

Quantum Natural Proof

ANONYMOUS AUTHOR(S)

Q-Dafny.

1 INTRODUCTION

Quantum computers offer unique capabilities that can be used to program substantially faster algorithms compared to those written for classical computers. For example, Grover's search algorithm [Grover 1996, 1997] can query unstructured data in sub-linear time (compared to linear time on a classical computer), and Shor's algorithm [Shor 1994] can factorize a number in polynomial time (compared to the sub-exponential time for the best known classical algorithm).

It is well known that quantum computers provide quantum supremacy. Most quantum algorithms are not classically simulatable because of the property. Unfortunately, the current quantum computers are limited and noisy, so it is hard to test the algorithms in a quantum computer as well as a classical computer. However, an algorithm not being efficiently simulatable in a classical computer does not mean that it is not efficiently provable in the computer. An analogy is that for a lottery system, we can easily prove that someone can win a lottery in a certain probability, but it is extremely hard to find out who will win the lottery. For example, simulating Shor's algorithm [Burgholzer et al. 2021] is hard, but its property has been proved on paper and in Coq [Peng et al. 2022]. It is highly likely that we can discover an efficient and automatic proof framework to extract and certify the properties for quantum algorithms.

Unfortunately, previous quantum proof frameworks [Chareton et al. 2020; Hietala et al. 2021b; Le et al. 2022; Ying 2012] permit proofs with many human-interactions. It is unlikely that algorithms can be proved automatically in these frameworks. For example, the Shor's algorithm proof in Quantum Separation Logic [Le et al. 2022] was proved by assuming that the property of the oracle loop was given and the last part was omitted. The complete Shor's algorithm proof in VOQC [Hietala et al. 2021b] was done by intelligent researchers who spent two years. The reason is that these systems specify quantum programs with quantum states, such as density matrices and high denominational Hilbert spaces. These structures cannot be effectively captured in a classical computer. Although there is a mechanism [Yu and Palsberg 2021] to reduce the complexity of representing these quantum states, the key is that they lack the establishment of "computer understandable" axioms and theories to permit proof automation in verifying quantum algorithms. More sadly, the establishment is itself a hard problem.

On the other hand, there are many classical data state structures that have already had good proof automation theorems, such as first order array, linked list, and graph theories. If we can connect quantum states with these existing classical theories, many quantum algorithms can be reasoned by these existing classical proof theories. We propose Quantum Natural Proof (QNP), a system to map quantum algorithms to classical programs that run on the classical Hoare Logic style proof system, and efficiently and automatically certify quantum program properties. The key observation is that although quantum algorithms running on quantum computers have quantum supremacy, the structures of many algorithms are simple. In addition, the states appearing in these algorithms can be mapped to classical array structures.

Figure 1 provides the QNP development plan. In the first stage, we develop the Q-Dafny proof system including a quantum language (QWhile) that contains operations for writing oracles (QQASM/OQIMP), a type system to record the qubit state forms for easing property proofs, as well as a proof system built on top of the classical Hoare Logic array theory. In addition, Q-Dafny is compiled to Dafny, such that a quantum program with its user-defined properties is compiled to


```

99  method Shor ( a : int, N : int, n : int, x : Q[n], y : Q[n] )
100    requires (n > 0)
101    requires (1 < a < N)
102    requires (N < 2^(n-1))
103    requires (gcd(a, N) == 1)
104    requires ( type(x) = Tensor n (Nor 0))
105    requires ( type(y) = Tensor n (Nor 0))
106    ensures (a*xv mod N == yv)
107    ensures (gcd(q, t) == 1)
108    ensures (is_even(t))
109    ensures (a^(t/2) + 1 divides N || a^(t/2) - 1 divides N)
110    //ensures (r.pos > (4 * e^(-2) / PI ^ 2) / (Nat.log2 N)^4)
111  {
112    x *= H ;
113    y *= cl(y+1); //cl might be omitted.
114    for (int i = 0; i < n; x[i]; i++)
115      invariant (0 ≤ i ≤ n)
116      invariant (saturation(x[0..i]))
117      invariant (type(x[0..i],y) = Tensor n (ch (2^i) {k | j baseof x[0..i] && k = (j,a^j mod N)}))
118      //psum(k=b,M,p(k),b(k)) = sum_{k=b}^M p(k)*b(k)
119      invariant ((x[0..i],y) == psum(k=0,2^i,1,(k,a^k mod N)))
120    {
121      y *= cl(a^(2^i) * y mod N);
122    }
123
124    M z := measure(y); //∀ x in x[0..n), valid(x) ==> z.base = a^(x.base) mod N
125    x *= RQFT;
126    M r := measure(x); //r is r.pos and r.base
127    int t := conFrac(r); //continued fraction of r.base and maintain r.pos (probability)
128  }
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147

```

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

To overcome these obstacles, we identify three different quantum state representations (Figure 11) in Q-Dafny, which are associated with different axioms for proof automation. Then, the Q-Dafny session type system tracks the transformation of different state representations for different sessions, collections of qubit array segments that are possibly entangled. Conceptually, the different state representations guarantee the Q-Dafny proof system can be expressed in terms of the Hoare Logic array theory. In compiling Q-Dafny to Dafny, for proving a local property for a session, we generate and identify different axioms for different state representations, and we also generate axioms that handle the transformation between state representations.

Enabling Complicated Oracle and Reflection Operations. The first stage session type system tracks only qubit bases. As a consequence, the set of quantum algorithms that are efficiently reasonable has the scheme $QFT; U; QFT^{-1}$, where QFT and QFT^{-1} are Hadamard or QFT gates that switch qubits array state forms between normal and superposition, U is an arbitrary combination of QWhile operations that use Hadamard or QFT gates in a manageable manner. For rich expressiveness, we permit two kinds of managed operations that might involve Hadamard or QFT gates: oracle assignments written in OQASM/OQIMP and quantum reflection operations (amplify and diffuse in Figure 3). The concept of "management" refers to the fact that there is a type system in OQASM/OQIMP to ensure the use of Hadamard or QFT gates do not cause entanglement, as well as the Hadamard or QFT gate usage in quantum reflections has a fixed implementation, such that its semantics is well-known and its property reasoning is captured by Q-Dafny.

148	Nat. Num	m, n	$\in \mathbb{N}$
149	Real	r	$\in \mathbb{R}$
150	Variable	x, y	
151	C-Mode Value	n	
152	M-Mode Value	(r, n)	
153	QASM Expr	μ	
154	Predicates	T	$::= x : \tau \mid x : \{\zeta : \tau\} \dots$
155	SMode	g	$::= C \mid Q$
156	Mode	q	$::= g \mid M$
157	Factor	l	$::= x \mid x[a]$
158	Arith Expr	p, a	$::= l \mid a + a \mid a * a \mid \dots$
159	Bool Expr	b	$::= x[a] \mid (a = a) @ x[a] \mid (a < a) @ x[a] \mid \neg b \dots$
160	Gate Expr	op	$::= H \mid QFT^{[-1]}$
161	C/M Moded Expr	e	$::= a \mid \text{measure}(y)$
162	Statement	s	$::= \{ \} \mid \text{let } x = e \text{ in } s \mid l \leftarrow op \mid \bar{l} \leftarrow a(\mu)$
163			$\mid s ; s \mid \text{if } (b) \{s\} \mid \text{for } (\text{int } i = a_1 ; i < a_2 ; b(i) ; f(i)) T(i) \{s\}$
164			$\mid \text{amplify}\{x \leftarrow a\} \mid \text{diffuse}(l)$

Fig. 3. Core QWhile Syntax, QASM Syntax is in Figure 12

Almost all near term quantum algorithms are definable in terms of the above scheme, such as GHZ, Grover's, Shor's, and Childs' Boolean equation algorithms. To prove properties about advanced quantum algorithms, the session type system needs to track qubit phases, which will complicate its design, and it is the major task of stage 3 in the QNP development.

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 Syntax

QWhile is a high-level language for describing quantum programs, which permits quantum control and for-loop statements. Figure 3 introduces the QWhile syntax.

A QWhile program consists of a sequence of C-like statements s . Values and variables (ranged by x and y) in a statement are classified as three different categories, named *modes*: C , M , and Q . C and M mode variables represent classical values. C mode variables are computed through classical computation, while M mode variables are originated from a quantum variable measurement. We represent C -mode values as natural numbers, while M -mode values are pairs of reals and natural numbers. The reals represent the conceptual occurrence probability of the result measurement, and the natural numbers are the measurement results. Any further arithmetic operations on M -mode values are applied on the measurement results, such as $(r, n_1) + n_2 = (r, n_1 + n_2)$.

Q mode variables refer to quantum qubit arrays. Each Q mode variable x represents a qubit array, whose size is defined by a global map Σ . We use $x[a]$ to represent the a -th position qubit in x . It is worth noting that the variable x in $x[a]$ must be Q mode. We named a variable x or an array index $x[a]$ as a factor. In a QWhile program, C and M mode variables act like stack variables and

Session	λ	$::=$	$\overline{(x, n, m)}$
Bit	c	$::=$	$0 \mid 1$
BitString	\bar{c}	$::=$	$\mathbb{N} \rightarrow c$
BitString Indexed Set	β	$::=$	$\{\bar{c}\} \mid \infty$
Phase Type	w	$::=$	$\bigcirc \mid \infty$
Type Element	t	$::=$	$\text{Nor } \bar{c} \mid \text{Had } w \mid \text{CH } n \beta$
Type	τ	$::=$	$\bigotimes_n t$
Phase	$\alpha(n)$	$::=$	$e^{2\pi i \frac{1}{n}}$
Amplitude	θ	$::=$	r
Phi State	$\Phi(n)$	$::=$	$\frac{1}{\sqrt{2}}(0\rangle + \alpha(n) 1\rangle)$
Quantum State	ρ	$::=$	$\alpha \bar{c}\rangle \mid \bigotimes_{k=0}^m \Phi(n_k)\rangle \mid \sum_{k=0}^m \theta_k \bar{c}_k\rangle$

Fig. 4. QWhile Sessions and Types and Quantum States

$$\begin{array}{c}
\frac{\Omega(x) \neq Q}{\Omega \vdash x : (\Omega(x), \perp)} \quad \frac{\Omega(x) = Q}{\Omega \vdash x : (Q, [(x, 0, \Sigma(x))])} \quad \frac{\Omega(x) = Q}{\Omega \vdash x[n] : (Q, [(x, n, n+1)])} \\
\\
\frac{\Omega \vdash a_1 : (q_1, \zeta_1) \quad \Omega \vdash a_2 : (q_2, \zeta_2)}{\Omega \vdash a_1 + a_2 : (q_1 \sqcup q_2, \zeta_1 \uplus \zeta_2)} \quad \frac{\Omega \vdash a_1 : (q_1, \zeta_1) \quad \Omega \vdash a_2 : (q_2, \zeta_2)}{\Omega \vdash a_1 * a_2 : (q_1 \sqcup q_2, \zeta_1 \uplus \zeta_2)} \\
\\
\frac{\Omega \vdash a_1 : (q_1, \zeta_1) \quad \Omega \vdash a_2 : (q_2, \zeta_2)}{\Omega \vdash (a_1 = a_2)@x[n] : (q_1 \sqcup q_2, \zeta_1 \uplus \zeta_2 \uplus \zeta_3)} \quad \frac{\Omega \vdash a_1 : (q_1, \zeta_1) \quad \Omega \vdash a_2 : (q_2, \zeta_2)}{\Omega \vdash (a_1 < a_2)@x[n] : (q_1 \sqcup q_2, \zeta_1 \uplus \zeta_2 \uplus \zeta_3)} \quad \frac{\Omega \vdash b : (q, \zeta)}{\Omega \vdash \neg b : (q, \zeta)} \\
\\
\frac{}{\Omega \vdash H : (Q, \perp)} \quad \frac{}{\Omega \vdash X : (Q, \perp)} \quad \frac{}{\Omega \vdash \text{RZ}^{[-1]} n : (Q, \perp)} \quad \frac{\Omega \vdash e : (q, \zeta_2 \uplus \zeta_1)}{\Omega \vdash e : (q, \zeta_1 \uplus \zeta_2)} \\
\\
Q \sqcup q = Q \quad q \sqcup Q = Q \quad C \sqcup C = C \quad M \sqcup C = M \quad C \sqcup M = M \quad C \leq M \leq Q \\
\perp \uplus I = I \quad I \uplus \perp = I \quad [(x, v_1, v_2)] \uplus [(y, v_3, v_4)] = [(x, v_1, v_2), (y, v_3, v_4)] \\
[v_2, v_2] \cap [v_3, v_4] \neq \emptyset \Rightarrow [(x, v_1, v_2)] \uplus [(x, v_3, v_4)] = [(x, \min(v_1, v_3), \max(v_2, v_4))]
\end{array}$$

Fig. 5. Arith, Bool, Gate Mode Checking

they must be bounded by variables introduced by `let` statements; while Q mode factors represent arrays in a "quantum heap" and are bounded by Σ .

The statements s in the first row in Figure 3 are $\{\}$ (SKIP) and assignment operations. Classic assignment (`let x = e in s`) evaluates e and assigns the value to C or M mode variable x that is used in s . Expressions e can be an arithmetic or a measurement operation. `let x = measure(y) in ...` assigns the measurement result of qubit array y to x . $l \leftarrow op$ and $\bar{l} \leftarrow a(\mu)$ are quantum assignments. The former applies a simple quantum gate (H or QFT) to a single qubit ($x[a]$) or a qubit array (x)¹. $\bar{l} \leftarrow a(\mu)$ is a generalized quantum assignment that implements quantum oracle circuits a and applies an \mathbb{Q} QASM operation on a list of qubit array \bar{l} . We assume all arithmetic (a) and Boolean (b) expressions are reversible. μ is the circuit implementation of the expression a in \mathbb{Q} QASM, whose syntax is given in Section 3. One can utilize \mathbb{Q} QASM expressions (μ) to implement singleton X and $\text{RZ}^{[-1]} n$ gates, thus, the QWhile syntax is universal with respect to quantum gates. In addition, \mathbb{Q} QIMP is a reversible arithmetic C-like language built on top of \mathbb{Q} QASM. The reversible expression

¹QFT gate must apply to a qubit array variable x

a is based on \mathbb{Q} IMP operations. For simplicity, we write $\bar{l} \leftarrow a$ in examples here to mean that it applies a \mathbb{Q} IMP circuit that computes a to \bar{l} .

The second row of statements in Figure 3 are control-flow operations. $s_1 ; s_2$ is a sequential operation. $\text{if } (b) \{s\}$ is a conditional and b might contain quantum factors. Every quantum factor l appearing in b must not appear in s . In QWhile , we define a `well_formed` predicate to check such property. $\text{for } (\text{int } i = a_1 ; i < a_2 ; b(i) ; f(i)) T(i) \{s\}$ is a possibly quantum for-loop depending on if the Boolean guard $b(i)$ contains quantum factors (also needing to be `well_formed`). A classical variable i is introduced and it is initialized as the lower bound a_1 , increments in each loop step by the monotonic increment function $f(i)$, and ends at the upper bound a_2 . $T(i)$ is the type predicate (syntax in fig. 3) where for i -th loop step, $T(i)$ is true in the beginning and $T(f(i))$ is maintained after the execution.

The last row contains quantum reflection operations, which are used to adjust the occurrence probability of some quantum states in a quantum qubit array. For example, in Grover's search algorithm [Grover 1996], the Grover's diffusion operation is a quantum reflection that increases the occurrence probability of a target quantum state in a qubit array being in superposition. We identify two kinds of quantum reflections that previous works tend to combine together. The first one is an amplifier (`amplify`{ $x \leftarrow a$ }) that amplifies the occurrence probability of the target state, which is represented by a classical value a , in a quantum qubit array x . The second one is a diffusion operator (`diffuse`(x)) that distributes a qubit array x to a superposition of all possible bases in x with possibly different amplitudes². For example, apply the distributor on a two-qubit array x having state $|00\rangle$ results in a superposition of $-\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$. In general, if an n -qubit array x has the state $\sum_{j=0} \alpha_j |x_j\rangle$, the result of applying a distributor is $\frac{1}{2^n} (2 \sum_{i=0}^{2^n} (\sum_{j=0} \alpha_j) |i\rangle - \sum_{j=0} \alpha_j |x_j\rangle)$.

2.2 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 11. 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 ranges, 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 $\otimes_n \text{Nor } \bar{c}$ means that every qubit is in normal basis (either $|0\rangle$ or $|1\rangle$), and \bar{c} describes basis states for the qubits. The type corresponds to a single qubit basis state $\alpha(n) |\bar{c}\rangle$, where the global phase $\alpha(n)$ has the form $e^{2\pi i \frac{1}{n}}$ and \bar{c} 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 turn to local phases when we turn a session from type Nor to the entanglement type.

$\otimes_n \text{Had } w$ 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 ∞ means the phase amplitude for each qubit is unknown. If a session has such type, it then has the value form $\otimes_{k=0}^m |\Phi(n_k)\rangle$, 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 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 QFT^{-1} operation, the state has type $\otimes_n \text{CH } m \infty$, and the

²The possible bases do not depend on x , and it is only related to the qubit size of x ; i.e., if x is a two qubit array, then the operation reflects the superposition of all possible two qubit states as: $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$.

only quantum operation to apply is a measurement. If a session has type $\otimes_n \text{CH } m \beta$ 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 $\otimes_n \text{CH } 2 \{\bar{0}^n, \bar{1}^n\}$, where \bar{c}^n is a n -bit string having bit c .

The type $\otimes_n \text{CH } m \beta$ corresponds to the value form $\sum_{k=0}^m \theta_k |\bar{c}_k\rangle$. θ_k is an amplitude real number, and \bar{c}_k is the basis. Basically, $\sum_{k=0}^m \theta_k |\bar{c}_k\rangle$ represents a size m array of basis states that are pairs of θ_k and \bar{c}_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: $g, \Omega, \mathcal{T} \vdash s : \zeta \triangleright \tau$, where g is the context mode, mode environment Ω maps variables to modes, 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 8 has several tasks. First, it enforces context mode restrictions. Context mode g is either C or Q . Q represents the current expression lives inside a quantum conditional or loop, while C refers to other cases. In a Q context, one cannot perform M -mode operations, i.e., no measurement is allowed. There are other well-formedness enforcement. For example, the session of the Boolean guard b in a conditional/loop is disjoint with the session in the conditional/loop body, i.e., qubits used in a Boolean guard cannot appear in its conditional/loop body.

Second, the type system enforces mode checking for variables and expressions in Figure 14. In QWhile, C -mode variables are evaluated to values during type checking. In a `let` statement (Figure 8), C -mode expression is evaluated to a value n , and the variable x is replaced by n in s . The expression mode checking (Figure 14) has the judgement: $\Omega \vdash (a \mid b \mid op) : (q, \zeta)$.³ It takes a mode environment Ω , and an expression (a , b , or op), and judges if the expression has the mode q with the possible session ζ if the mode is Q and C/M mode variable has no sessions. 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 14 and the phase rotation value in $\text{RZ}^{[-1]} n$. It is worth noting that the session computation (\oplus) is also commutative as the last rule in Figure 14.

Third, by generating the session of an expression, the QWhile type system assigns a type τ for the session indicating the state format of session, which will be discussed shortly below. Recall that a session is a list of quantum qubit ranges. In quantum computation, qubits can entangled with each other. We utilize type τ (Figure 11) to state entanglement properties appearing in a group of qubits. It is worth noting that the entanglement property refers to *eventual entanglement*, i.e. a group of qubits that are eventually entangled. Entanglement classification is tough and might not be necessary. In most near term quantum algorithms, such as Shor's algorithm [Shor 1994] and Childs' Boolean equation algorithm (BEA) [Childs et al. 2007], programmers care about if qubits eventually become entangled during a quantum loop execution. This is why the normal basis type ($\otimes_n \text{Nor } \bar{c}$) can also be a subtype of a entanglement type ($\otimes_n \text{CH } 1 \{\bar{c}\}$) in our system (Figure 7).

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,

³For simplicity, we wrote (q, \perp) as q .

	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. 6. Control Gate Entanglement Situation

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

Fig. 7. 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 $\text{if } (x[i]) \{e(y)\}$ is described in Figure 6.

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.

The situation when y being type Had is not commonly appeared in quantum algorithms. The only known example is the phase kickback pattern, which applies an \mathcal{X} kind operation on y of type Had. If y is in uniform superposition, applying any \mathcal{X} kind operation does not cause entanglement. If y is not uniform superposition, the only \mathcal{X} kind operation not causing entanglement is an \mathcal{X} gate, which is exactly the phase kickback pattern. The application of a \mathcal{R} operation on y being type Had does create entanglement, but there is no known algorithms utilizing such entanglement, because the entanglement relies on phase amplitudes, which are expensive to maintain due to noisy qubits and gates. The QWhile type system forbids such operations.

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 2.3.

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

If $x[i]$ and y are in different entanglement sessions, the situation is similar to when $x[i]$ having Had and y having CH type. It merges the two sessions together through the saturation $x[i]$. For example, in BEA, The quantum Boolean guard computes the following operation $(z < i)@x[i]$ on a Had type variable z (state: $\sum_{k=0}^{2^n} |k\rangle$) and a Nor type factor $x[i]$ (state: $|0\rangle$). The result is an entanglement $\sum_{k=0}^{2^n} |k, k < i\rangle$, where the $x[i]$ position stores the Boolean bit result $k < i$.⁴ The

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

393					
394	TSKIP	TEXP	TA-NOR		
395	$\frac{}{g, \Omega, \mathcal{T} \vdash \{\} : \emptyset \triangleright \perp}$	$\frac{g, \Omega[x \mapsto C], \mathcal{T} \vdash s[v/x] : \zeta \triangleright \tau}{g, \Omega, \mathcal{T} \vdash \text{let } x = v \text{ in } s : \zeta \triangleright \tau}$	$\frac{\Omega \vdash \bar{l} : (Q, \zeta) \quad \llbracket a \rrbracket \bar{c} = \alpha \bar{c}' }{g, \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{Nor } \bar{c}] \vdash \bar{l} \leftarrow a : \zeta \triangleright \bigotimes_n \text{Nor } \bar{c}'}$		
396					
397	TA-HAD		TMEA		
398	$\frac{\Omega \vdash \bar{l} : (Q, \zeta) \quad \zeta \subseteq \zeta' \quad (\forall \bar{c}. \llbracket a \rrbracket \bar{c} = \bar{c}')}{g, \Omega, \mathcal{T} [\zeta' \mapsto \bigotimes_n \text{Had } \bigcirc] \vdash \bar{l} \leftarrow a : \zeta' \triangleright \bigotimes_n \text{Had } \bigcirc}$		$\frac{\Omega \vdash y : (Q, \zeta) \quad \zeta \cup \zeta' \quad C, \Omega[x \mapsto M], \mathcal{T} \vdash s : \zeta' \triangleright \tau}{C, \Omega, \mathcal{T} \vdash \text{let } x = \text{measure}(y) \text{ in } s : \zeta' \triangleright \tau}$		
399					
400					
401	TA-CH				
402		$\Omega \vdash \bar{l} : (Q, \zeta) \quad \zeta' = \zeta_l \uplus \zeta \uplus \zeta_r$			
403	$\frac{}{g, \Omega, \mathcal{T} [\zeta' \mapsto \bigotimes_n \text{CH } k \beta] \vdash \bar{l} \leftarrow a : \zeta' \triangleright \bigotimes_n \text{CH } k \{ \bar{c}_l \cdot \bar{c} \cdot \bar{c}_r \mid \bar{c}_l \cdot \bar{c} \cdot \bar{c}_r \in \beta(\downarrow \zeta_l) \cdot \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r) \wedge \llbracket a \rrbracket \bar{c} = \alpha \bar{c}' \}}$				
404					
405	TSEQ-1		TSEQ-2		
406	$\frac{g, \Omega, \mathcal{T} \vdash s_1 : \zeta \triangleright \bigotimes_n \text{Nor } \bar{c} \quad \zeta \cup \zeta'}{g, \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{Nor } \bar{c}] \vdash s_2 : \zeta' \triangleright \bigotimes_{n'} \text{Nor } \bar{c}'}$		$\frac{g, \Omega, \mathcal{T} \vdash s_1 : \zeta \triangleright \bigotimes_n \text{Nor } \bar{c} \quad \zeta \cup \zeta'}{g, \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{Nor } \bar{c}] \vdash s_2 : \zeta' \triangleright \bigotimes_{n'} \text{Nor } \bar{c}'}$		
407					
408	$\frac{}{g, \Omega, \mathcal{T} \vdash s_1 ; s_2 : \zeta \uplus \zeta' \triangleright \bigotimes_{n+n'} \text{Nor } \bar{c} \cdot \bar{c}'}$		$\frac{}{g, \Omega, \mathcal{T} \vdash s_1 ; s_2 : \zeta' \uplus \zeta \triangleright \bigotimes_{n+n'} \text{Nor } \bar{c}' \cdot \bar{c}}$		
409					
410					
411	TIF				
412	$\frac{\Omega(x) = Q \quad \mathcal{T}([(x, v, v+1)]) = \bigotimes_1 \text{Had } w \quad Q, \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{CH } k \beta] \vdash s : \zeta \triangleright \bigotimes_n \text{CH } k \beta'}{g, \Omega, \mathcal{T} [\zeta \mapsto \bigotimes_n \text{CH } k \beta] \vdash \text{if } (x[v]) \{s\} : [(x, v, v+1)] \uplus \zeta \triangleright \bigotimes_{n+1} \text{CH } 2k \{ \bar{c} \in (0 \cdot \beta 1 \cdot \beta') \}}$				
413					
414					
415	TLOOP				
416	$\frac{\Omega[i \mapsto C] \vdash f(i) : C \quad \mathcal{T}([x, v_1, v_2]) = \bigotimes_{v_2-v_1} \text{Had } w \quad \{[x, v_1, i] \uplus \zeta \mapsto \bigotimes_{i-v_1} \text{CH } a(i) \beta(i)\} \models T(i)}{g, \Omega[i \mapsto C], \mathcal{T}([x, v_1, i] \uplus \zeta \mapsto \bigotimes_{i-v_1} \text{CH } a(i) \beta(i)) \vdash \text{if } (x[i]) \{s\} : [x, v_1, f(i)] \uplus \zeta \triangleright \bigotimes_{f(i)-v_1} \text{CH } a(f(i)) \beta(f(i))}$				
417					
418	$\frac{}{g, \Omega, \mathcal{T} \vdash \text{for } (\text{int } i = v_1 ; i < v_2 ; x[i] ; f(i)) T(i) \{s\} : [[(x, v_1, v_2)] \uplus \zeta \triangleright \bigotimes_{v_2-v_1} \text{CH } (a(v_2 - v_1)) (\beta(v_2 - v_1))}$				
419					
420					
421	TDIS				
422	$\frac{\Omega \vdash x : (Q, \zeta) \quad \zeta' = \zeta_l \uplus \zeta \uplus \zeta_r \quad \text{size}(\zeta) = n' \quad m' = \text{size}(\{ \bar{c}_l \cdot \mathcal{P}(n') \cdot \bar{c}_r \mid \bar{c}_l \cdot \bar{c} \cdot \bar{c}_r \in \beta(\downarrow \zeta_l) \cdot \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r) \})}{g, \Omega, \mathcal{T} [\zeta' \mapsto \bigotimes_n \text{CH } m \beta] \vdash \text{diffuse}(x) : \zeta' \triangleright \bigotimes_n \text{CH } m' \{ \bar{c}_l \cdot \mathcal{P}(n') \cdot \bar{c}_r \mid \bar{c}_l \cdot \bar{c} \cdot \bar{c}_r \in \beta(\downarrow \zeta_l) \cdot \beta(\downarrow \zeta) \cdot \beta(\downarrow \zeta_r) \}}$				
423					
424					
425	$\bar{c} \cdot \bar{c}'$: list concatenation. $\zeta \cup \zeta'$: The two sessions are disjoint.				
426	$\beta(\downarrow \zeta)$: Get the position range of ζ in the session and form a new set				
427	of bitstrings containing only the positions in the range.				
428	$\llbracket a \rrbracket \bar{c} $: \mathbb{Q}_{ASM} semantics of interpreting reversible expression a in Figure 16.				
429					

Fig. 8. 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 8. As we have mentioned above, the type system tracks possible eventual entanglement for a group of qubits, which we named a session. As we mentioned above, the type judgment is given $g, \Omega, \mathcal{T} \vdash s : \zeta \triangleright \tau$.

Rule TSKIP judges that program $\{\}$ has no sessions and its empty is \perp . TEXP is the type rule for C -mode expressions. The expression a is evaluated and variable x is substituted in s . TMEA is a similar rule as TEXP, but for M -mode variable. 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 $\mathbb{Q}QASM$ (Figure 16) and the arithmetic language $\mathbb{Q}QIMP$ is the role model of such semantic function. In TA-HAD, when \bar{l} is in uniform superposition (Had \bigcirc), 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 ζ of the whole entanglement session ζ' , such that $\zeta' = \zeta_l \cdot \zeta \cdot \zeta_r$. Here, for every element $\bar{c}_l \cdot \bar{c} \cdot \bar{c}_r \in \beta$, we find out the corresponding part \bar{c} belonging to the session ζ ($\bar{c}_l \cdot \bar{c} \cdot \bar{c}_r \triangleright \zeta_l \cdot \zeta \cdot \zeta_r$), and updates the \bar{c} 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, 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 ζ' , since it is obvious that $\zeta \subseteq \zeta'$. If s_1 and s_2 are of type CH, we only permit the case when $\zeta \subseteq \zeta'$ for simplicity. It has no technical difficulty to allow τ 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 two distinct entanglement sessions inside a conditional block. In addition, 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 $x[i]$ having type Had. The result type of such conditionals is an CH type by merging the bit $x[i]$ into the entanglement session with a double state number ($2k$). It is worth noting that by the subtyping relation in Figure 7, 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 CH, its behavior is similar to the Had case.

Rule TLOOP describes the type for a for-loop. It is a generalization of the conditional case when the Boolean guard $x[i]$ having type Had. The predicate $\{[x, v_1, i] \uplus \zeta \mapsto \bigotimes_{i=v_1} \text{CH } a(i) \beta(i)\} \models T(i)$ refers to an existence of a type $(\bigotimes_{i=v_1} \text{CH } a(i) \beta(i))$ for a session $[x, v_1, i] \uplus \zeta$ that models the type predicate invariant $T(i)$ for all i ; i.e., the predicate expression $T(i)$ expresses a type for the session $[x, v_1, i] \uplus \zeta$, where it contains part of the qubit array x from v_1 to i . If such model exists and the type judgment for $\text{if } (x[i]) \{s\}$ is provable, we can infer that the final type of the whole session is a replacement of i in the one step session type $\bigotimes_{f(i)=v_1} \text{CH } a(f(i)) \beta(f(i))$ with the range size $v_2 - v_1$, written as $\bigotimes_{v_2-v_1} \text{CH } (a(v_2 - v_1)) (\beta(v_2 - v_1))$.

Rule TDIS types a distributor. The rule finds the right part of x in the session ζ' . For every right session bitstring \bar{c} in $\bar{c}_l \cdot \bar{c} \cdot \bar{c}_r$, it generates a set of new type sequence by replacing \bar{c} with elements in the power set $\mathcal{P}(n')$, where n' is the bit length of \bar{c} . 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.

2.3 Logic Proof System

The reason of having the session type system in Figure 8 is to enable the proof system given in Figure 9. Every proof rule is a structure as $g, \Omega \vdash \{T\} \{P\} s \{T'\} \{Q\}$, where g and Ω are the type entities mentioned in Section 2.2. 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 $(\zeta \mapsto \tau) \in \mathcal{T}'$. P and Q are the pre- and post- Hoare conditions for statement s . We denote $(\mathcal{T}, \mathcal{T}') \models (T, s, T') : \zeta \triangleright \tau$ as the property described above. For simplicity, we abbreviate the rules in Figure 9 as a five tuple structure $\{T\} \{P\} s \{T'\} \{Q\}$.

$$\begin{array}{c}
\text{PA-NOR} \\
\frac{(\mathcal{T}, \mathcal{T}') \models (T, \bar{l} \leftarrow a, T') : \zeta \triangleright \bigotimes_n \text{Nor } \bar{c}}{\{T\} \{P[\llbracket a \rrbracket \zeta / \bar{c}]\} \bar{l} \leftarrow a \{T'\} \{P\}} \\
\\
\text{PA-CH} \\
\frac{(\mathcal{T}, \mathcal{T}') \models (T, \bar{l} \leftarrow a, T') : \zeta' \triangleright \bigotimes_n \text{CH } m \beta \quad \mathcal{T}(\bar{l}) = \zeta}{\{T\} \{P[\forall k < m. \llbracket a \rrbracket(\zeta[k]) / \zeta[k]]\} \bar{l} \leftarrow a \{T'\} \{P\}} \\
\\
\text{P-MEA} \\
\frac{\begin{array}{l} \mathcal{T} \models T \quad \mathcal{T}(y) = \{\zeta' : \bigotimes_n \text{CH } m \beta\} \quad \zeta = (y, 0, \Sigma(y)) \quad v < m \\ \zeta'' = \zeta' \setminus \zeta \quad n' = \text{In}(\zeta'') \quad m' = \text{size}(\beta') \quad \beta' = \{\bar{c} \mid \bar{c} \cdot \bar{c}' = \beta[k](\downarrow \zeta'') \cdot \beta[k](\downarrow \zeta) \wedge \bar{c}' = \text{bs}(\zeta[v])\} \\ A = \{\frac{1}{2^{m'}} \text{as}(\zeta''[k]) \bar{c} \mid \bar{c} \cdot \bar{c}' = \beta[k](\downarrow \zeta'') \cdot \beta[k](\downarrow \zeta) \wedge \bar{c}' = \text{bs}(\zeta[v])\} \\ T'' = T[x \mapsto M, y \mapsto \perp] [\forall z : . \mathcal{T}(y) = \{\zeta' : \bigotimes_n \text{CH } m \beta\} \Rightarrow z \mapsto \{\zeta'' : \bigotimes_{n'} \text{CH } m' \beta'\}] \\ \{T''\} \{P[(\text{as}(\zeta[v])^2, \text{bs}(\zeta[v]))/x, A/\zeta'', \perp/\zeta']\} s \{T'\} \{Q\} \end{array}}{\{T\} \{P\} \text{let } x = \text{measure}(y) \text{ in } s \{T'\} \{Q\}} \\
\\
\text{P-IF} \\
\frac{(\mathcal{T}, \mathcal{T}') \models (T, s, T') : \zeta \triangleright \bigotimes_n \text{CH } m \beta \quad \mathcal{T}(x, v, v+1) = \bigotimes_1 \text{Had } w}{\{T\} \{P[\forall k < 2m. (k \% 2) \cdot (\llbracket s \rrbracket(\zeta[k])) / ((x, v, v+1) \uplus \zeta)] \text{ if } (x[v]) \{s\} \{T'\} \{P\}\}} \\
\\
\text{P-LOOP} \\
\frac{(\mathcal{T}, \mathcal{T}') \models (T(i), \text{if } (x[i]) \{s\}, T(f(i))) : \zeta \triangleright \tau \quad \mathcal{T}(x, i, i+1) = \bigotimes_1 \text{Had } w}{\{T(i)\} \{P(i)\} \text{ if } (x[i]) \{s\} \{T(f(i))\} \{P(f(i))\}} \\
\{T(a_1)\} \{P(a_1)\} \text{ for } (\text{int } i = a_1; i < a_2; x[i]; f(i)) T(i) \{s\} \{T(a_2)\} \{P(a_2)\} \\
\\
\text{P-DIS} \\
\frac{(\mathcal{T}, \mathcal{T}') \models (T, \text{diffuse}(x), T') : \zeta \triangleright \bigotimes_{n'} \text{CH } m' \beta' \quad \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')}{\{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] \\
\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])\} \\
\text{All rules are abbreviation of the structure } g, \Omega \vdash \{T\} \{P\} s \{T'\} \{Q\}.
\end{array}$$

Fig. 9. Selected Proof System Rules

The proof system is an imitation of the classical Hoare Logic array theory. We view the three different quantum state forms in Figure 11 as arrays, 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. This proof system permits the utilization of classical proof automation infrastructure, such as Dafny, because quantum program semantics based density matrices and vectors are hard to map to an invariant state based on classical computer structures, like arrays and linked lists.

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 $\llbracket a \rrbracket \zeta / \bar{c}$ 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.

Rule P-MEA is the rule for partial/complete measurement. y might be a part of a session ζ' . After the measurement, The M -mode x is with the measurement result $(\text{as}(\zeta[v])^2, \text{bs}(\zeta[v]))$ coming

from one basis state of y , $\text{as}(\zeta[v])$ is the amplitude and $\text{bs}(\zeta[v])$ is the base. y is removed, and the session ζ' is shrunk to a new session ζ'' without the y part, and its new CH type is re-calculated by selecting those elements in β whose y session part has the bitstring same as the measurement result bitstring $\text{bs}(\zeta[v])$. For each element in the array ζ'' , it get the new base in β' and computes its amplitude by $\frac{1}{2^{m'}} \text{as}(\zeta')$, where $\text{as}(\zeta')$ is the old array amplitude value.

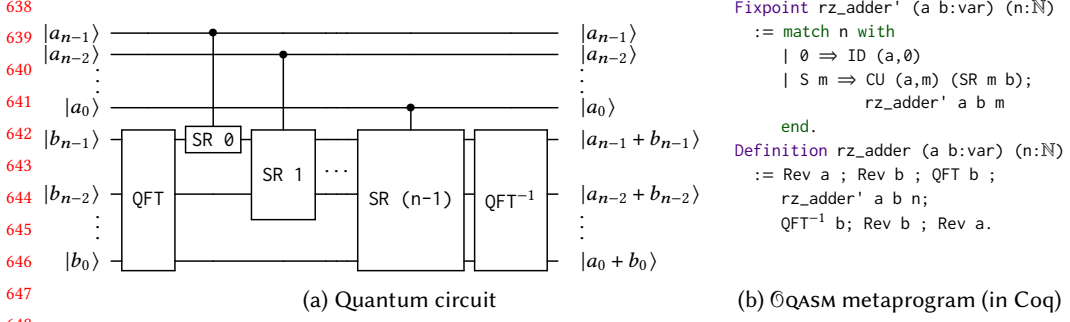
Rule P-IF deals with a quantum conditional where the Boolean guard $x[i]$ is of type \otimes_1 . Had w . For each element in array ζ , we add a bit 0 or 1 on top of the base depending on if the index lives in $[0, m)$ or $[m, 2m)$. Rule P-Loop is an initiation of the classical while rule in Hoare Logic but the loop guard can be quantum. 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 distributor $\text{diffuse}(x)$. The quantum semantics for $\text{diffuse}(x)$ 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, $\text{diffuse}(x)$ 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 $\theta_k(\overline{c_x} \cdot \overline{c_1})$, for any bitstring z in $\mathcal{P}(\Sigma(x))$, if $z \cdot \overline{c_1}$ is not a base for $\zeta[j]$ for any j , then the state is $\frac{1}{2^{n-1}} \sum_{k=0} \theta_k(z \cdot \overline{c_1})$; if the base of $\zeta[j]$ is $z \cdot \overline{c_1}$, then the state for $\zeta[j]$ is $\frac{1}{2^{n-1}} (\sum_{k=0} \theta_k) - \theta_j(z \cdot \overline{c_1})$.

REFERENCES

- Stephane Beauregard. 2003. Circuit for Shor's Algorithm Using $2n+3$ Qubits. *Quantum Info. Comput.* 3, 2 (March 2003), 175–185.
- Lukas Burgholzer, Hartwig Bauer, and Robert Wille. 2021. Hybrid Schrödinger-Feynman Simulation of Quantum Circuits With Decision Diagrams. arXiv:2105.07045 [quant-ph]
- Christophe Charetton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. 2020. Toward Certified Quantum Programming. *arXiv e-prints* (2020). arXiv:2003.05841 [cs.PL]
- 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).
- 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
- Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021a. Proving Quantum Programs Correct. In *Proceedings of the Conference on Interactive Theorem Proving (ITP)*.
- Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021b. A Verified Optimizer for Quantum Circuits. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.
- Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. 2022. A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 36 (jan 2022), 27 pages. <https://doi.org/10.1145/3498697>
- Guang Hao Low and Isaac L. Chuang. 2017. Optimal Hamiltonian Simulation by Quantum Signal Processing. *Physical Review Letters* 118, 1 (Jan 2017).
- Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. 2022. A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm. <https://doi.org/10.48550/ARXIV.2204.07112>
- Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. 2013. Natural proofs for structure, data, and separation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 231–242. <https://doi.org/10.1145/2491956.2462169>
- 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>
- Mingsheng Ying. 2012. Floyd–Hoare Logic for Quantum Programs. *ACM Trans. Program. Lang. Syst.* 33, 6, Article 19 (Jan. 2012), 49 pages. <https://doi.org/10.1145/2049706.2049708>
- Nengkun Yu and Jens Palsberg. 2021. Quantum Abstract Interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) (PLDI 2021)*. Association for Computing

Machinery, New York, NY, USA, 542–558. <https://doi.org/10.1145/3453483.3454061>

Fig. 10. Example \mathbb{Q} ASM program: QFT-based adder

651 Bit	b	$::=$	$0 \mid 1$
652 Natural number	n	\in	\mathbb{N}
653 Real	r	\in	\mathbb{R}
654 Phase	$\alpha(r)$	$::=$	$e^{2\pi i r}$
655 Basis	τ	$::=$	$\text{Nor} \mid \text{Phi } n$
656 Unphased qubit	\bar{q}	$::=$	$ b\rangle \mid \Phi(r)\rangle$
657 Qubit	q	$::=$	$\alpha(r)\bar{q}$
658 State (length d)	φ	$::=$	$q_1 \otimes q_2 \otimes \dots \otimes q_d$

Fig. 11. \mathbb{Q} ASM state syntax

3 \mathbb{Q} ASM: AN ASSEMBLY LANGUAGE FOR QUANTUM ORACLES

We designed \mathbb{Q} ASM to be able to express efficient quantum oracles that can be easily tested and, if desired, proved correct. \mathbb{Q} ASM operations leverage both the standard computational basis and an alternative basis connected by the quantum Fourier transform (QFT). \mathbb{Q} ASM's type system tracks the bases of variables in \mathbb{Q} ASM programs, forbidding operations that would introduce entanglement. \mathbb{Q} ASM states are therefore efficiently represented, so programs can be effectively tested and are simpler to verify and analyze. In addition, \mathbb{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 \mathbb{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 10. The Coq function `rz_adder` generates an \mathbb{Q} ASM program that adds two natural numbers a and b , each of length n qubits.

3.1 \mathbb{Q} ASM States

An \mathbb{Q} ASM program state is represented according to the grammar in Figure 11. 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 \mathbb{Q} ASM program on any well-formed \mathbb{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* 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 $\iota ::= \text{ID } p \mid X \ p \mid \text{RZ}^{[-1]} \ n \ p \mid \iota ; \iota$
 $\mid \text{SR}^{[-1]} \ n \ x \mid \text{QFT}^{[-1]} \ n \ x \mid \text{CU } p \ \iota$
 $\mid \text{Lshift } x \mid \text{Rshift } x \mid \text{Rev } x$

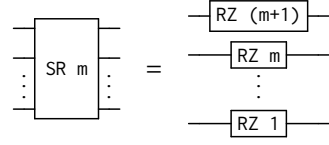


Fig. 12. \mathbb{Q} ASM syntax. For an operator OP , $\text{OP}^{[-1]}$ indicates that the operator has a built-in inverse available.

Fig. 13. SR unfolds to a series of RZ instructions

$b \in \{0, 1\}$), which is a computational basis value. A Φ 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 \mathbb{Q} ASM Syntax, Typing, and Semantics

[Liyi: add RZ gate back]

Figure 12 presents \mathbb{Q} ASM's syntax. An \mathbb{Q} ASM program consists of a sequence of instructions ι . Each instruction applies an operator to either a variable x , which represents a group of qubits, or a position p , which identifies a particular offset into a variable x .

The instructions in the first row correspond to simple single-qubit quantum gates—ID p , $X \ p$, and $\text{RZ}^{[-1]} \ n \ p$ —and instruction sequencing. The instructions in the next row apply to whole variables: $\text{QFT } n \ x$ 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 13). Operation $\text{CU } p \ \iota$ 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 10(b) uses many of these instructions; e.g., it uses QFT and 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 12, 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* Ω and a predefined *size environment* Σ , 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 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 `QIMP` in ??. The algorithm generates Σ by taking an `QIMP` program and scanning through all the variable initialization statements. Select type rules are given in Figure 14; the rules not shown (for ID, Rshift, Rev, RZ^{-1} , and SR^{-1}) are similar.

$\frac{X \quad \Omega(x) = \text{Nor} \quad n < \Sigma(x)}{\Sigma; \Omega \vdash X(x, n) \triangleright \Omega}$	$\frac{RZ \quad \Omega(x) = \text{Nor} \quad n < \Sigma(x)}{\Sigma; \Omega \vdash RZ q(x, n) \triangleright \Omega}$
$\frac{SR \quad \Omega(x) = \text{Phi } n \quad m < n}{\Sigma; \Omega \vdash SR m x \triangleright \Omega}$	$\frac{QFT \quad \Omega(x) = \text{Nor} \quad n \leq \Sigma(x)}{\Sigma; \Omega \vdash QFT n x \triangleright \Omega[x \mapsto \text{Phi } n]}$
$\frac{RQFT \quad \Omega(x) = \text{Phi } n \quad n \leq \Sigma(x)}{\Sigma; \Omega \vdash QFT^{-1} n x \triangleright \Omega[x \mapsto \text{Nor}]}$	$\frac{CU \quad \Omega(x) = \text{Nor} \quad \text{fresh}(x, n) \quad \iota}{\Sigma; \Omega \vdash CU(x, n) \iota \triangleright \Omega}$
$\frac{LSH \quad \Omega(x) = \text{Nor}}{\Sigma; \Omega \vdash Lshift x \triangleright \Omega}$	$\frac{SEQ \quad \Sigma; \Omega \vdash \iota_1 \triangleright \Omega' \quad \Sigma; \Omega' \vdash \iota_2 \triangleright \Omega''}{\Sigma; \Omega \vdash \iota_1 ; \iota_2 \triangleright \Omega''}$

Fig. 14. Select QASM typing rules

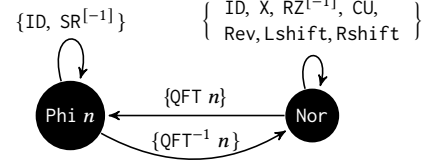


Fig. 15. Type rules' state machine

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 10) 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 15. 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 ι and input state φ to an output state φ' , written $\llbracket \iota \rrbracket \varphi = \varphi'$, shown in Figure 16.

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 16 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. 16. \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 SR^{-1}) applies an $m+1$ series of RZ (or RZ^{-1}) rotations where the i -th rotation applies a phase of $\alpha(\frac{1}{2^{m-i+1}})$ (or $\alpha(-\frac{1}{2^{m-i+1}})$). qt applies an approximate quantum Fourier transform; $|y\rangle$ is an abbreviation of $|b_1\rangle \otimes \dots \otimes |b_i\rangle$ (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). qt^{-1} is the inverse function of qt. Note that the input state to qt^{-1} is guaranteed to have the form $\bigotimes_{k=0}^{i-1} (|\Phi(\frac{y}{2^{n-k}})\rangle)$ because it has type $\text{Phi } n$. pm_l , pm_r , and 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 φ 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 $\alpha(r) |\Phi(\frac{v}{2^{n-k}})\rangle$.⁵

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

⁵Note 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. 17. Select \mathbb{Q} ASM inversion rules

THEOREM 3.2. [\mathbb{Q} ASM type soundness] If $\Sigma; \Omega \vdash \iota \triangleright \Omega'$ and $\Sigma; \Omega \vdash \varphi$ then there exists φ' such that $\llbracket \iota \rrbracket \varphi = \varphi'$ and $\Sigma; \Omega' \vdash \varphi'$.

Algebra. Mathematically, the set of well-formed d -qubit \mathbb{Q} ASM states for a given Ω can be interpreted as a subset \mathcal{S}^d of a 2^d -dimensional Hilbert space \mathcal{H}^d ,⁶ 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. 2021a]. Because \mathbb{Q} ASM'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 \mathbb{Q} ASM programs.

[Liyl: good?] Given a qubit size map Σ and type environment Ω , the set of \mathbb{Q} ASM programs that are well-typed with respect to Σ and Ω (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 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 \mathbb{Q} ASM 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 \mathbb{Q} ASM operation is valid in a traditional quantum language like SQIR. We then have the following two theorems to connect \mathbb{Q} ASM 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 \mathbb{Q} ASM to SQIR (??). Theorem 3.3 suggests that the space \mathcal{S}^d is closed under the application of any well-typed \mathbb{Q} ASM operation. Theorem 3.4 says that \mathbb{Q} ASM oracles can be safely applied to superpositions over classical states.⁷

\mathbb{Q} ASM programs are easily invertible, as shown by the rules in Figure 17. 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 17). This allows us to reuse

⁶A 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.

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

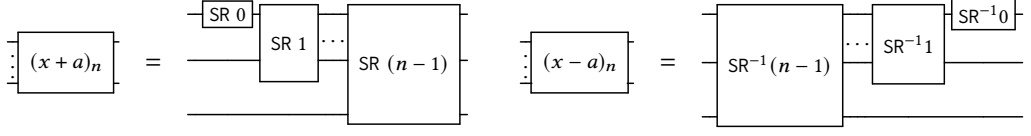


Fig. 18. Addition/subtraction circuits are inverses

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 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, $\llbracket \iota; \iota' \rrbracket \varphi = \varphi$.