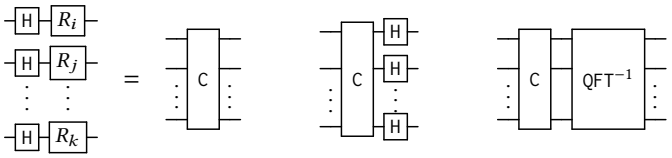

Fig. 22. Example quantum pattern as part of a graph representation in the quantum walk algorithm [Loke 2017]. R_i is a z-axis rotation by $2\pi/2^i$ (written `RZ i` in ℚASM+).

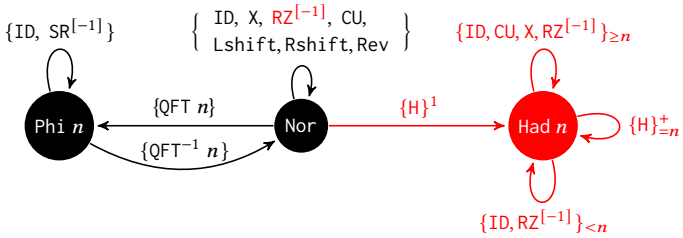

Fig. 23. Typing state machine. Red indicates new additions to ℚASM+.

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

Fig. 24. ℚASM+ syntax. For an operator OP, OP$^{[-1]}$ indicates that the operator has a built-in inverse available.

A ℚASM+: EXTENDING ℚASM WITH ADDITIONAL GATES

ℚASM+ extends ℚASM by adding an Rz gate, a Hadamard gate, and a Had n basis, which uses the same qubit form as the Phi basis (i.e., $\alpha(r_1)\,|\Phi(r_2)\rangle$). ℚASM+ is useful for defining common circuit patterns that appear as subcomponents of quantum algorithms. For example, the circuit C in Figure 22 is used in several places in the encoding of the graph in a quantum walk algorithm [Loke 2017], as shown on the right of Figure 22. The behaviors of circuit like C, which consist only of Hadamard and Rz gates, can be efficiently tested using ℚASM+. Recall from Figure 2 that QFT and AQFT circuits also have this form, meaning that we can use ℚASM+ to evaluate different implementations of AQFT/QFT to use when compiling ℚASM to SQIR.

A.1 ℚASM+ Syntax, Type System, and Semantics

The added syntax in ℚASM+ is marked in red in Figure 24. Every valid ℚASM program is also valid in ℚASM+.

All ℚASM type rules are also valid in ℚASM+. To make the extension useful for defining some quantum patterns beyond oracle circuits, we violate the type reversibility property of ℚASM+ by adding additional typing rules summarized by the red parts of Figure 8 and listed in the top of Figure 25. Rules RZ, RRZ, RZ-HAD, and RRZ-HAD deal with RZ gates. Rules HAD-1 and HAD-2 enforce that once a variable is in the Had n basis, it can never return to Nor. These rules represent the gradual transition of a Nor basis variable to a Had basis variable. Given a Had i basis variable x, H can only be applied to (x, i), and the result is in the Had $i + 1$ basis, as indicated by the $\{\mathsf{H}\}^{+}_{=n}$ label

Typing:

$$\text{RZ}\ \frac{\Omega(x) = \mathsf{Nor} \qquad n < \Sigma(x)}{\Sigma; \Omega \vdash \mathsf{RZ}\ q\ (x, n) \triangleright \Omega} \qquad \text{RRZ}\ \frac{\Omega(x) = \mathsf{Nor} \qquad n < \Sigma(x)}{\Sigma; \Omega \vdash \mathsf{RZ}^{-1}\ q\ (x, n) \triangleright \Omega} \qquad \text{RZ-HAD}\ \frac{\Omega(x) = \mathsf{Had}\ n \qquad m < n}{\Sigma; \Omega \vdash \mathsf{RZ}\ q\ (x, m) \triangleright \Omega}$$

$$\text{RRZ-HAD}\ \frac{\Omega(x) = \mathsf{Had}\ n \qquad m < n}{\Sigma; \Omega \vdash \mathsf{RZ}^{-1}\ q\ (x, m) \triangleright \Omega} \qquad \text{HAD-1}\ \frac{\Omega(x) = \mathsf{Nor} \qquad 0 < \Sigma(x)}{\Sigma; \Omega \vdash \mathsf{H}\ (x, 0) \triangleright \Omega[x \mapsto \mathsf{Had}\ 1]} \qquad \text{HAD-2}\ \frac{\Omega(x) = \mathsf{Had}\ n \qquad n < \Sigma(x)}{\Sigma; \Omega \vdash \mathsf{H}\ (x, n) \triangleright \Omega[x \mapsto \mathsf{Had}\ n+1]}$$

$$\text{NOR-HAD}\ \frac{\Sigma; \Omega[\mathsf{var}(\iota) \mapsto \mathsf{Nor}] \vdash \iota \triangleright \Omega' \qquad \Omega'(\mathsf{var}(\iota)) = \mathsf{Nor} \qquad \Omega(\mathsf{var}(\iota)) = \mathsf{Had}\ n \qquad n \le \mathsf{pos}(\iota) < \Sigma(x)}{\Sigma; \Omega \vdash \iota \triangleright \Omega'[\mapsto \mathsf{Had}\ n]}$$

Semantics:

$$\begin{aligned}
&[\![\mathsf{RZ}\ m\ (x,i)]\!]\varphi && = \varphi[(x,i) \mapsto \uparrow \mathsf{rz}(m, \downarrow \varphi(x,i))] \\
&&& \text{where } \mathsf{rz}(m, |0\rangle) = |0\rangle \quad \mathsf{rz}(m, |1\rangle) = \alpha(\tfrac{1}{2^m})\,|1\rangle \quad \mathsf{rz}(m, |\Phi(r)\rangle) = |\Phi(r + \tfrac{1}{2^m})\rangle \\
&[\![\mathsf{RZ}^{-1}\ m\ (x,i)]\!]\varphi && = \varphi[(x,i) \mapsto \uparrow \mathsf{rrz}(m, \downarrow \varphi(x,i))] \\
&&& \text{where } \mathsf{rrz}(m, |0\rangle) = |0\rangle \quad \mathsf{rrz}(m, |1\rangle) = \alpha(-\tfrac{1}{2^m})\,|1\rangle \quad \mathsf{rrz}(m, |\Phi(r)\rangle) = |\Phi(r - \tfrac{1}{2^m})\rangle \\
&[\![\mathsf{H}\ (x,i)]\!]\varphi && = \varphi[(x,i) \mapsto |\Phi(\tfrac{b}{2})\rangle] \quad \text{where } \downarrow \varphi(x,i) = |b\rangle
\end{aligned}$$

Fig. 25. ⟦QASM+ additional typing and semantics rules

in Figure 8. For $j < i$, (x, j) has the form $\alpha(b)\,|\Phi(b')\rangle$; thus, only ID and RZ^{-1} gates are allowed. For $j \ge i$, (x, j) has the form $\alpha(b)\,|\Phi(c)\rangle$; thus, all allowed Nor gates are permitted.

The ⟦QASM+ semantics extends the ⟦QASM semantics (Figure 9) with rules for H and RZ gates, as shown in the bottom of Creffig:pqasm-sem.

To support the new Had basis, we extend Definition 3.1 as follows:

Definition A.1 (Well-formed ⟦QASM state). A state φ is *well-formed*, written $\Sigma; \Omega \vdash \varphi$, iff:

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

We have re-proved type soundness (Theorem 3.2) for ⟦QASM+, but the subgroupoid (Theorem 3.3) and type reversibility (Theorem 3.5) theorems *do not* hold. We can re-state the quantum summation formula (Theorem 3.4) as follows, allowing input in the Nor basis and output in the Had basis.

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

A.2 Case Study: Compare and Contrast QFT and approximate QFT circuits

The AQFT circuit implementation being analyzed here is described in Figure 2. There are other possible ways of implementing AQFT circuits, such as removing more RZ gates in the middle of the qubit array, rather than removing them for the end array qubits. All of these implementations can be defined in ⟦QASM+. In analyzing the AQFT semantics, we utilize the random testing framework to test if a semantic pattern is correct based on the circuit in Figure 2, with bitstring inputs representing Nor basis values. The procedure works as follows: we first guess a semantic property, and use the testing framework to test its correctness. If it fails, we then use the framework to see the distance

between the property and the circuit implementation, then we repair the semantic property to approach the correct circuit semantics. This semantic property gives us the situation when inputs are in Nor. Once we have the property, we can then use the summation formula (Theorem A.2) to infer the behavior of the quantum circuit with superposition inputs.

There are many ways ℚASM+ can be useful. First, QFT/AQFT circuits are designed to disentangle the entanglement [Barenco et al. 1996; Hales and Hallgren 2000; Nam et al. 2020] to achieve physical implementation benefits. This makes it easy to analyze in ℚASM+ because we guarantees the non-entanglement so that states are separably analytical. However, it is actually hard to analyze the reversed QFT/AQFT circuits $\mathtt{QFT}^{-1}/\mathtt{AQFT}^{-1}$, because their non-entanglement properties are not trivial. A simple way of analyzing $\mathtt{QFT}^{-1}/\mathtt{AQFT}^{-1}$ circuits is to infer the reversed matrix value based on the QFT circuit behavior. By given the semantic output of applying QFT/AQFT devices on an input in ℚASM+, we can infer the semantic output for the reversed circuit by computing the inverse value for each qubit, since we are sure that none of the qubits are entangled.

Second, a lot of the times, we also want to learn about the side-effects of these quantum patterns in a quantum algorithm. We utilize the ℚASM+ random testing framework to cooperate with SQIR verification framework to finish such analysis. Hietala et al. [Hietala et al. 2021b] previously proved the semantics of Quantum Phase Estimation (QPE) with the QFT circuit that utilizes a $\mathtt{QFT}^{-1}$ device. We mimic a approximate QPE by replacing the $\mathtt{QFT}^{-1}$ with our AQFT device ($\mathtt{AQFT}^{-1}$). By using random testing to estimate the maximum distance between circuits and their approximate circuits, we learn about the semantics for QFT and AQFT. We then utilize the reversed circuit formula described above to infer the semantics of $\mathtt{QFT}^{-1}$ and $\mathtt{AQFT}^{-1}$. After that we can plug the two semantic statements into the summation theorem in Theorem A.2. We then compile the circuits and proofs to SQIR as two SQIR semantic statements for $\mathtt{QFT}^{-1}$ and ($\mathtt{AQFT}^{-1}$. The final maximum distance theorem between QPE and approximate QPE utilizes the two semantic statements in terms of two summation forms, and it is listed as:

Theorem A.3. [QPE Maximum Distance] For a QPE circuit with a QFT device as $\mathtt{QPE}(\mathtt{QFT}^{-1})$, and an approximate QPE circuit with an $\mathtt{AQFT}^{-1}$ device as $\mathtt{QPE}(\mathtt{AQFT}^{-1})$, if n is the number of qubits for the input of $\mathtt{QFT}^{-1}$ and $\mathtt{AQFT}^{-1}$, and i is the precision number of AQFT, if $O(n) \gg O(n-i)$, for any ϵ and eigenvalue input $|u\rangle$, there exists k, such that for $k < n$, $\|\mathtt{QPE}(\mathtt{QFT}^{-1})\,|u\rangle - \mathtt{QPE}(\mathtt{AQFT}^{-1}\,|u\rangle)\| < \epsilon$.

This theorem shows that the maximum distance of a QFT and an approximate QFT is insignificant in QPE circuits. The QFT substitution for an AQFT is a good approximation in the QPE circuit. The above mechanism shows how we can evaluate and discover different quantum pattern usage in different quantum algorithms.

B ℚIMP: SEMANTICS, TYPING, AND COMPILATION

Though it is common practice, writing oracles in a host metalanguage—like using Coq to write ℚASM programs—is tedious and error prone. To make writing arithmetic-based quantum oracles easier, we developed ℚIMP, a source-level, imperative language. This section describes ℚIMP's design rationale and features, and sketches its formal semantics, metatheory, and partially-verified compilation to ℚASM.

B.1 Syntax, and Overview

The grammar of core ℚIMP is given in Figure 26. An ℚIMP program P is a sequence of global variable declarations $\overline{\tau\ x}$ followed by a sequence of function definitions $\overline{d}$, where the last of these acts as the "main" function. A function definition d declares parameters and local variables; its body consists of a statement s; and it concludes by returning a value v. All variables are initialized as 0.

Bitstring	b		
Nat. Num	m, n		
Variable	x, y		
Mode	q	::=	$C \mid Q$
Base type	ω	::=	$\texttt{bool} \mid \texttt{fixedp} \mid \texttt{nat}$
Type	τ	::=	$\omega^q \mid \texttt{array}\ n\ \omega^q$
Value	v	::=	$l \mid (\omega)b$
LValue	l	::=	$x \mid x[v]$
Bool Expr	e	::=	$v < v \mid v = v \mid \texttt{even}\ v \mid \ldots$
Operator	op	::=	$+ \mid - \mid \times \mid \otimes \mid \ldots$
Statement	s	::=	$l \leftarrow v\ op\ v \mid l \overset{op}{\longleftarrow} v \mid l \leftarrow v \mid l \leftarrow f\ \bar{v}$
		$\mid$	$\texttt{inv}\ l \mid \texttt{for}\ x\ v\ s \mid \texttt{if}\ e\ s\ s \mid s\ ;\ s$
FunDef	d	::=	$\texttt{def}\ f\ (\overline{\omega^q\ x})\ \{\ \overline{\tau\ y}; s; \texttt{return}\ v\}$
Program	P	::=	$\overline{\tau\ x};\ \overline{d}$

Fig. 26. Core ℚIMP syntax

$$\frac{}{\Xi \vdash \sigma; l \leftarrow v_1\ op\ v_2 \longrightarrow \sigma[l \mapsto \mathsf{app}^{op}(v_1, v_2)]}(\texttt{bin}) \qquad \frac{\sigma; v \longrightarrow n \quad \Xi \vdash \sigma[x \mapsto 0]; s; n \longrightarrow \sigma'}{\Xi \vdash \sigma; \texttt{for}\ x\ v\ s \longrightarrow \sigma'}(\texttt{for\_t})$$

$$\frac{}{\Xi \vdash \sigma; s; 0 \longrightarrow \sigma}(\texttt{for\_0}) \qquad \frac{\Xi \vdash \sigma; s \longrightarrow \sigma' \quad \Xi \vdash \sigma'; s; n \longrightarrow \sigma''}{\Xi \vdash \sigma; s; \mathsf{S}\ n \longrightarrow \sigma''}(\texttt{for\_n})$$

Fig. 27. Select ℚIMP semantics

Variables x have types τ, which are either primitive types ω^q or arrays thereof, of size n. A primitive type pairs a base type ω with a quantum mode q. There are three base types: type `nat` indicates non-negative (natural) numbers; type `fixedp` indicates fixed-precision real numbers in the range $(-1, 1)$; and type `bool` represents booleans. The programmer specifies the number of qubits to use to represent `nat` and `fixedp` numbers when invoking the VQO compiler. We discuss modes q and our rationale for the choice of primitive types, below.

ℚIMP statements s consist of assignments, loops, conditionals, and sequences of statements, as is typical. Assignments are always made to lvalues l, which are either variables or array locations, and come in four forms. $l \leftarrow v\ op\ v$ assigns to l the result of applying binary operator op on two value parameters v. Values can be either lvalues or literals $(\omega)b$, where b is a bitstring and ω is the base type that indicates its interpretation. $l \overset{op}{\longleftarrow} v$ is a unary assignment; it is equivalent to $l \leftarrow l\ op\ v$. $l \leftarrow v$ initializes l to a value. $l \leftarrow f\ \bar{v}$ assigns to l the result of calling function f with arguments $\bar{v}$.

B.2 Semantics

We define a big-step operational semantics for ℚIMP; select rules are shown in Figure 27. The main judgment has the form $\Xi \vdash \sigma; s \longrightarrow r$, which states that under function environment Ξ and input store σ, statement s evaluates to result r. Here, Ξ is a partial map from function variables f to their definitions, and σ is a partial map from variables x and array locations $x[n]$ to a *history* of literal values $\overline{(\omega)b}$. A result r is either an output store σ' or a run-time failure `Error`, which arises due to an out-of-bounds index or a division by zero. The semantics also defines the standard subsidiary judgments, e.g., $\Xi; \sigma \vdash e \longrightarrow (\omega)b$ evaluates e under Ξ and σ to a literal result.

$$\frac{}{\Gamma \vdash (\omega)b : \omega^C} \qquad \frac{\Gamma(x) = \mathsf{array}\ n\ \omega^q \quad \Gamma \vdash v : \omega^C}{\Gamma \vdash x[v] : \omega^q} \qquad \frac{\Gamma(x) = \omega^q}{\Gamma \vdash x : \omega^q} \qquad \frac{\Gamma \vdash v : \omega^q \quad q \sqsubset Q}{\Gamma \vdash v : \omega^Q}$$

$$\frac{\Gamma \vdash e : \mathsf{bool}^q \quad \Xi; \Gamma; q \sqcup q' \vdash s_1 \quad \Xi; \Gamma; q \sqcup q' \vdash s_2}{\Xi; \Gamma; q' \vdash \mathtt{if}\ e\ s_1\ s_2}(\mathtt{if})$$

$$\frac{\Gamma \vdash v_1 : \omega^C \quad \Gamma \vdash v_2 : \omega^C \quad \Gamma(l) = \omega^C}{\Xi; \Gamma; C \vdash l \leftarrow v_1\ op\ v_2}(\mathtt{binop\_c}) \qquad \frac{\begin{array}{c} l \neq v_1 \quad l \neq v_2 \\ \Gamma \vdash v_1 : \omega^Q \quad \Gamma \vdash v_2 : \omega^Q \quad \Gamma(l) = \omega^Q \end{array}}{\Xi; \Gamma; q \vdash l \leftarrow v_1\ op\ v_2}(\mathtt{binop\_q})$$

$$\frac{\Gamma(x) = \mathsf{nat}^C \quad \Gamma \vdash v : \mathsf{nat}^C \quad \Xi; \Gamma; q \vdash s}{\Xi; \Gamma; q \vdash \mathtt{for}\ x\ v\ s}(\mathtt{for}) \qquad \frac{\Xi; \Gamma; q \vdash s_1 \quad \Xi; \Gamma; q \vdash s_2}{\Xi; \Gamma; q \vdash s_1\ ;\ s_2}(\mathtt{seq})$$

$$\frac{\begin{array}{c} \Gamma \vdash l : \omega^q \qquad \Gamma' \vdash v : \omega^{q'} \qquad q' \sqsubset q \\ \Xi(f) = ((\tau_1\ x_1 ... \tau_n\ x_n), \overline{\tau\ y}, s, v, \Gamma') \quad \Gamma \vdash v_1 : \omega_1^C ... \Gamma \vdash v_n : \omega_n^C \quad \Gamma'(x_1) = \omega_1^C ... \Gamma'(x_n) = \omega_n^C \end{array}}{\Xi; \Gamma; q \vdash l \leftarrow f\ v_1 ... v_n}(\mathtt{call})$$

$$\frac{\Gamma' = \mathsf{gen\_}\Gamma(\Gamma, \overline{\tau\ x} @ \overline{\tau\ y}) \quad \Xi; \Gamma'\ ; C \vdash s \quad \Gamma' \vdash v : \omega^q}{\Xi\ ;\ \Gamma \vdash (\mathtt{def}\ f\ (\overline{\tau\ x})\ \{\ \overline{\tau\ y}; s; \mathtt{return}\ v\}) \triangleright \Xi[f \mapsto (\overline{\tau\ x}, \overline{\tau\ y}, s, v, \Gamma')]}(\mathtt{fun})$$

Fig. 28. Select ⒸQIMP type rules

The ⒸQIMP semantics is largely standard except for the treatment of σ: An assignment of v to a Q-mode variable x pushes v to the history $\sigma(x)$, and likewise for assignments to $x[n]$. Assignment to a C-mode variable x replaces x's history with the singleton $[v]$. Lookup of x returns the topmost literal of $\sigma(x)$, while `inv` x pops off the top element of $\sigma(x)$. For example, in Fig. 13, executing the statement z=pow$(x/2, 2 * n + 1)$ updates the topmost element of $\sigma(z)$ to $(\frac{x}{2})^{2n+1}$. After the computation of `inv` z, this new value for z is erased, and the store entry for z reverts to what it was before. Evaluating $l \leftarrow f\ \bar{v}$ performs uncomputation automatically. It amounts to evaluating f's statement s (stored in Ξ) under σ extended with mappings for the function's parameters to $\bar{v}$ and with mappings for local variables to 0. When execution of s concludes, the returned result will be copied to l, and the added mappings will be dropped from the output σ (effectively uncomputing them). Moreover any updates to global variables will be reverted, just as if `inv` had been applied for each modification thereto. To evaluate a program P, we populate Ξ with P's function definitions, set the initial σ to map P's global variables to $(\omega)0$ (where ω is extracted from the variable's declared type), and then evaluate `main`'s statement s (which has no arguments); `main`'s `return` v is evaluated using the σ' that results.

B.3 Typing

The semantics of ⒸQIMP defines what a program P would do if we ran it directly, but our intention is not to do this, but to compile P to a quantum oracle. We can think of P as a computation whose inputs are the Q-mode global variables; all of the C-mode globals—and function calls, loops, etc.—will be inlined away by the compiler, so that what remains can be compiled directly to a quantum circuit. When that quantum circuit is executed, it will produce the result indicated by the ⒸQIMP semantics.

To ensure that this is all possible, the ⒸQIMP type system restricts how various constructs are used, based on when their variables are available. This is the standard partial evaluation process [Jones et al. 1993]. The ⒸQIMP type system is given in Fig. 28. It defines three judgments. The first is $\Gamma \vdash v : \omega^q$, which states that under assumptions Γ, value v has primitive type ω^q; a similar judgment (not shown) handles boolean expressions e. As usual Γ maps variables x to types τ. Arrays are not

first class, and always classical, so the array index rule requires the index to have mode C. Modes q are organized as a lattice with $C \sqsubset Q$ for the purposes of subtyping: a value known at compile-time (C) can have its use deferred to run-time (Q) if need be.

The second judgment has form $\Xi; \Gamma; q \vdash s$, which states that in context q and under assumptions Ξ and Γ, statement s is well formed. The context q indicates the mode of the data that determines whether the current statement was reached, either classical C (compile time) or quantum Q (run time). At the outset, the context q is C—the program's execution depends on no prior result—but the context can change at a conditional. If the guard e's type is $\texttt{bool}^Q$, then which branch executes depends on a quantum result, so the branches should be checked in mode Q. If the guard e's type is $\texttt{bool}^C$, then the branches should be checked in the current mode q. Both notions are captured in a single rule, with the branches checked in mode $q' \sqcup q$, where q' is the current mode and q is from the guard's type.

Rule `(binop_c)` types the assignment to a C-mode (compile-time) lvalue. Here, both operands must be C mode too, i.e., known at compile time. Such a statement can only be typed in context C: It will have no run-time effect, so its execution must not be conditional on quantum data. Rule `(binop_q)` considers assignments to Q-mode variables, which can occur in any context q. Both operands are also in Q mode (possibly made so through subtyping), and moreover must be different from the output-variable (l), though the operands can be the same. This restriction is leveraged to simplify compilation (the $l \xleftarrow{op} v$ form can be used to update the left-hand side). If both operands are the same, one is copied out to a temporary register, to avoid violating the no cloning rule.

Rule `call` checks that the input arguments and the parameters for the function have the same size, the types are matched, and the return value (v) of the function is a subtype of the to-be-assigned variable (l).

A `for` loop may be evaluated in whatever context q its body s may be evaluated in. During compilation it will be unrolled, so we require the iterator x and the bound value v to be type C.

The rule (`fun`), which type checks a function, has form $\Xi; \Gamma \vdash d \triangleright \Xi'$ where Ξ and Ξ' are partial maps from function names to a tuple of (i) a list of function arguments, (ii) a list of declarations ($\overline{\tau\ y}$) in the function, (iii) a function body statement, (iv) the return value for the function, and (v) a map (Γ) recording the types for variables in the function and global variables declared in the program. The judgement outputs Ξ' containing the function information and acting as the new function environment. $\texttt{gen\_}\Gamma(\Gamma, \overline{\tau\ x}@\overline{\tau\ y})$ adds variable-type information to Γ for the two lists $\overline{\tau\ x}$ and $\overline{\tau\ y}$. The list $\overline{\tau\ x}$ should contain only C-mode variables since we require all function arguments to be C (users can always use global Q-mode variables), while the local declaration list $\overline{\tau\ y}$ contains C or Q mode variables. The type judgment for functions is similar to the `(fun)` rule except that the input is a list of functions, and the type judgment for the whole program is similar to the `(call)` rule except that we also need to generate a type environment for global variables.

Well-formedness of `inv` x statements is checked in conjunction with typing, as described in Section 4.3.

B.4 Soundness

We have proved the soundness of the ℚIMP type system. Here we only show the type soundness for the ℚIMP statements s, as the theorem is the main theorem in proving ℚIMP type soundness. Before introducing the type soundness, we introduce a map Ψ from variables to natural numbers that represent the number of elements defined for an array. If the variable is not an array, its number is 1 in Ψ. This information is generated from the global and local variable declarations, and we disallow a program to create an 0 length array. Based on Ψ, Γ and S for a program, we then have the following store consistency definition:

Definition B.1. (Store Consistency) Let s be an ℚIMP statement, Γ is the type environment such that $\Gamma \vdash s$; Ψ is a map from variables to natural numbers and for all every variable x in s, $\Psi(x) \neq 0$; S is a store; we say that S is consistent with Γ on Ψ, if and only if for every variable x and index i, if $\Gamma(x)$ is defined and $i < \Psi(x)$, then there exists a value list $\overline{v}$, such that $\S(x, i) = \overline{v}$ and $\texttt{length}(\overline{v}) > 0$.

The type soundness for statements is divided into two theorems: the progress and preservation theorems. The progress theorem makes sure that every type checked program in ℚIMP can take a step:

Theorem B.2. (Progress) Let s be a (inv) well-formed ℚIMP statement, Γ and Γ' are the type environment such that $\Gamma \vdash s \triangleright \Gamma'$; Ψ is a map from variables to natural numbers, such that for all every variable x in s, $\Psi(x) \neq 0$; S is a store; and S is consistent with Γ' on Ψ, then there exists v, $\Xi \vdash_s S; s \longrightarrow v$.

The value v above can be an `Error` state or a final store evaluated from the big-step semantics ($\vdash_s$). The progress theorem make sure that every well-typed program is evaluated to something. Another important theorem is the preservation theorem. The ℚIMP operational semantics is in big-step format. It is actually trivial that the final type environment after an evaluation of a statement s is the same as the input type environment before the evaluation. We choose to prove another form of preservation related to the store consistency:

Theorem B.3. (Preservation) Let s be a (inv) well-formed ℚIMP statement, Γ and Γ' are the type environment such that $\Gamma \vdash s \triangleright \Gamma'$; Ψ is a map from variables to natural numbers, such that for all every variable x in s, $\Psi(x) \neq 0$; S and S' are stores; S is consistent with Γ' on Ψ; and $\Xi \vdash_s S; s \longrightarrow S'$; then S' is consistent with Γ' on Ψ.

The preservation theorem makes sure that if a program is evaluated to a non-`Error` state with the resulting store S', the store is still consistent. The proofs of the two theorems are done by induction on the statement s. We mechanicalized the proof in Coq.

B.5 Compilation to ℚASM

The compilation from ℚIMP to ℚASM is a combination of traditional compilation work with quantum compilation factors. In one hand, the compilation of assignments, for-loop and branching statements are all well-known as compilation techniques. For example, for compiling for-loop, we use loop unrolling, which is well studied.

The complication appears in the distinguish in C/Q mode values. This division has two consequences in the compilation: (1) we need to keep a store σ_c for C-mode variables because they are partially evaluated and not compiled, and the store value changes if we compile a C-mode assignment; (2) the compilation cases, of compiling arithmetic operations, face an exponential increase due to the division. The solution is to divide the compilation into two steps. We first assume there is uniformed ℚASM arithmetic programs for handling each individual arithmetic operation correctly. We compile an ℚIMP program to an ℚASM circuit that contains these programs. The correctness for this level is based on the assumptions that these ℚASM arithmetic programs works correctly. For example, an addition operation $l \xleftarrow{+} v$ is compiled to a pre-defined ℚASM program $\texttt{add}(l, v)$ that adds the value v to l regarding if v is a C or Q mode variable. We then compile these arithmetic program to different target ℚASM circuits. For example, $\texttt{add}(l, v)$ is compiled to a QFT-adder for adding a Q-mode variable and a C-mode value if the global flag is set to `QFT` and the type of v is in C. We then verify/validate that the correctness of different arithmetic programs by theorem proving or random testing.

The compilation process for ℚIMP statements is a partial function having the form: $\Xi; \Gamma; \Theta \vdash (\mathbb{N} * \sigma * s) \rightsquigarrow \kappa$, while the compilation process for an ℚIMP program is a partial function $P \rightsquigarrow \kappa$. There are two global setting data for compiling a program. The first one is a flag fl that is either `Classical` or `QFT` to set up the kind of circuit users want to compile an ℚIMP program to. `Classical` means to compile the program to a circuit based on classical arithmetic operations in ℚASM, while `QFT` means to compile it to a circuit based on *QFT* arithmetic operations in ℚASM. The other datum is a $\mathbb{N}$ number setting the bit-size number in each value in the program. There is also a function $\texttt{sizeof}(\Gamma, x)$ provides the bit-size for a variabel in ℚIMP. Here is an explanation of the different arguments in the forms:

- The arguments P and s are an ℚIMP program and statement.
- The argument Ξ is a function map similar to the one in Sec. B.2, and it is generated from the ℚIMP type system.
- Γ, generated from the ℚIMP type checking (we do type checks for compiling a program), is the type environment for variables in a function.
- The $\mathbb{N}$ number represents the scratch space number used in the system. The number is initialized as 0 when the program compilation calls a function one. In ℚIMP, only a branching statement needs one-bit scratch space.
- Θ is a map from ℚIMP variables to ℚASM variables. We generate Θ as a finitely bijective relation between ℚIMP LValue and ℚASM variables. We discuss Θ in details in a later paragraph. In ℚIMP, Global and local variable names are in two different categories. χ is an extra ℚASM variable that does not show up in Θ. Even though Θ is bijective, to save space, we do optimizations to reuse ℚASM variables mapped in Θ. In mechanism, we keep a variable counter in generating mapping variables for ℚIMP variables, so that mapping variables used in one function call can be reused.

```
g(int b){          f(){                f'(){
  int a,b;          int a,b;           int c,d;
  a = 10;           a = call g(b);     c = call g();
}                  }                  d = call g();
                                      }
```

  In the above example, let's assume that variable counter is 0, in the function `f`, `a` and `b` are mapped to 0 and 1 in Θ, and inside the function g, the two variables `a` and `b` are first renamed to a_1 and b_1, then their mappings in Θ are 2 and 3. In the right code, the mappings of c and d are mapped to 0 and 1, but the two consecutive function calls `g` actually generate the same mappings for `a` and `b` in `g`. In this case, they are both 2 and 3, because the variable mappings used in the first `g` are reused in the second one.
- σ is a dynamic store that stores the values for all C-mode variables.
- κ is the compilation result. It is either a failure (`Error`, described in Sec. B.2), or a value whose type depends on if it is for an ℚIMP program or statement. If it is for compiling an ℚIMP program P, then the value is u, a compiled circuit and a final scratch space number. If is for compiling an ℚIMP statement, then the value has the form (n, σ, u), where n is the final scratch space number, σ is the post-store for C-mode variables, and u is the generated circuit.
- We use χ to represent scratch space, and n is scratch space number. It represents ancillary qubits in a circuit. The input scratch space number in the transition rules points to the scratch space that is currently blank and read be used, while the output one represents the number of scratch spaces used in compiling a program.
- When we compile a value to ℚASM, we are compiling bits 0 and 1 to qubits $|0\rangle$ and $|1\rangle$, which are represented as `Nval 0 [0]` and `Nval 0 [0]`, where `[0]` is the bitstring of 0.

Fig. 29 provides a selected set of compilation rules from ℚIMP to ℚASM arithmetic operations. There are cases for compiling an ℚIMP program to `Error` states and we omit them here. The first

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

$$\frac{\Gamma \vdash l : \omega^C \quad \Xi \vdash_s \sigma; l \xleftarrow{op} v \longrightarrow \sigma'}{\Xi; \Gamma; \Theta \vdash (n, \sigma, l \xleftarrow{op} v) \rightsquigarrow (n, \sigma', \texttt{ID}\ (\Theta(l), 0))}\,(\texttt{bin\_c}) \qquad \frac{\Gamma \vdash e : \texttt{bool}^C \quad \Xi \vdash \sigma; e \longrightarrow (\texttt{bool})\texttt{true} \quad \Xi; \Gamma; \Theta \vdash (n, \sigma, s_1) \rightsquigarrow (n', \sigma', u)}{\Xi; \Gamma; \Theta; \vdash (n, \sigma, \texttt{if}\ e\ s_1\ s_2) \rightsquigarrow (n', \sigma', u)}\,(\texttt{if\_c})$$

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

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

$$\frac{\begin{array}{c}\emptyset\ ;\ \texttt{gen\_}\Gamma(\overline{\tau}@[\tau\ x]) \vdash \overline{d}@[\texttt{def main}\ \emptyset\ e] \triangleright \Xi \\ \Xi(\texttt{main}) = (\overline{\tau\ y'}, \overline{\tau\ y}, s, l, \Gamma) \quad \Theta = \texttt{gen\_}\Theta^Q(\overline{\tau}@[\tau\ x]@\overline{\tau\ y'}@\overline{\tau\ y}) \quad \Gamma \vdash l : \omega^Q \\ \sigma = \texttt{init\_}\sigma^C(\overline{\tau\ y'}@\overline{\tau\ y}) \quad \Xi; \Gamma; \Theta \vdash (0, \sigma, s) \rightsquigarrow (n, \sigma', u) \quad u' = u\ ;\ \texttt{copy}(\Theta(l), \Theta(x))\ ;\ \texttt{qinv}(u)\end{array}}{p \rightsquigarrow (n, u')}$$

Fig. 29. Select ⟦QIMP⟧ to ⟦QASM⟧ compilation rules

two rules compile a unary assignment operation. If l and v are both typed as Q-mode, we compile the assignment to a pre-defined ⟦QASM⟧ program in Coq based on the op syntax ($\texttt{get\_op}(op)$). $\texttt{get\_op}(op)$ provides the right ⟦QASM⟧ program for expressing the ⟦QIMP⟧ operations. For example, if op is an addition, $\texttt{get\_op}(op)$ maps it to a pre-defined ⟦QASM⟧ program, named add. The function takes two values (l and v in the unary assignment) and their bit-size number, and generates a ripple-carry adder or a QFT-based adder based on the input flag fl. This rule is the version for two Q-mode variables. We have another one for adding a Q-mode variable with a C-mode value. On the other hand, if l is in C, we update the store σ by computing the addition directly using the ⟦QIMP⟧ semantic function (Sec. B.2).

The rules (if_c) and (if_q) are for branching operations. If the Boolean guard is in C, we evaluate the expression to its value, and choose one of the branches for further compilation based on the result. In (if_c), we show the case for the true value. If the Boolean guard is in Q, we generate an ⟦QASM⟧ expression for it, and continue compiling the two branches, and compile the whole circuit as shown in Fig. 29 (if_q). The Boolean guard value is stored in the scratch space (χ, n), where χ is the scratch space variable, and n is the current scratch space number.

Rule (call) in Fig. 29 compiles a function call. $\texttt{add\_}\Theta^Q$ and $\texttt{add\_}\Sigma^Q$ extends the ⟦QIMP⟧ to ⟦QASM⟧ variable map and the ⟦QASM⟧ bit-size map with the new Q-mode variables in the local declaration list $\overline{\tau_y}$. The ⟦QIMP⟧ type system requires that all function arguments ($\overline{\tau_x}$) are in C-mode so that we do not need to worry about if there exists Q-mode variables in $\overline{\tau_x}$. $\texttt{init\_S}^C$ initializes the values of all C-mode variables in $\overline{\tau_x}$ and $\overline{\tau_y}$ to the store σ. For all variables in $\overline{\tau_x}$, its value is initialized as the corresponding one in $\overline{v}$, and for all variables in the local declaration list $\overline{\tau_y}$, the value is a bit-string representing 0. In the generated circuit, $\texttt{copy}(x, y)$ is a pre-defined function in ⟦QASM⟧ to copy the states of all qubits in x to y by using a series of $\texttt{CU}\ p_x\ \texttt{X}\ p_y$ ⟦QASM⟧ operations. There are another two function call compilation rules for the case when l and l' are in C. If l' is in C, we do not need

to generate a circuit for the function f; instead, we just generate a circuit to initialize the l' value in σ'' for l, while if l is in C, we just assign the l' value to l in σ without generating any circuit.

The last rule in Fig. 29 compiles an ℴQIMP program. The @ operation is a list concatenation operation. An execution of an ℴQIMP program means to execute the `main` function (the last function in the function list $\overline{d}$), and assigns its return value to the last variable in the global variable list ($\overline{\tau}$). In compiling a program P, we uses the type checking in ℴQIMP to generate a function map Ξ. `gen_`Θ^Q and `gen_`Σ^Q have similar feature as the `add_`Θ^Q and `add_`Σ^Q in rule `(call)`. We use `init_S`^{C} to initialize a C-mode store for the `main` function arguments and local variable declaration list, and scratch space number is initialized to 0. This is half of the story for a program compilation when the return value mode for `main` is Q; otherwise, we then just get its result in the final store, and initializes x with the result.

B.6 Proof of Compilation Correctness

Before we describe the main theorem we proved. We first discuss the equivalence relation on the final values. For an ℴQIMP program P, the equivalence of final values for the ℴQIMP semantics of the compilation is defined as the evaluation result of the program matches the execution result of the circuit generated from the P compilation. For an ℴQIMP statement s, the store is a little complicated. Given an ℴQIMP statment s and an initial state σ, we can evaluate s to get a final store σ_q by executing the ℴQIMP semantics, or we can compile s to a tuple (σ_c, u), and execute u in ℴQASM on an initialized ℴQASM state φ to get a final state φ_q. Given a type environment Γ for variables in σ_q, an ℴQIMP to ℴQASM variable map Θ, and a bit-size map Σ, we construct the following equivalence relation between σ_q and (σ_c, φ_q):

Definition B.4. (Statement Final Value Equivalence) Let s be an ℴQIMP statement, Γ the type environment, Θ an ℴQIMP to ℴQASM variable map, Σ a bit-size map such that for all $(x, i) \in \mathtt{dom}(\Theta)$, $\Sigma(\Theta(x, i))$ is defined and it is equal to $\mathtt{sizeof}(\Gamma, l)$. σ_q is an ℴQIMP state, and (σ_c, φ_q) a pair of ℴQIMP and ℴQASM states. Then $\Sigma; \Theta; \Gamma \vdash \sigma_q \simeq (\sigma_c, \varphi_q)$, if and only if, for all indexed-variables (x, i) having $\Gamma(x) = \omega^Q$, we have $[\sigma_q(x, i)]_{\Sigma(\Theta(x,i))} = \mathtt{cval}^{\Theta(x,i)}(\Sigma(\Theta(x, i)))\ \varphi_q$, and for all indexed-variables (x, i) having $\Gamma(x) = \omega^C$, we have $\sigma_q(x, i) = \sigma_c(x, i)$.

$$\mathtt{cval}^x\ 0\ \varphi = \lambda i.0 \quad \mathtt{cval}^x(n+1)\ \varphi = (\mathtt{cval}^x\ n\ \varphi)[n \mapsto\downarrow (\varphi(x, n))]$$
$$\mathtt{where}\ \ \downarrow (\mathtt{Nval}\ c\ b) = c$$

`cval` is for getting bit-values from n qubit state described above, while $[b]_n$ gets the first n bits of the bit-string based on the bit-size number of the type of a given ℴQIMP value. We show the main theorem to prove below:

Theorem B.5. Let $p = (\overline{\tau}@[\tau\ x])\ ;\ \overline{d}$ be a well-formed ℴQIMP program, Ξ is generated by the ℴQIMP function type rule in Fig. 28, Θ an ℴQIMP to ℴQASM variable map for P, Σ a record of the bit-string sizes for the mappings in Θ for the global variables in P. Then for every $\sigma_q\ \sigma_c\ r\ r'\ \varphi\ \varphi_q$, such that $\Sigma; \Theta; \Gamma \vdash \sigma_q \simeq (\sigma_c, \varphi)$, $p \rightsquigarrow \kappa$, $\Xi \vdash \sigma_q; p \longrightarrow r$, if r is `Error`, then κ is also `Error`; if r is a value v, then κ is (n, u), such that $\Sigma; \varphi \vdash u \longrightarrow \varphi_q$, thus, $\mathtt{cval}^{\Theta(x)}(\Sigma(\Theta(x)))\ \varphi_q = v$.

The above theorem only discusses what happens at the top level. It is best to question the compilation correctness of individual ℴQIMP statements. Here is the theorem for a statement:

Theorem B.6. Let s be a well-formed ℴQIMP statement, Ξ the proper function type environment, q a statement mode, Γ and Γ' type environments such that $\Xi; \Gamma; q \vdash s \triangleright \Gamma'$, Θ a proper ℴQIMP to ℴQASM variable map for s, Σ a record of the bit-string size for the mappings in Θ for Q-mode variables in s, n the current scratch space number, σ_c an initial state containing the values for the C-mode variables in s, σ_q an initial state containing the values for the variables in s, φ the ℴQASM

state such that $\Sigma; \Theta; \Gamma \vdash \sigma_q \simeq (\sigma_c, \varphi)$. Then, for every $r\ r'$, such that $\Xi; \Gamma; \Theta \vdash (n, \sigma_c, s) \rightsquigarrow \kappa$ and $\Xi \vdash_s \sigma_q\ ;\ s \longrightarrow r$, if r is `Error`, so do κ, if r is a store σ'_q, then $\kappa = (n', \sigma'_c, u)$ and $\Sigma; \varphi \vdash u \longrightarrow \varphi_q$ and $\Sigma; \Theta; \Gamma \vdash \sigma'_q \simeq (\sigma'_c, \varphi_q)$.

All the theorem proofs have been mechanized in Coq. The proof is by induction on the compilation judgment, and relies on proofs of correctness of the functions we define to compile individual ⓞQIMP operations to ⓞQASM, as discussed in Section 4.1.