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.

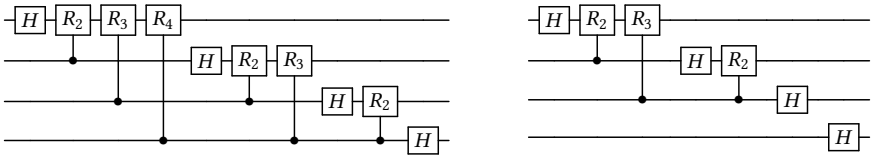


Fig. 1. Example quantum circuits: QFT over 4 qubits (left) and approximate QFT with 3 qubit precision (right). R_m is a z-axis rotation by $2\pi/2^m$.

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 QWHILE: A HIGH-LEVEL QUANTUM LANGUAGE

We introduce the language syntax and type system for QWhile and introduce the Q-Dafny Proof system. As a running example, we specify Shor's algorithm and its proof in Q-Dafny in Figure 2. The Q-Dafny to Dafny compiler is under construction, but the compiled version of the Shor's algorithm proof has been finalized and can be found at https://github.com/inQWIRE/VQO/blob/naturalproof/Q-Dafny/examples/Shor-compiled.dfy.

2.1 Sessions, Kinds, Types, and States

The QAFNY element component syntax is represented according to the grammar in Figure 3. 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

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

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 $\uplus$ 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 3) that are corresponding to three different types (τ in Figure 3). 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

Bit	d	::=	$0 \mid 1$
Bitstring	c	$\in$	d^+
Indexed bitstring set	$\overline{c}(m)$	::=	$\{c_0, c_1, ..., c_{m-1}\}$
Nat. Num	m, n	$\in$	$\mathbb{N}$
Real	r	$\in$	$\mathbb{R}$
Complex Number	z	$\in$	$\mathbb{C}$
Phase	$\alpha(r)$	::=	$e^{2\pi i r}$
Program/Session Variable	x, y		
Mode	g	::=	$\mathtt{C} \mid \mathtt{M}$
Classical Value	v	::=	$n \mid (r, n)$
Range	l	::=	$\underline{x[n..m]}$
Session	λ	::=	$\overline{x[n..m]}$
Full Mode (Kind)	fg	::=	$g \mid \mathtt{Q}\ n$
Option	$p \in\ 'a\ \mathtt{opt}$	::=	$'a \mid \infty$
Uniform Distribution	$\bigcirc$		
Type	τ	::=	$\mathtt{Nor}\ (c\ \mathtt{opt}) \mid \mathtt{Had}\ (\bigcirc\ \mathtt{opt}) \mid \mathtt{CH}\ (\overline{c}(m)\ \mathtt{opt})$
Quantum States	q	::=	$\lvert c\rangle \mid \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n} (\lvert 0\rangle + \alpha(r_j) \lvert 1\rangle) \mid \Sigma_{j=0}^{m} z_j \lvert c_j\rangle$

Fig. 3. Qafny element syntax. In $\overline{c}(m)$, $\overline{c}$ is a bitstring set and m is the element number, and it can be abbreviated as $\overline{c}$. Each element $x[n..m]$ in a session $\overline{x[n..m]}$ represents the number range $[n, m)$ in a qubit array x.

$\mathtt{Nor}\ \infty$	$\sqsubseteq_n$	$\mathtt{CH}\ \infty$
$\mathtt{Nor}\ c$	$\sqsubseteq_n$	$\mathtt{CH}\ \{c\}$
$\mathtt{CH}\ \overline{c}(1)$	$\sqsubseteq_n$	$\mathtt{Nor}\ \overline{c}[0]$
$\mathtt{Had}\ p$	$\sqsubseteq_n$	$\mathtt{CH}\ \{(\!\lvert j\rvert\!) \mid j \in [0, 2^n)\}(2^n)$
$\mathtt{CH}\ \{0, 1\}$	$\sqsubseteq_1$	$\mathtt{Had}\ \infty$
$\mathtt{CH}\ p$	$\sqsubseteq_n$	$\mathtt{CH}\ \infty$

(a) Subtyping

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

(b) State Equivalence

Fig. 4. Qafny type/state relations. $\overline{c}[n]$ produces the n-th element in set $\overline{c}$. $\{(\!\lvert j\rvert\!) \mid j \in [0, 2^n)\}(2^n)$ defines a set $\{(\!\lvert j\rvert\!) \mid j \in [0, 2^n)\}$ 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, $(\!\lvert j\rvert\!)$ turns a number j to a bitstring, $(\!\lvert j\rvert\!)[k]$ takes the k-th element in the bitstring $(\!\lvert j\rvert\!)$, and $\lvert j\rangle$ is an abbreviation of $\lvert(\!\lvert j\rvert\!)\rangle$.

uniformly distributed superposition; otherwise, we represent the type as $\mathtt{Had}\ \infty$. The third kind of state is of CH type ($\mathtt{CH}\ (\overline{c}(m)\ \mathtt{opt})$), having the state form $\Sigma_{j=0}^{m} z_j \lvert c_j\rangle$, referring to that qubits in such session are possibly entangled. The state $\Sigma_{j=0}^{m} z_j \lvert c_j\rangle$ can be viewed as an m element set of pairs $z_j \lvert c_j\rangle$, where z_j and c_j are the j-th amplitude and basis. The well-formed restrictions for the state are three: 1) $\sum_{j=0}^{m} \lvert z_j\rvert^2 = 1$ (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 $\lvert c\rangle$ and CH type state $\Sigma_{j=0}^{m} z_j \lvert c_j\rangle$, 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 $\lvert c\rangle$ and $\Sigma_{j=0}^{m} z_j \lvert c_j\rangle$, respectively. For Nor type, this means that the type factor c (in $\mathtt{Nor}\ c$) and the state qubit format $\lvert c\rangle$ must be equal; for CH type ($\mathtt{CH}\ \overline{c}(m)$), if the state is $\Sigma_{j=0}^{m} z_j \lvert c_j\rangle$, the j-th element $\overline{c}[j]$ is equal to c_j. Additionally, Qafny types permit subtyping relations that

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

Fig. 5. Core QAFNY syntax. ⟦QASM is in Section 3. For an operator OP, $\mathtt{OP}^{[-1]}$ indicates that the operator has a built-in inverse available. Arithmetic expressions 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 whole qubit array.

correspond to state equivalent relations in Figure 4. 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 4a 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 $\{c\}$ [1]. 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.

2.2 Syntax

The QAFNY program operations in Figure 5 are designed based on separations of different functionality instead of quantum gates in many other languages. A program consists of a sequence of C-like statements s. The first row in Figure 5 are the classical and quantum data-flow operations. $\{\}$ is a SKIP operation. The let operation ($\mathtt{let}\ x\ =\ e\ \mathtt{in}\ s$) 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 we only interacts a M value with a C one in a binary arithmetic operation, i.e., the $(r_1,n_1)+(r_2,n_2)$ is disallowed in QAFNY . $\mathtt{let}\ x\ =\ \mathtt{init}\ a\ \mathtt{in}\ s$ initializes an a-length qubit array named x with the value $|0\rangle^{\otimes a}$ [2], and is used in statement s, while $\mathtt{let}\ x\ =\ \mathtt{measure}(y)\ \mathtt{in}\ s$ measures quantum qubit array y, stores the result in x an a-length qubit array named x and is used in s. The measurement turns the expression $\mathtt{measure}(y)$ to a ghost expression $\mathtt{ret}(y,(r,n))$, which does not appear in a QAFNY source program but appears during semantic evaluation, and it records the intermediate measurement result of qubit array y as (r,n). $l \leftarrow op$ 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 $\mathtt{QFT}^{-1}$ gate in the end of the quantum phase estimation algorithm. We only permit op to be H and $\mathtt{QFT}^{[-1]}$ gates. The other gate applications are done through $\lambda \leftarrow \mu$ that performs quantum oracle computation, such as quantum arithmetic operation. While let operation only performs classical arithmetic computation, quantum oracle arithmetic operation is performed through ⟦QASM

[1] If a qubit array only consists of 0 and 1, it can be viewed as an entanglement of unique possibility.
[2] $|0\rangle^{\otimes n}$ means $|\underbrace{00...0}_{\text{n}}\rangle$.

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

$\mathtt{pmut}((c_1.i_1.i_2.c_2), n) = (c_1.i_2.i_1.c_2)$ 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. 6. 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.

expressions [Li et al. 2021], which can be used to define most reversible arithmetic operations such as the ones in Figure 5. For example, the Shor's algorithm implementation in Figure 2 utilizes the modulo multiplication operation $a^{(2^i)} * y \mathit{mod} N$ on qubit array y, which can be expressed as an ⒪QASM circuit. A syntactic restriction is placed on $\lambda \leftarrow \mu$, such that the session λ represents the exact quantum qubits mentioned in expression μ.

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 conditional and b might contain quantum parameter. Every quantum parameter l appearing in b must not appear in s. In the QAFNY type system, we define a `well_formed` predicate to check such property. Apparently, quantum gate applications are essentially reversible. 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 we have an additional bit $x[a]$ to hold the result of computing $a_1 = a_2$ or $a_1 < a_2$. `for` $(\mathtt{int}\ x := a_1;\ x < a_2\ \&\&\ b\ ;\ {+}{+}x)$ $\{s\}$ is a possibly quantum for-loop depending on if the Boolean guard b contains quantum parameters. A classical variable x is introduced and it is initialized as the lower bound a_1, increments in each loop step defined by ${+}{+}x$, and ends at the upper bound a_2. In QAFNY implementation, ${+}{+}x$ and $x < a_2$ can be arbitrary monotonic increment and comparison functions. For simplicity, we restrict the two to be ${+}{+}x$ and $x < a_2$.

The last row contains quantum reflection operations, including quantum amplifier ($x \leftarrow \mathtt{amp}(a)$) and diffusion operation ($l \leftarrow \mathtt{dis}$), which are used to increase and average the occurrence likelihood of some quantum bases in a quantum state, respectively. For example, if a session $x[0, n] \uplus \lambda$ has a CH type state $\sum_{j=0}^{m} z_j |c_j\rangle$, $c_k = (\!|a|\!) \uplus c$, and $|(\!|a|\!)| = n$; then, $x \leftarrow \mathtt{amp}(a)$ increases the amplitude z_k for c_k.

2.3 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 7 for simplicity. We also assume that variables introduced in `let` expressions are all distinct through proper alpha conversions. σ

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

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

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

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

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

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

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

Fig. 7. Qafny type system. $\llbracket \mu \rrbracket c$ is the ◷qasm semantics of interpreting reversible expression μ in Figure 14. Boolean expression b can be $a_1 = a_2, a_1 < a_2$ or `true`. $b[(\lfloor n \rfloor)/\lambda]$ means that we treat b as a ◷qasm μ expression, replace qubits in array λ with bits in bitstring $(\lfloor n \rfloor)$, and evaluate it to a Boolean value. $\sigma(y) = \{\lambda \mapsto \tau\}$ produces the map entry $\lambda \mapsto \tau$ and the range $y[0..|y|]$ is in λ. $\sigma(\lambda) = \tau$ is an abbreviation of $\sigma(\lambda) = \{\lambda \mapsto \tau\}$. $FV(-)$ produces a session by union all qubits appearing in $-$.

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. We have partial order relations defined for type environments in Figure 6, which will be explained shortly. Selected type rules are given in Figure 7; the rules not mentioned are similar and listed in Section 4.

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

Second, the type system tracks the basis state of every qubit in sessions. In rule TA-CH, we find that the oracle μ is applied on λ belonging to a session $\lambda \uplus \lambda'$. Correspondingly, the session's type is $\mathtt{CH}\ \overline{c}(m)$, for each bitstring $c_1.c_2 \in \overline{c}$, with $|c_1| = |\lambda|$, we apply μ on the c_1 and leave c_2 unchanged. Here, we utilize the ◷qasm semantics that describes transitions from a `Nor` state to

another Nor one, and we generalize it to apply the semantic function on every element in the CH type. During the transition, the number of elements m does not change. Similarly, applying a partial measurement on range $y[0..j]$ of the session $y[0..j] \uplus \lambda$ in rule TMEA-N can be viewed as a array filter, i.e., for an element $c_1.c_2$ in set $\overline{c}$ of the type $\mathtt{CH}\ \overline{c}(m)$, with $|c_1| = j$, we keep only the ones with $c_1 = (\!|n|\!)$ (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 6, 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 6 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 6. Merging two sessions is valuable for analyzing the behavior of quantum conditionals. In rule TIF, the session $(\lambda_1 \uplus x[j, \mathtt{S}\ j])$ for the Boolean guard $(b@x[j])$ and the session for (λ_2) the body can be two separative sessions. Here, we first merge the two session through the mer rule in Figure 6 by computing the Cartesian product of the two type bases, such that if the two sessions are both CH types $\lambda_1 \uplus x[j, \mathtt{S}\ j] \mapsto \mathtt{CH}\ \overline{c_1}(m_1)$ and $\lambda_2 \mapsto \mathtt{CH}\ \overline{c_2}(m_2)$, the result is of type $\mathtt{CH}\ (\overline{c_1} \times \overline{c_2})(m_1 * m_2)$. After that, the quantum conditional behavior can be understood as applying a partial map function on the size $m_1 * m_2$ array of bitstrings, and we only apply the conditional body's effect on the second part (the $\overline{c_2}$ part) of some bitstrings whose first part is checked to be true by applying the Boolean guard b. **[Liyi: see how to merge the following to above]** Based on the new CH type with the set $\overline{c_1} \times \overline{c_2}$, the quantum conditional creates a new set based on $\overline{c_1} \times \overline{c_2}$, i.e., for each element $(\!|n|\!).d.c$ in the set, with $|(\!|n|\!)| = |\lambda_1|$, we compute Boolean guard b value by substituting qubit variables in b with the bitstring $(\!|n|\!)$, and the result $b[(\!|n|\!)/\lambda_1] \oplus d$ is true or not (d represents the bit value for the qubit at $x[j, \mathtt{S}\ j]$); if it is true, we replace the c bitstring by applying the conditional body on it; otherwise, we keep c to be the same. In short, the quantum conditional behavior can be understood as applying a partial map function on an m array of bitstrings, and we only apply the conditional body's effect on the second part of some bitstrings whose first part is checked to be true by applying the Boolean guard b.

Fourth, the type system enforces that the C classical variables can be evaluated to values in the compilation time. [3], 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 Section 4. 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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Qafny type system. $\llbracket \mu \rrbracket c$ is the ⟦qasm semantics of interpreting reversible expression μ in Figure 14. 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 $-$.

the step evaluation, rule TMEA types the partial measurement results as a classical `M` mode variable x and a possible quantum leftover λ as `CH` ∞. 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 `CH` $\overline{c'}(|\overline{c'}|)$. This does not violate type preservation because we have the subtyping relation `CH` $\overline{c'}(|\overline{c'}|) \sqsubseteq_{|\lambda|}$ `CH` ∞.

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.

2.4 Qafny Semantics and Type Soundness

2.5 Logic Proof System

The reason of having the session type system in Figure 7 is to enable the proof system given in **??**. Every proof rule is a structure as $\Omega \vdash_g \{T\}\ \{P\}\ s\ \{T'\}\ \{Q\}$, where g and Ω are the type entities mentioned in Section 2.3. 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 9 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 2 line 12, we apply a series of H gates to array x. Its post-condition is $[(x,0,n)] = \bigotimes_{k=0}^{n} |\Phi(0)\rangle$, where $[(x,0,n)]$ is the session representing register variable x. Thus, replacing the session $[(x,0,n)]$ with the H application results in a pre-condition as $\mathtt{H}[(x,0,n)] = \bigotimes_{k=0}^{n} |\Phi(0)\rangle$, which means that $[(x,0,n)]$ has the state $|0\rangle^n$.

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

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

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

Rule P-DIS proves a diffusion operator $\mathtt{diffuse}(x)$. 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 $\frac{1}{2^{n-1}}(\sum_{k=0}\theta_k) - \theta_j(z \cdot \overline{d_1})$.

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.

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

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

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

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]

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

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. 2021. Proving Quantum Programs Correct. In *Proceedings of the Conference on Interative Theorem Proving (ITP)*.

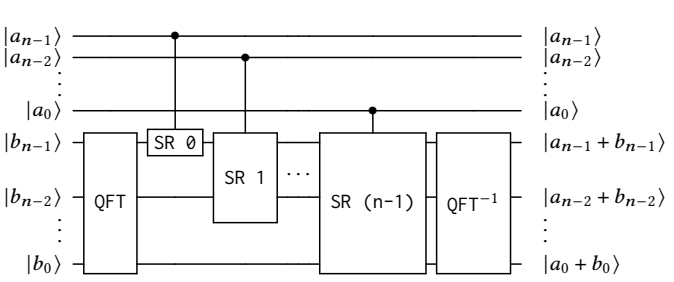
Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2021. Verified Compilation of Quantum Oracles. https://doi.org/10.48550/ARXIV.2112.06700

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.

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



(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. 9. 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` $\mid$ `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. 10. 𝒪QASM state syntax

3 𝒪QASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

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

This section presents 𝒪QASM states and the language's syntax, semantics, typing, and soundness results. As a running example, we use the QFT adder [Beauregard 2003] shown in Figure 8. The Coq function `rz_adder` generates an 𝒪QASM program that adds two natural numbers `a` and `b`, each of length `n` qubits.

3.1 𝒪QASM States

An 𝒪QASM program state is represented according to the grammar in Figure 9. A state φ of d qubits is a length-d tuple of qubit values q; the state models the tensor product of those values. This means that the size of φ is $O(d)$ where d is the number of qubits. A d-qubit state in a language like SQIR is represented as a length 2^d vector of complex numbers, which is $O(2^d)$ in the number of qubits. Our linear state representation is possible because applying any well-typed 𝒪QASM program on any well-formed 𝒪QASM state never causes qubits to be entangled.

A 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

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

Fig. 11. ℴQASM syntax. For an operator OP, $\texttt{OP}^{[-1]}$ indicates that the operator has a built-in inverse available.

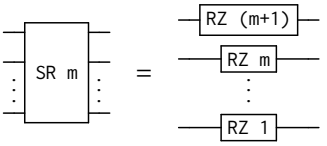


Fig. 12. SR unfolds to a series of RZ instructions

$b \in \{0, 1\}$), 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.

3.2 ℴQASM Syntax, Typing, and Semantics

[Liyi: add RZ gate back]

Figure 10 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—$\texttt{ID}\ p$, $\texttt{X}\ p$, and $\texttt{RZ}^{[-1]}\ n\ p$ —and instruction sequencing. The instructions in the next row apply to whole variables: $\texttt{QFT}\ n\ x$ 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 11). Operation $\texttt{CU}\ p\ \iota$ 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 8(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 10, instructions $\texttt{Lshift}\ x$, $\texttt{Rshift}\ x$, and $\texttt{Rev}\ x$ are *position shifting operations*. Assuming that x has d qubits and x_k represents the k-th qubit state in x, $\texttt{Lshift}\ x$ changes the k-th qubit state to $x_{(k+1)\%d}$, $\texttt{Rshift}\ x$ 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 3.5, Section 3.3), 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 17; the rules not shown (for ID, Rshift, Rev, $\texttt{RZ}^{-1}$, and $\texttt{SR}^{-1}$) are similar.

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

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

$$\textsc{SR}\ \frac{\Omega(x) = \texttt{Phi}\ n \qquad m < n}{\Sigma;\Omega \vdash \texttt{SR}\ m\ x \triangleright \Omega}$$

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

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

$$\textsc{CU}\ \frac{\Omega(x) = \texttt{Nor} \qquad \texttt{fresh}\ (x,n)\ \iota \qquad \Sigma;\Omega \vdash \iota \triangleright \Omega \qquad \texttt{neutral}(\iota)}{\Sigma;\Omega \vdash \texttt{CU}\ (x,n)\ \iota \triangleright \Omega}$$

$$\textsc{LSH}\ \frac{\Omega(x) = \texttt{Nor}}{\Sigma;\Omega \vdash \texttt{Lshift}\ x \triangleright \Omega}$$

$$\textsc{SEQ}\ \frac{\Sigma;\Omega \vdash \iota_1 \triangleright \Omega' \qquad \Sigma;\Omega' \vdash \iota_2 \triangleright \Omega''}{\Sigma;\Omega \vdash \iota_1\ ;\ \iota_2 \triangleright \Omega''}$$

Fig. 13. Select ⟦QASM typing rules

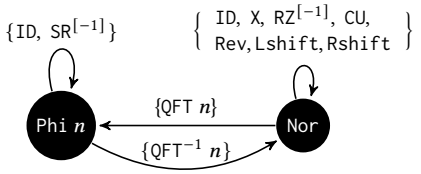


Fig. 14. Type rules' state machine

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 the quantum *no-cloning rule*. For example, we can apply the CU in `rz_adder'` (Figure 8) 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 13. 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 3.1) 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 14.

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 14 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 $\texttt{RZ}^{-1}$) is an z-axis phase rotation operation. Since it applies to Nor-basis, it applies a global phase.

$$
\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(\frac{1}{2^m})\,|1\rangle \\
[\![\texttt{RZ}^{-1}\ m\ (x,i)]\!]\varphi & = \varphi[(x,i) \mapsto \uparrow \texttt{rrz}(m, \downarrow \varphi(x,i))] & \text{where } \texttt{rrz}(m, |0\rangle) = |0\rangle \quad \texttt{rrz}(m, |1\rangle) = \alpha(-\frac{1}{2^m})\,|1\rangle \\
[\![\texttt{SR}\ m\ x]\!]\varphi & = \varphi[\forall i \le m.\ (x,i) \mapsto \uparrow |\Phi(r_i + \frac{1}{2^{m-i+1}})\rangle] & \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 - \frac{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(\frac{y}{2^{n-k}})\rangle) \\
[\![\texttt{QFT}^{-1}\ n\ x]\!]\varphi & = \varphi[x \mapsto \uparrow \texttt{qt}^{-1}(\Sigma(x), \downarrow \varphi(x), n)] & \\
[\![\texttt{Lshift}\ x]\!]\varphi & = \varphi[x \mapsto \texttt{pm}_l(\varphi(x))] & \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. 15. ℚASM semantics

By Theorem 3.4, 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 ℚASM. SR (or $\texttt{SR}^{-1}$) applies an $m+1$ series of RZ (or $\texttt{RZ}^{-1}$) rotations where the i-th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$ (or $\alpha(-\frac{1}{2^{m-i+1}})$). qt applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes \cdots \otimes |b_i\rangle$ (assuming $\Sigma(y) = i$) and n is the degree of approximation. If $n = i$, then the operation is the standard QFT. Otherwise, each qubit in the state is mapped to $|\Phi(\frac{y}{2^{n-k}})\rangle$, which is equal to $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(\frac{y}{2^{n-k}})\,|1\rangle)$ when $k < n$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ when $n \le k$ (since $\alpha(n) = 1$ for any natural number n). $\texttt{qt}^{-1}$ is the inverse function of qt. Note that the input state to $\texttt{qt}^{-1}$ is guaranteed to have the form $\bigotimes_{k=0}^{i-1}(|\Phi(\frac{y}{2^{n-k}})\rangle)$ because it has type Phi n. $\texttt{pm}_l$, $\texttt{pm}_r$, and $\texttt{pm}_a$ are the semantics for Lshift, Rshift, and Rev, respectively.

3.3 ℚASM Metatheory

Soundness. We prove that well-typed ℚASM 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 ℚASM state.

Definition 3.1 (Well-formed ℚASM state). A state φ is *well-formed*, written $\Sigma; \Omega \vdash \varphi$, iff:

- For every $x \in \Omega$ such that $\Omega(x) = \texttt{Nor}$, for every $k < \Sigma(x)$, $\varphi(x,k)$ has the form $\alpha(r)\,|b\rangle$.
- For every $x \in \Omega$ such that $\Omega(x) = \texttt{Phi}\ n$ and $n \le \Sigma(x)$, there exists a value v such that for every $k < \Sigma(x)$, $\varphi(x,k)$ has the form $\alpha(r)\,|\Phi(\frac{v}{2^{n-k}})\rangle$.[4]

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

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

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

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

Fig. 16. Select ℚASM inversion rules

Theorem 3.2. [ℚASM 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 ℚASM states for a given Ω can be interpreted as a subset $\mathcal{S}^d$ of a 2^d-dimensional Hilbert space $\mathcal{H}^d$,[5] 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. 2021]. Because ℚASM'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 ℚASM programs.

[Liyi: good?] Given a qubit size map Σ and type environment Ω, the set of ℚASM 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 3.1. From the ℚASM 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. 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 3.3. $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d) \subseteq (\{\iota\}, \Sigma, \mathcal{H}^d)$ is a subgroupoid.

Theorem 3.4. 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 3.3 suggests that the space $\mathcal{S}^d$ is closed under the application of any well-typed ℚASM operation. Theorem 3.4 says that ℚASM oracles can be safely applied to superpositions over classical states.[6]

ℚASM programs are easily invertible, as shown by the rules in Figure 15. 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 15). This allows us to reuse

[5] 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.

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

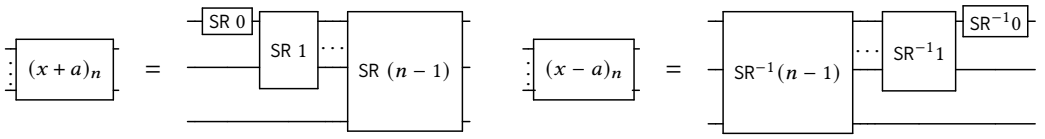


Fig. 17. Addition/subtraction circuits are inverses

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

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

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

Fig. 18. Arith, Bool, Gate Mode Checking

the proof of addition in the proof of subtraction. The inversion function satisfies the following properties:

Theorem 3.5. [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, $[\![\iota; \iota']\!]\varphi = \varphi$.

4 THE FULL DEFINITIONS OF QAFNY

4.1 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 $\bigotimes_n$ 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.

$\bigotimes_n$ 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 7 has several tasks. First, it enforces context mode restrictions. Context mode g is either Cor Q . Q represents the current expression lives inside a quantum conditional or loop, while Crefers to other cases. In a Q context, one cannot perform M-mode operations, i.e., no measurement is allowed. There are other well-formedness enforcement. For example, the session of the Boolean guard b in a conditional/loop is disjoint with the session in the conditional/loop body, i.e., qubits used in a Boolean guard cannot appear in its conditional/loop body.

Second, the type system enforces mode checking for variables and expressions in Figure 17. In QWhile, C-mode variables are evaluated to values during type checking. In a `let` statement (Figure 7), C-mode expression is evaluated to a value n, and the variable x is replaced by n in s. The expression mode checking (Figure 17) 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, such as the index value $x[n]$ in difference expressions in Figure 17. It is worth noting that the session computation ($\uplus$) is also commutative as the last rule in Figure 17.

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 9) to state entanglement properties appearing in a group of qubits. It is worth noting that the entanglement property refers to *eventual entanglement*, .i.e. a group of qubits that are eventually entangled. Entanglement classification is tough and might not be necessary. In most near term quantum algorithms, such as Shor's algorithm [Shor 1994] and Childs' Boolean equation algorithm (BEA) [Childs et al. 2007], programmers care about if qubits eventually become entangled during a quantum loop execution. This is why the normal basis type ($\bigotimes_n$ Nor $\overline{d}$) can also be a subtype of a entanglement type ($\bigotimes_n$ CH $1\{\overline{d}\}$) in our system (Figure 19).

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

A qubit can be described as $\alpha_1 |b_1\rangle + \alpha_2 |b_2\rangle$, where α_1/α_2 are phase amplitudes, and b_1/b_2 are bases. For simplicity, we assume that when we applying a quantum operation on a qubit array y, we either solely change the qubit amplitudes or bases. We identify the former one as $\mathcal{R}$ kind, referring to its similarity of applying an RZ gate; and the latter as $\mathcal{X}$ kind, referring to its similarity of applying an X gate. The entanglement situation between $x[i]$ and y after applying a control statement `if` $(x[i])$ $\{e(y)\}$ is described in Figure 18.

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

	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. 19. 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. 20. Session Type Subtyping

entanglement creation. When y has type Nor, if y's operation is of $\mathcal{X}$ kind, an entanglement between $x[i]$ and y is created, such as the GHZ algorithm; if the operation is of $\mathcal{R}$ kind, there is not entanglement after the control application, such as the Quantum Phase Estimation (QPE) algorithm.

When $x[i]$ and y are both of type Had, if we apply an $\mathcal{X}$ kind operation on y, it does not create entanglement. An example application is the phase kickback pattern. If we apply a $\mathcal{R}$ operation on y, this does create entanglement. This kind of operations appears in state preparations, such as preparing a register x to have state $\sum_{t=0}^{N} i^{-t} |t\rangle$ in Childs' Boolean equation algorithm [Childs et al. 2007]. The main goal for preparing such state is not to entanglement qubits, but to prepare a state with phases related to its bases.

The case when $x[i]$ and y has type Had and CH, respectively, happens in the middle of executing a quantum loop, such as in the Shor's algorithm and BEA. Applying both $\mathcal{X}$ and $\mathcal{R}$ kind operations 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 Section 2.5.

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 $k < i$. [7] 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.

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