $\lambda \uplus \lambda'$ in the bottom to λ' in the upper level. The unmask function $\mathcal{U}$ assembles the result state of applying s to the masked λ state with the other parts hidden in the unmasked state (the part marked black in Figure 10a). Function $\mathcal{U}$ is usually appeared to be a pair with both b and $\neg b$. In the post-condition manipulation, we substitute $\lambda \uplus \lambda'$ with $\mathcal{U}(\neg b, \lambda \uplus \lambda')$ in P, representing the unchanged and masked state, substitute λ' with $\mathcal{U}(b, \lambda \uplus \lambda')$ in Q', representing the result of applying s on the unmasked part, and assemble them together through the separation operation $*$. $\mathcal{U}$'s modeling in Figure 10b 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.

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

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

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

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

(a) Conditional Analogy

(b) Mask/Unmask Function Modeling

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

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

(c) Semantic/Proof Rules

SIF

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

PIF

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

SLOOP

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

PLOOP

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

$\theta := q \mid \lambda$

$q(\lambda,\neg b) = \Sigma_{k=0}^{m} z_k \,|c_{k1}.c_{k2}\,\beta_k\rangle$ where $\forall k.\ |c_{k1}| = |\lambda| \wedge [\![\neg b[c_{k1}/\lambda]]\!]$

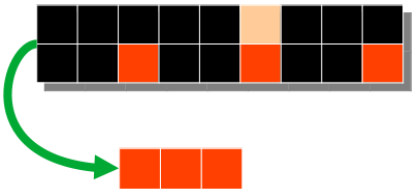
$R = FV(\emptyset,b) = \lambda \wedge \forall j.\ |c_{j1}| = |\lambda| \wedge [\![b[c_{j1}/\lambda]]\!]$

Fig. 10. 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.

The proof is built from bottom up. We first cut the Had type state into two sessions ($x[0..n - \mathsf{S}\ j]$ and $x[j]$), join $x[j]$ with session $\{x[0..j], y[0..n]\}$, and double the state elements to be $0.C(j) + 1.C(j)$, which is proved by applying the consequence rules. Notice that the type environment is also transitioned from σ to σ_1. By the same strategy of the $\mathcal{U}$ rule in Figure 10b, we combine the two $\mathcal{U}$ terms into the final result. The second step applies rule PIF to substitute session $\{x[0..\mathsf{S}\ j], y[0..n]\}$ with the mask construct $\mathcal{M}(b, \{y[0..n]\})$ in the pre-condition and create two $\mathcal{U}$ terms in the post-condition. The step on the top applies the modulo multiplication on every element in the masked state $\mapsto C(j).1$ by rule PA-CH.

Rules for Measurement. As Figure 11a 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

(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

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

PMEA

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

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

Fig. 11. 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.

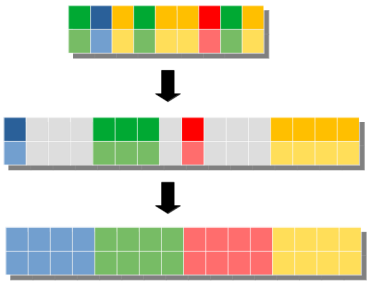
all the first parts in the old state and collecting the elements whose first part is equal to the key. The second step actually collects elements in a periodical manner as shown in the analogy in Figure 11a, where the marked red basis states appear in a periodical pattern in the whole state. This behavior is universally true for quantum operations, and many quantum algorithms utilize the periodical pattern of quantum computation.

In rule SMEA, we pick an n-length bitstring c as the marked pink key, and elect m basis states $\Sigma_{j=0}^{m} z_j |c.c_j\rangle$ that has the key c. In the post-state, we update the remaining session λ to $\Sigma_{j=0}^{m} \frac{z_j}{\sqrt{r}} |c_j\rangle$ with the adjustment of amplitude $\frac{1}{\sqrt{r}}$, and replace the variable x in the statement s with the value $(r, \{\!|c|\!\})$. In designing the proof rule PMEA, the operation $\mathcal{F}(x,n,\lambda)$ is invented, whose modeling is in Figure 11b, 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 11c replaces the session $y[0..n] \cup \lambda$ in P with the measurement result session $\mathcal{F}(x,n,\lambda)$ and updates the type state Ω and σ.

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

$C = \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} |(|j|).(|a^j \ \%\ N|)\rangle \quad C' = \Sigma_{j=0}^{2^n} \frac{1}{\sqrt{2^n}} |(|a^j \ \%\ N|).(|j|)\rangle \quad \sigma = \{\{x[0..\texttt{S}\ j], y[0..n]\} : \texttt{CH}\}$
$D = \frac{1}{\sqrt{s}} \Sigma_{k=0}^{s} |t + kp\rangle \quad E = p = \texttt{ord}(a, N) \wedge u = (\frac{s}{2^n}, a^t \ \%\ N) \wedge s = \texttt{rnd}(\frac{2^n}{p})$

(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

SDis
$$\frac{FV(\Omega, l) = \lambda \qquad \varphi(\lambda) = \{\lambda \uplus \lambda' \mapsto q\}}{(\varphi, l \leftarrow \texttt{dis}) \longrightarrow (\varphi[\lambda \uplus \lambda' \mapsto \mathcal{D}(|\lambda|, q)], \{\})}$$

PDis
$$\frac{FV(\Omega, l) = \lambda \qquad \sigma(\lambda) = \{\lambda \uplus \lambda : \texttt{CH}\}}{\Omega;\sigma \vdash_g \{P[\mathcal{D}(|\lambda|, \lambda \uplus \lambda')/\lambda \uplus \lambda']\}\, l \leftarrow \texttt{dis}\, \{P\}}$$

Fig. 12. Semantic and Proof Rules for Diffusion Operations. $\mathcal{D}$ models the diffusion operations.

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 ($l \leftarrow \texttt{dis}$) reorient the amplitudes of basis states based on the basis state corresponding to l. They are analogized to an aggregate operation of reshape and mean computation, both appeared in many programming languages [8]. The aggregate operation first applies a reshape, where elements are regrouped into a normal form, as the first agree of Figure 12a. 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 12b. 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 12a), as the third rule in Figure 12b. 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.

[8] Such as Python

The mean computation analogy (the second arrow in Figure 12a) 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 $l \leftarrow \texttt{dis}$, which applies the $\mathcal{D}$ function to the session $\lambda \uplus \lambda'$, where λ corresponds to the l's session. Proof rule PDIS replaces session $\lambda \uplus \lambda'$ 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 $|\lambda \uplus \lambda'|$, as the whole session. In either case, the summation formula in $\mathcal{D}$'s modeling (Figure 12b) 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 5.

3.2 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.

Definition 3.1 (Well-formed session domain). A type environment σ's session domain is *well-formed*, written $\Omega \vdash \texttt{dom}(\sigma)$, iff for every session $\lambda \in \texttt{dom}(\sigma)$:

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

Definition 3.2 (Well-formed QAFNY state). A state φ is *well-formed*, written $\Omega; \sigma \vdash \varphi$, iff $\texttt{dom}(\sigma) = \texttt{dom}(\varphi)$, $\Omega \vdash \texttt{dom}(\sigma)$, and:

- For every $\lambda \in \sigma$ such that $\sigma(\lambda) = \texttt{Nor}$, $\varphi(\lambda) = |c\rangle$ and $|c| = |\lambda|$.
- For every $\lambda \in \sigma$ such that $\sigma(\lambda) = \texttt{Had}$, $\varphi(\lambda) = \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (|0\rangle + \alpha(r_j)\,|1\rangle)$ and $|\lambda| = n$.
- For every $\lambda \in \sigma$ such that $\sigma(\lambda) = \texttt{CH}$, $\varphi(\lambda) = \sum_{j=0}^{m} z_j\,|c_j\,\beta_j\rangle$ and $|\lambda| = |c_j|$ for all j.

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 3.3. *[QAFNY type progress]*If $\Omega; \sigma \vdash_g s \triangleright \sigma'$ and $\Omega; \sigma \vdash \varphi$, then either $s = \{\}$, or there exists φ' and s' such that $(\varphi, s) \longrightarrow (\varphi', s')$.

THEOREM 3.4. *[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 state representations for a session λ 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 3.5 (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 $(\varphi, s) \longrightarrow^* (\varphi', \{\})$, $\sigma' \preceq \sigma^*$ and $\varphi' \equiv \varphi^*$.

The QAFNY proof system correctness is defined by the soundness and relatively completeness theorems below, which has been formalized and proved in Coq. The QAFNY proof system only describes the quantum portion 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 3.6. *[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 3.7. *[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 $\Omega; \sigma \vdash_g \{P\}\, s\, \{Q\}$.

4 QAFNY PROOF SYSTEM AND PROGRAM COMPILATION

We discuss the QAFNY compilation to the Dafny proof system and the compilation of quantum programs to SQIR, a quantum circuit language.

4.1 Translation from QAFNY to SQIR

VQO translates QASM to SQIR by mapping QASM positions to SQIR concrete qubit indices and expanding QASM instructions to sequences of SQIR gates. Translation is expressed as the judgment $\Sigma \vdash (\gamma, \iota) \longrightarrow (\gamma', \epsilon)$ where Σ maps QASM variables to their sizes, ϵ is the output SQIR circuit, and γ maps an QASM position p to a SQIR concrete qubit index (i.e., offset into a global qubit register). At the start of translation, for every variable x and $i < \Sigma(x)$, γ maps (x, i) to a unique concrete index chosen from 0 to $\sum_x(\Sigma(x))$.

Figure 13 depicts a selection of translation rules.[9] The first rule shows how to translate `X` p, which has a directly corresponding gate in SQIR. The second rule left-shifts the qubits of the target variable in the map γ, and produces an identity gate (which will be removed in a subsequent optimization pass). For example, say we have variables x and y in the map γ and variable x has three qubits so γ is $\{(x, 0) \mapsto 0, (x, 1) \mapsto 1, (x, 2) \mapsto 2, (y, 0) \mapsto 3, ...\}$. Then after `Lshift` x the γ map becomes $\{(x, 0) \mapsto 1, (x, 1) \mapsto 2, (x, 2) \mapsto 0, (y, 0) \mapsto 3, ...\}$. The last two rules translate the

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

$$\frac{}{\Sigma \vdash (\gamma, \texttt{X}\ p) \to (\gamma, \texttt{X}\ \gamma(p))} \qquad \frac{\gamma' = \gamma[\forall i.\ i < \Sigma(x) \Rightarrow (x,i) \mapsto \gamma(x,(i+1)\%\Sigma(x))]}{\Sigma \vdash (\gamma, \texttt{Lshift}\ x) \to (\gamma', \texttt{ID}\ (\gamma'(x,0)))}$$

$$\frac{\Sigma \vdash (\gamma, \iota) \to (\gamma, \epsilon) \qquad \epsilon' = \texttt{ctrl}(\gamma(p), \epsilon)}{\Sigma \vdash (\gamma, \texttt{CU}\ p\ \iota) \to (\gamma, \epsilon')} \qquad \frac{\Sigma \vdash (\gamma, \iota_1) \to (\gamma', \epsilon_1) \qquad \Sigma \vdash (\gamma', \iota_2) \to (\gamma'', \epsilon_2)}{\Sigma \vdash (\gamma, \iota_1\ ;\ \iota_2) \to (\gamma'', \epsilon_1\ ;\ \epsilon_2)}$$

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

CU and sequencing instructions. In the CU 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 CU in Figure 32. `ctrl` generates the controlled version of an arbitrary SQIR program using standard decompositions [Nielsen and Chuang 2011, Chapter 4.3].

We have proved 𝒪QASM-to-SQIR translation correct. To formally state the correctness property we relate d-qubit 𝒪QASM states to SQIR states, which are vectors of 2^d complex numbers, via a function $[\![-]\!]^d_\gamma$, where γ is the virtual-to-physical qubit map. For example, say that our program uses two variables, x and y, and both have two qubits. The qubit states are $|0\rangle$ and $|1\rangle$ (meaning that x has type Nor), and $|\Phi(r_1)\rangle$ and $|\Phi(r_2)\rangle$ (meaning that y has type Phi). Furthermore, say that $\gamma = \{(x,0) \mapsto 0, (x,1) \mapsto 1, (y,0) \mapsto 2, (y,1) \mapsto 3\}$. This 𝒪QASM program state will be mapped to the 2^4-element vector $|0\rangle \otimes |1\rangle \otimes (|0\rangle + e^{2\pi i r_1}|1\rangle) \otimes (|0\rangle + e^{2\pi i r_2}|1\rangle)$.

Theorem 4.1. [𝒪QASM translation correctness] Suppose $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ and $\Sigma \vdash (\gamma, \iota) \to (\gamma', \epsilon)$. Then for $\Sigma; \Omega \vdash \varphi$ and $[\![\iota]\!]\varphi = \varphi'$, we have $[\![\epsilon]\!] \times [\![\varphi]\!]^d_\gamma = [\![\varphi']\!]^d_{\gamma'}$ where $[\![\epsilon]\!]$ is the matrix interpretation of ϵ per SQIR's semantics.

The proof of translation correctness is by induction on the 𝒪QASM program ι. Most of the proof simply shows the correspondence of operations in ι to their translated-to gates ϵ in SQIR, except for shifting operations, which update the virtual-to-physical map.

Note that to link a complete, translated oracle ι into a larger SQIR program may require that $\gamma = \gamma'$, i.e., `neutral`(ι), so that logical inputs match logical outputs. This requirement is naturally met for programs written to be reversible, as is the case for all arithmetic circuits in this paper, e.g., `rz_adder` from Figure 23.

Compilation for 𝒪QIMP statements is a partial function of the form $\Xi; \Gamma; \Theta \vdash (n * \sigma * s) \rightsquigarrow \kappa$, where:

- Ξ is a map from function variables to their definitions;
- Γ is the standard type environment;
- Θ is a map from 𝒪QIMP variables to 𝒪QASM variables;
- n is the current amount of scratch space (i.e., number of ancilla qubits) needed;
- σ is a dynamic store that records the values for all C-mode variables;
- s is an 𝒪QIMP statement;
- and κ is the result of compilation, which is either `Error` or a value (n, σ, u), where n is the final amount of scratch space, σ is the resulting store for C-mode variables, and u is the generated 𝒪QASM circuit.

Ξ and Γ are generated during type checking and the amount of scratch space (n) is initialized to 0.

Compilation assumes (correct) 𝒪QASM implementations of primitive arithmetic operators. An addition operation $l \overset{+}{\leftarrow} v$ in 𝒪QIMP will be compiled to an 𝒪QASM program `add`(l, v) that will either apply a constant or general adder from Section 5, depending on whether v has mode C or Q (if both l and v have mode C, then the result is precomputed). Compilation expects two global settings: flag

$$\frac{\Gamma \vdash l : \omega^Q \quad \Gamma \vdash v : \omega^Q \quad u = \texttt{get\_op}(op)(fl, \Theta(l), \Theta(v), sz)}{\Xi; \Gamma; \Theta \vdash (n, \sigma, l \xleftarrow{op} v) \rightsquigarrow (n, \sigma, u)}\texttt{(bin\_q)}$$

$$\frac{\Gamma \vdash l : \omega^C \quad \Gamma \vdash v : \omega^C \quad \Xi \vdash_s \sigma; l \xleftarrow{op} v \longrightarrow \sigma'}{\Xi; \Gamma; \Theta \vdash (n, \sigma, l \xleftarrow{op} v) \rightsquigarrow (n, \sigma', \texttt{ID}\ (\Theta(l), 0))}\texttt{(bin\_c)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{bool}^C \quad \Xi \vdash \sigma; e \longrightarrow (\texttt{bool})\texttt{true} \\ \Xi; \Gamma; \Theta \vdash (n, \sigma, s_1) \rightsquigarrow (n', \sigma', u)\end{array}}{\Xi; \Gamma; \Theta; \vdash (n, \sigma, es_1s_2) \rightsquigarrow (n', \sigma', u)}\texttt{(if\_c)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{bool}^Q \quad \Xi; \Gamma; \Theta \vdash (n, \sigma, e) \rightsquigarrow (n_e, \sigma_e, u_e) \quad \Xi; \Gamma; \Theta \vdash (n_e, \sigma_e, s_1) \rightsquigarrow (n_1, \sigma_1, u_1) \\ \Xi; \Gamma; \Theta \vdash (n_1, \sigma_e, s_2) \rightsquigarrow (n_2, \sigma_2, u_2) \quad u' = u_e\ ;\ \texttt{CU}\ (\chi, n)\ u_1\ ;\ \texttt{X}\ (\chi, n)\ ;\ \texttt{CU}\ (\chi, n)\ u_2\end{array}}{\Xi; \Gamma; \Theta \vdash (n, \sigma, es_1s_2) \rightsquigarrow (n_2, \sigma_e, u')}\texttt{(if\_q)}$$

$$\frac{\begin{array}{c}\Xi(f) = (\overline{\tau\ x}, \overline{\tau\ y}, s, l', \Gamma') \quad \Theta' = \texttt{add\_}\Theta^Q(\Theta, \Gamma', \overline{\tau\ y}) \quad \Gamma' \vdash l' : \omega^Q \quad \Gamma \vdash l : \omega^Q \\ \sigma' = \texttt{init\_}\sigma^C(\overline{\tau\ x}, \overline{v}, \overline{\tau\ y}) \qquad \Xi; \Gamma'; \Theta' \vdash (n, \sigma', s) \rightsquigarrow (n', \sigma'', u) \\ u' = u\ ;\ \texttt{copy}(\Theta'(l'), \Theta(l))\ ;\ \texttt{qinv}(u)\end{array}}{\Xi; \Gamma; \Theta \vdash (n, \sigma, l \leftarrow f\ \overline{v}) \rightsquigarrow (n, \sigma, u')}\texttt{(call)}$$

Fig. 14. Select 𝒪QIMP to 𝒪QASM compilation rules

$fl \in \{\texttt{Classical}, \texttt{QFT}\}$ indicates whether 𝒪QASM operators should use Toffoli-based or QFT-based arithmetic, and size $sz \in \mathbb{N}$ fixes the bit size of operations.

?? provides a select set of compilation rules from 𝒪QIMP to 𝒪QASM. The first two rules compile an assignment operation. If l and v are both typed as Q-mode, we compile the assignment to a pre-defined 𝒪QASM program, which we look up using `get_op`. For example, if op is an addition, `get_op`(op) produces the 𝒪QASM program `add`, described above. The rule for the case where one variable has mode Q and the other variables has mode C is similar. On the other hand, if l and v are both typed as C-mode, we update the store σ by computing the addition directly using the 𝒪QIMP semantics (**??**) and generate the trivial 𝒪QASM program `ID`.

The rules `(if_c)` and `(if_q)` are for branching operations. If the Boolean guard has mode C, we evaluate the expression to its value and choose one of the branches for further compilation; `(if_c)` shows the case where e evaluates to `true`. If the Boolean guard has mode Q, we generate an 𝒪QASM expression to compute the guard and store the result in (χ, n), where χ is the scratch space variable and n is the current scratch space index. We then compile the two branches conditioned on (χ, n), as shown in rule `(if_q)`.

Rule `(call)` compiles a function call. `add_`Θ^Q extends the 𝒪QIMP-to-𝒪QASM variable map with the new Q-mode variables in the local declaration list $\overline{\tau\ y}$. The 𝒪QIMP type system requires that all function arguments ($\overline{\tau\ x}$) have mode C. `init_`σ^C initializes the values of all variables in $\overline{\tau\ x}$ to their corresponding values in $\overline{v}$ and all C-mode variables in $\overline{\tau\ y}$ to 0. `copy`(x, y) is an 𝒪QASM program that copies the states of all qubits in x to y using a series of `CU` p_x `X` p_y operations. In the case where l' (the return value of f) has mode C, we do not need to generate a circuit for the function f; instead, we just generate a circuit to set l to the value of l'. If l also has mode C, we just update l's in σ without generating any circuit.

We prove that compilation is correct: Given an 𝒪QIMP program P that compiles to an 𝒪QASM circuit C, evaluating P according to the 𝒪QIMP semantics will produce a value consistent with evaluating C according to the 𝒪QASM semantics. The proof is mechanized in Coq and proceeds

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

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

Fig. 15. Second half of the Shor's algorithm quantum part in Qafny.

by induction on the compilation judgment, relying on proofs of correctness for 𝒪QASM arithmetic operators, as discussed in Section 4.1.

5 EVALUATION: ARITHMETIC OPERATORS IN 𝒪QASM

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.

5.1 Implemented Operators

Figure 16 and Figure 17 tabulate the arithmetic operators we have implemented in 𝒪QASM.

The addition and modular multiplication circuits (parts (a) and (d) of Figure 16) 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 Figure 15. 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.

Part (b) of Figure 16 shows results for 𝒪QASM implementations of multiplication (without the modulo) and part (c) shows results for modular division by a constant, which is useful in Taylor series expansions used to implement operators like sine and cosine. Figure 17 lists additional operations we have implemented in 𝒪QASM for arithmetic and Boolean comparison using natural and fixed-precision numbers.

5.2 Validating Operator Correctness

As shown in Figure 16, we have fully verified the adders and modular multipliers used in Shor's algorithm. These constitute the first proved-correct implementations of these functions, as far as we are aware.

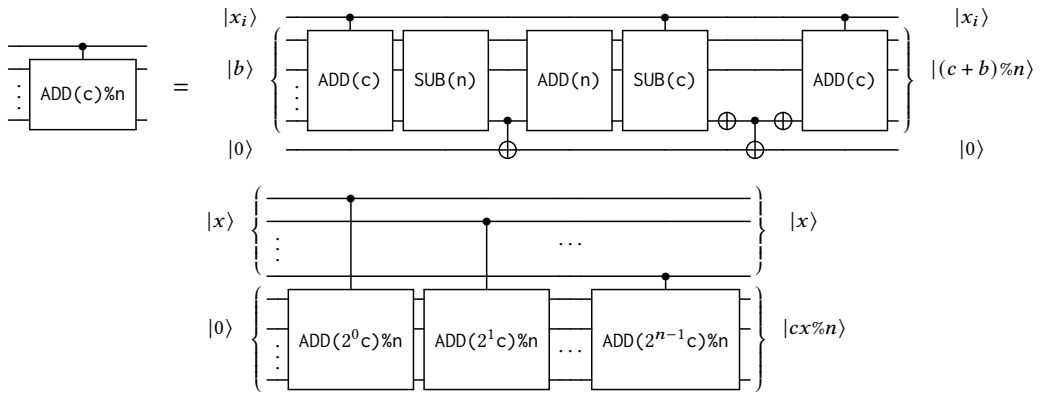


Fig. 16. Structure of modular multiplication circuits

	# qubits	# gates	Verified
⟮QASM TOFF	33	423	✓
⟮QASM QFT	32	1206	✓
⟮QASM QFT (const)	16	756 ± 42	✓
Quipper TOFF	47	768	
Quipper QFT	33	6868	
Quipper TOFF (const)	31	365 ± 11	

(a) Addition circuits (16 bits)

	# qubits	# gates	QC time (16 / 60 bits)
⟮QASM TOFF	49	11265	6 / 74
⟮QASM TOFF (const)	33	1739 ± 367	3 / 31
⟮QASM QFT	48	4339	4 / 138
⟮QASM QFT (const)	32	1372 ± 26	4 / 158
Quipper TOFF	63	8060	
Quipper TOFF (const)	41	2870 ± 594	

(b) Multiplication circuits (16 bits)

	# qubits	# gates	QC time (16 / 60 bits)
⟮QASM TOFF (const)	49	28768	16 / 397
⟮QASM QFT (const)	34	15288	5 / 412
⟮QASM AQFT (const)	34	5948	4 / 323
Quipper TOFF	98	37737	

(c) Division/modulo circuits (16 bits)

	# qubits	# gates	Verified
⟮QASM TOFF (const)	41	56160	✓
⟮QASM QFT (const)	19	18503	✓

(d) Modular multiplication circuits (8 bits)

Fig. 17. Comparison of ⟮QASM and Quipper arithmetic operations. In the "const" case, one argument is a classically-known constant parameter. For (a)-(b) we present the average (± standard deviation) over 20 randomly selected constants c with $0 < c < 2^{16}$. For the division/modulo circuits x mod n, we only consider the gate counts for the maximum iteration case when $n = 1$; the Quipper version assumes n is a variable, but they use the same algorithm as us by guessing a maximum iteration number for n. In (d), we use the constant 255 ($= 2^8 - 1$) for the modulus and set the other constant to 173 (which is invertible mod 255). Quipper supports no QFT-based circuits aside from an adder. "QC time" is the time (in seconds) for QuickChick to run 10,000 tests.

All other operations in the figure were tested with QuickChick. To ensure these tests were efficacious, we confirmed they could find hand-injected bugs; e.g., we reversed the input bitstrings for the QFT adder (Figure 23) and confirmed that testing found the endianness bug. The tables in Figure 16 give the running times for the QuickChick tests—the times include the cost of extracting the Coq code to OCaml, compiling it, and running it with 10,000 randomly generated inputs. We tested these operations both on 16-bit inputs (the number that's relevant to the reported qubit and

type	Verified	Randomly Tested
Nat /Bool	$[x{-}N]_q$ $[N{-}x]_q$ $[x{-}y]_{q,t}$ $[x{=}N]_{q,t}$ $[x{<}N]_{q,t}$ $[x{=}y]_{q,t}$ $[x{<}y]_{q,t}$	$[x{+}N]_a$ $[x{+}y]_a$ $[x{-}N]_t$ $[N{-}x]_t$ $[x{-}y]_q$ $[x\%N]_{a,q,t}$ $[x/N]_{a,q,t}$
FixedP	$[x{+}N]_q$ $[x{+}y]_t$ $[x{-}N]_q$ $[N{-}x]_q$ $[x{-}y]_t$ $[x{=}N]_{q,t}$ $[x{<}N]_{q,t}$ $[x{=}y]_{q,t}$ $[x{<}y]_{q,t}$	$[x{+}N]_t$ $[x{+}y]_q$ $[x{-}N]_t$ $[N{-}x]_t$ $[x{-}y]_q$ $[x{*}N]_{q,t}$ $[x{*}y]_{q,t}$ $[x/N]_{q,t}$

x, y = variables, N = constant,
$[\,]_{a,q,t}$ = AQFT-based (a), QFT-based (q), or Toffoli-based (t)

All testing is done with 16-bit/60-bit circuits.

Fig. 18. Other verified & tested operations

gate sizes) and 60-bit inputs. For the smaller sizes, tests complete in a few seconds; for the larger sizes, in a few minutes. For comparison, we translated the operators' ℚASM programs to SQIR, converted the SQIR programs to OpenQASM 2.0 [Cross et al. 2017], and then attempted to simulate the resulting circuits on test inputs using the DDSim [Burgholzer et al. 2021], a state-of-the-art quantum simulator. Unsurprisingly, the simulation of the 60-bit versions did not complete when running overnight.

We also verified and property-tested several other operations, as shown in Figure 17.

During development, we found two bugs in the original presentation of the QFT-based modular multiplier [Beauregard 2003]. The first issue was discovered via random testing and relates to assumptions about the endianness of stored integers. The binary number in Figure 6 of the paper uses a little-endian format whereas the rest of the circuit assumes big-endian. Quipper's implementation of this algorithm solves the problem by creating a function in their Haskell compiler to reverse the order of qubits. In ℚASM, we can use the `Rev` operation (which does not insert `SWAP`s) to correct the format of the input binary number.

The second issue was discovered during verification. Beauregard [2003] indicates that the input x should be less than 2^n where n is the number of bits. However, to avoid failure the input must *actually* be less than N, where N is the modulus defined in Shor's algorithm. To complete the proof of correctness, we needed to insert a preprocessing step to change the input to $x\%N$. The original on-paper implementation of the ripple-carry-based modular multiplier [Markov and Saeedi 2012] has the same issue.

5.3 Operator Resource Usage

Figure 16 compares the resources used by ℚASM operators with counterparts in Quipper. In both cases, we compiled the operators to OpenQASM 2.0 circuits,[10] and then ran the circuits through the VOQC optimizer [Hietala et al. 2021b] to ensure that the outputs account for inefficiencies in automatically-generated circuit programs (e.g., no-op gates inserted in the base case of a recursive function). VOQC outputs the final result to use gates preferred by the Qiskit compiler [Cross 2018], which are the single-qubit gates U_1, U_2, U_3 and the two-qubit gate $CNOT$.

We also provide resource counts (computed by the same procedure) for our implementations of 8-bit modular multiplication. Quipper does not have a built-in operation for modular multiplication (which is different from multiplication followed by a modulo operator in the presence of overflow).

[10] We converted the output Quipper files to OpenQASM 2.0 using a compiler produced at Dalhousie University [Bian 2020].

We define all of the arithmetic operations in Figure 16 for arbitrary input sizes; the limited sizes in our experiments (8 and 16 bits) are to account for inefficiencies in VOQC. For the largest circuits (the modular multipliers), running VOQC takes about 10 minutes.

Comparing QFT and Toffoli-based operators. The results show that the QFT-based implementations always use fewer qubits. This is because they do not need ancillae to implement reversibility. For both division/modulo and modular multiplication (used in Shor's oracle), the savings are substantial because those operators are not easily reversible using Toffoli-based gates, and more ancillae are needed for uncomputation.

The QFT circuits also typically use fewer gates. This is partially due to algorithmic advantages of QFT-based arithmetic, partially due to VOQC (VOQC reduced QFT circuit gate counts by 57% and Toffoli circuit gate counts by 28%) and partially due to the optimized decompositions we use to convert many-qubit gates to the one- and two-qubit gates supported by VOQC.[11] We found during evaluation that gate counts are highly sensitive to the decompositions used: Using a more naïve decomposition of the controlled-Toffoli gate (which simply computes the controlled version of every gate in the standard Toffoli decomposition) increased the size of our Toffoli-based modular multiplication circuit by 1.9x, and a similarly naïve decomposition of the controlled-controlled-*Rz* gate increased the size of our QFT-based modular multiplication circuit by 4.4x. Unlike gate counts, qubit counts are more difficult to optimize because they require fundamentally changing the structure of the circuit; this makes QFT's qubit savings for modular multiplication even more impressive.

In addition, when we test different constant inputs for different arithmetic circuits, we find that the gate counts for Toffoli-based circuits tend to vibrate more than the QFT-based ones. This means that Toffoli-based circuits are more sensitive towards input constant changes. For example, with different constant inputs, the gate counts of the QFT-based Multiplication circuits are in the range 1372 ± 26, while the gate counts of our Toffoli-based circuits are in the range 1739 ± 367, and the Quipper ones are in the range 2870 ± 594.

Overall, our results suggest that QFT-based arithmetic provides better performance, so when compiling ⓪QIMP programs (like the sine function in **??**) to ⓪QASM, we should bias towards using the QFT-based operators.

Comparing to Quipper. Overall, Figure 16(a)-(c) shows that operator implementations in ⓪QASM consume resources comparable to those available in Quipper, often using fewer qubits and gates, both for Toffoli- and QFT-based operations. In the case of the QFT adder, the difference is that the Quipper-to-OpenQASM converter we use has a more expensive decomposition of controlled-*Rz* gates.[12] In the other cases (all Toffoli-based circuits), we made choices when implementing the oracles that improved their resource usage. Nothing fundamental stopped the Quipper developers from having made the same choices, but we note they did not have the benefit of the ⓪QASM type system and PBT framework. Quipper has recently begun to develop a random testing framework based on QuickCheck [Claessen and Hughes 2000], but it only applies to Toffoli-based (i.e., effectively classical) gates.

[11] We use the decompositions for Toffoli and controlled-Toffoli at https://qiskit.org/documentation/_modules/qiskit/circuit/library/standard_gates/x.html; the decomposition for controlled-*Rz* at https://qiskit.org/documentation/_modules/qiskit/circuit/library/standard_gates/u1.html; and the decomposition for controlled-controlled-*Rz* at https://quantumcomputing.stackexchange.com/questions/11573/controlled-u-gate-on-ibmq. The decompositions we use are all proved correct in the SQIR development. All of these decompositions are ancilla free.

[12] Bian [2020] decomposes a controlled-*Rz* gate into a circuit that uses two Toffoli gates, an *Rz* gate, and an ancilla qubit. In VOQC, each Toffoli gate is decomposed into 9 single-qubit gates and 6 two-qubit gates. In contrast, VQO's decomposition for controlled-*Rz* uses 3 single-qubit gates, 2 two-qubit gates, and no ancilla qubits.

Precision	# gates	Error
16 bits (full)	1206	± 0
15 bits	1063	± 1
14 bits	929	± 3

(a) Varying the precision in a 16-bit adder

# iters. ($I+1$)	TOFF	QFT	AQFT	% savings
1	1798	1794	1717	4.5 / 4.5
4	7192	4432	3488	48.5 / 21.2
8	14384	8017	4994	65.2 / 37.7
12	21576	11637	5684	73.6 / 51.1
16	28768	15288	5948	79.3 / 61.1

(b) Gate counts for TOFF vs. QFT vs. AQFT division/modulo circuits, the left saving numbers in the savings column are comparing TOFF vs. AQFT, and the right one are QFT vs. AQFT

Fig. 19. Effects of approximation

5.4 Approximate Operators

ⓞQASM's efficiently-simulable semantics can be used to predict the effect of using approximate components, which enables a new workflow for optimizing quantum circuits: Given an exact circuit implementation, replace a subcomponent with an approximate implementation; use VQO's PBT framework to compare the outputs between the exact and approximate circuits; and finally decide whether to accept or reject the approximation based on the results of these tests, iteratively improving performance.

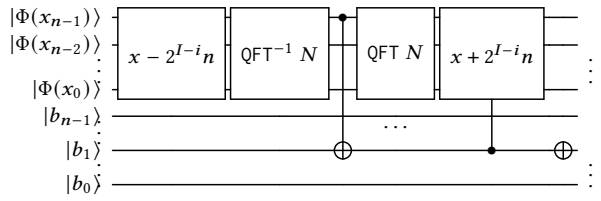
In this section, we use VQO's PBT framework to study the effect of replacing QFT circuits with AQFT circuits (Figure 23) in addition and division/modulo circuits.

Approximate Addition. Figure 18(a) shows the results of replacing QFT with AQFT in the QFT adder from Figure 16(a). As expected, a decrease in precision leads to a decrease in gate count. On the other hand, our testing framework demonstrates that this also increases error (measured as absolute difference accounting for overflow, maximized over randomly-generated inputs). Random testing over a wider range of inputs suggests that dropping b bits of precision from the exact QFT adder always induces an error of at most $\pm 2^b - 1$. This exponential error suggests that the "approximate adder" is not particularly useful on its own, as it is effectively ignoring the least significant bits in the computation. However, it computes the most significant bits correctly: if the inputs are both multiples of 2^b then an approximate adder that drops b bits of precision will always produce the correct result.

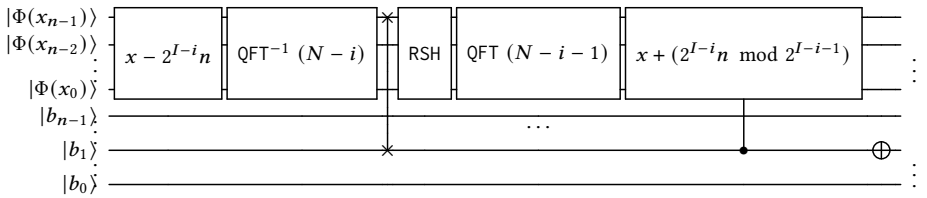
Exact Division/Modulo using an Approximate Adder. Even though the approximate adder is not particularly useful for addition, there are still cases where it can be useful as a subcomponent. For example, the modulo/division circuit relies on an addition subcomponent, but does not need every bit to be correctly added.

Figure 19(a) shows one step of an N-bit QFT-based modulo circuit that computes $x \bmod n$ for constant n. The algorithm runs for $I+1$ iterations, where $2^{N-1} \leq 2^I n < 2^N$, with the iteration counter i increasing from 0 to I (inclusive). In each iteration, the circuit in Figure 19(a) computes $x - 2^{I-i}n$ and uses the result's most significant bit (MSB) to check whether $x < 2^{N-1-i}$. If the MSB is 0, then $x \geq 2^{N-1-i}$ and the circuit continues to next iteration; otherwise, it adds $2^{I-i}n$ to the result and continues.

We can improve the resource usage of the circuit in Figure 19(a) by replacing the addition, subtraction, and QFT components with approximate versions, as shown in Figure 19(b). At the start of each iteration, $x < 2^{N-i}$, so it is safe to replace components with versions that will perform the intended operation on the lowest $(N-i)$ bits. The circuit in Figure 19(b) begins by subtracting



(a) QFT-based



(b) AQFT-based (addition and subtraction are approximate)

Fig. 20. One step of the QFT/AQFT division/modulo circuit

the top $(N-i)$ bits, and then converts x back to the `Nor` basis using an $(N-i)$-bit precision QFT. It then swaps the MSB with an ancilla, guaranteeing that the MSB is 0. Next, it uses a `Rshift` to move the cleaned MSB to become the lowest significant bit (effectively, multiplying x by 2) and uses a $(N-i-1)$-bit precision QFT to convert back to the `Phi` basis. Finally, it conditionally adds back the top $(N-i-1)$ bit of the value $(2^{I-i}n \mod 2^{I-i-1})$, ignoring the original MSB.

The result is a division/modulo circuit that uses approximate components, but, as our testing assures, is exactly correct. Figure 18(b) shows the required resources for varying numbers of iterations. Compared to QFT-based circuit, for a single iteration, the approximation provides a 4.5% savings, and the saving increases with more iterations. For $n = 1$, we need 16 iterations. In this case, the AQFT-based division/modulo circuit uses 61.1% fewer gates than the QFT-based implementation. If we compare the AQFT-based division/modulo circuit with the Toffoli-based one, the result is more significant. For 16 iterations, the AQFT-based division/modulo circuit uses 79.3% fewer gates than the Toffoli-based implementation.

6 EVALUATION: ⓞQIMP ORACLES AND PARTIAL EVALUATION

The prior section considered arithmetic operators implemented in ⓞQASM. They are building blocks for operators we have programmed using ⓞQIMP, which include sine, arcsine, cosine, arccosine, and exponentiation on fixed-precision numbers. We used ⓞQIMP's source semantics to test each operator's correctness. These operators are useful in near-term applications; e.g., the arcsine and sine functions are sub-components to repair the phase values in constructing Hamiltonian simulations [Feynman 1982] by the quantum walk algorithm [Childs 2009].

As discussed in **??**, one of the key features of ⓞQIMP is *partial evaluation* during compilation to ⓞQASM. The simplest optimization similar to partial evaluation happens for a binary operation $x := x \odot y$, where y is a constant value. Figure 16 hints at the power of partial evaluation for this case—all constant operations (marked "const") generate circuits with significantly fewer qubits and gates. Languages like Quipper take advantage of this by producing special circuits for operations that use classically-known constant parameters.

Partial evaluation takes this one step further, pre-evaluating as much of the circuit as possible. For example, consider the fixed precision operation $\frac{x*y}{M}$ where M is constant and a natural number,

	# qubits	# gates
OQIMP (x, y const)	16	16
OQIMP TOFF (x const)	33	1739 ± 376
OQIMP QFT (x const)	16	1372 ± 26
OQIMP TOFF	33	61470
OQIMP QFT	32	25609

(a) Fixed-precision circuits for $\frac{x*y}{M}$ with $M = 5$ (16 bits)

	# qubits
OQIMP TOFF	418
OQIMP QFT	384
Quipper	6142

(b) Sine circuits (64 bits)

Fig. 21. Effects of partial evaluation

and x and y are two fixed precision numbers that may be constants. This is a common pattern, appearing in many quantum oracles (recall the $\frac{8^n*x}{n!}$ in the Taylor series decomposition of sine). In Quipper, this is expression compiled to $r_1 = \frac{x}{M}; r_2 = r1 * y$. The ℗QIMP compiler produces different outputs depending on whether x and y are constants. If they both are constant, ℗QIMP simply assigns the result of computing $\frac{x*y}{M}$ to a quantum variable. If x is a constant, but y is not, ℗QIMP evaluates $\frac{x}{M}$ classically, assigns the value to r_1, and evaluates r_2 using a constant multiplication circuit. If they are both quantum variables, ℗QIMP generates a circuit to evaluate the division first and then the multiplication.

In Figure 20 (a) we show the size of the circuit generated for $\frac{x*y}{M}$ where zero, one, or both variables are classically known. It is clear that more classical variables in a program lead to a more efficient output circuit. If x and y are both constants, then only a constant assignment circuit is needed, which is a series of X gates. Even if only one variable is constant, it may lead to substantial savings: In this example, if x is constant, the compiler can avoid the division circuit and use a constant multiplier instead of a general multiplier. These savings quickly add up: Figure 20 (b) shows the qubit size difference between our implementation of sine and Quippers'. Both the TOFF and QFT-based circuits use fewer than 7% of the qubits used by Quipper's sine implementation. [13]

7 CASE STUDY: GROVER'S SEARCH

Here we present a case study of integrating an oracle implemented with VQO into a full quantum algorithm, Grover's search algorithm, implemented and verified in SQIR.

Grover's search algorithm [Grover 1996, 1997], described in Section 1, has implications for cryptography, in part because it can be used to find collisions in cryptographic hash functions [Bernstein 2010]. Thus, the emergence of quantum computers may require lengthening hash function outputs.

We have used ℗QIMP to implement the ChaCha20 stream cipher [Bernstein 2008] as an oracle for Grover's search algorithm. This cipher computes a hash of a 256-bit key, a 64-bit message number, and a 64-bit block number, and it is actively being used in the TLS protocol [Langley et al. 2016; Rescorla 2018]. The procedure consists of twenty cipher rounds, most easily implemented when segmented into quarter-round and double-round subroutines. The only operations used are bitwise XOR, bit rotations, and addition modulo 2^{32}, all of which are included in ℗QIMP; the implementation is given in Figure 21.

To test our oracle implementation, we wrote our specification as a Coq function on bitstrings. We then defined correspondence between these bitstrings and program states in ℗QASM semantics and conjectured that for any inputs, the semantics of our compiled oracle matches the corresponding

[13] ℗QIMP also benefits from its representation of fixed-precision numbers (**??**), which is more restrictive than Quipper's. Our representation of fixed-precision numbers reduces the qubit usage of the sine function by half, so about half of the qubit savings can be attributed to this.

```
Q nat[4] qr(Q nat x1, Q nat x2, Q nat x3, Q nat x4) {
  x1 += x2; x4 ⊕= x1; x4 <<<= 16;
  x3 += x4; x2 ⊕= x3; x4 <<<= 12;
  x1 += x2; x2 ⊕= x1; x4 <<<= 8;
  x3 += x4; x2 ⊕= x3; x4 <<<= 7
  return [x1, x2, x3, x4];
}

void chacha20(Q nat[16] x) {
  for(C nat i = 20; i > 0; i -= 2) {
    [x[0], x[4], x[8], x[12]] = qr(x[0], x[4], x[8], x[12]);
    [x[1], x[5], x[9], x[13]] = qr(x[1], x[5], x[9], x[13]);
    [x[2], x[6], x[10], x[14]] = qr(x[2], x[6], x[10], x[14]);
    [x[3], x[7], x[11], x[15]] = qr(x[3], x[7], x[11], x[15]);
    [x[0], x[5], x[10], x[15]] = qr(x[0], x[5], x[10], x[15]);
    [x[1], x[6], x[11], x[12]] = qr(x[1], x[6], x[11], x[12]);
    [x[2], x[7], x[8], x[13]] = qr(x[2], x[7], x[8], x[13]);
    [x[3], x[4], x[9], x[14]] = qr(x[3], x[4], x[9], x[14]);
  }
}
```

Fig. 22. ChaCha20 implementation in ©QIMP

outputs from our specification function. Using random testing (**??**), we individually tested the quarter-round and double-round subroutines as well as the whole twenty-round cipher, performing a sort of unit testing. We also tested the oracle for the boolean-valued function that checks whether the ChaCha20 output matches a known bitstring rather than producing the output directly. This oracle can be compiled to SQIR using our verified compiler, and then the compiled oracle can be used by Grover's algorithm to invert the ChaCha20 function and find collisions. Grover's algorithm was previously implemented and verified in SQIR [Hietala et al. 2021a], and we have modified this implementation and proof to allow for oracles with ancillae like the ones generated by our compiler; thus, our successful QuickChick tests combined with the previously proved theorems for Grover's algorithm provide confidence that we can find Chacha20's hash collisions in a certain probability through Grover's algorithm.

REFERENCES

Adriano Barenco, Artur Ekert, Kalle-Antti Suominen, and Päivi Törmä. 1996. Approximate quantum Fourier transform and decoherence. *Physical Review A* 54, 1 (Jul 1996), 139–146. https://doi.org/10.1103/physreva.54.139

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

Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20 *(The State of the Art of Stream Ciphers)*. ECRYPT Network of Excellence in Cryptology, 273–278. https://cr.yp.to/papers.html#chacha

Daniel J. Bernstein. 2010. Grover vs. McEliece. In *Post-Quantum Cryptography*, Nicolas Sendrier (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–80. https://doi.org/10.1007/978-3-642-12929-2_6

Xiaoning Bian. 2020. Compile Quipper quantum circuit to OpenQasm 2.0 program. https://www.mathstat.dal.ca/~xbian/QasmTrans/ [Online; accessed 8-July-2021].

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

Lukas Burgholzer, Hartwig Bauer, and Robert Wille. 2021. Hybrid Schrödinger-Feynman Simulation of Quantum Circuits With Decision Diagrams. arXiv:2105.07045 [quant-ph]

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

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

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

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

Thomas G. Draper. 2000. Addition on a Quantum Computer. *arXiv e-prints*, Article quant-ph/0008033 (Aug. 2000), quant-ph/0008033 pages. arXiv:quant-ph/0008033 [quant-ph]

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

Richard P Feynman. 1982. Simulating physics with computers. *International journal of theoretical physics* 21, 6/7 (1982), 467–488.

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

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

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

L. Hales and S. Hallgren. 2000. An improved quantum Fourier transform algorithm and applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 515–525. https://doi.org/10.1109/SFCS.2000.892139

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)*.

A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. 2016. *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. RFC 7905. https://doi.org/10.17487/RFC7905

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2021. Verified Compilation of Quantum Oracles. https://doi.org/10.48550/ARXIV.2112.06700

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.

Yunseong Nam, Yuan Su, and Dmitri Maslov. 2020. Approximate quantum Fourier transform with O(n log(n)) T gates. *npj Quantum Information* 6, 1 (Mar 2020). https://doi.org/10.1038/s41534-020-0257-5

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

E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. https://doi.org/10.17487/RFC8446

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

```
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. 23. Shor's Algorithm in Q-Dafny

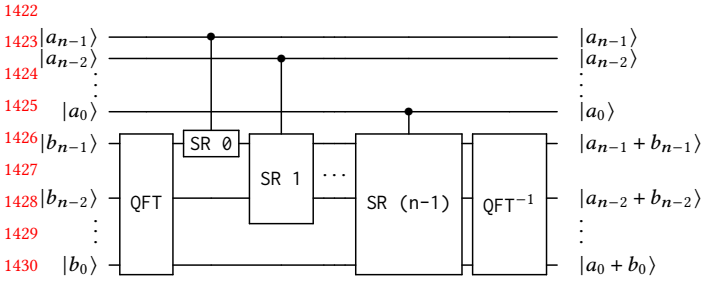
A ℚASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

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

This section presents ℚASM states and the language's syntax, semantics, typing, and soundness results. As a running example, we use the QFT adder [Beauregard 2003] shown in Figure 23. The Coq function `rz_adder` generates an ℚASM program that adds two natural numbers `a` and `b`, each of length `n` qubits.

A.1 ℚASM States

An ℚASM program state is represented according to the grammar in Figure 24. A state φ of d qubits is a length-d tuple of qubit values q; the state models the tensor product of those values. This means that the size of φ is $O(d)$ where d is the number of qubits. A d-qubit state in a language like SQIR is represented as a length 2^d vector of complex numbers, which is $O(2^d)$ in the number of qubits. Our linear state representation is possible because applying any well-typed ℚASM program on any well-formed ℚASM state never causes qubits to be entangled.



(a) Quantum circuit

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

(b) 𝒪QASM metaprogram (in Coq)

Fig. 24. Example 𝒪QASM program: QFT-based adder

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

Fig. 25. 𝒪QASM state syntax

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

Fig. 26. 𝒪QASM syntax. For an operator OP, $\mathrm{OP}^{[-1]}$ indicates that the operator has a built-in inverse available.

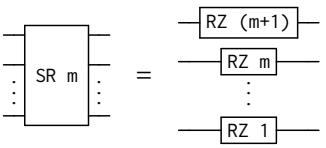


Fig. 27. SR unfolds to a series of RZ instructions

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

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

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

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

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

Fig. 28. Select ℴQASM typing rules

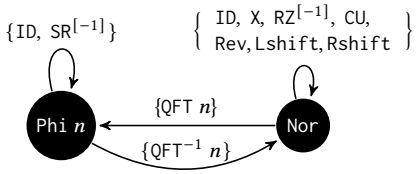


Fig. 29. Type rules' state machine

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 26). 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 23(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 25, 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 32; the rules not shown (for ID, Rshift, Rev, $\texttt{RZ}^{-1}$, and $\texttt{SR}^{-1}$) are similar.

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

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

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

Fig. 30. ©QASM semantics

the quantum *no-cloning rule*. For example, we can apply the CU in `rz_adder'` (Figure 23) 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 28. 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 ©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 29.

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 29 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 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 SR^{-1}) applies an $m+1$ series of RZ (or RZ^{-1}) rotations where the i-th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$ (or $\alpha(-\frac{1}{2^{m-i+1}})$). qt applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes \cdots \otimes |b_i\rangle$ (assuming $\Sigma(y) = i$) and n is the degree of approximation. If $n = i$, then the operation is the standard QFT. Otherwise, each qubit in the state is mapped to $|\Phi(\frac{y}{2^{n-k}})\rangle$, which is equal to $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(\frac{y}{2^{n-k}})\,|1\rangle)$ when $k < n$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ when $n \leq k$ (since $\alpha(n) = 1$ for any natural number n). qt^{-1} is the inverse function of qt. Note that the input state to 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. pm_l, pm_r, and 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 $\Omega(x) = \mathsf{Nor}$, for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|b\rangle$.
- For every $x \in \Omega$ such that $\Omega(x) = \mathsf{Phi}\ n$ and $n \leq \Sigma(x)$, there exists a value v such that for every $k < \Sigma(x)$, $\varphi(x, k)$ has the form $\alpha(r)\,|\Phi(\frac{v}{2^{n-k}})\rangle$.[14]

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 $\mathcal{H}^d$,[15] 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.

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

[15] 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. 31. Select ⟦QASM inversion rules

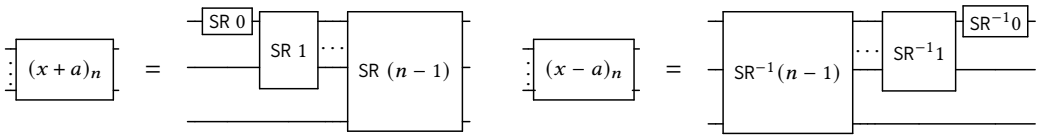


Fig. 32. Addition/subtraction circuits are inverses

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

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

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

We prove these theorems as corollaries of the compilation correctness theorem from ⟦QASM to SQIR (Theorem 4.1). Theorem A.3 suggests that the space $\mathcal{S}^d$ is closed under the application of any well-typed ⟦QASM operation. Theorem A.4 says that ⟦QASM oracles can be safely applied to superpositions over classical states.[16]

⟦QASM programs are easily invertible, as shown by the rules in Figure 30. 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 30). 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.

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

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

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

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

Fig. 33. Arith, Bool, Gate Mode Checking

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

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

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

The QWhile type system has the type judgment: $\Omega, \mathcal{T} \vdash_g s : \zeta \triangleright \tau$, where g is the context mode, mode environment Ω maps variables to modes or sessions (q in Figure 5), type environment $\mathcal{T}$ maps a session to its type, s is the statement being typed, ζ is the session of s, and τ is ζ's type. The QWhile type system in Figure 37 has several tasks. First, it enforces context mode restrictions. Context mode g is either C or Q . Q represents the current expression lives inside a quantum conditional or loop, while C refers to other cases. In a Q context, one cannot perform M-mode operations, i.e., no measurement is allowed. There are other well-formedness enforcement. For example, the session of the Boolean guard b in a conditional/loop is disjoint with the session in the conditional/loop body, i.e., qubits used in a Boolean guard cannot appear in its conditional/loop body.

Second, the type system enforces mode checking for variables and expressions in Figure 32. In QWhile, C-mode variables are evaluated to values during type checking. In a `let` statement (Figure 37), C-mode expression is evaluated to a value n, and the variable x is replaced by n in s. The expression mode checking (Figure 32) has the judgment: $\Omega \vdash (a \mid b) : q$. It takes a mode environment Ω, and an expression (a, b), and judges if the expression has the mode g if it contains only classical values, or a quantum session ζ if it contains some quantum values. All the supposedly C-mode locations in an expression are assumed to be evaluated to values in the type checking step,

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

Fig. 34. Control Gate Entanglement Situation

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

Fig. 35. Session Type Subtyping

such as the index value $x[n]$ in difference expressions in Figure 32. It is worth noting that the session computation ($\uplus$) is also commutative as the last rule in Figure 32.

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 24) 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 ($\otimes_n$ Nor $\overline{d}$) can also be a subtype of a entanglement type ($\otimes_n$ CH $1\{\overline{d}\}$) in our system (Figure 34).

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 $\alpha_1 \left|b_1\right\rangle + \alpha_2 \left|b_2\right\rangle$, 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 33.

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 $\sum_{t=0}^{N} i^{-t} \left|t\right\rangle$ 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

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

(a) Subtyping

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

(b) State Equivalence

Fig. 36. Qafny type/state relations. $\overline{c}[n]$ produces the n-th element in set $\overline{c}$. $\{(\!|j|\!)|j \in [0,2^n)\}(2^n)$ defines a set $\{(\!|j|\!)|j \in [0,2^n)\}$ with the emphasis that it has 2^n elements. $\{0,1\}$ is a set of two single element bitstrings 0 and 1. $\cdot$ is the multiplication operation, $(\!|j|\!)$ turns a number j to a bitstring, $(\!|j|\!)[k]$ takes the k-th element in the bitstring $(\!|j|\!)$, and $|j\rangle$ is an abbreviation of $|(\!|j|\!)\rangle$.

result in entanglement. In this narrative, algorithm designers intend to merge an additional qubit $x[i]$ into an existing entanglement session y. $x[i]$ is commonly in uniform superposition, but there can be some additional local phases attached with some bases, which we named this situation as saturation, i.e., In an entanglement session written as $\sum_{i=0}^{n}|x_l,y,x_r\rangle$, 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 $k < i$. [17] 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

[17] 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 $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$, referring to that qubits in such session are possibly entangled. The state $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$ 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 $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$, 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 $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$, 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 $\Sigma_{j=0}^{m} z_j\,|c_j\rangle$, 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 35. 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 35a 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 $\{c\}$ [18]. 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 $\Sigma_{j=0}^{1} z_j\,|c_j\rangle$ can be used as a Nor type state $|c_0\rangle$ with the fact that z_0 is now a global phase that can be neglected.

C.1 Type Checking: A Quantum Session Type System

In Qafny , typing is with respect to a *kind environment* Ω and a *finite type environment* σ, which map Qafny variables to kinds and map sessions to types, respectively. The typing judgment is written as $\Omega; \sigma \vdash_g s \triangleright \sigma'$, which states that statements s is well-typed under the context mode g and environments Ω and σ, the sessions representing s is exactly the domain of σ' as $\mathtt{dom}(\sigma')$, and s transforms types for the sessions in σ to types in σ'. Ω describes the kinds for all program variables. Ω is populated through `let` expressions that introduce variables, and the Qafny type system enforces variable scope; such enforcement is neglected in Figure 37 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

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

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

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

Fig. 37. 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.

TPAR
$$\frac{\sigma \preceq \sigma' \qquad \Omega, \sigma' \vdash_g s \triangleright \sigma''}{\Omega, \sigma \vdash_g s \triangleright \sigma''}$$

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

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

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

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

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

TLOOP
$$\frac{\forall j \in [n_1, n_2)\ .\ \Omega, \sigma[\uparrow \sigma'[j/x]] \vdash_g \mathtt{if}\ (b)\ s \triangleright \sigma'[\mathtt{S}\ j/x]}{\Omega, \sigma \vdash_g \mathtt{for}\ (\mathtt{int}\ \mathtt{int}\ x := n_1 \in [x < n_2, b)\ \&\&\ {+}{+}x)\ s \triangleright \sigma'[n_2/x]}$$

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

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

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

Fig. 38. QAFNY type system. $\llbracket \mu \rrbracket c$ is the ⒪QASM semantics of interpreting reversible expression μ in Figure 29. 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 ⒪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 36, which will be explained shortly. Selected type rules are given in Figure 37; 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 $\texttt{CH}\ \overline{c}(m)$, 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 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 $\texttt{CH}\ \overline{c}(m)$, 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 36, 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 36 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 36. Merging two sessions is valuable for analyzing the behavior of quantum conditionals. In rule TIF, the session $(\lambda_1 \uplus x[j, \texttt{S}\ j])$ 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 36 by computing the Cartesian product of the two type bases, such that if the two sessions are both `CH` types $\lambda_1 \uplus x[j, \texttt{S}\ j] \mapsto \texttt{CH}\ \overline{c_1}(m_1)$ and $\lambda_2 \mapsto \texttt{CH}\ \overline{c_2}(m_2)$, the result is of type $\texttt{CH}\ (\overline{c_1} \times \overline{c_2})(m_1 * m_2)$. 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 $x[j, \mathsf{S}\ j]$); 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. [19], 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 $\mathsf{CH}\ \infty$. 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 $\mathsf{CH}\ \overline{c'}(|\overline{c'}|)$. This does not violate type preservation because we have the subtyping relation $\mathsf{CH}\ \overline{c'}(|\overline{c'}|) \sqsubseteq_{|\lambda|} \mathsf{CH}\ \infty$.

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

C.2 QAFNY Semantics and Type Soundness

We define the semantics of an QASM program as a partial function $[\![\ ]\!]$ from an instruction ι and input state φ to an output state φ', written $[\![\iota]\!]\varphi = \varphi'$, shown in Figure 29.

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 29 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 SR^{-1}) applies an $m + 1$ series of RZ (or RZ^{-1}) rotations where the i-th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$ (or $\alpha(-\frac{1}{2^{m-i+1}})$). qt applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes \cdots \otimes |b_i\rangle$ (assuming $\Sigma(y) = i$) and n is the degree of approximation. If $n = i$, then the operation is the standard QFT. Otherwise, each qubit in the state is mapped to $|\Phi(\frac{y}{2^{n-k}})\rangle$, which is equal to $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(\frac{y}{2^{n-k}})\,|1\rangle)$ when $k < n$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ when $n \leq k$ (since $\alpha(n) = 1$ for any natural number n). $\mathtt{qt}^{-1}$ is the inverse function of qt. Note that the input state to $\mathtt{qt}^{-1}$ is guaranteed to have the form $\bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{n-k}})\rangle)$ because it has type Phi n. $\mathtt{pm}_l$, $\mathtt{pm}_r$, and $\mathtt{pm}_a$ are the semantics for Lshift, Rshift, and Rev, respectively.

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

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

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

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

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

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

Fig. 39. QAFNY small step semantics. $[\![\mu]\!]c$ is the ⟦QASM semantics of interpreting reversible expression μ in Figure 29. 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 37 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 24 as arrays with elements in different forms, and use the session types to guide the occurrence of a specific form at a time. Sessions, like the array variables in the classical Hoare Logic theory, represent the stores of quantum states. The state changes are implemented by the substitutions of sessions with expressions containing operation's semantic transitions. The substitutions can happen for a single index session element or the whole session.

Rule PA-NOR and PA-CH specify the assignment rules. If a session ζ has type `Nor`, it is a singleton array, so the substitution $[\![a]\!]\zeta/\zeta$ means that we substitute the singleton array by a term with the a's application. When ζ has type `CH`, term $\zeta[k]$ refers to each basis state in the entanglement. The assignment is an array map operation that applies a to every element in the array. For example, in Figure 22 line 12, we apply a series of `H` gates to array x. Its post-condition is $[(x, 0, n)] =$

$\bigotimes_{k=0}^{n} |\Phi(0)\rangle$, where $[(x,0,n)]$ is the session representing register variable x. Thus, replacing the session $[(x,0,n)]$ with the $\mathtt{H}$ application results in a pre-condition as $\mathtt{H}[(x,0,n)] = \bigotimes_{k=0}^{n} |\Phi(0)\rangle$, which means that $[(x,0,n)]$ has the state $|0\rangle^n$.

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

Rule P-IF deals with a quantum conditional where the Boolean guard $b(@x[v])$ is of type $\bigotimes_n$ $\mathtt{CH}\ 2m(\beta_1 \cdot 0 \cup \beta_2 \cdot 1)$. 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 $\mathtt{if}\ (x[i])\ s$, 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 $\mathtt{diffuse}(x)$. The quantum semantics for $\mathtt{diffuse}(x)$ 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, $\mathtt{diffuse}(x)$ with the session ζ is an array operation as follows: assume that $\zeta = (x,0,\Sigma(x)) \uplus \zeta_1$, for every k, if $\zeta[k]$'s value is $\theta_k(\overline{d_x} \cdot \overline{d_1})$, for any bitstring z in $\mathcal{P}(\Sigma(x))$, if $z \cdot \overline{d_1}$ is not a base for $\zeta[j]$ for any j, then the state is $\frac{1}{2^{n-1}}\sum_{k=0}\theta_k(z \cdot \overline{d_1})$; if the base of $\zeta[j]$ is $z \cdot \overline{d_1}$, then the state for $\zeta[j]$ is $\frac{1}{2^{n-1}}(\sum_{k=0}\theta_k) - \theta_j(z \cdot \overline{d_1})$.