

# 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  $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$  where  $\alpha, \beta$  are complex numbers such that  $|\alpha|^2 + |\beta|^2 = 1$ . The  $\alpha$  and  $\beta$  are called *amplitudes*. We frequently write the qubit vector as  $\alpha|0\rangle + \beta|1\rangle$  where  $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  and  $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$  are *computational basis states*. When both  $\alpha$  and  $\beta$  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  $|0\rangle \otimes |1\rangle$ ;  $\mathbb{Q}$ 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  $\mathbb{Q}$ 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.

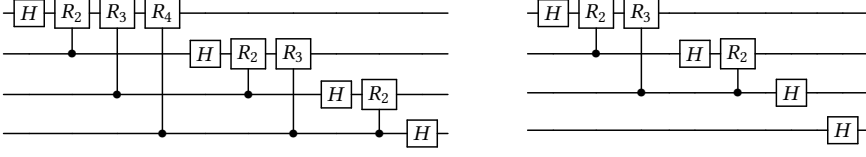


Fig. 1. Example quantum circuits: QFT over 4 qubits (left) and approximate QFT with 3 qubit precision (right).  $R_m$  is a z-axis rotation by  $2\pi/2^m$ .

**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  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ . 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 q. The former represents classical values, represented as a natural number  $n$ , that do

```

99 1 method Shor ( a : int, N : int, n : int, m : int, x : Q[n], y : Q[n] )
100 2   requires (n > 0)
101 3   requires (1 < a < N)
102 4   requires (N < 2^(n-1))
103 5   requires (N^2 < 2^m ≤ 2 * N^2)
104 6   requires (gcd(a, N) == 1)
105 7   requires ( type(x) = Tensor n (Nor 0))
106 8   requires ( type(y) = Tensor n (Nor 0))
107 9   ensures (gcd(N, r) == 1)
108 10  ensures (p.pos ≥ 4 / (PI ^ 2))
109 11  {
110 12    x *= H ;
111 13    y *= cl(y+1); //cl can be omitted.
112 14    for (int i = 0; i < n; x[i]; i++)
113 15      invariant (0 ≤ i ≤ n)
114 16      invariant (saturation(x[0..i]))
115 17      invariant (type(y, x[0..i]) = Tensor n (ch (2^i) {k | j baseof x[0..i] && k = (a^j mod N, j)}))
116 18      //psum(k=b,M,p(k),b(k)) = sum_{k=b}^M p(k)*b(k)
117 19      invariant ((y, x[0..i]) == psum(k=0, 2^i, 1, (a^k mod N, k)))
118 20    {
119 21      y *= cl(a^(2^i) * y mod N);
120 22    }
121 23
122 24  M z := measure(y); //partial measurement, actually measure(y,r) r is the period
123 25  x *= RQFT;
124 26  M p := measure(x); //p.pos and p.base
125 27  var r := post_period(m, p.base) // ∃ t. 2^m * t / r = p.base
126 28  }
127 29

```

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

Bitstring	$c$	$\in$	$(0 \mid 1)^+$
Indexed bitstring set	$\bar{c}(m)$	$::=$	$\{c_0, c_1, \dots, c_{m-1}\}$
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	$g$	$::=$	$c \mid q$
Classical Value	$v$	$::=$	$\frac{n}{r} \mid (r, n)$
Session	$\lambda$	$::=$	$\overline{x[n..m]}$
Full Mode (Kind)	$k$	$::=$	$g \mid \lambda$
Option	$p \in 'a \text{ opt}$	$::=$	$'a \mid \infty$
Uniform Distribution	$\bigcirc$		
Type	$\tau$	$::=$	$\text{Nor } (c \text{ opt}) \mid \text{Had } (\bigcirc \text{ opt}) \mid \text{CH } (\bar{c}(m) \text{ opt})$
Quantum States	$q$	$::=$	$ c\rangle \mid \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^n ( 0\rangle + \alpha(r_j)  1\rangle) \mid \sum_{j=0}^m z_j  c_j\rangle$

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

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

148	Nor $\infty$	$\sqsubseteq_n$	CH $\infty$	$ c\rangle$	$\equiv_n$	$\sum_{j=0}^1  c\rangle$
149	Nor $c$	$\sqsubseteq_n$	CH $\{c\}$	$\sum_{j=0}^1 z_j  c_j\rangle$	$\equiv_n$	$ c_0\rangle$
150	CH $\bar{c}(1)$	$\sqsubseteq_n$	Nor $\bar{c}[0]$	$\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^n ( 0\rangle + \alpha(r_j)  1\rangle)$	$\equiv_n$	$\sum_{j=0}^{2^n} \frac{\alpha(\sum_{k=0}^n r_k \cdot \langle j  \rangle [k])}{\sqrt{2^n}}  j\rangle$
151	Had $p$	$\sqsubseteq_n$	CH $\{\langle j  \rangle   j \in [0, 2^n)\} (2^n)$	$\sum_{j=0}^2 z_j  c_j\rangle$	$\equiv_1$	$\frac{1}{\sqrt{2}} \bigotimes_{j=0}^1 ( 0\rangle + \frac{\sqrt{2}z_1}{z_0}  1\rangle)$
152	CH $\{0, 1\}$	$\sqsubseteq_1$	Had $\infty$			
153	CH $p$	$\sqsubseteq_n$	CH $\infty$			when $c_0 = 0 \quad c_1 = 1$
154	(a) Subtyping			(b) State Equivalence		

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

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 values are defined in terms of sessions ( $\lambda$ ), which can be viewed as clusters of possibly entangled qubits, where the number of qubits is exactly the session length, i.e.,  $\text{len}(x[n..m])$ . Each session consists of different disjoint ranges represented as  $x[n..m]$  that refers the range  $[n, m]$  in a quantum array named  $x$ . For simplicity, we assume that different variable names referring to different quantum arrays without aliasing. 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 ( $\tau$  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 uniformly distributed superposition; otherwise, we represent the type as Had  $\infty$ . The third kind of state is of CH type (CH ( $\bar{c}(m)$  opt)), having the state form  $\sum_{j=0}^m z_j |c_j\rangle$ , referring to that qubits in such session are possibly entangled. The state  $\sum_{j=0}^m z_j |c_j\rangle$  can be viewed as an  $m$  element set of pairs  $z_j |c_j\rangle$ , where  $z_j$  and  $c_j$  are the  $j$ -th amplitude and basis. The well-formed restrictions for the state are three: 1)  $\sum_{j=0}^m |z_j|^2 = 1$  ( $z_j$  is a complex number); 2) length of  $c_j$  is  $n$  for all  $j$  and  $m \leq 2^n$ ; 3) any two bases  $c_j$  and  $c_k$  are distinct if  $j \neq k$ .

In QAFNY, the quantum types and states are associated through bases and equational properties. For each quantum state  $q$ , especially for Nor type state  $|c\rangle$  and CH type state  $\sum_{j=0}^m z_j |c_j\rangle$ , the type factors are either  $\infty$  meaning no bases can be tracked, or having the form  $c$  and  $\bar{c}(m)$  that track the bases of the state  $|c\rangle$  and  $\sum_{j=0}^m z_j |c_j\rangle$ , respectively. For Nor type, this means that the type factor  $c$  (in Nor  $c$ ) and the state qubit format  $|c\rangle$  must be equal; for CH type (CH  $\bar{c}(m)$ ), if the state is  $\sum_{j=0}^m z_j |c_j\rangle$ , the  $j$ -th element  $\bar{c}[j]$  is equal to  $c_j$ . Additionally, QAFNY types permit subtyping relations that 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  $\{c\}$ <sup>1</sup>. Correspondingly, state equivalence relation  $\equiv_n$  describes the two state forms to be equivalent; specifically, the left state term can be used as the

<sup>1</sup>If a qubit array only consists of 0 and 1, it can be viewed as an entanglement of unique possibility.

197	QASM Expr	$\mu$	
198	Parameter	$l$	$::= x \mid x[a]$
199	Arith Expr	$a$	$::= x \mid n \mid (r, n) \mid a + a \mid a * a \mid \dots$
200	Bool Expr	$b$	$::= x[a] \mid (a = a) @ x[a] \mid (a < a) @ x[a] \mid \neg b \mid \dots$
201	Gate Expr	$op$	$::= H \mid QFT^{[-1]}$
202	C/M Moded Expr	$e$	$::= a \mid \text{init } a \mid \text{measure}(y)$
203			$\mid \text{ret}((y, (r, n)))$
204	Statement	$s$	$::= \{\} \mid \text{let } x = e \text{ in } s \mid l \leftarrow op \mid \lambda \leftarrow \mu$
205			$\mid s ; s \mid \text{if } (b) \{s\} \mid \text{for } (\text{int } i := a_1; i < a_2 \ \&\& \ b; \odot i) \{s\}$
206			$\mid x \leftarrow \text{amp}(a) \mid l \leftarrow \text{dis}$
207			

Fig. 5. Core QAFNY syntax. QASM is in Section 3. For an operator  $OP$ ,  $OP^{[-1]}$  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.  $\odot i$  is either  $++i$  or  $--i$ .  $x[a]$  represents the  $a$ -th element in the qubit array  $x$ , while a quantum variable  $x$  represents the whole qubit array.

right one, e.g., a single element entanglement state  $\sum_{j=0}^1 z_j \mid c_j \rangle$  can be used as a Nor type state  $\mid c_0 \rangle$  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 ( $\text{let } x = e \text{ in } s$ ) 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 q classical variable. For simplicity, we assume that we only interacts a q 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.  $\text{let } x = \text{init } a \text{ in } s$  initializes an  $a$ -length qubit array named  $x$  with the value  $\mid 0 \rangle^{\otimes a}$ <sup>2</sup>, and is used in statement  $s$ , while  $\text{let } x = \text{measure}(y) \text{ in } s$  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  $\text{measure}(y)$  to a ghost expression  $\text{ret}((y, (r, n)))$ , 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  $QFT^{-1}$  gate in the end of the quantum phase estimation algorithm. We only permit  $op$  to be  $H$  and  $QFT^{[-1]}$  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 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  $a(2^i) * y \bmod N$  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  $\lambda$  represents the exact quantum qubits mentioned in expression  $\mu$ .

The second row of statements in Figure 5 are control-flow operations.  $s_1 ; s_2$  is a sequential operation.  $\text{if } (b) \{s\}$  is a conditional and  $b$  might contain quantum parameter. Every quantum

<sup>2</sup>  $\mid 0 \rangle^{\otimes n}$  means  $\underbrace{\mid 00 \dots 0 \rangle}_n$ .

$\Omega \vdash x : \Omega(x)$	$\frac{\Omega(x) = (x, 0, \Sigma(x))}{\Omega \vdash x[n] : [(x, n, n+1)]}$	$\frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2}{\Omega \vdash a_1 + a_2 : q_1 \sqcup q_2}$	$\frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2}{\Omega \vdash a_1 * a_2 : q_1 \sqcup q_2}$
$\frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2 \quad \Omega \vdash a_3 : q_3}{\Omega \vdash (a_1 = a_2)@x[n] : q_1 \sqcup q_2 \sqcup q_3}$	$\frac{\Omega \vdash a_1 : q_1 \quad \Omega \vdash a_2 : q_2 \quad \Omega \vdash a_3 : q_3}{\Omega \vdash (a_1 < a_2)@x[n] : q_1 \sqcup q_2 \sqcup q_3}$	$\frac{\Omega \vdash b : q}{\Omega \vdash \neg b : q}$	$\frac{\Omega \vdash e : \zeta_2 \sqcup \zeta_1}{\Omega \vdash e : \zeta_1 \sqcup \zeta_2}$
$\zeta_1 \sqcup \zeta_2 = \zeta_1 \sqcup \zeta_2 \zeta \sqcup g = \zeta \quad g \sqcup \zeta = \zeta \quad c \sqcup c = c \quad q \sqcup c = q \quad c \sqcup q = q \quad c \leq q \leq \zeta$ $\perp \sqcup l = l \quad l \sqcup \perp = l \quad [(x, v_1, v_2)] \sqcup [(y, v_3, v_4)] = [(x, v_1, v_2), (y, v_3, v_4)]$ $[v_2, v_2] \cap [v_3, v_4] \neq \emptyset \Rightarrow [(x, v_1, v_2)] \sqcup [(x, v_3, v_4)] = [(x, \min(v_1, v_3), \max(v_2, v_4))]$			

Fig. 6. Arith, Bool, Gate Mode Checking

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  $(\text{int } i := a_1; i < a_2 \ \&\& \ b; \odot i) \{s\}$  is a possibly quantum for-loop depending on if the Boolean guard  $b$  contains quantum parameters. A classical variable  $i$  is introduced and it is initialized as the lower bound  $a_1$ , increments/decrements in each loop step defined by a monotonic function  $\odot i$ , and ends at the upper bound  $a_2$ . For simplicity, we require  $\odot i$  to be either  $++i$  or  $--i$ .

The last row contains quantum reflection operations, including quantum amplifier ( $x \leftarrow \text{amp}(a)$ ) and diffusion operation ( $l \leftarrow \text{dis}$ ), which are used to increase and average the occurrence likelihood of some quantum bases in a quantum state, respectively. For example, if a session  $x[0, n] \sqcup \lambda$ <sup>3</sup> has a CH type state  $\sum_{j=0}^m z_j |c_j\rangle$ ,  $c_k = (|a|) \sqcup c$ , and  $\text{len}(|a|) = n$ ;  $x \leftarrow \text{amp}(a)$  then increases the amplitude  $z_k$  for  $c_k$ .

### 2.3 Type Checking: A Quantum Session Type System

The QWhile type system can be viewed as a mapping from lists of factors ( $x$  or  $x[n]$ ) to QWhile types in Figure 12. Generally, factors represent a range of locations in a "quantum heap". Variable  $x$  can be viewed as a range  $(x, 0, \Sigma(x))$ , meaning the heap range starting at  $x$  and ending at  $x + n$ . Index  $x[n]$  can be viewed as a range  $(x, n, n + 1)$ . Thus, a list of **quantum** factors is essentially a list of disjoint fragments, which it is called a *session*.

A type is written as  $\otimes_n t$ , where  $n$  refers to the total number of qubits in a session, and  $t$  describes the qubit state form. A session being type  $\otimes_n \text{Nor } \bar{d}$  means that every qubit is in normal basis (either  $|0\rangle$  or  $|1\rangle$ ), and  $\bar{d}$  describes basis states for the qubits. The type corresponds to a single qubit basis state  $\alpha(n) |\bar{d}\rangle$ , where the global phase  $\alpha(n)$  has the form  $e^{2\pi i \frac{1}{n}}$  and  $\bar{d}$  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.

$\otimes_n \text{Had } w$  means that every qubit in the session has the state:  $(\alpha_1 |0\rangle + \alpha_2 |1\rangle)$ ; the qubits are in superposition but they are not entangled.  $\bigcirc$  represents the state is a uniform superposition, while  $\infty$  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  $\otimes_n \text{CH } m\beta$  are supposedly entangled (eventual entanglement below).  $m$  refers to the number of possible different entangled states in the session, and the bitstring indexed set  $\beta$  describes each of these states, while every element in  $\beta$  is indexed by  $i \in [0, m)$ .  $\beta$

<sup>3</sup> $\sqcup$  is the session and bitstring concatenation operation.



can also be  $\infty$  meaning that the entanglement structure is unknown. For example, in quantum phase estimation, after applying the  $\text{QFT}^{-1}$  operation, the state has type  $\bigotimes_n \text{CH } m\infty$ . In such case, the only quantum operation to apply is a measurement. If a session has type  $\bigotimes_n \text{CH } m\beta$  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  $\bigotimes_n \text{CH } 2\{\bar{0}^n, \bar{1}^n\}$ , where  $\bar{d}^n$  is a  $n$ -bit string having bit  $d$ .

The type  $\bigotimes_n \text{CH } m\beta$  corresponds to the value form  $\sum_{k=0}^m \theta_k |\bar{d}_k\rangle$ .  $\theta_k$  is an amplitude real number, and  $\bar{d}_k$  is the basis. Basically,  $\sum_{k=0}^m \theta_k |\bar{d}_k\rangle$  represents a size  $m$  array of basis states that are pairs of  $\theta_k$  and  $\bar{d}_k$ . For a session  $\zeta$  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  $\theta_k$  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  $\Omega$  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,  $\zeta$  is the session of  $s$ , and  $\tau$  is  $\zeta$ 's type. The QWhile type system in Figure 9 has several tasks. First, it enforces context mode restrictions. Context mode  $g$  is either cor or 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 15. In QWhile, c-mode variables are evaluated to values during type checking. In a let statement (Figure 9), c-mode expression is evaluated to a value  $n$ , and the variable  $x$  is replaced by  $n$  in  $s$ . The expression mode checking (Figure 15) has the judgment:  $\Omega \vdash (a \mid b) : q$ . It takes a mode environment  $\Omega$ , and an expression  $(a, b)$ , and judges if the expression has the mode  $g$  if it contains only classical values, or a quantum session  $\zeta$  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 15. It is worth noting that the session computation ( $\uplus$ ) is also commutative as the last rule in Figure 15.

Third, by generating the session of an expression, the QWhile type system assigns a type  $\tau$  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  $\tau$  (Figure 12) 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  $(\bigotimes_n \text{Nor } \bar{d})$  can also be a subtype of a entanglement type  $(\bigotimes_n \text{CH } 1\{\bar{d}\})$  in our system (Figure 8).

**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  $\alpha_1/\alpha_2$  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,

	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. 7. Control Gate Entanglement Situation

$$\bigotimes_n \text{Nor } \bar{d} \sqsubseteq \bigotimes_n \text{CH } 1\{\bar{d}\} \quad \bigotimes_n \text{CH } 2^n \beta \sqsubseteq \bigotimes_n \text{CH } 2^n \infty \quad \bigotimes_n \text{Had } \bigcirc \sqsubseteq \bigotimes_n \text{CH } 2^n \mathcal{P}(n)$$

Fig. 8. Session Type Subtyping

referring to its similarity of applying an RZ gate; and the latter as  $\mathcal{X}$  kind, referring to its similarity of applying an  $\mathcal{X}$  gate. The entanglement situation between  $x[i]$  and  $y$  after applying a control statement if  $(x[i]) \{e(y)\}$  is described in Figure 7.

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

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

The case when  $x[i]$  and  $y$  has type Had and CH, respectively, happens in the middle of executing a quantum loop, such as in the Shor's algorithm and BEA. Applying both  $\mathcal{X}$  and  $\mathcal{R}$  kind operations 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 2.4.

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  $k < i$ .<sup>4</sup> The

<sup>4</sup>When  $k < i$ ,  $x[i] = 1$  while  $\neg(k < i)$ ,  $x[i] = 0$ .



TA-NOR

$$\frac{\Omega \vdash \bar{l} : \zeta \quad \llbracket a \rrbracket |\bar{d}\rangle = \alpha |\bar{d}'\rangle}{\Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{Nor } \bar{d}] \vdash_g \bar{l} \leftarrow a : \zeta \triangleright_g \bigotimes_n \text{Nor } \bar{d}'}$$

TA-HAD

$$\frac{\Omega \vdash \bar{l} : \zeta \quad \zeta \subseteq \zeta' \quad (\forall \bar{d}. \llbracket a \rrbracket |\bar{d}\rangle = |\bar{d}'\rangle)}{\Omega, \mathcal{T} [\zeta' \mapsto \bigotimes_n \text{Had } \bar{O}] \vdash_g \bar{l} \leftarrow a : \zeta' \triangleright \bigotimes_n \text{Had } \bar{O}}$$

TMEA

$$\frac{\begin{array}{l} \Omega \vdash y : \zeta \quad n' = \text{size}(\zeta') \quad m' = \text{size}(\{\bar{c}' | \bar{c} \cdot \bar{c}' \in \beta \cdot \beta' \wedge \bar{c} = \text{val}(x)\}) \\ \Omega[x \mapsto q], \mathcal{T} [\zeta' \mapsto \bigotimes_{n'} \text{CH } m' \{\bar{c}' | \bar{c} \cdot \bar{c}' \in \beta \cdot \beta' \wedge \bar{c} = \text{val}(x)\}] \vdash_c s : \zeta'' \triangleright \tau \end{array}}{\Omega, \mathcal{T} [\zeta \uplus \zeta' \mapsto \bigotimes_n \text{CH } m(\beta \cdot \beta')] \vdash_c \text{let } x = \text{measure}(y) \text{ in } s : \zeta'' \triangleright \tau}$$

TEXP

$$\frac{\Omega[x \mapsto c], \mathcal{T} \vdash_g s[v/x] : \zeta \triangleright \tau}{\Omega, \mathcal{T} \vdash_g \text{let } x = v \text{ in } s : \zeta \triangleright \tau}$$

TA-CH

$$\frac{\Omega \vdash \bar{l} : \zeta \quad \zeta' = \zeta \uplus \zeta_r}{\Omega, \mathcal{T} [\zeta' \mapsto \bigotimes_n \text{CH } k\beta] \vdash_g \bar{l} \leftarrow a : \zeta' \triangleright \bigotimes_n \text{CH } k\{\bar{d}' \cdot \bar{d}_r \mid \bar{d} \cdot \bar{d}_r \in \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r) \wedge \llbracket a \rrbracket |\bar{d}\rangle = \alpha |\bar{d}'\rangle\}}$$

TA-Mu

$$\frac{\Omega, \mathcal{T} [\zeta_1 \uplus \zeta_3 \uplus \zeta_2 \uplus \zeta_4 \mapsto \bigotimes_n \text{CH } k\beta_1 \cdot \beta_3 \cdot \beta_2 \cdot \beta_4] \vdash_g s : \zeta_1 \uplus \zeta_3 \uplus \zeta_2 \uplus \zeta_4 \triangleright \bigotimes_n \text{CH } k\beta_1 \cdot \beta'_3 \cdot \beta'_2 \cdot \beta_4}{\Omega, \mathcal{T} [\zeta_1 \uplus \zeta_2 \uplus \zeta_3 \uplus \zeta_4 \mapsto \bigotimes_n \text{CH } k\beta_1 \cdot \beta_2 \cdot \beta_3 \cdot \beta_4] \vdash_g s : \zeta_1 \uplus \zeta_2 \uplus \zeta_3 \uplus \zeta_4 \triangleright \bigotimes_n \text{CH } k\beta_1 \cdot \beta'_2 \cdot \beta'_3 \cdot \beta_4}$$

TSEQ-1

$$\frac{\begin{array}{l} \Omega, \mathcal{T} \vdash_g s_1 : \zeta \triangleright \bigotimes_n \text{Nor } \bar{d} \quad \zeta \uplus \zeta' \\ \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{Nor } \bar{d}] \vdash_g s_2 : \zeta' \triangleright \bigotimes_{n'} \text{Nor } \bar{d}' \end{array}}{\Omega, \mathcal{T} \vdash_g s_1 ; s_2 : \zeta \uplus \zeta' \triangleright \bigotimes_{n+n'} \text{Nor } \bar{d} \cdot \bar{d}'}$$

TSEQ-2

$$\frac{\begin{array}{l} \Omega, \mathcal{T} \vdash_g s_1 : \zeta \triangleright \bigotimes_n \text{CH } k\beta \\ \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{CH } k\beta] \vdash_g s_2 : \zeta' \triangleright \tau \end{array}}{\Omega, \mathcal{T} \vdash_g s_1 ; s_2 : \zeta' \triangleright \tau}$$

TIF

$$\frac{\begin{array}{l} \Omega \vdash b(@x[v]) : \zeta \quad \mathcal{T}(\zeta) = \bigotimes_n \text{CH } k(\beta_1 \cdot 0 \cup \beta_2 \cdot 1) \\ \zeta \uplus \zeta' \quad \text{last}(\zeta) = (x, v, v+1) \quad \Omega, \mathcal{T} \vdash_q s : \zeta' \triangleright \bigotimes_{n'} \text{CH } k'\beta' \end{array}}{\Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{CH } k\beta] \vdash_g \text{if } (x[v]) \{s\} : \zeta \uplus \zeta' \triangleright \bigotimes_{n+n'} \text{CH } (k \times k')(\beta_1 \cdot 0 \cup \beta_2 \cdot 1 \cup \beta' \cdot 1 \cup \beta')}$$

TLOOP

$$\frac{\begin{array}{l} v_1 \leq v < v_2 \quad \Omega \vdash f(v) : c \quad \Omega \vdash b(@x[v]) : \zeta(v) \\ \zeta(v) \uplus \zeta'(v) \quad \text{last}(\zeta) = (x, v, v+1) \quad \Omega, \mathcal{T} \vdash_g \text{if } (b(@x[v])) \{s\} : \zeta(v) \uplus \zeta'(v) \triangleright \tau(v) \end{array}}{\Omega, \mathcal{T} \vdash_g \text{for } (\text{init } iv_1 ; i < v_2 \ \&\& \ b(@x[i]) ; f(i)) \{T(i)\} P(i)s : \zeta(v_2) \uplus \zeta'(v_2) \triangleright \tau(v_2)}$$

TDIS

$$\frac{\Omega \vdash x : \zeta \quad \zeta' = \zeta \uplus \zeta_r \quad n' = \text{size}(\zeta) \quad m' = \text{size}(\{\mathcal{P}(n') \cdot \bar{d}_r \mid \bar{d} \cdot \bar{d}_r \in \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r)\})}{\Omega, \mathcal{T} [\zeta' \mapsto \bigotimes_n \text{CH } m\beta] \vdash_g \text{diffuse}(x) : \zeta' \triangleright \bigotimes_n \text{CH } m'\{\mathcal{P}(n') \cdot \bar{d}_r \mid \bar{d} \cdot \bar{d}_r \in \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r)\}}$$

$\bar{d} \cdot \bar{d}'$  : list concatenation.  $\zeta \uplus \zeta'$  : The two sessions are disjoint.  
 $\beta(\downarrow \zeta)$  : Get the position range of  $\zeta$  in the session and form a new set  
of bitstrings containing only the positions in the range.  
 $\llbracket a \rrbracket |\bar{d}\rangle$  :  $\mathbb{Q}$ ASM semantics of interpreting reversible expression  $a$  in Figure 17.

Fig. 9. Session Type System

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.

**Session Type System.** Selected type rules are given in Figure 9. As we have mentioned above, the type system tracks possible eventual entanglement for a group of qubits, which we named a session. The type judgment is given as  $\Omega, \mathcal{T} \vdash_g s : \zeta \triangleright \tau$ .

Rule TEXP is the type rule for c-mode expressions. The expression  $a$  is evaluated and variable  $x$  is substituted with the value  $v$  in  $s$ . TMEA is a similar rule as TEXP, but for  $M$ -mode variable. We allow partial measurements in QWhile. Thus, we need to find out a possible entanglement session  $\zeta \uplus \zeta'$  that contains  $y$ 's session ( $\zeta$ ), that is going to disappear because of measurement. Then, we re-calculate the entanglement type information for  $\zeta'$ .

TA-NOR, TA-HAD and TA-CH are rules for quantum assignments with different input types.  $\llbracket a \rrbracket$  appearing in these rules is a semantic function for interpreting the expressions  $a$ . The semantic function takes an expression in Nor type and output a Nor value, i.e., inputting classical values and output classical results. The semantics of  $\mathbb{Q}_{\text{QASM}}$  (Figure 17) and the arithmetic language  $\mathbb{Q}_{\text{QIMP}}$  is the role model of such semantic function. In TA-HAD, when  $\bar{l}$  is in uniform superposition (Had  $\odot$ ), for every bit in  $\bar{l}$ , if the semantic function judges that its global phase keeps in uniformity, i.e., 1, the output type is still a uniform superposition without entanglement. In TA-CH, the factor  $\bar{l}$  that is assigning might be a sub-session  $\zeta$  of the whole entanglement session  $\zeta'$ , such that  $\zeta' = \zeta \cdot \zeta_r$ . Here, for every element  $\bar{d} \cdot \bar{d}_r \in \beta$ , we find out the corresponding part  $\bar{d}$  belonging to the session  $\zeta$  ( $\bar{d} \cdot \bar{d}_r \in \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r)$ ), and updates the  $\bar{d}$  in the result type.

Rules TSeq-1 and TSeq-2 describe the type for a sequence operation. If  $s_1$  and  $s_2$  are of type Nor or Had (rule TSeq-1), the output session order can be mutated as long as the two sessions are disjoint. If the two sessions are not disjoint, we only need to keep the type for  $\zeta'$ , since it is obvious that  $\zeta \subseteq \zeta'$ . If  $s_1$  and  $s_2$  are of type CH (rule TSeq-2), we only permit the case when  $\zeta \subseteq \zeta'$  for simplicity. It has no technical difficulty to allow  $\tau$  to be a list and two entanglement sessions to bind together, but it makes the type system a lot messier, and there is no current algorithms that require the modification of two distinct entanglement sessions inside a conditional block. On the other hand, if two distinct entanglement sessions live in a conditional block, the block can always be split into two different conditionals with the same Boolean guard.

Rule TIF describes the type for conditionals when the Boolean guard  $b(@x[v])$  having type CH, and  $x[v]$  is the result bit storing the Boolean evaluation result. The result type of such conditionals is an CH type by merging the session of  $b(@x[v])$  into the entanglement session. Assume that the CH bases for  $b(@x[v])$  are  $\beta_1 \cdot 0$  and  $\beta_2 \cdot 1$ , meaning that we can split nicely for all possible bases of a quantum state to two different sets where the last bit, which represents the  $x[v]$  position, is 0 and 1. For the 0 set, we extend  $\beta_1 \cdot 0$  to  $\beta_1 \cdot 0 \cdot \beta$  by doing a cross product of the elements in set  $\beta_1 \cdot 0$  and set  $\beta$ . For the 1 set, the cross product is applied on set  $\beta_1 \cdot 0$  and set  $\beta'$ , where  $\beta'$  is the result type bases of the body statement  $s$ . It is worth noting that by the subtyping relation in Figure 8, any type can be turned to a CH type. When the Boolean guard has type Nor, it is no more than a classical conditional. When the Boolean guard has type Had, its behavior is similar to the CH case.

Rule TLOOP describes the type for a for-loop. It is a generalization of the conditional case when the Boolean guard  $b(@x[i])$  having type CH. In the type rule, we pick a number  $v$  to represent variable  $i$ , and check if a single loop step  $\text{if } (b(@x[v])) \{s\}$  is well typed. If so, we can conclude that when we replace  $v$  by  $v_2$ , the for-loop is type checked.

Rule TDIS types a diffusion operator. The rule finds the right part of  $x$  in the session  $\zeta'$ . For every right session bitstring  $\bar{d}$  in  $\bar{d}_l \cdot \bar{d} \cdot \bar{d}_r$ , it generates a set of new type sequence by replacing  $\bar{d}$  with elements in the power set  $\mathcal{P}(n')$ , where  $n'$  is the bit length of  $\bar{d}$ . Here, we need to compute the size of the new bitstring set as  $m'$ . Sometimes, this computation can be hard, but for most quantum algorithms, depending on the session data structures, the size can be computed effectively.

PA-NOR	PA-CH
$(\mathcal{T}, \mathcal{T}') \models_g (T, \bar{l} \leftarrow a, T') : \zeta \triangleright \bigotimes_n \text{Nor } \bar{d}$	$(\mathcal{T}, \mathcal{T}') \models_g (T, \bar{l} \leftarrow a, T') : \zeta \uplus \zeta' \triangleright \bigotimes_n \text{CH } m\beta \quad \mathcal{T}(\bar{l}) = \zeta$
$\Omega \vdash_g \{T\} \{P[\llbracket a \rrbracket \zeta / \zeta]\} \bar{l} \leftarrow a \{T'\} \{P\}$	$\Omega \vdash_g \{T\} \{P[\forall k < m. \llbracket a \rrbracket (\zeta[k]) / \zeta[k]]\} \bar{l} \leftarrow a \{T'\} \{P\}$
P-MEA	
$(\mathcal{T}[\forall \zeta' . \zeta \uplus \zeta' \mapsto \bigotimes_{n+n'} \text{CH } (m \times m')(\beta \cdot \beta')], \mathcal{T}'[\forall \zeta' . \zeta' \mapsto \bigotimes_{n'} \text{CH } m'\beta']) \models_c (T, y, T'') : \zeta \triangleright \bigotimes_n \text{CH } m\beta$	
$v < m \quad \Omega[x \mapsto M, y \mapsto \perp] \vdash_c \{T''\} \{P[(\text{as}^2(\zeta[v]), \text{bs}(\zeta[v]))/x, \perp/\zeta]\} s \{T'\} \{Q\}$	
$\Omega \vdash_c \{T\} \{P\} \text{let } x = \text{measure}(y) \text{ in } s \{T'\} \{Q\}$	
P-IF	
$(\mathcal{T}, \mathcal{T}') \models_q (T, s, T') : \zeta' \triangleright \bigotimes_{n'} \text{CH } m'\beta'$	
$\Omega \vdash b(@x[v]) : \zeta \uplus [(x, v, v+1)] \quad \mathcal{T}(\zeta \uplus [(x, v, v+1)]) = \bigotimes_n \text{CH } 2m(\beta_1 \cdot 0 \cup \beta_2 \cdot 1)$	
$\Omega \vdash_g \{T\} \{P[(\zeta \uplus 0 \uplus \zeta') ++ (\zeta \uplus 1 \uplus \llbracket s \rrbracket \zeta') / \zeta \uplus [(x, v, v+1)] \uplus \zeta'] \text{ if } (b(@x[v])) \{s\} \{T'\} \{P\}$	
P-LOOP	
$(\mathcal{T}, \mathcal{T}') \models_g (T(i), \text{if } (b(@x[i])) \{s\}, T(f(i))) : \zeta \triangleright \tau$	
$\Omega \vdash_g \{T(i)\} \{P(i)\} \text{if } (x[i]) \{s\} \{T(f(i))\} \{P(f(i))\}$	
$\Omega \vdash_g \{T(a_1)\} \{P(a_1)\} \text{for } (\text{init } ia_1; i < a_2 \ \&\& \ x[i]; f(i)) \{T(i)\} P(i)s \{T(a_2)\} \{P(a_2)\}$	
P-DIS	
$(\mathcal{T}, \mathcal{T}') \models_g (T, \text{diffuse}(x), T') : \zeta \triangleright \bigotimes_{n'} \text{CH } m'\beta'$	
$\mathcal{T}(x) = \{\zeta : \bigotimes_{n'} \text{CH } m\beta\} \quad \zeta = \zeta' \uplus (x, 0, \Sigma(x)) \quad \beta_1 \cdot \beta_2 = \beta(\downarrow (x, 0, \Sigma(x))) \cdot \beta(\downarrow \zeta')$	
$\Omega \vdash_g \{T\} \{P[\text{dis}(n', \zeta, \beta, \beta_1, \beta_2) / \zeta]\} \text{diffuse}(x) \{T'\} \{P\}$	
$\text{In}(\zeta) : \text{length of } \zeta \quad \text{as}(\zeta[v]) : \text{the amplitude of } \zeta[v] \quad \text{bs}(\zeta[v]) : \text{the base of } \zeta[v] \quad ++ : \text{array concatenation.}$	
$\text{dis}(n, \zeta, \beta, \beta_1, \beta_2) \equiv \{\frac{1}{2^{n-1}} (\sum_k \text{as}(\zeta[k]) - \text{as}(\zeta[j])) \beta_1[j] \mid \beta_1[j] \in \mathcal{P}(n)\}$	
$\cup \bigcup_{j \in \beta_2} \{\frac{1}{2^n} \sum_k \text{as}(\zeta[k]) z \mid z \in \mathcal{P}(n) \wedge (\forall k. z \cdot \beta_2[j] \neq \beta[k])\}$	

Fig. 10. Selected Proof System Rules

## 2.4 Logic Proof System

The reason of having the session type system in Figure 9 is to enable the proof system given in Figure 10. Every proof rule is a structure as  $\Omega \vdash_g \{T\} \{P\} s \{T'\} \{Q\}$ , where  $g$  and  $\Omega$  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 12 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  $\zeta$  has type Nor, it is a singleton array, so the substitution  $\llbracket a \rrbracket \zeta / \zeta$  means that we substitute the singleton array by a term with the  $a$ 's application. When  $\zeta$  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  $\mathbb{H}$  application results in a pre-condition  $\mathbb{H}[(x, 0, n)] = \bigotimes_{k=0}^n |\Phi(0)\rangle$ , 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  $\zeta$ , but it might be a part of an entangled session  $\zeta \uplus \zeta'$ . After the measurement,  $M$ -mode  $x$  has the measurement result  $(\text{as}(\zeta[v])^2, \text{bs}(\zeta[v]))$  coming from one possible basis state of  $y$  (picking a random index  $v$  in  $\zeta$ ),  $\text{as}(\zeta[v])$  is the amplitude and  $\text{bs}(\zeta[v])$  is the base. We also remove  $y$  and its session  $\zeta$  ( $\perp/\zeta$ ) in the new pre-condition because it is measured away. The removal means that the entangled session  $\zeta \uplus \zeta'$  is replaced by  $\zeta'$  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  $\bigotimes_n \text{CH } 2m(\beta_1 \cdot 0 \cup \beta_2 \cdot 1)$ . 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  $(\zeta \uplus 0 \uplus \zeta') ++ (\zeta \uplus 1 \uplus \llbracket s \rrbracket \zeta')$ , where we double the array: if the  $x[v]$  position is 0, we concatenate the current session  $\zeta'$  for the conditional body, if  $x[v] = 1$ , we apply  $\llbracket s \rrbracket$  on the array  $\zeta'$  and concatenate it to  $(\zeta \uplus 1)$ .

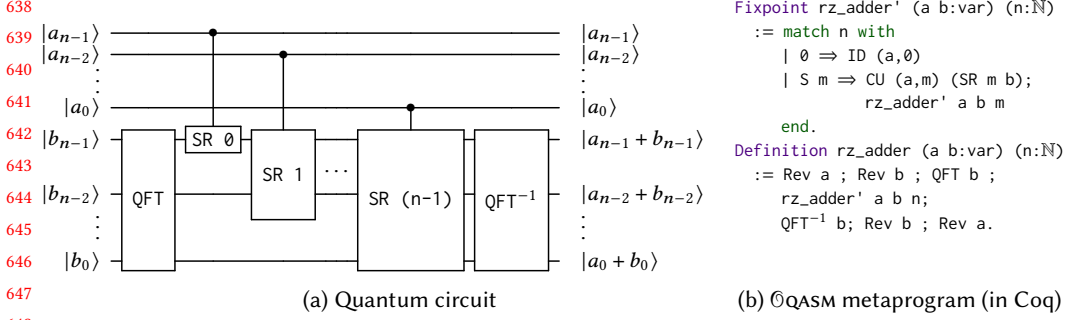
Rule P-Loop is an initiation of the classical while rule in Hoare Logic with the loop guard possibly having quantum variables. In QWhile, we only has for-loop structure and we believe it is enough to specify any current quantum algorithms. For any  $i$ , if we can maintain the loop invariant  $P(i)$  and  $T(i)$  with the post-state  $P(f(i))$  and  $T(f(i))$  for a single conditional  $\text{if } (x[i]) \{s\}$ , 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  $\text{diffuse}(x)$ . The quantum semantics for  $\text{diffuse}(x)$  is  $\frac{1}{2^n} (2 \sum_{j=0}^{2^n} (\sum_{j=0}^n \alpha_j) |i\rangle - \sum_{j=0}^n \alpha_j |x_j\rangle)$ . As an array operation,  $\text{diffuse}(x)$  with the session  $\zeta$  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 Interactive 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>

Fig. 11. Example  $\textcircled{\text{Q}}$ ASM 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	$\tau$	$::=$	$\text{Nor} \mid \text{Phi } n$
Unphased qubit	$\bar{q}$	$::=$	$ b\rangle \mid  \Phi(r)\rangle$
Qubit	$q$	$::=$	$\alpha(r)\bar{q}$
State (length $d$ )	$\varphi$	$::=$	$q_1 \otimes q_2 \otimes \dots \otimes q_d$

Fig. 12.  $\textcircled{\text{Q}}$ ASM state syntax

### 3 $\textcircled{\text{Q}}$ ASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

We designed  $\textcircled{\text{Q}}$ ASM to be able to express efficient quantum oracles that can be easily tested and, if desired, proved correct.  $\textcircled{\text{Q}}$ ASM operations leverage both the standard computational basis and an alternative basis connected by the quantum Fourier transform (QFT).  $\textcircled{\text{Q}}$ ASM's type system tracks the bases of variables in  $\textcircled{\text{Q}}$ ASM programs, forbidding operations that would introduce entanglement.  $\textcircled{\text{Q}}$ ASM states are therefore efficiently represented, so programs can be effectively tested and are simpler to verify and analyze. In addition,  $\textcircled{\text{Q}}$ ASM 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  $\textcircled{\text{Q}}$ ASM 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 11. The Coq function `rz_adder` generates an  $\textcircled{\text{Q}}$ ASM program that adds two natural numbers  $a$  and  $b$ , each of length  $n$  qubits.

#### 3.1 $\textcircled{\text{Q}}$ ASM States

An  $\textcircled{\text{Q}}$ ASM program state is represented according to the grammar in Figure 12. A state  $\varphi$  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  $\varphi$  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  $\textcircled{\text{Q}}$ ASM program on any well-formed  $\textcircled{\text{Q}}$ ASM state never causes qubits to be entangled.

A qubit value  $q$  has one of two forms  $\bar{q}$ , scaled by a global phase  $\alpha(r)$ . The two forms depend on the *basis*  $\tau$  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  $\iota ::= \text{ID } p \mid \text{X } p \mid \text{RZ}^{[-1]} n p \mid \iota ; \iota$   
 $\mid \text{SR}^{[-1]} n x \mid \text{QFT}^{[-1]} n x \mid \text{CU } p \iota$   
 $\mid \text{Lshift } x \mid \text{Rshift } x \mid \text{Rev } x$

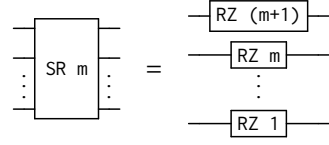


Fig. 13.  $\mathbb{Q}$ ASM syntax. For an operator  $\text{OP}$ ,  $\text{OP}^{[-1]}$  indicates that the operator has a built-in inverse available.

Fig. 14. 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  $\varphi$ . 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 $\mathbb{Q}$ ASM Syntax, Typing, and Semantics

[ Liyi: add RZ gate back ]

Figure 13 presents  $\mathbb{Q}$ ASM's syntax. An  $\mathbb{Q}$ ASM program consists of a sequence of instructions  $\iota$ . Each instruction applies an operator to either a variable  $x$ , which represents a group of qubits, or a position  $p$ , which identifies a particular offset into a variable  $x$ .

The instructions in the first row correspond to simple single-qubit quantum gates—ID  $p$ , X  $p$ , and  $\text{RZ}^{[-1]} n p$ —and instruction sequencing. The instructions in the next row apply to whole variables: QFT  $n x$  applies the AQFT to variable  $x$  with  $n$ -bit precision and  $\text{QFT}^{-1} n x$  applies its inverse. If  $n$  is equal to the size of  $x$ , then the AQFT operation is exact.  $\text{SR}^{[-1]} n x$  applies a series of RZ gates (Figure 14). Operation CU  $p \iota$  applies instruction  $\iota$  *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 11(b) uses many of these instructions; e.g., it uses QFT and  $\text{QFT}^{-1}$  and applies CU to the  $m$ th position of variable  $a$  to control instruction SR  $m b$ .

In the last row of Figure 13, 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  $\mathbb{Q}$ ASM 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  $\mathbb{Q}$ ASM to allow reasoning about a larger class of quantum programs, while still guaranteeing a lack of entanglement. In ??, we show how  $\mathbb{Q}$ ASM 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  $\mathbb{Q}$ ASM, typing is with respect to a *type environment*  $\Omega$  and a predefined *size environment*  $\Sigma$ , which map  $\mathbb{Q}$ ASM variables to their basis and size (number of qubits), respectively. The typing judgment is written  $\Sigma; \Omega \vdash \iota \triangleright \Omega'$  which states that  $\iota$  is well-typed under  $\Omega$  and  $\Sigma$ , and transforms the variables' bases to be as in  $\Omega'$  ( $\Sigma$  is unchanged). [ Liyi: good? ]  $\Sigma$  is fixed because the number of qubits in an execution is always fixed. It is generated in the high level language compiler, such as `QIMP` in ??. The algorithm generates  $\Sigma$  by taking an `QIMP` program and scanning through all the variable initialization statements. Select type rules are given in Figure 15; the rules not shown (for ID, Rshift, Rev,  $\text{RZ}^{-1}$ , and  $\text{SR}^{-1}$ ) are similar.

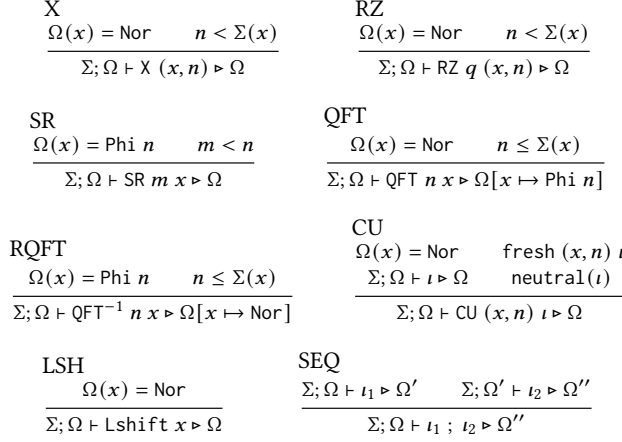


Fig. 15. Select QASM typing rules

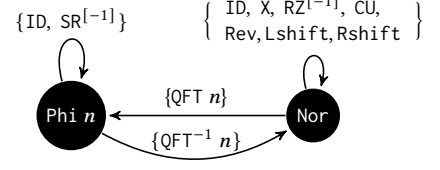


Fig. 16. Type rules' state machine

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 11) 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 16. For example, `QFT n` transforms a variable from Nor to Phi `n` (rule QFT), while `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 `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  $\llbracket \cdot \rrbracket$  from an instruction  $\iota$  and input state  $\varphi$  to an output state  $\varphi'$ , written  $\llbracket \iota \rrbracket \varphi = \varphi'$ , shown in Figure 17.

Recall that a state  $\varphi$  is a tuple of  $d$  qubit values, modeling the tensor product  $q_1 \otimes \dots \otimes q_d$ . The rules implicitly map each variable  $x$  to a range of qubits in the state, e.g.,  $\varphi(x)$  corresponds to some sub-state  $q_k \otimes \dots \otimes q_{k+n-1}$  where  $\Sigma(x) = n$ . Many of the rules in Figure 17 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  $\alpha(b)\bar{q}$  to an unphased qubit  $\bar{q}$ .

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.

785	$\llbracket \text{ID } p \rrbracket \varphi$	$= \varphi$	
786	$\llbracket X(x, i) \rrbracket \varphi$	$= \varphi[x, i \mapsto \uparrow \text{xg}(\downarrow \varphi(x, i))]$	where $\text{xg}( 0\rangle) =  1\rangle \quad \text{xg}( 1\rangle) =  0\rangle$
787	$\llbracket \text{CU}(x, i) \iota \rrbracket \varphi$	$= \text{cu}(\downarrow \varphi(x, i), \iota, \varphi)$	where $\text{cu}( 0\rangle, \iota, \varphi) = \varphi \quad \text{cu}( 1\rangle, \iota, \varphi) = \llbracket \iota \rrbracket \varphi$
788	$\llbracket \text{RZ } m(x, i) \rrbracket \varphi$	$= \varphi[x, i \mapsto \uparrow \text{rz}(m, \downarrow \varphi(x, i))]$	where $\text{rz}(m,  0\rangle) =  0\rangle \quad \text{rz}(m,  1\rangle) = \alpha(\frac{1}{2^m})  1\rangle$
789	$\llbracket \text{RZ}^{-1} m(x, i) \rrbracket \varphi$	$= \varphi[x, i \mapsto \uparrow \text{rrz}(m, \downarrow \varphi(x, i))]$	where $\text{rrz}(m,  0\rangle) =  0\rangle \quad \text{rrz}(m,  1\rangle) = \alpha(-\frac{1}{2^m})  1\rangle$
790	$\llbracket \text{SR } m x \rrbracket \varphi$	$= \varphi[\forall i \leq m. (x, i) \mapsto \uparrow  \Phi(r_i + \frac{1}{2^{m-i+1}})\rangle]$	when $\downarrow \varphi(x, i) =  \Phi(r_i)\rangle$
791	$\llbracket \text{SR}^{-1} m x \rrbracket \varphi$	$= \varphi[\forall i \leq m. (x, i) \mapsto \uparrow  \Phi(r_i - \frac{1}{2^{m-i+1}})\rangle]$	when $\downarrow \varphi(x, i) =  \Phi(r_i)\rangle$
792	$\llbracket \text{QFT } n x \rrbracket \varphi$	$= \varphi[x \mapsto \uparrow \text{qt}(\Sigma(x), \downarrow \varphi(x), n)]$	where $\text{qt}(i,  y\rangle, n) = \bigotimes_{k=0}^{i-1} ( \Phi(\frac{y}{2^{n-k}})\rangle)$
793	$\llbracket \text{QFT}^{-1} n x \rrbracket \varphi$	$= \varphi[x \mapsto \uparrow \text{qt}^{-1}(\Sigma(x), \downarrow \varphi(x), n)]$	
794	$\llbracket \text{Lshift } x \rrbracket \varphi$	$= \varphi[x \mapsto \text{pm}_l(\varphi(x))]$	where $\text{pm}_l(q_0 \otimes q_1 \otimes \dots \otimes q_{n-1}) = q_{n-1} \otimes q_0 \otimes q_1 \otimes \dots$
795	$\llbracket \text{Rshift } x \rrbracket \varphi$	$= \varphi[x \mapsto \text{pm}_r(\varphi(x))]$	where $\text{pm}_r(q_0 \otimes q_1 \otimes \dots \otimes q_{n-1}) = q_1 \otimes \dots \otimes q_{n-1} \otimes q_0$
796	$\llbracket \text{Rev } x \rrbracket \varphi$	$= \varphi[x \mapsto \text{pm}_a(\varphi(x))]$	where $\text{pm}_a(q_0 \otimes \dots \otimes q_{n-1}) = q_{n-1} \otimes \dots \otimes q_0$
797	$\llbracket \iota_1; \iota_2 \rrbracket \varphi$	$= \llbracket \iota_2 \rrbracket (\llbracket \iota_1 \rrbracket \varphi)$	
800			
801		$\downarrow \alpha(b)\bar{q} = \bar{q} \quad \downarrow (q_1 \otimes \dots \otimes q_n) = \downarrow q_1 \otimes \dots \otimes \downarrow q_n$	
802		$\varphi[x, i \mapsto \uparrow \bar{q}] = \varphi[x, i \mapsto \alpha(b)\bar{q}]$	where $\varphi(x, i) = \alpha(b)\bar{q}_i$
803		$\varphi[x, i \mapsto \uparrow \alpha(b_1)\bar{q}] = \varphi[x, i \mapsto \alpha(b_1 + b_2)\bar{q}]$	where $\varphi(x, i) = \alpha(b_2)\bar{q}_i$
804		$\varphi[x \mapsto q_x] = \varphi[\forall i < \Sigma(x). (x, i) \mapsto q_{(x,i)}]$	
805		$\varphi[x \mapsto \uparrow q_x] = \varphi[\forall i < \Sigma(x). (x, i) \mapsto \uparrow q_{(x,i)}]$	

Fig. 17.  $\mathbb{Q}$ QASM semantics

By Theorem 3.4, when we compile 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  $\mathbb{Q}$ QASM. SR (or  $\text{SR}^{-1}$ ) applies an  $m+1$  series of RZ (or  $\text{RZ}^{-1}$ ) 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  $|b_1\rangle \otimes \dots \otimes |b_i\rangle$  (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  $n$ ).  $\text{qt}^{-1}$  is the inverse function of qt. Note that the input state to  $\text{qt}^{-1}$  is guaranteed to have the form  $\bigotimes_{k=0}^{i-1} (|\Phi(\frac{y}{2^{n-k}})\rangle)$  because it has type  $\text{Phi } n$ .  $\text{pm}_l$ ,  $\text{pm}_r$ , and  $\text{pm}_a$  are the semantics for `Lshift`, `Rshift`, and `Rev`, respectively.

### 3.3 $\mathbb{Q}$ QASM Metatheory

**Soundness.** We prove that well-typed  $\mathbb{Q}$ 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  $\mathbb{Q}$ QASM state.

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

- For every  $x \in \Omega$  such that  $\Omega(x) = \text{Nor}$ , for every  $k < \Sigma(x)$ ,  $\varphi(x, k)$  has the form  $\alpha(r) |b\rangle$ .
- For every  $x \in \Omega$  such that  $\Omega(x) = \text{Phi } n$  and  $n \leq \Sigma(x)$ , there exists a value  $v$  such that for every  $k < \Sigma(x)$ ,  $\varphi(x, k)$  has the form  $\alpha(r) |\Phi(\frac{v}{2^{n-k}})\rangle$ .<sup>5</sup>

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

<sup>5</sup>Note that  $\Phi(x) = \Phi(x + n)$ , where the integer  $n$  refers to phase  $2\pi n$ ; so multiple choices of  $v$  are possible.

$$\begin{array}{c}
\text{X } (x, n) \xrightarrow{\text{inv}} \text{X } (x, n) \quad \text{SR } m \ x \xrightarrow{\text{inv}} \text{SR}^{-1} \ m \ x \quad \text{QFT } n \ x \xrightarrow{\text{inv}} \text{QFT}^{-1} \ n \ x \quad \text{Lshift } x \xrightarrow{\text{inv}} \text{Rshift } x \\
\\
\frac{\iota \xrightarrow{\text{inv}} \iota'}{\text{CU } (x, n) \ \iota \xrightarrow{\text{inv}} \text{CU } (x, n) \ \iota'} \quad \frac{\iota_1 \xrightarrow{\text{inv}} \iota'_1 \quad \iota_2 \xrightarrow{\text{inv}} \iota'_2}{\iota_1 ; \iota_2 \xrightarrow{\text{inv}} \iota'_2 ; \iota'_1}
\end{array}$$

Fig. 18. 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  $\varphi'$  such that  $\llbracket \iota \rrbracket \varphi = \varphi'$  and  $\Sigma; \Omega' \vdash \varphi'$ .

**Algebra.** Mathematically, the set of well-formed  $d$ -qubit QASM states for a given  $\Omega$  can be interpreted as a subset  $\mathcal{S}^d$  of a  $2^d$ -dimensional Hilbert space  $\mathcal{H}^d$ ,<sup>6</sup> and the semantics function  $\llbracket \cdot \rrbracket$  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.

[Liyl: good?] Given a qubit size map  $\Sigma$  and type environment  $\Omega$ , the set of QASM programs that are well-typed with respect to  $\Sigma$  and  $\Omega$  (i.e.,  $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ ) form an algebraic structure  $(\{\iota\}, \Sigma, \Omega, \mathcal{S}^d)$ , where  $\{\iota\}$  defines the set of valid program syntax, such that there exists  $\Omega', \Sigma; \Omega \vdash \iota \triangleright \Omega'$  for all  $\iota$  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  $\llbracket \iota \rrbracket \varphi = \varphi'$ ,  $\Sigma; \Omega' \vdash \varphi'$ , 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 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  $i \in [0, 2^d)$ ,  $\llbracket \iota \rrbracket |y_i\rangle = |y'_i\rangle$ , then  $\llbracket \iota \rrbracket (\sum_{i=0}^{2^d-1} |y_i\rangle) = \sum_{i=0}^{2^d-1} |y'_i\rangle$ .

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.<sup>7</sup>

QASM programs are easily invertible, as shown by the rules in Figure 18. 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 18). This allows us to reuse

<sup>6</sup>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 subset, not a subspace 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.

<sup>7</sup>Note that a superposition over classical states can describe any quantum state, including entangled states.

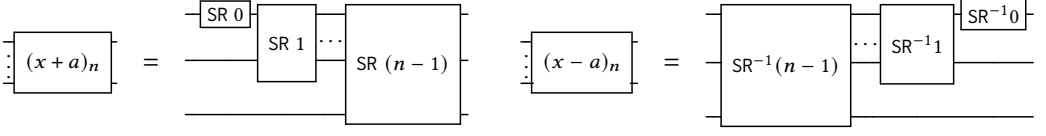


Fig. 19. Addition/subtraction circuits are inverses

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  $\iota$ , such that  $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ , its inverse  $\iota'$ , where  $\iota \xrightarrow{\text{inv}} \iota'$ , is also well-typed and we have  $\Sigma; \Omega' \vdash \iota' \triangleright \Omega$ . Moreover,  $\llbracket \iota; \iota' \rrbracket \varphi = \varphi$ .