# Synthesis of Compact and Expressive Quantum-Circuit Optimizations

WEI QIANG, Columbia University, USA
RONGHUI GU, Columbia University, USA and CertiK, USA

Today's quantum devices are noisy, so reducing circuit size is critical for reliable execution. Existing rule-based optimizers often rely on large rule sets that are difficult to manage and still miss long-distance transformations. We present Qsymb, a framework for synthesizing compact and expressive quantum-circuit rewrite rules with formal guarantees. We formalize symbolic rewrite rules in which a symbolic gate represents infinitely many subcircuits. We then define canonical symbolic rules of the form $L; S = S; R$ and prove that they constitute a compact generative core from which general symbolic rules can be derived. On top of this formal foundation, given a gate set, Qsymb synthesizes (1) a small, non-derivable concrete rule set that is complete up to chosen size and qubit bounds, and (2) a small but expressive canonical symbolic rule set that captures transformations beyond finite or monomial-only patterns. We further present rule anchoring to derive optimization-effective rules from canonical symbolic rules. Together, these results provide both expressiveness and guarantees: soundness of synthesized rules via validation, non-derivability, and bounded completeness. On the IBM-Eagle gate set, Qsymb strictly outperforms state-of-the-art rewrite-based optimizers (Qiskit, Guoq, Quartz, TKET, and Queso) in two-qubit-gate reduction on 90%, 67%, 82%, 85%, and 83% of standard quantum algorithm benchmarks, respectively; on Nam gate set, the corresponding rates are 88%, 74%, 81%, 86%, and 82.9%. It achieves final average two-qubit-gate reductions of 27.44% and 29.95%, respectively.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Hardware** → *Quantum computation*; • **Theory of computation** → *Equational logic and rewriting*.

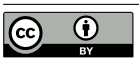
Additional Key Words and Phrases: quantum circuit optimization, rewrite-rule inference, equality saturation, symbolic rewriting

**ACM Reference Format:**
Wei Qiang and Ronghui Gu. 2026. Synthesis of Compact and Expressive Quantum-Circuit Optimizations. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 318 (October 2026), 28 pages. https://doi.org/10.1145/3839450

## 1 Introduction

Quantum computing promises to revolutionize computation by exploiting quantum-mechanical phenomena such as superposition and entanglement to perform tasks that are intractable for classical machines [37, 39]. Yet, until recently, quantum hardware has seen limited realization in practice. Today's quantum hardware—often referred to as Noisy Intermediate-Scale Quantum (NISQ) devices [32]—remains severely impacted by short coherence times, gate errors, and restricted qubit connectivity. Recent developments have implemented error-corrected logical qubits and demonstrated the potential to reduce logical errors [3]. As a result, the efficiency of compiled quantum circuits has a direct and often dominant impact on whether an algorithm can be executed reliably on real hardware. Because each operation is noisy, reducing circuit size is essential for

Authors' Contact Information: Wei Qiang, Columbia University, New York, USA, wei.qiang@cs.columbia.edu; Ronghui Gu, Columbia University, New York, USA and CertiK, New York, USA, ronghui.gu@columbia.edu.




ACM 2475-1421/2026/10-ART318
https://doi.org/10.1145/3839450

lowering error rates. Without optimization, circuits can produce results that are indistinguishable from random noise.

One promising direction is to treat circuit optimization as a process of equational rewriting, where small, equivalent circuit fragments are expressed as rewrite rules. The most prominent approach is peephole optimization, which uses local, bounded-size rewrite rules. Some work relies on domain experts to manually write a small set of rules [13, 18, 25], while others synthesize rewrite rules automatically [42, 44]. However, existing techniques still produce a large number of rules, many of which are redundant or derivable from others, and they still fail to capture the full range of useful transformations. For example, previous works synthesize either rewrite rules with symbolic parameters [44] or symbolic rules over monomial gates [42]; the latter represent only a finite set of monomial subcircuits, do not induce superposition, and are limited in expressiveness. Both approaches generate large rule sets yet still suffer from limited expressiveness.

The goal of this paper is to answer the following question: Can we automatically synthesize quantum-circuit optimizations that are compact, expressive, and effective in practice? Qsymb formalizes symbolic rewrite rules in which symbolic variables represent infinite subcircuit spaces, and their associated constraints characterize the full solution set for which the left-hand side and right-hand side are equivalent. We also formalize canonical symbolic rules of the form $L; S = S; R$, from which many other symbolic rules can be derived. These canonical rules provide a compact generative basis for deriving additional rules. Qsymb then introduces a novel approach to quantum optimization via rewrite-rule synthesis that addresses both limitations in compactness and expressiveness. We present a framework that, given a gate set, automatically synthesizes: (1) a much smaller set of concrete rewrite rules that are non-derivable yet complete for circuits up to a given size and qubit count, and (2) a small, expressive set of canonical symbolic rewrite rules in which symbolic variables represent infinite subcircuit spaces satisfying specific constraints, including subcircuits that induce superposition. Although the canonical symbolic rule set is complete up to the $(n, q)$ bounds, many of its rules are size-preserving. To address this, we introduce a novel technique for composing more useful rules from the canonical symbolic rule set, enabling more effective optimization. We demonstrate the effectiveness of our synthesized rules by applying them to optimize quantum circuits using common techniques such as equality saturation [40] and stochastic search. Our experiments show that our simple optimizer, using the generated rule set, achieves superior optimization results compared to prior approaches that rely on larger, less structured rule sets. To summarize, Qsymb contributes the following:

- Advances the existing rule inference approach using equality saturation in the synthesis of quantum concrete rules.
- Formalizes the notion of symbolic rule and presents a novel synthesis technique that generates symbolic rules efficiently and significantly reduces the term space without compromising completeness.
- Introduces a novel rule-composition approach, rule anchoring, that derives more useful rules from the rule set while keeping it small.
- Shows that the generated rule set can be integrated with simple optimization algorithms and outperforms state-of-the-art rewrite-based optimizers.

## 2 Background

### 2.1 Quantum Gates

A quantum bit (qubit) has two basis states, 0 and 1. A basis state is represented as $|0\rangle$ or $|1\rangle$. A qubit can also be in a linear combination (superposition) of its basis states, $\alpha|0\rangle + \beta|1\rangle$, where $\alpha$ and $\beta$ are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. $\alpha$ and $\beta$ are called amplitudes. A two-qubit state has

four basis states $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ and can be represented as a linear combination of these four basis states. The leftmost qubit is labeled $q_0$, which is the most significant bit, and the rightmost qubit is the least significant bit. Further, an $n$-qubit state can be represented as a linear combination of its $2^n$ basis states. Measurement of a qubit state $\alpha|0\rangle + \beta|1\rangle$ will collapse the qubit to $|0\rangle$ with probability $|\alpha|^2$ and collapse to $|1\rangle$ with probability $|\beta|^2$. A vector representation of a one-qubit state $\alpha|0\rangle + \beta|1\rangle$ is $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$. A two-qubit state $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$ can be represented as a vector $\begin{bmatrix}\alpha\\ \beta\\ \gamma\\ \delta\end{bmatrix}$.

Quantum operations transform qubit states of a quantum system. For example, the quantum X gate is analogous to the classical NOT gate: it transforms a qubit state $\alpha|0\rangle + \beta|1\rangle$ to $\beta|0\rangle + \alpha|1\rangle$. The Hadamard gate (H) transforms a qubit state $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. It transforms a basis state to a state in superposition, meaning that measuring the qubit will have a 50% chance of being 0 and a 50% chance of being 1. Controlled NOT (CNOT or CX) is a two-qubit gate that flips the second qubit if the first qubit is 1.

Quantum operations are represented as unitary matrices. For example, the X gate is represented as

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$

Applying the X gate to a one-qubit state $\alpha|0\rangle + \beta|1\rangle$ is equivalent to multiplying the matrix X by the state vector $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$:

$$X\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\beta\\ \alpha\end{bmatrix}$$

Quantum circuits are sequences of quantum gates applied to a set of qubits.

### 2.2 Path Sum Representation

A gate can be written in path-sum notation as

$$|x\rangle \to \sum_{y\in\mathbb{Z}_2} \phi(x,y,\theta)\,|f(x,y)\rangle$$

Here, $x$ is the input qubit state, $y$ is the path variable that sums over all possible paths, $\theta$ is the symbolic angle parameter, $\phi(x,y,\theta)$ is the phase polynomial that computes the amplitude of each path, and $f(x,y)$ is the output function that computes the output qubit state for each path. Intuitively, each path represents a resulting output state. If the gate induces superposition, there are multiple paths. For example, the H gate can be represented as:

$$|x\rangle \to \frac{1}{\sqrt{2}} \sum_{y\in\mathbb{Z}_2} e^{i\pi xy}\,|y\rangle$$

If the gate induces no superposition, there is only one path, which we call a monomial gate. For example, the X gate can be represented as: $|x\rangle \to |\neg x\rangle$.

### 2.3 Matrix Semantics

The semantics of a quantum circuit is a unitary matrix of size $2^n \times 2^n$, where $n$ is the number of qubits. Each entry in the matrix represents the amplitude of transforming one basis state into

another. The H gate has the following matrix semantics:

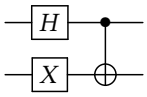
$$[\![H]\!] = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

Many gates supported by modern quantum devices take symbolic real-valued parameters; for example, the RZ gate is represented as follows:

$$[\![R_z^\theta]\!] = \begin{bmatrix}e^{-i\theta/2} & 0\\ 0 & e^{i\theta/2}\end{bmatrix}$$

The semantics of a quantum circuit is computed by matrix multiplication and the tensor product. For example, a circuit with two gates $G_1$ and $G_2$ applied sequentially has the semantics of $M = M_2 \cdot M_1$, where $M_1$ and $M_2$ are the matrix semantics of $G_1$ and $G_2$. A circuit with two gates $G_1$ and $G_2$ applied in parallel on two qubits has the semantics of $M = M_1 \otimes M_2$, where $\otimes$ is the tensor product operator.

**Example 2.1.** Consider a two-qubit circuit in which the H gate is applied to the first qubit and the X gate is applied to the second qubit in parallel, followed by a CNOT gate in which the first qubit is the control and the second is the target. The graph representation of the circuit is




Then, the matrix semantics of the circuit is computed as follows: $([\![CNOT]\!])([\![H]\!] \otimes [\![X]\!])$.

### 2.4 Quantum Rewrite Rule

Two circuits are equivalent if their matrix semantics are equivalent up to some global phase $\theta$, i.e., $\forall \vec{p}, \exists \theta, M_1(\vec{p}) = e^{i\theta} M_2(\vec{p})$, where $\vec{p}$ are the parameters of the circuits and $M_1, M_2$ are their matrix representations. To eliminate the existential quantifier, we follow an approach similar to that of Quartz [44] and enumerate a list of $\theta$ values that are linear combinations of the symbolic parameters $\vec{p}$ and have been shown to suffice for synthesis. Then, we need only check $M_1(\vec{p}) = M_3(\vec{p})$, where $M_3(\vec{p}) = e^{i\theta} M_2(\vec{p})$. A rewrite rule is a pair of semantically equivalent circuits. Rewrite rules can be composed to form a new rule that transforms a longer circuit. Fig. 1 shows a list of commonly used rewrite rules. Fig. 1(a) can be written as $cx\ q_0\ q_1; cx\ q_0\ q_1 \longrightarrow empty$.

An Equivalence Circuit Class (*ECC*) is a set of equivalent quantum circuits. *ECCs* denotes a set of equivalence classes. Any two quantum circuits in an ECC form a valid circuit rewrite. A set of ECCs is $(n, q)$-complete if any valid circuit transformation (rewrite) within size bound $n$ and qubit bound $q$ can be derived by composing transformations from the ECCs. The size of a circuit is its total number of gates, and $q$ denotes the number of qubits used by the circuit.

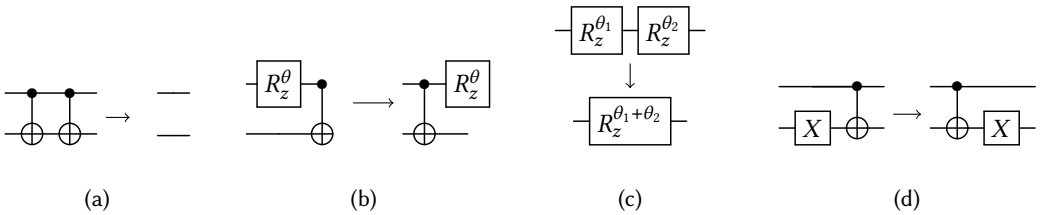



Fig. 1. Quantum rewrite rule example

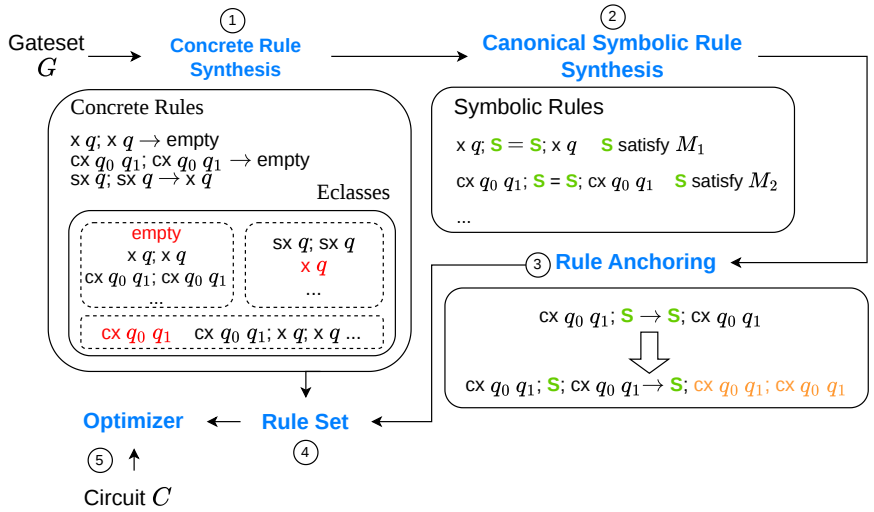



Fig. 3. Overview of Qsymb

## 2.5 Equality Saturation

An e-graph is a data structure commonly used in theorem provers [9, 16, 27] to compactly store equality relations between terms and congruence relations derived from them. An e-graph consists of a set of e-classes (equivalence classes), each representing a set of equivalent terms, with each term represented as an e-node. Equality saturation [40] is an optimization technique that utilizes e-graphs to perform program rewrites. The main idea is to add all possible equivalent terms of a program to the e-graph using a set of rewrite rules and then extract the optimal term according to a cost function. The process of adding equivalent terms to the e-graph is called saturation. During saturation, for each rewrite rule, if the left-hand side (lhs) of the rule matches an e-class in the e-graph, then the right-hand side (rhs) of the rule is added to the same e-class. Fig. 2 is an example of an e-graph and equality saturation.

This process is repeated until no more terms can be added to the e-graph. A major advantage of equality saturation engines is that they are non-destructive: they retain all equivalences derived, regardless of the order in which the rules are applied. Finally, the optimal term is extracted from the e-graph by searching for the term with the lowest cost, as defined by a specific cost function. We utilize equality saturation to check the derivability of a rewrite rule from a set of smaller rewrite rules. If $\mathbb{U}$ is a domain of rewrite rules (e.g., all possible rules within a gate set), a set of rewrite rules is non-derivable if no rule in the set is derivable by composing other smaller rules in $\mathbb{U}$. The size of a quantum-circuit rewrite rule is the maximum number of gates on either its lhs or rhs. Therefore, the rules in Fig. 1 are non-derivable because no smaller rules exist from which they can be composed.

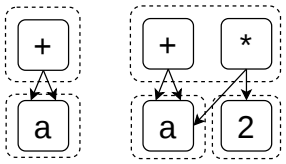



Fig. 2. The e-graph on the left contains the term $a + a$. Running equality saturation using the rewrite rule $a + a \leftrightarrow a * 2$ yields the e-graph on the right.

## 3 Overview

This section describes Qsymb, a synthesis engine that generates a compact set of symbolic rewrite rules that can represent an infinite space of subcircuits. Each rule has a symbolic subcircuit whose constraints characterize the full set of solutions under which the rule is valid. Qsymb proves that

(1) the rule set is non-derivable, meaning that no rule in the set can be derived from the others, and (2) any symbolic rule whose concrete part lies within the $(n, q)$ bounds can be derived from the canonical rules generated by Qsymb.

Fig. 3 provides an overview of our synthesis pipeline. The synthesis engine takes a gate set, which includes a list of gates and their semantics. In step ①, Qsymb synthesizes a small set of non-derivable concrete rules that are $(n, q)$-complete for given values of $n$ and $q$ by enumerating all possible circuit terms up to size $n$ and qubit count $q$, grouping them into equivalence classes, and inferring non-derivable rewrite rules from each equivalence class. In each e-class (dotted round box), the red circuit is the class representative, typically the shortest circuit. In step ②, given the previously synthesized e-classes, Qsymb synthesizes a set of canonical symbolic rewrite rules. Each e-class representative is enumerated as either the lhs or rhs concrete part of a symbolic rule. In each symbolic rule, the variable $S$ denotes any subcircuit that satisfies constraint $M_i$, where $M_i$ characterizes the full solution set under which the lhs and rhs are equivalent. In step ③, Qsymb designs a novel approach that selectively appends prefixes or suffixes to existing canonical symbolic rules so that a concrete rule can be applied afterward. These canonical symbolic rules serve as a generative basis for composing larger and more useful rules that reduce circuit size and collaborate seamlessly with concrete rules. In the figure, the orange part of the anchored rule can match the lhs of a concrete rule, which can reduce that part to an empty circuit. In step ④, concrete rules and anchored symbolic rules are collected into the final rule set, which is used as input to our optimizer.

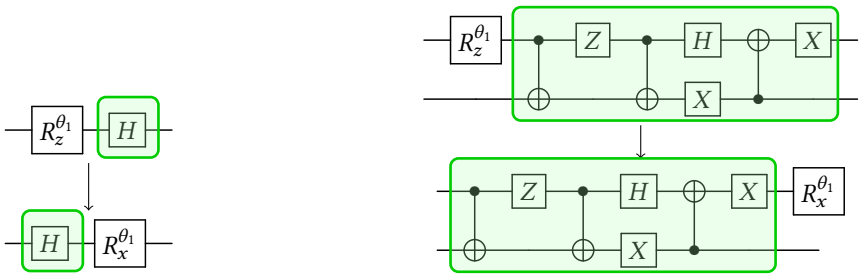
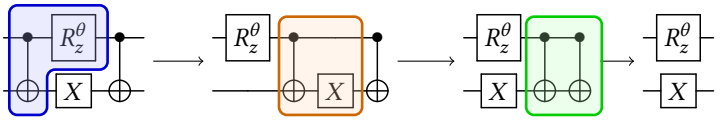



Fig. 4. Variants of RZ-RX transformation

**Step ①. Small $(n, q)$-Complete Concrete Rule Set.** Qsymb first constructs a small, non-derivable $(n, q)$-complete rule set that can represent all equivalences up to a given circuit size $n$ and qubit count $q$. It enumerates all circuits of sizes 1 through $n$ on $q$ qubits while removing redundant circuits that are derivable from the rules already learned.

**Example 3.1.** The figure below shows a transformation $cx\ q0\ q1; rz(\theta)\ q0; x\ q1; cx\ q0\ q1 \rightarrow rz(\theta)\ q0; x\ q1$. This transformation can be derived using the rules in Fig. 1 by applying the rules in Fig. 1(b), Fig. 1(d), and Fig. 1(a), in that order. Therefore, this rule is not learned because smaller rules already capture the equivalence.



Prior work [42, 44] often produces thousands to tens of thousands of rewrite rules, even when restricted to circuits of sizes smaller than six. To address this, given a gate set $G$, we construct a much smaller, non-derivable $(n, q)$-complete rule set that can represent all equivalences up to a given circuit size $n$ and qubit count $q$. The constructed equivalences are then used in symbolic-rule synthesis.

**Step ②. Expressive Symbolic Rule Set.** One limitation of concrete rules is that they are guaranteed to capture only equivalences of concrete circuits enumerated within $(n, q)$. Figure 4 shows two rewrite rules that share a concrete part but differ in their intermediate parts. For both intermediate parts, RZ can move to the right and become RX. In fact, the space of such intermediate parts is infinite, and there are infinitely many such subcircuits longer than $n$ that concrete rules are not guaranteed to derive. The purpose of an expressive symbolic rule is to represent the full space of such intermediate parts, compactly encoded as a constraint.

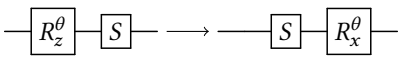



Fig. 5. Circuit graph representation of an RZ-RX transform symbolic rule

**Example 3.2.** Fig. 5 shows the symbolic rewrite rule that transforms an RZ gate at the start into an RX gate at the end, with a symbolic subcircuit $S$ in between. Qsymb synthesizes such rules and the constraints on $S$ so that the lhs and rhs are equivalent when the constraints are satisfied. The constraints are represented as a symbolic matrix with variables and are expressive enough to support rules in which $S$ can represent an infinite subcircuit space that may induce superposition. For example, Qsymb allows the rule in Fig. 5 to match subcircuits that may induce superposition, such as Example (f) in Fig. 4.

If there is no superposition in $S$, then $S$ is a symbolic monomial gate: modeling the constraint space of $S$ involves only finite permutations in which one qubit state transforms into another because each state is represented by a finite-length bit string. Below is a monomial-gate representation that transforms a qubit state $x$ into another state without superposition.

$$|x\rangle \rightarrow |f(x)\rangle$$

If $x$ is a one-qubit state, then there are two interpretations of the function: $\{|0\rangle \rightarrow |0\rangle, |1\rangle \rightarrow |1\rangle\}$ and $\{|0\rangle \rightarrow |1\rangle, |1\rangle \rightarrow |0\rangle\}$. One can simply enumerate all possible finite permutations to model the constraint space and determine the interpretations under which the rule is valid. However, supporting the synthesis of symbolic rules with a symbolic gate whose solution space includes monomial subcircuits and subcircuits that induce superposition is challenging. Here is the path sum of a circuit with superposition:

$$|x\rangle \rightarrow \sum_{y \in \mathbb{Z}_2} \phi(x, y, \theta)\, |f(x, y)\rangle$$

Every amplitude $\phi$ in every path is a complex number and is subject to the constraint $\sum_i |\phi_i|^2 = 1$. Consequently, the path sum admits an infinite number of interpretations. For example, interpretations include $\{|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),\ |1\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\}$, $\{|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle),\ |1\rangle \rightarrow \frac{1}{\sqrt{2}}(i\,|0\rangle + |1\rangle)\}$, $\{|0\rangle \rightarrow \cos\theta\,|0\rangle + \sin\theta\,|1\rangle,\ |1\rangle \rightarrow -\sin\theta\,|0\rangle + \cos\theta\,|1\rangle\}$, etc., where $\theta$ can take infinitely many values. While prior work [42] supports only symbolic rules in which $S$ can match monomial subcircuits, Qsymb models the solution space of $S$ as a symbolic matrix to compactly

represent the full solution space, which consists of an infinite number of interpretations of concrete circuits.

**Canonical Rules.** Enumerating all possible symbolic rules of the form $G_i; S; G_j \longrightarrow G_i'; S; G_j'$ causes combinatorial explosion. However, we prove that every symbolic rule $G_i; S; G_j \longrightarrow G_i'; S; G_j'$ has a canonical rule represented as $L; S = S; R$, such that $G_i; S; G_j \longrightarrow G_i'; S; G_j'$ can be derived from its canonical form, and many symbolic rules can be reduced to the same canonical form. This means that, during candidate enumeration, we only need to place the symbolic variable at the two boundaries (prefix/suffix), instead of considering all placements where $S$ appears in the interior of the circuit, while capturing the same amount of equivalences.

**Example 3.3.** The CX cancellation rule (1) has a canonical symbolic rule (2).

$$cx\ q_0\ q_1; S; cx\ q_0\ q_1 \longrightarrow S \tag{1}$$

$$cx\ q_0\ q_1; S = S; cx\ q_0\ q_1 \tag{2}$$

Intuitively, one can always derive (1) from (2) by right-appending $cx\ q_0\ q_1$ to both sides and applying the concrete rule $cx\ q_0\ q_1; cx\ q_0\ q_1 \longrightarrow empty$.

Even with canonical forms, there are still several challenges to address: (1) Enumerating all canonical forms of symbolic rules and checking each candidate individually remains intractable because the search space is still exponential if we naively enumerate $L$ and $R$. (2) Directly solving $S$ for unitary solutions requires solving a list of non-linear equations, which are computationally hard [4]. Qsymb introduces a property-grouping method that groups pairs of $L$ and $R$ with similar properties and proves that, once $L$ and $R$ are in the same group, there exists a solution for $S$ such that $[\![S]\!][\![L]\!] = [\![R]\!][\![S]\!]$. This solves the first challenge by grouping candidates that have solutions for $S$, thereby avoiding checks of all $N^2$ candidates. Qsymb solves the second challenge by avoiding the direct solution of the unitary constraints on $S$ and deriving only general solutions for $S$ without determining whether a unitary solution exists. However, we show that the property-grouping method can determine whether a unitary solution exists for $S$, allowing us to derive valid rules and check whether a subcircuit satisfies the constraints on $S$ without solving the unitary constraints.

**Step ③. Anchoring canonical rules.** After generating canonical symbolic rules with concrete parts bounded by size $n$ and qubit count $q$, Qsymb proves that other symbolic rules whose canonical forms are within those bounds can be derived from the canonical rules. Once canonical symbolic rules are generated, they serve as a generative basis that Qsymb uses to derive more useful rules. However, composing all possible rules is impractical because the set of possible rules is too large. Although the canonical rules already capture the equivalences, they are mostly size-preserving. Qsymb introduces a novel rule anchoring technique by selectively appending the same prefix or suffix to both sides of a canonical rule. The resulting symbolic rules are designed to collaborate with concrete rules: after a symbolic rule is applied, a concrete rule can be applied to the resulting term to reduce gate size.

**Example 3.4.** Consider the same rule from Example 3.3: $cx\ q_0\ q_1; S = S; cx\ q_0\ q_1$. Qsymb first observes that the concrete suffix on the right-hand side, $cx\ q_0\ q_1$, matches the prefix of the lhs of the concrete rule in Fig. 1(a): $cx\ q_0\ q_1; cx\ q_0\ q_1 \longrightarrow empty$. Then, Qsymb "anchors" the canonical rule by appending $cx\ q_0\ q_1$ to the right of both sides:

$$cx\ q_0\ q_1; S; cx\ q_0\ q_1 \longrightarrow S; \boxed{cx\ q_0\ q_1; cx\ q_0\ q_1} \tag{3}$$

Observe that the circuit in the orange box is now the lhs of the rule $cx\ q_0\ q_1; cx\ q_0\ q_1 \longrightarrow empty$, which enables that rule to apply after anchoring. Intuitively, during synthesis, we retain the anchored rule (3) rather than explicitly deriving rule (1) from the canonical rule. The additional

suffix enables rule (3) to match CX gates on both sides of $S$ and move one CX gate next to the other. This creates the orange-boxed subcircuit, to which the concrete cancellation rule in Fig. 1(a) can be applied at optimization time to cancel the two cx gates.

**Steps ④ and ⑤. Applying the rules.** After generating both symbolic and concrete rules, we demonstrate that they can be applied using a simple optimizer that combines equality saturation with a stochastic algorithm. Concrete rules are used directly in equality saturation, while symbolic rules are applied stochastically. Our experiments show that, with symbolic rules, this simple optimizer can already outperform most state-of-the-art rewrite-based optimizers.

**Theoretical Conclusions.** (1) We prove that every symbolic rule has a canonical form and that every symbolic rule can be reduced to and derived from its canonical form. (2) We prove that, for any canonical symbolic rule, if $[\![L]\!]$ and $[\![R]\!]$ share the same set of eigenvalues, there exists a unitary solution for $S$ such that $L; S = S; R$. (3) We prove that Qsymb generates a set of non-derivable canonical symbolic rules. To keep the main text concise, longer proofs of lemmas and theorems are provided in Appendices A and B.

## 4 Synthesizing Concrete Rules

Previous synthesis engines [42, 44] generate hundreds or thousands of quantum rewrite rules, many of which are derivable from smaller rules. Qsymb synthesizes a much smaller concrete rule set that can derive larger rules while guaranteeing $(n, q)$-completeness. It also removes rules that are derivable from previously generated rules. Qsymb combines Ruler-style rule inference [26] with equality saturation to infer a non-derivable rule set, and uses an efficient Polynomial Identity Filter (PIF) [42] to group equivalent circuits. Together, these components synthesize concrete rules that represent circuit equivalence classes (ECCs) within $(n, q)$ bounds.

### 4.1 Bottom-Up Synthesis Algorithm

*Our Algorithm.* Algorithm 1 shows how Qsymb synthesizes concrete rules. Lines 5–6 describe grouping all enumerated terms into classes. PIF [42] groups equivalent quantum circuits efficiently with a low failure rate. Using PIF, Qsymb groups equivalent terms into classes while keeping the probability of mixing semantically different circuits extremely low. The PIF map groups terms by a PIF-computed hash. Within each group, Qsymb filters out rules that are derivable from the verified rule set $L$ learned in previous iterations, thereby removing duplicates. In Lines 9–19, the e-graph adds each term of size $i$ in a class and runs saturation with the current learned rules to merge derivable terms. In Lines 15–17, if $A$ and $B$ cannot be proven equivalent under the current learned rules, the tuple $(A, B)$ is added to $C$. Thus, $C$ tracks likely equivalences that are not yet derivable. Method `Choose` selects new rules from $C$. It prioritizes candidates using heuristics (e.g., smaller rules first) via a priority queue initialized with $C$. `Choose` pops candidates in priority order. For each selected rule $r$, it checks whether $r$ is derivable from existing rules $R \cup L$. If it is not derivable, it validates equivalence with the SMT solver. Method `derivable` checks derivability under $R \cup L$ using equality saturation. In Line 21, newly derived rules $E$ are canonicalized to reduce duplication and then added to the learned rule set $L$.

**Correctness.** Because PIF is used only for grouping, every rule selected by `Choose` is validated with SMT-based equivalence checking, following Quartz [44]. Therefore, all rules in $L$ are validated, and $L$ always remains sound. In Line 13, the e-graph is saturated only with $L$, preserving soundness during saturation. Qsymb discards derivable rules early, so only a small set of non-derivable candidates proceeds to SMT verification.

**Algorithm 1** Concrete Rule Synthesis

```
1: function SynthesizeConcrete(n, q)
2:     PIF ← InitPIFMap()
3:     L ← ∅
4:     for i := 1 to n do
5:         X ← enumerate(i, q)
6:         PIF.insert(x) for each x ∈ X
7:         C ← ∅
8:         for each class EC grouped by PIF do
9:             egraph ← EmptyEgraph()
10:            for each term T ∈ EC do
11:                egraph.add(T)
12:            end for
13:            egraph.run_saturation(L)
14:            for all (A, B) ∈ egraph.eclasses × egraph.eclasses do
15:                if A, B not in same eclass of egraph then
16:                    C ← C ∪ {(A, B)}
17:                end if
18:            end for
19:        end for
20:        E ← ∅ ;
21:        E ← Choose(C, L)
22:        L ← L ∪ Canonicalize(E)
23:    end for
24:    return L
25: end function

26: function Choose(C, L)
27:    R ← ∅
28:    q ← PriorityQ(C)
29:    while q is not empty do
30:        r ← q.pop()
31:        if derivable(r, L ∪ R) then
32:            continue
33:        end if
34:        if smt.check_equivalent(r.lhs, r.rhs) then
35:            R ← R ∪ {r}
36:        end if
37:    end while
38:    return R
39: end function
```

**Completeness.** Given a gate set $G$, function `enumerate(i, q)` in Line 3 of Algorithm 1 enumerates all circuit terms of size $i$ on $q$ qubits. Therefore, all rewrite rules formed by terms with size up to $n$ and qubit count $q$ can be derived by the rule set synthesized by Algorithm 1.

**Remove Redundant Rules.** `Choose` removes rules that are already derivable by running equality saturation with learned rules $L$ and newly selected rules. If a rule's lhs and rhs are already in the same e-class, that rule is derivable from existing rules. Therefore, `Choose` selects only rules that are not yet derivable. QSymb also adopts pruning techniques from Quartz [44]: (1) `Enumerate` selects one representative term of size $i$ from each equivalence class to construct terms of size $i + 1$; and (2) circuits with common prefix or suffix subcircuits are filtered out. In `Choose`, the priority queue selects smaller and more abstract rules first. For example, $rz(\theta_1); rz(\theta_2) \rightarrow rz(\theta_1 + \theta_2)$ is selected

before $rz(\pi); rz(\pi) \longrightarrow rz(\pi + \pi)$; for rules with the same structure, the most general valid rule is selected first.

**Rule Canonization.** The `Canonicalize` function in Algorithm 1 converts rewrite rules into canonical form to reduce duplication. For example, we generated rules like

$$x\ q0; x\ q0 \longrightarrow empty$$

$$x\ q1; x\ q1 \longrightarrow empty$$

The two rules have the same meaning, except they act on different qubits. Thus, all such rules can be canonized into a single rule: $x\ q; x\ q \longrightarrow empty$.

## 5 Synthesizing Symbolic Rewrite Rules

Large concrete rule sets are hard to manage and can cause both synthesis and optimization costs to grow rapidly as term size increases. On the other hand, very small rule sets may suffer in optimization quality because long-distance rewrites are difficult to compose. In such cases, either many iterations of equality saturation [40] are required, or search algorithms such as beam search [42, 44] take a long time to discover large equivalences. For long circuits, equality saturation can quickly fill the e-graph with e-nodes. To address these limitations, we synthesize symbolic rules with symbolic variables, allowing a rule to represent infinitely many subcircuits for which the lhs and rhs are equivalent. These rules can match long-distance subcircuits as long as the constraints are satisfied. Previous work supports symbolic gate parameters [42, 44] or symbolic monomial gates [42], which represent only a finite set of subcircuit interpretations. However, real circuits often contain superposition, which requires more expressive symbolic rules that can represent subcircuits with superposition.

### 5.1 Symbolic Rule

A symbolic rule $R_s$ can be represented by the equation $G_1; S; G_2 = G_1'; S; G_2'$, where $G_1, G_2, G_1', G_2'$ are concrete circuit sequences, and $S$ is a symbolic variable representing a set of subcircuits such that, for any subcircuit $C$ in the set, $G_1; C; G_2 = G_1'; C; G_2'$. The $S$ on the lhs is used to match a subcircuit $C$, and the $S$ on the rhs indicates where $C$ should be substituted. Fig. 6 shows a circuit-graph representation of the symbolic rule in Example 3.3. The goal of Qsymb is to find a representation of $S$ that captures the complete set of unitary interpretations of $S$ under which the equation holds.

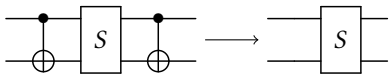



Fig. 6. Graph representation of a CX-cancellation symbolic rule.

$$S = \begin{bmatrix} a & b & c & c \\ d & m & f & f \\ g & h & j & k \\ g & h & k & j \end{bmatrix}$$

Fig. 7. Symbolic matrix that represents the intermediate subcircuit of CX cancellation.

### 5.2 Constraints

Qsymb represents the solution of $S$ as a symbolic matrix that captures all unitary matrices to which $S$ can be instantiated while keeping the lhs and rhs equivalent. During matching, a symbolic subcircuit can match any subcircuit whose unitary matrix is represented by this symbolic matrix.

Fig. 7 shows how the intermediate part of a symbolic rule (e.g., CX cancellation) can be represented as a $4 \times 4$ symbolic unitary matrix $S$, where $a, b, c, \ldots, j$ are complex variables satisfying the unitary constraint $SS^\dagger = I$. Expanding this constraint yields a set of non-linear equations, illustrated in Fig. 8. In the figure, $|x|^2$ denotes $xx^*$, where $x^*$ is the complex conjugate of $x$. To determine whether $S$ has valid solutions, Qsymb must find a symbolic matrix form (as in Fig. 7) that satisfies both constraints: (1) $G_1; S; G_2 = G_1'; S; G_2'$ and (2) $S$ has a unitary solution. However, enumerating circuits in (1) can cause combinatorial explosion, and solving non-linear equations in (2) is known to be computationally expensive [4] and to scale poorly as the matrix dimension increases. We now discuss how Qsymb avoids checking the non-linear equations and infers candidate solutions for $S$.

$$\begin{cases} |a|^2 + |b|^2 + |c|^2 + |c|^2 = 1 \\ |d|^2 + |m|^2 + |f|^2 + |f|^2 = 1 \\ |g|^2 + |h|^2 + |j|^2 + |k|^2 = 1 \\ ad^* + bm^* + cf^* + cf^* = 0 \\ \ldots \end{cases}$$

Fig. 8. The constraints corresponding to Fig. 7

### 5.3 Canonical Symbolic Rules and Property Grouping

Having defined symbolic rules and constraints, we now describe how Qsymb synthesizes symbolic rules. There are two key challenges. (1) Unlike concrete-rule synthesis, where likely equivalent concrete terms can be grouped efficiently using probabilistic equivalence checking, symbolic terms are hard to group because symbolic variables have infinitely many possible instantiations. (2) If there are $N$ concrete terms, checking whether each candidate admits a valid symbolic constraint requires considering an exponential number of candidates. For example, for two concrete candidate terms $G_1; G_2; G_3$ and $G_4; G_5$, we need to insert a symbolic variable at every position and produce an exponentially large number of symbolic-rule candidates: $G_1; S; G_2; G_3 = G_4; S; G_5$, $G_1; G_2; S; G_3 = G_4; G_5; S$, etc. Checking every candidate to determine whether such an $S$ exists is intractable. We now describe how Qsymb addresses these challenges.

*5.3.1* ***Canonical Form of Symbolic Rules.*** Given a candidate symbolic rule $G_1; S; G_2 = G_1'; S; G_2'$, where $S$ is a symbolic variable representing a subcircuit satisfying the constraint $M$, we describe how Qsymb derives the symbolic-matrix constraint or discards the candidate if no valid $S$ exists. We show that every candidate of this form can be reduced to, and derived from, a canonical-form representation $L; S = S; R$.

**Theorem 5.1.** *Given a gate set $A$, where each gate has a finite sequence of gates in $A$ that implements its inverse, and two circuits with a symbolic gate, $G_1; S; G_2$ and $G_1'; S; G_2'$, where $S$ is a variable that represents subcircuits satisfying the constraint $M$ and each $G_i$ is a concrete circuit, there exist circuits $L$ and $R$ implemented over $A$ such that $[\![L]\!] = [\![G_1]\!][\![G_1']\!]^{-1}$ and $[\![R]\!] = [\![G_2]\!]^{-1}[\![G_2']\!]$. $G_1; S; G_2$ and $G_1'; S; G_2'$ are equivalent iff $L; S$ and $S; R$ are equivalent.*

Proof. If each gate has a finite inverse implementation in $A$, there is always a finite circuit sequence that implements the inverse of $G_1$ and $G_2$. We can left-append a circuit that implements $[\![G_1]\!]^{-1}$ and right-append a circuit that implements $[\![G_2]\!]^{-1}$ to both sides of the equation to obtain the canonical form. The reverse direction is similar. □

We call $([\![L]\!], [\![R]\!])$ the canonical form of $G_1; S; G_2 = G_1'; S; G_2'$. A canonical form is valid if the constraint $M$ on $S$ contains at least one solution such that the rule's lhs and rhs are equivalent. The rule $L; S = S; R$ is one circuit-level representation of this canonical form, which we call a canonical symbolic rule. Since each matrix can have many equivalent circuit representations, we discuss below how Qsymb enumerates only one representation for each canonical form.

The theorem's premise that each gate has a finite inverse implementation in the gate set holds for gate sets that arise in practice, including native gate sets for real hardware such as IBM [1], IonQ [14],

Rigetti, and others, as well as standard textbook gate sets such as Nam [25] and Clifford+T [11]. This theorem allows Qsymb to enumerate only canonical symbolic rules, where $S$ appears at the two ends of the rule and on opposite sides (i.e., $L; S$ on the lhs and $S; R$ on the rhs). This significantly reduces the symbolic-rule search space. After generating a canonical symbolic rule represented as $L; S = S; R$ with symbolic variable $S$, we can derive rules of the form $G_1; S; G_2 = G_1'; S; G_2'$ by left-appending subcircuits that implement the unitary matrix $A$ and right-appending subcircuits that implement the unitary matrix $B$ to both sides, such that $G_1 = AL$ and $G_2 = RB$.

**Example 5.2.** The RZ-merging symbolic rule: $rz(\gamma)\ q0; S; rz(\theta)\ q1 = S; rz(\gamma + \theta)\ q1$ has the canonical form:

$$rz(\gamma)\ q0; S = S; rz(\gamma)\ q1$$

We can derive the RZ-merging symbolic rule by right-appending $rz(\theta)\ q1$ to both sides of the canonical form and then using a simple concrete rule in Fig. 1(c) to merge two RZ gates on the right-hand side.

*5.3.2* ***Enumerate Candidates.*** Even though we have reduced the search space of symbolic rules to canonical form, it is still intractable to enumerate all possible candidates for $L, R$. The representation of a canonical form is not unique because circuits $L$ and $R$ can be represented by many equivalent circuits in an equivalence class. Qsymb further reduces the search space by enumerating one concrete circuit to represent $L$ or $R$ from each equivalence class generated during concrete-rule synthesis and appending a symbolic variable $S$ to it. Thus, for each canonical form, we enumerate only one representation. The intuition is that, while concrete rules already capture the equivalences between concrete circuits, we need to select only one representative from each equivalence class to serve as the concrete part of a canonical symbolic rule.

*5.3.3* ***Group Symbolic Terms.*** For concrete terms, we can efficiently group circuits approximately using PIF. However, for symbolic terms, because of the infinite number of possible instantiations of symbolic gates, it is hard to group symbolic terms into equivalence classes. Qsymb groups symbolic terms with a variable $S$ using necessary conditions for the existence of a solution. If the concrete components $L$ and $R$ are placed in the same group, there may exist an instantiation of $S$ satisfying $L; S = S; R$; if they are placed in different groups, no such instantiation exists.

**Lemma 5.3.** *Given unitary matrices $L_m, R_m$, there exists a unitary matrix $S$ such that $SL_m = R_mS$ if and only if $L_m$ and $R_m$ have the same eigenvalues, and each eigenvalue appears the same number of times (i.e., has the same multiplicity) in both matrices.*

Proof. If there exists a unitary matrix $S$ such that $SL_m = R_mS$, then multiplying both sides on the right by $S^\dagger$ gives $SL_mS^\dagger = R_mSS^\dagger = R_m$. Hence, $L_m$ and $R_m$ are unitarily similar matrices, so they have the same set of eigenvalues with the same multiplicities.

Conversely, if $L_m$ and $R_m$ have the same set of eigenvalues with the same multiplicities, then by the spectral theorem, there exist unitary matrices $U$ and $V$ such that $L_m = UDU^\dagger$ and $R_m = VDV^\dagger$, where $D$ is a diagonal matrix whose diagonal entries are the eigenvalues of $L_m$ and $R_m$. Let $S = VU^\dagger$. Then

$$SL_m = VU^\dagger UDU^\dagger = VDU^\dagger = VDV^\dagger VU^\dagger = R_mS.$$

□

**Lemma 5.4.** *Given unitary matrices $L_m$ and $R_m$, if there exists a unitary matrix $S$ such that $SL_m = R_mS$, then $L_m$ and $R_m$ have the same trace.*

Proof. The trace of a matrix is the sum of its eigenvalues. Thus, if $L$ and $R$ have the same set of eigenvalues with the same multiplicities, they must have the same trace. □

With the above two lemmas, QSYMB can group symbolic terms into classes, each of which contains symbolic terms that have the same trace and the same set of eigenvalues with the same multiplicities. This significantly reduces the search space of symbolic terms to be considered. Because computing traces is relatively efficient, we first group terms by their traces and then further test each term's concrete eigenvalues within each trace group. Because identical traces are a necessary but insufficient condition, they are used as a filter to eliminate a large number of non-existent symbolic rules early, before computing the eigenvalues of $L$ and $R$. For $L$ and $R$ that have symbolic variables such as angles, we again utilize probabilistic grouping by sampling concrete parameters to eliminate many non-existent symbolic rules whose lhs and rhs do not have equal traces. After that, for each rule in the group, we check whether the eigenvalues are symbolically equivalent with an SMT solver.

*5.3.4 Solving Symbolic Matrix.* Once we have grouped symbolic terms into classes, QSYMB can enumerate each circuit pair $(L, R)$ and construct solutions to the equation $[\![S]\!][\![L]\!] = [\![R]\!][\![S]\!]$. Solving this equation is equivalent to solving a set of linear equations and finding the nullspace of the matrix $K([\![R]\!], [\![L]\!]) = I_n \otimes [\![R]\!] - [\![L]\!]^{\mathsf{T}} \otimes I_n$, if $[\![L]\!] \in \mathbb{C}^{n\times n}$ and $[\![R]\!] \in \mathbb{C}^{n\times n}$, where $I_n$ is the identity matrix of size $n$.

**Example 5.5.** Given a canonical symbolic rule $cx\ q0\ q1; S = S; cx\ q0\ q1$, QSYMB tries to solve for $S$ such that $[\![S]\!][\![CX]\!] = [\![CX]\!][\![S]\!]$. Solving the equation gives a list of basis matrices $\{B_1, ..., B_k\}$ that correspond to the solution space of $S$. After assigning a coefficient variable $c_1..c_k$ to each basis matrix, the derived symbolic matrix is equivalent, modulo variable renaming, to the symbolic matrix in Fig. 7.

### 5.4 Synthesis Algorithm

Algorithm 2 shows the algorithm QSYMB uses to synthesize symbolic rules. Line 2 shows that QSYMB first collects representative terms from each equivalence class produced by concrete-rule synthesis. ECs are computed by running equality saturation on the final concrete rule set $L$ and extracting a representative from each e-node. Line 3 groups the circuit terms $T$ by their concrete traces, with the symbolic angles in $L, R$ evaluated using $n$ sampled concrete values. This step eliminates many invalid candidates early. Lines 4–14 show that, for each grouped class $C$, QSYMB enumerates each pair of terms $(L, R)$ in $C$. Line 6 solves $[\![S]\!][\![L]\!] = [\![R]\!][\![S]\!]$ as a linear system. The result is a set of basis matrices whose linear combinations form the solution space of $S$. In Line 8, we validate the eigenvalues of $L$ and $R$ symbolically to determine whether a unitary solution for $S$ exists. The grouping steps significantly reduce the number of systems of linear equations to be solved and eliminate a large number of non-existent symbolic rules early, avoiding the need to solve $S$'s unitary constraints directly; these constraints form a list of non-linear equations.
**Correctness** By Lemmas 5.3 and 5.4, for each pair of terms $L, R$, there exists a unitary matrix $S$ such that $S[\![L]\!] = [\![R]\!]S$ iff $eigenvalues(L) = eigenvalues(R)$. For $L, R$ that have symbolic parameters such as angles, we first sample concrete angles and compute each circuit's concrete trace and then validate the equivalence of the eigenvalues symbolically in Line 8 using the Z3 SMT solver [8] if $B$ is nonempty. Thus, all $B$ derived by solving $L; S = S; R$ have unitary solutions to the equations.
**Completeness** Because we take one representative from each equivalence class generated by Algorithm 1, and Algorithm 1 generates an $(n, q)$-complete non-derivable concrete rule set, the terms $(L, R)$ are canonicalized to representatives, and Algorithm 2 generates all canonical symbolic rules within qubit bound $q$ and with concrete parts within bound $n$. It is not necessary to include all terms in an equivalence class because, once we have a canonical form represented as $L; S = S; R$, applications of concrete rules can always replace $L$ and $R$ with any of their previously generated

**Algorithm 2** Symbolic Rule Synthesis

```
1: function SynthesizeSymb(ECs)
2:     T ← { Representative(EC) | for each equivalence class EC ∈ ECs}
3:     CC ← group T by (concrete_trace(T))                ▷ CC is a set of equivalence classes
4:     for each class C ∈ CC do
5:         for each pair of terms (L, R) in C do
6:             B ← SolveIntertwiner(L, R)                 ▷ Solve [[S]][[L]] = [[R]][[S]]
7:             if B ≠ {} then
8:                 if eigenvalues(L) = eigenvalues(R) then   ▷ validate the eigenvalues
9:                     SymbolicRules ← SymbolicRules ∪ {(L, R, B)}
10:                end if
11:            end if
12:        end for
13:    end for
14: end function
```

equivalent terms. More intuitively, the concrete rules have already captured the equivalence relations between concrete terms.

**Lemma 5.6.** *For any symbolic rule $\phi$ and its canonical form represented as $L; S = S; R$, if $L$ and $R$ are within size $n$ and qubit bound $q$, then it can be derived from the canonical symbolic rules and concrete rules generated by Algorithms 2 and 1.*

Proof. It follows from Theorem 5.1 that $\phi$ can be derived from the canonical rule. □

**Non-derivability** We also show that each generated canonical symbolic rule is non-derivable from the other generated canonical symbolic rules.

**Definition 5.7.** (Symbolic Matrix Satisfiability) Given a Symbolic Matrix $S$ with variables $v_1, v_2, ...v_n \in \mathbb{C}$, we denote $M \models S$ if there exists a model $M$ that assigns complex numbers or complex variables to $v_1, v_2, ...v_n$.

A model $M$ can be viewed as a matrix containing concrete numbers and/or symbolic parameters. The matrix semantics of a quantum circuit is a unitary matrix that may also contain symbolic parameters (e.g., angles), which can be checked against $S$. If it is a model of $S$, then it satisfies the constraints of $S$.

**Definition 5.8.** For any two symbolic matrices $S, S'$, we denote $S \subseteq S'$ if $M \models S$ implies $M \models S'$ for any model M.

**Lemma 5.9.** *Given a unitary matrix $L$ and a matrix $S$ with symbolic parameters, there exists a unique unitary matrix $R$, up to variable renaming, such that $SL = RS$.*

*Remark* 5.10. The symmetric version also holds for a unique $L$ such that $SL = RS$, given $S$ and $R$. The lemma implies that the two unitary symbolic matrices $L$ and $S$ determine the $R$ such that $SL = RS$. The lemma is later used to prove the following theorem.

**Theorem 5.11.** *Any canonical symbolic rule $R_s$ is not derivable from the other generated canonical symbolic rules.*

These results imply that we generate a compact symbolic-rule set that represents a large family of circuit equivalences, because symbolic variables can represent infinitely many unitary matrices.

### 5.5 Checking Constraints

Once we have a valid symbolic rule $R$, we match a subcircuit by syntactically matching the lhs pattern of the symbolic rule, producing a substitution that maps the variable $S$ to a concrete subcircuit $C$, $\delta = \{S : C\}$. The concrete part of the candidate must match the rule's concrete part, while the symbolic part $S$ can match any subcircuit. After matching, we check whether $\delta(S)$ satisfies the constraint on $S$ by testing whether the unitary matrix $M$ of $C$ is a model of symbolic matrix $S$, i.e., $M \models S$. QSYMB checks whether each entry of $M$ can be assigned consistently to variables in $S$ (the same variable in $S$ must map to the same value in $M$). If such an assignment exists, the constraint is satisfied. Since $M$ is already unitary (from the matrix semantics of the matched subcircuit), we do not re-check unitarity constraints. QSYMB avoids directly solving the non-linear equations in subsection 5.2 by grouping terms for which unitary solutions for $S$ exist during candidate enumeration and by solving linear equations to obtain a general symbolic matrix without explicit unitarity constraints. Intuitively, we do not need to solve for $S$ to check whether $M \models S$ for a given unitary matrix $M$.

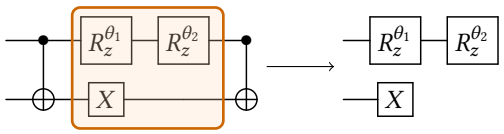



Fig. 9. Example of matching a symbolic rule on a CX-framed subcircuit.

**Example 5.12.** The symbolic rule

$$cx\ q0\ q1; S; cx\ q0\ q1 \longrightarrow S;$$

can match a circuit in Fig. 9; the symbolic part $S$ is matched to the subcircuit in the orange box. The concrete unitary matrix $M$ is:

$$M = (I \otimes [\![X]\!])([\![RZ(\theta_2)]\!][\![RZ(\theta_1)]\!] \otimes I) = \begin{bmatrix} 0 & e^{-i(\theta_1+\theta_2)/2} & 0 & 0 \\ e^{-i(\theta_1+\theta_2)/2} & 0 & 0 & 0 \\ 0 & 0 & 0 & e^{i(\theta_1+\theta_2)/2} \\ 0 & 0 & e^{i(\theta_1+\theta_2)/2} & 0 \end{bmatrix}$$

It is straightforward to check that there exists an assignment of complex numbers to $S$ in Fig. 7 such that $M \models S$:

$$a = 0, b = e^{-i(\theta_1+\theta_2)/2}, c = 0, d = e^{-i(\theta_1+\theta_2)/2}, m = 0, f = 0, g = 0, h = 0, k = e^{i(\theta_1+\theta_2)/2}, j = 0$$

Thus, the constraint is satisfied, and we can apply the symbolic rule to optimize the matched subcircuit. QSYMB also supports matching a $k$-qubit symbolic matrix $S$ ($k < n$) with an $n$-qubit subcircuit $C$. For example, if $S$ is a $4 \times 4$ matrix that constrains only qubits $x_1, x_2$, and the matched subcircuit contains $n > 2$ qubits $x_1, x_2, \ldots, x_n$, then QSYMB checks that $C$ transforms $x_1, x_2$ correctly regardless of other qubits' states. In addition, QSYMB checks that the remaining qubits yield consistent results for $L; C$ and $C; R$, since $S$ constrains only $x_1, x_2$. If interactions from $x_1, x_2$ change other qubits in a way that makes $L; C$ and $C; R$ inconsistent, the symbolic rule is not applicable.

### 5.6 Anchoring Canonical Rules

Most canonical symbolic rules are size-preserving and are useful when we want to move $L$ and $R$ around the symbolic part $S$. The $L$ part can move to the right side of $S$ and become $R$. However, size-reducing rules can sometimes transform circuits directly, speeding up the search process, rather

$$\textsc{Init}\quad \frac{}{\Delta := \Delta_{init}}$$

$$\textsc{Append-1}\quad \frac{L;S;R = B_s \in \Delta \qquad \mathit{suffix}\ L\ L_c \qquad L_c \to R_c \in \Delta}{\Delta := \{L_c; s; R = \mathit{dropBack}(L_c, L); B_s\} \cup \Delta}$$

$$\textsc{Append-2}\quad \frac{A_s = L;S;R \in \Delta \qquad \mathit{prefix}\ R\ L_c \qquad L_c \to R_c \in \Delta}{\Delta := \{A_s; \mathit{dropFront}(L_c, R) = L;S;L_c\} \cup \Delta}$$

$$\textsc{Append-3}\quad \frac{A_s = L;S;R \in \Delta \qquad \mathit{suffix}\ L\ L_c \qquad L_c \to R_c \in \Delta}{\Delta := \{\mathit{dropBack}(L_c, L); A_s = L_c; S; R\} \cup \Delta}$$

$$\textsc{Append-4}\quad \frac{L;S;R = B_s \in \Delta \qquad \mathit{prefix}\ R\ L_c \qquad L_c \to R_c \in \Delta}{\Delta := \{L;S;L_c = B_s; \mathit{dropFront}(L_c, R)\} \cup \Delta}$$

Fig. 10. Inference rules for anchoring canonical rules

than first transforming the circuit using a size-preserving rule and then applying a size-reducing rule. We can do this by appending prefixes and suffixes to the canonical rule. However, naively enumerating all possible suffixes and prefixes will cause a combinatorial explosion, which is why we derive canonical rules instead of all possible rules in the first place. Canonical symbolic rules serve as the core, compact generative basis from which Qsymb can construct rules. They can be viewed as templates that can be instantiated into multiple concrete or symbolic transformations by prefixing or suffixing the circuits. Yet, not all instantiations are meaningful. For example, left-appending or right-appending a circuit that implements the identity matrix $I$ to both sides produces a trivial symbolic rule.

To avoid generating redundant variants, Qsymb introduces a novel technique that selectively instantiates only those symbolic rules that are likely to "collaborate" with concrete rules to reduce gates by appending carefully chosen prefixes or suffixes to both sides. Fig. 10 shows the inference rules we use to derive anchored rules. $\Delta$ is the current rule context that stores all derived rules. $L, R, L_c, R_c$ are concrete subcircuits that contain no symbolic variables. $S$ is a symbolic subcircuit variable. $A_s$ and $B_s$ denote subcircuits that contain a symbolic variable. *prefix/suffix X Y* indicates whether $X$ is a proper prefix/suffix of $Y$. *dropfront/dropback X Y* indicates that, if $Y$ is a prefix/suffix of $X$, the operation drops $Y$ from the front/back of $X$ and yields the remainder. We assume that concrete rules $L_c \to R_c$ are size-decreasing. Rule Init initializes the rule set with all currently generated rules. Append-2 first checks whether the concrete part $R$ of the rhs of the canonical symbolic rule is a prefix of the lhs $L_c$ of any concrete rule. If so, it can extend the rhs of the symbolic rule by transforming $R$ into $L_c$ and extend the lhs of the symbolic rule by adding, to the right of $A_s$, the part that remains after removing the prefix $R$ from $L_c$. The other Append-x rules are similar.

**Lemma 5.13.** *$\Delta$ always maintains a set of valid rules.*

**Lemma 5.14.** *If $R \in \Delta \setminus \Delta_{init}$ is applied to a circuit $C$ and results in $C'$, then a concrete rule in $\Delta$ can subsequently be applied to $C'$. We define the full set of anchored rules as $\Delta \setminus \Delta_{init}$ together with the canonical symbolic rules from which no additional rules are generated through anchoring.*

**Example 5.15.** Consider the canonical symbolic rule from Example 5.2: $rz(\gamma)\ q_0;\ S = S;\ rz(\gamma)\ q_1$. Qsymb first observes that the concrete suffix on the right-hand side, $rz(\gamma)\ q_1$, matches the prefix of the lhs of the concrete rule $rz(\alpha)\ q_1;\ rz(\beta)\ q_1 \to rz(\alpha+\beta)\ q_1$. Note that matching the symbolic parameters creates a substitution $\delta = \{\alpha : \gamma\}$. Hence, Qsymb can right-append $\delta(rz(\beta)\ q_1) = rz(\beta)\ q_1$ to both sides of the canonical symbolic rule, producing the new symbolic rule:

$$rz(\gamma)\ q_0;\ S;\ rz(\beta)\ q_1 \to S;\ rz(\gamma)\ q_1;\ rz(\beta)\ q_1. \tag{5.1}$$

Although the new rule (5.1) only appends a suffix to both sides, it "enables" the application of a concrete rule in Fig. 1(c), which can transform (5.1) to a size-reducing rule:

$$rz(\gamma)\ q_0;\ S;\ rz(\beta)\ q_1\ \rightarrow\ S;\ rz(\gamma+\beta)\ q_1. \tag{5.2}$$

Only the symbolic rule (5.1) is retained as an anchored extension of the canonical rule, while the subsequent transformation (5.2) can result from applying existing concrete rules during runtime optimization.

## 6 Applying Rules

We demonstrate that our generated rule sets can be easily integrated using standard techniques such as equality saturation [40] and a search-based stochastic algorithm such as simulated annealing [17]. We also show in the next section that these simple techniques achieve good results. Our generated rule sets are small and compact, allowing for the composition of a large number of rules. We easily integrate our concrete rules into existing equality-saturation engines [41, 48]. However, because of the limitations of equality saturation, an engine can run for only a small number of iterations before e-node explosion, when too many circuit terms are stored in the e-graph. Although the generated concrete rules can be easily used by equality saturation to compose larger rules, the e-graph is rarely able to saturate because it quickly fills with e-nodes when optimizing large circuits. In our experience, equality saturation can run for only up to 15 iterations before the number of e-nodes explodes. For a larger circuit, it can run for only around 5–10 iterations. Thus, we combine our concrete rules using equality saturation and apply symbolic rules using a simple stochastic algorithm that randomly selects and applies a symbolic rule. If the application reduces the current cost, it is accepted. Otherwise, it may be accepted with a small probability. Our intuition is that concrete rules can achieve global optimization within a limited circuit window (by running a limited number of equality-saturation iterations) and can quickly optimize short-distance subcircuits to an optimal size, whereas symbolic rules can achieve long-distance optimization by matching long-distance patterns and can serve as an exploration mechanism after concrete rules can no longer be applied. After an anchored rule is applied, concrete rules in an e-graph can again yield progress.

Algorithm 3 shows the overall search-based optimization algorithm. The algorithm runs in a while loop until timeout. In each outer iteration, it runs 1 to $N$ iterations of equality saturation, alternating with the application of symbolic rules. In each symbolic step, it randomly selects a symbolic rule and matches it against the current search circuit. If a matched subcircuit is found, it applies the symbolic rule to obtain a new circuit. The `Match` method scans the circuit graph once per iteration (linear in gate count). $O(n^2)$ behavior arises only when many overlapping matches are attempted within the same window. However, with anchored symbolic rules, the added prefixes and suffixes limit a symbolic rule's matches to subcircuits that, after rewriting, enable a later size-reducing concrete rule. If the new circuit has a lower cost than the current circuit, it is accepted as the new search state. Otherwise, it may still be accepted with a small probability. It then builds an empty e-graph from the current circuit, saturates it with concrete rules for $k$ iterations, and extracts the lowest-cost circuit according to the cost model, which can be either the number of two-qubit gates or the total number of gates. After $N$ iterations, the algorithm starts over from $k = 1$. When timeout is reached, the best circuit encountered is returned. $T$ is a temperature hyperparameter that adjusts the acceptance probability [17]: larger $T$ yields higher acceptance probability. We allow users to specify a window with a lower bound and an optional upper bound on the sizes of subcircuits that $S$ can match, such that a symbolic rule matches $S$ to a subcircuit larger than $l$ and smaller than $u$, while concrete rules find equivalences in subcircuits smaller than $l$. The lower bound ensures that symbolic and concrete rules do not overlap in their roles. This setting also provides flexible control: a larger or unlimited window allows $S$ to match larger subcircuits and

**Algorithm 3** Circuit Optimizer

```
function Optimize(c, N, ConcreteRules, SymbolicRules, T)
    currentCircuit ← c
    bestCircuit ← c
    while not Timeout do
        for k in 1 to N do
            rule ← RandomSelect(SymbolicRules)
            matchedSubcircuit ← Match(currentCircuit, rule.lhs)
            if matchedSubcircuit ≠ None then
                newCircuit ← ApplyRule(currentCircuit, rule, matchedSubcircuit)
                if Cost(newCircuit) < Cost(currentCircuit) then
                    currentCircuit ← newCircuit
                else
                    p ← exp(-(Cost(newCircuit)-Cost(currentCircuit))/T)
                    if Random(0,1) < p then
                        currentCircuit ← newCircuit
                    end if
                end if
            end if
            if Cost(currentCircuit) < Cost(bestCircuit) then
                bestCircuit ← currentCircuit
            end if
            eGraph ← EmptyEGraph(currentCircuit)
            eGraph ← eGraph.saturate(ConcreteRules, k)
            currentCircuit ← ExtractBest(eGraph, CostModel)
            if Cost(currentCircuit) < Cost(bestCircuit) then
                bestCircuit ← currentCircuit
            end if
        end for
    end while
    return bestCircuit
end function
```

prioritize broader transformations, while a smaller window focuses on smaller subcircuits so that each match runs faster and more rules can be explored. This does not mean that the system cannot optimize circuits larger than $u$: such optimizations can be performed through multiple symbolic rewrites.

## 7 Implementation and Evaluation

We implemented Qsymb in approximately 3.2k lines of Java and 1.0k lines of Python. The synthesis engine (including both concrete-rule and symbolic-rule synthesis) is implemented in Java. Our concrete rule term enumerator is built on Queso's enumerator [42]. The constraint-checking module is implemented in Python using SymPy [23] to model and solve linear systems over symbolic matrices. For equality saturation, we use the open-source e-graph library egglog [48]. In this section, we evaluate Qsymb and answer the following research questions.

- (1) How do Qsymb's rules compare to rewrite rules generated by other quantum synthesis engines?
- (2) How do the formalized techniques help speed up the process of symbolic rule generation?
- (3) How does Qsymb, when used with simple optimization techniques, compare to existing state-of-the-art quantum circuit optimizers?
- (4) What effect do symbolic rules have on optimization? What is the difference among using only concrete rules, using canonical symbolic rules, and using anchored rules?

***Experimental Setup***. All experiments were run on an AMD Ryzen 9 9950X (16 cores) with 128GB RAM, running Ubuntu 22.04.

### 7.1 Rewrite Rule Synthesis

Table 1. Concrete rule-set size (number of rules) generated by Qsymb that fully derive prior rules

| Gateset | Gates | #Rules (Qsymb) | #Rules (Prior) | $n$ | Time |
|---|---|---|---|---|---|
| ibm-eagle | cx, rz x, sx | **179** | 4615 (Queso) | 5 | 770s |
| nam | x h rz cx | **194** | 8816 (Quartz, Queso) | 5 | 341s |
| rigetti | rx1 rx2 rx3 rz cz | **46** | 8773 (Quartz, Queso) | 5 | 484s |
| ion | rx ry rz rxx | **180** | 11777 (Queso) | 3 | 237s |

***Size Comparison.*** We evaluate the concrete rules generated by Qsymb on four gate sets: (1) IBM-Eagle for IBM devices, (2) Rigetti's gate set, (3) an ion-trap gate set [14], and (4) the gate set of Nam et al. [25]. The IBM and Rigetti gate sets target superconducting devices, which currently constitute the largest physically realized quantum hardware. We conduct a size-comparison study to show that our concrete rule set (rule-set size = number of rules) remains compact while still deriving rules from prior work. We collect the gate-set grammars used by previous systems [42, 44] so that the generated Qsymb rules can cover their concrete rule spaces for a fair comparison. Table 1 reports both rule-set size (number of rules) and the maximum rule-length bound used during synthesis. For the IBM-Eagle, Nam, and Rigetti gate sets, we use a maximum rule length of 5, whereas for the ion gate set, we use a rule length of 3. Quartz does not report rules for the IBM-Eagle and ion gate sets. We choose synthesis bounds that allow synthesis to finish within 15 minutes and show optimization effectiveness in the next subsection.

*Results.* Across all four gate sets, Qsymb generates concrete rule sets that are 26x–191x smaller while still deriving all rules generated by Queso and Quartz.

***Symbolic Rules.*** Because Qsymb synthesizes symbolic rules that differ from prior approaches, we compare end-to-end optimization performance against tools that use monomial symbolic rules in the next subsection. Here, we evaluate how our formalized techniques accelerate symbolic-rule synthesis and filter large numbers of invalid candidates. To measure the effect of property-based grouping, we compare Qsymb with grouping enabled and disabled. Since a symbolic gate can match arbitrarily long subcircuits, in practice we generate only canonical symbolic rules with short concrete parts, then use anchoring to construct longer, optimization-effective rules. In practice, we use a canonical symbolic rule size of 3 for the IBM, Nam, and Rigetti gate sets, and 2 for the ion gate set; all are synthesized within 5 minutes. Table 3 reports the number of canonical symbolic rules and synthesis time with and without property-based grouping. We then anchor canonical rules to generate longer symbolic rules. Table 2 reports the number of anchored symbolic rules and anchoring time.

Table 2. Anchoring cost of symbolic rules

| **Gateset** | **#Rules** | **Time (s)** | **size** |
|---|---|---|---|
| ibm-eagle | 436 | 0.6 | 5 |
| nam | 484 | 1.1 | 5 |
| rigetti | 97 | 0.5 | 5 |
| ion | 607 | 0.8 | 3 |

*Results.* Property-based grouping significantly reduces the search space for canonical symbolic-rule generation, yielding an average speedup of 11.6x. Without grouping, synthesis is inefficient

Table 3. Synthesis of canonical symbolic rules with and without property grouping

| Gateset | #Rules | Cost w/ Grouping (s) | Cost w/o Grouping (s) | Speedup | size |
|---|---|---|---|---|---|
| ibm-eagle | 101 | **2.4** | 40.9 | **17.0x** | 3 |
| nam | 139 | **8.4** | 61.0 | **7.3x** | 3 |
| rigetti | 81 | **19.0** | 199.5 | **10.5x** | 3 |
| ion | 59 | **2.4** | 27.5 | **11.5x** | 2 |

because candidates from both sides of a rule must be enumerated and checked one by one. Table 2 shows that anchoring runs in seconds, because we anchor only canonical rules that can enable a later concrete rewrite and we generate a small number of concrete rules.

***Summary of Questions 1 and 2.*** The concrete rewrite rules generated by QSYMB are much smaller than those of prior quantum rewrite synthesizers, while still semantically covering their rules. Grouping by symbolic properties significantly accelerates symbolic-rule synthesis. Anchoring efficiently derives useful rules from the canonical rule set without exhaustive enumeration.

### 7.2 Optimization Performance

To evaluate symbolic-rule quality and application effectiveness, we conduct three experiments: (1) comparing full QSYMB (concrete + symbolic rules) against state-of-the-art optimizers; (2) comparing full QSYMB against two ablations (concrete-only and concrete + canonical symbolic rules); and (3) measuring the effect of anchoring on optimization performance.

***Metrics.*** We use two-qubit-gate reduction to evaluate QSYMB. We focus on two-qubit gates because, on NISQ hardware, they typically have much higher error rates than single-qubit gates. The metric is the percentage reduction in two-qubit-gate count relative to the original circuit.

$$\text{Percentage Reduction} = \frac{\text{Original Count} - \text{Optimized Count}}{\text{Original Count}} * 100\%$$

In addition, to evaluate practical quality, we compute circuit fidelity. The fidelity of a gate $g$ is $1 - error\_rate(g)$, and the fidelity of a circuit is $\prod_{i=1}^{n} fidelity(g_i)$. Gate error rates are taken from Qiskit's calibration data for IBM Washington [1].

***Benchmarks.*** We use benchmarks from prior rewrite-based optimizers [42, 44] and approximate optimization work [30]. Our benchmark suite contains 135 circuits spanning near-term and long-term algorithms, including QAOA [10], VQE [31], QPE [20], QFT [7], Grover [12], and Shor [34]. It also includes benchmark families used by Queso and Quartz. All input circuits are decomposed into the target gate set. The circuits contain an average of 1371.8 gates; the maximum is 17438, and the minimum is 15. They use an average of 13.3 qubits; the maximum is 36, and the minimum is 4.

***Experiment Setup.*** We set temperature $T = 10.0$ and run $N = 9$ equality-saturation iterations in Algorithm 3. Each target has a 60-minute timeout. We evaluate on the IBM-Eagle and Nam gate sets, using all generated concrete rules and anchored symbolic rules, with 3 trials per benchmark. We set the symbolic-match window to $[10, \infty]$ so that symbolic and concrete rules do not overlap in the same local region. Each data point is averaged over 3 trials. We compare against state-of-the-art rewrite-based tools: Quartz [44], Queso [42], Qiskit [1], and TKET [35]. We also compare with Guoq [43], which combines resynthesis and rewriting; for fairness, we compare against its rewrite engine. Guoq's rewrite engine is adapted from Queso and therefore uses a similar rule set. Among the compared tools, Queso and Guoq use monomial symbolic rules, Quartz uses concrete rules with symbolic parameters, and Qiskit/TKET use only concrete rules. We run Guoq, Quartz, and Queso using the settings and trial procedures reported in their papers. For Qiskit, we use optimization level 3. Table 4 summarizes each tool's approach and symbolic-rule support. Column "Used Symb"

denotes whether the tool uses symbolic rules. Column "Synthesize Symb" denotes whether the tool itself synthesizes symbolic rules.

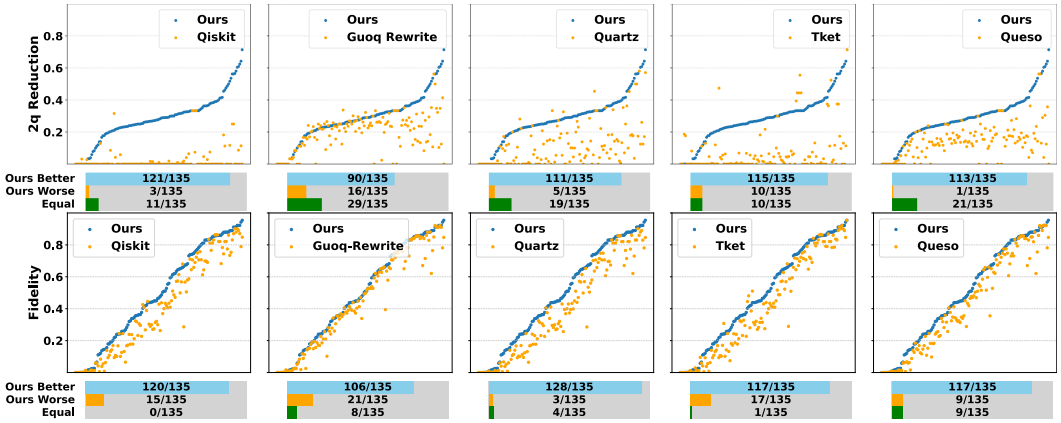



Fig. 11. Qsymb versus state-of-the-art optimizers on IBM-Eagle gateset

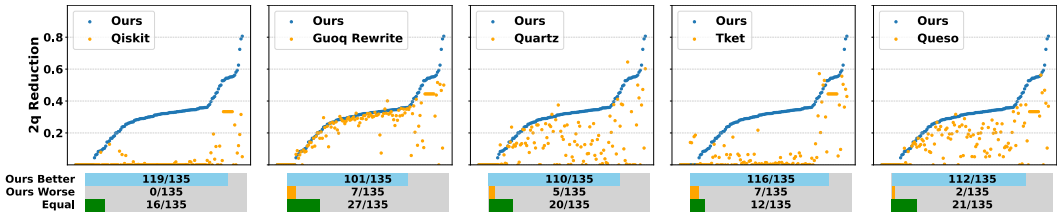



Fig. 12. Qsymb versus state-of-the-art optimizers on Nam gateset

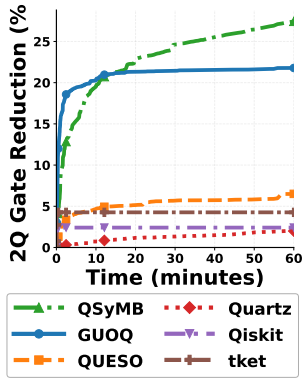
Table 4. Comparison of tools and their approaches

| Tool | Opt Approach | Used Symb | Synthesize Symb |
|---|---|---|---|
| Qsymb | Eqsat & Simulated Annealing | ✓ | full infinite space |
| Guoq-Rewrite [43] | Simulated Annealing | ✓ | × |
| Queso [42] | Beam Search | ✓ | monomial finite space |
| TKET [35] | fixed passes | × | × |
| Qiskit [1] | fixed passes | × | × |
| Quartz [44] | Beam Search | × | × |

***Results.*** Fig. 11 shows two-qubit reduction and fidelity against state-of-the-art optimizers on the IBM-Eagle gate set. The x-axis orders benchmarks by Qsymb's result (from smaller to larger reduction/fidelity). We observe that Qsymb outperforms other optimizers on most benchmarks. In two-qubit reduction, Qsymb strictly outperforms Qiskit [1], Guoq [43], Quartz [44], TKET [35], and Queso [42] on 90%, 67%, 82%, 85%, and 83% of benchmarks, respectively. For fidelity, Qsymb strictly outperforms Qiskit [1], Guoq [43], Quartz [44], TKET [35], and Queso [42] on 89%, 78%, 94%, 87%, and 87% of the benchmarks. Search-based tools (Quartz, Queso, and Guoq) continue optimization until timeout, while fixed-pass tools (Qiskit and TKET) terminate after their pass sequence completes. The average runtimes for Qiskit and TKET are 9.3s and 12.1s, respectively. Fig. 12 shows two-qubit reductions for the Nam gate set across optimizers. Since Nam does not have real-hardware fidelity data, we show only two-qubit reduction. Qsymb strictly outperforms

Qiskit [1], Guoq [43], Quartz [44], TKET [35], and Queso [42] on 88%, 74%, 81%, 86%, and 82.9% of the benchmarks, respectively.

**Runtime.** The figure on the right shows a runtime comparison of the average two-qubit reduction against state-of-the-art optimizers on the IBM-Eagle gate set, computed as $1 - \frac{mean(\sum \text{optimized\_2q})}{mean(\sum \text{original\_2q})}$. The results for the Nam gate set are similar to those for IBM-Eagle, so we omit them for brevity. As shown, Qsymb starts slower than Guoq in the first 10 minutes. However, as Guoq slows down, Qsymb continues to make progress, converges faster than Guoq, and reaches better final results. For fixed-sequence optimizers such as Qiskit and TKET, optimization terminates after a few seconds and the results then remain unchanged. Qsymb performs significantly better than Queso and Quartz, both by converging faster and by achieving better final results. Qsymb reaches a final average two-qubit reduction rate of 27.44% and average fidelity of 0.49 on IBM-Eagle; Qsymb reaches a final average two-qubit reduction rate of 29.95% on the Nam gate set.




*Summary of Question 3.* By combining compact concrete rules, expressive symbolic rules for long-distance matching, and anchoring, Qsymb significantly outperforms state-of-the-art rewrite-rule-based optimizers in both convergence rate and final results. The runtime results show that Qsymb continues to make progress after ten minutes, in contrast to prior work, demonstrating the effectiveness of our synthesized optimizations.

### 7.3 Ablation Study

We further evaluate the impact of symbolic rules by comparing full Qsymb against two ablations: Qsymb with only concrete rules, and Qsymb with concrete rules plus canonical symbolic rules. To isolate the effect of our concrete-rule synthesis algorithm, we also compare the concrete rules generated by Algorithm 1 against the same number of rules randomly selected from more than six thousand rules generated by Queso. We use the same temperature and iteration settings as in the previous experiment. Additional ablation studies are provided in Appendix C, including the effect of symbolic-rule window sizes and the impact of anchored rules versus canonical rules over time.

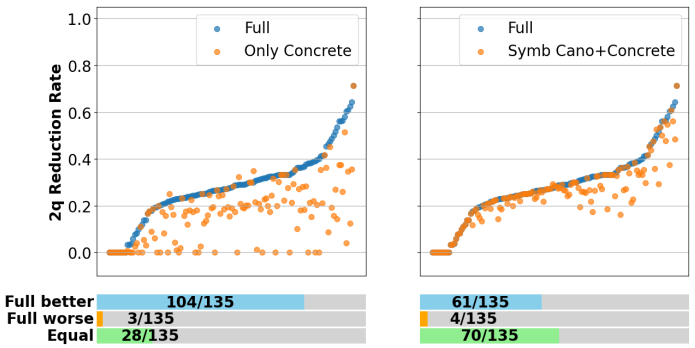



Fig. 13. Full Qsymb vs. Qsymb with only concrete rules vs. Qsymb with canonical symbolic rules

**Results.** Fig. 13 compares full Qsymb against two ablations: Qsymb with only concrete rules, and Qsymb with concrete rules plus only canonical symbolic rules. The results show that full Qsymb

performs as well as or better than the concrete-only variant in 97% of cases and is strictly better in 78% of cases. With anchored symbolic rules—formed by appending prefixes and suffixes to many canonical rules—full Qsymb further improves optimization, performing as well as or better than the canonical-symbolic-plus-concrete variant in 97% of cases, and strictly better in 31% of cases. In the best case, full Qsymb improves two-qubit-gate reduction by 21%. Fig. 14 compares Qsymb using the concrete rules generated by Algorithm 1 against an equal number of rules randomly selected from the Queso rule set [42] of over six thousand rules. For a fair comparison, we perform three trials. In each trial, we independently sample a new random rule set with the same number of rules. Fig. 14 reports the mean across the three trials. The randomly selected rules make almost no optimization progress because each set consists of 179 rules randomly selected from more than six thousand and is unlikely to derive the complete equivalences within the $(n, q)$ bounds.

*Summary of Question 4.* The results show that the combined approach achieves significantly better optimization than either component alone. Rule anchoring further improves optimization beyond what is achieved by canonical symbolic rules and concrete rules alone. The concrete-vs-random comparison also shows the importance of Algorithm 1: a compact, non-derivable, and $(n, q)$-complete concrete rule set is substantially more effective than an equally sized random subset of a much larger rule set.

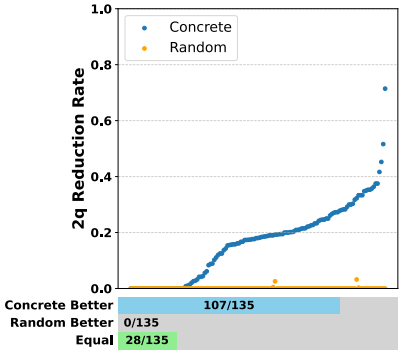



Fig. 14. Concrete Qsymb vs. the same number of randomly selected Queso rules

## 8 Discussion and Future Work

As in prior bounded rewrite-rule synthesizers [42, 44], the search space grows exponentially with qubit count and rule size. In practice, however, we choose rule sizes that allow optimization to converge quickly while still producing strong results. Thus, like Queso [42] and Quartz [44], Qsymb restricts synthesis to rules over at most 3 qubits. For example, Quartz reports that larger rule sizes ($n$) or rule qubit counts ($q$) do not necessarily yield better optimization results and can sometimes degrade performance. For Qsymb, most larger-size concrete rules are eliminated by the e-graph, so searching for larger-size rules that will mostly be eliminated is not an effective strategy. Our evaluation shows that simple optimizers using small-size non-derivable concrete rules together with e-graphs and symbolic rules can outperform existing work.

One interesting direction for advancing the optimization algorithm is to use machine-learning techniques to guide the application of symbolic rules. For example, using an LLM to analyze the structure of a circuit and apply symbolic rules in a guided way could potentially improve optimization performance. Additionally, integrating Qsymb's rewrite-rule engine with resynthesis techniques [30, 47] to further improve optimization is left for future work.

## 9 Related Work

*Synthesis of Quantum Optimizations.* Several works have explored the synthesis of quantum circuit optimizations. Queso [42] is a quantum circuit optimization rule synthesizer that generates rewrite rules and checks equivalence using efficient polynomial testing, but it does not perform SMT-based checking, which can lead to low-probability false positives. Queso only supports symbolic circuits with monomial gates, which represent only a small and finite fraction of the solution space captured by symbolic rules. Quartz [44] is another quantum circuit optimization rule synthesizer

that generates rewrite rules by enumerating all possible circuits within given size and qubit bounds, and it supports symbolic parameters such as angles. Qsymb follows Queso's approach to group circuits and uses equality saturation for concrete-rule synthesis, resulting in a much smaller rule set that is validated by an SMT solver while ensuring $(n, q)$-completeness. In addition, unlike these two prior works, Qsymb supports synthesis of symbolic rules whose symbolic variables represent infinite interpretations of subcircuits, and introduces a property-grouping method that enables the generation of such expressive symbolic rules.

*Quantum Circuit Optimizers.* Most quantum-circuit compilers use a fixed set of hand-crafted optimizations [1, 2, 5, 13, 25, 35, 36]. PyZX [19] represents a circuit as a ZX-diagram and applies the graphical rewrite rules of ZX calculus. Many works have developed verified compilers for quantum circuits; for example, VOQC [13] is a formally verified quantum circuit optimizer built on top of Coq. Giallar [38] utilized an automated verification approach based on SMT solvers to verify Qiskit optimization passes. Quarl [21] utilizes reinforcement learning to schedule the application of rules generated by Quartz. Some other works have used resynthesis techniques to optimize quantum circuits [30, 47]. The approach partitions circuits into several subcircuits and resynthesizes each subcircuit's unitary matrix to get a new optimized subcircuit. Many other optimization works [6, 15, 22, 24] adopt superoptimization, finding the optimal solution for small programs that are later used in peephole optimizers. Qsymb utilizes a symbolic unitary matrix to represent the constraint of a symbolic subcircuit in a rewrite rule.

*Equality Saturation.* With recent progress in fast, efficient equality-saturation engines [41, 48], many works have used equality saturation [40] to optimize programs in various domains, including classical programs [41, 48], math libraries [29], and tensor programs [46]. Qsymb's concrete rule inference is most related to general equality-saturation-based rule-inference engines such as Ruler [26] and its successor Enumo [28]. Qsymb integrates a domain-specific polynomial identity filter [42] to group quantum circuits and employs circuit-pruning techniques to reduce the search space. Quasar [45] is concurrent work that runs sequential and graph-level equality saturation in parallel on quantum circuits and uses equality saturation to filter concrete rules from prior work. Qsymb uses equality saturation to synthesize concrete rules bottom-up for any given gate set $G$ and synthesizes symbolic rewrite rules.

## 10 Acknowledgments

We thank the anonymous reviewers for valuable feedback that helped improve this paper. This work was supported in part by an NSF CAREER Award CCF-2239484, an Amazon Research Award, a VMware Systems Research Award, and an NSF grant CCF-2124080. Ronghui Gu is a co-founder of and has an equity interest in CertiK.

## 11 Data-Availability Statement

The source code, benchmarks, and scripts are open-sourced at https://github.com/VeriGu/QSymb. The artifact is also archived on Zenodo [33].

## References

[1] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel

Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. *Qiskit: An Open-source Framework for Quantum Computing*. doi:10.5281/zenodo.2562111

[2] Matthew Amy and Vlad Gheorghiu. 2020. staq—A full-stack quantum processing toolkit. *Quantum Science and Technology* 5, 3 (2020), 034016. doi:10.1088/2058-9565/ab9359

[3] Dolev Bluvstein, Simon J. Evered, Alexandra A. Geim, Sophie H. Li, Hengyun Zhou, Tom Manovitz, Sepehr Ebadi, Madelyn Cain, Marcin Kalinowski, Dominik Hangleiter, J. Pablo Bonilla Ataides, Nishad Maskara, Iris Cong, Xun Gao, Pedro Sales Rodriguez, Thomas Karolyshyn, Giulia Semeghini, Michael J. Gullans, Markus Greiner, Vladan Vuletić, and Mikhail D. Lukin. 2024. Logical quantum processor based on reconfigurable atom arrays. *Nature* 626, 7997 (2024), 58–65. doi:10.1038/s41586-023-06927-3

[4] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-540-74113-8

[5] Colin Campbell, Frederic T. Chong, Denny Dahl, Paige Frederick, Palash Goiporia, Pranav Gokhale, Benjamin Hall, Salahedeen Issa, Eric Jones, Stephanie Lee, Andrew Litteken, Victory Omole, David Owusu-Antwi, Michael A. Perlin, Rich Rines, Kaitlin N. Smith, Noah Goss, Akel Hashim, Ravi Naik, Ed Younis, Daniel Lobser, Christopher G. Yale, Benchen Huang, and Ji Liu. 2023. Superstaq: Deep Optimization of Quantum Programs. In *Proceedings of the* 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). IEEE Computer Society, Los Alamitos, CA, USA, 1020–1032. doi:10.1109/QCE57702.2023.00116

[6] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 313–326. doi:10.1145/3093337.3037754

[7] D. Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring. arXiv:quant-ph/0201067 [quant-ph] https://arxiv.org/abs/quant-ph/0201067

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[9] David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473. doi:10.1145/1066100.1066102

[10] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. arXiv:1411.4028 [quant-ph] https://arxiv.org/abs/1411.4028

[11] Daniel Gottesman. 1998. Theory of fault-tolerant quantum computation. *Physical Review A* 57, 1 (Jan. 1998), 127–137. doi:10.1103/physreva.57.127

[12] 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. doi:10.1145/237814.237866

[13] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for Quantum circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. doi:10.1145/3434318

[14] IonQ. 2022. IonQ Native Gates. Online documentation. Accessed: 2025-11-10. https://ionq.com/docs/getting-started-with-native-gates

[15] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

[16] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (2002), 304–314. doi:10.1145/543552.512566

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680. arXiv:https://www.science.org/doi/pdf/10.1126/science.220.4598.671 doi:10.1126/science.220.4598.671

[18] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. In *Proceedings of the 16th International Conference on Quantum Physics and Logic (QPL 2019) (Electronic Proceedings in Theoretical Computer Science, Vol. 318)*, Bob Coecke and Matthew Leifer (Eds.). Open Publishing Association, Chapman University, Orange, CA, USA, 229–241. doi:10.4204/EPTCS.318.14

[19] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), 229–241. doi:10.4204/eptcs.318.14

[20] A. Yu. Kitaev. 1995. Quantum measurements and the Abelian Stabilizer Problem. arXiv:quant-ph/9511026 [quant-ph] https://arxiv.org/abs/quant-ph/9511026

[21] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2024. Quarl: A Learning-Based Quantum Circuit Optimizer. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 114 (April 2024), 28 pages. doi:10.1145/3649831

[22] Zhengyang Liu, Stefan Mada, and John Regehr. 2024. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 326 (Oct. 2024), 25 pages. doi:10.1145/3689766

[23] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. doi:10.7717/peerj-cs.103

[24] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-based pruning for speeding up superoptimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 177 (Nov. 2020), 24 pages. doi:10.1145/3428245

[25] Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (2018), 1–12. doi:10.1038/s41534-018-0072-4

[26] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (Oct. 2021), 28 pages. doi:10.1145/3485496

[27] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA. AAI8011683.

[28] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 258 (Oct. 2023), 29 pages. doi:10.1145/3622834

[29] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. doi:10.1145/2813885.2737959

[30] Tirthak Patel, Ed Younis, Costin Iancu, Wibe de Jong, and Devesh Tiwari. 2022. QUEST: systematically approximating Quantum circuits for higher output fidelity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 514–528. doi:10.1145/3503222.3507739

[31] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5, 1 (July 2014), 4213. doi:10.1038/ncomms5213

[32] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. doi:10.22331/q-2018-08-06-79

[33] Wei Qiang and Ronghui Gu. 2026. Synthesis of Compact and Expressive Quantum-Circuit Optimizations (Artifact). doi:10.5281/zenodo.21508595

[34] P. W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (SFCS '94)*. IEEE Computer Society, USA, 124–134. doi:10.1109/SFCS.1994.365700

[35] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (nov 2020), 014003. doi:10.1088/2058-9565/ab8e92

[36] Runzhou Tao, Yunong Shi, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2019. CertiQ: a mostly-automated verification of a realistic quantum compiler. arXiv:1908.08963 https://arxiv.org/abs/1908.08963

[37] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T. Chong, and Ronghui Gu. 2021. Gleipnir: toward practical error analysis for Quantum programs. 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, 48–64. doi:10.1145/3453483.3454029

[38] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: push-button verification for the qiskit Quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 641–656. doi:10.1145/3519939.3523431

[39] Runzhou Tao, Hongzheng Zhu, Jason Nieh, Jianan Yao, and Ronghui Gu. 2025. Quantum Virtual Machines. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA,

411–428. https://www.usenix.org/conference/osdi25/presentation/tao

[40] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. *SIGPLAN Not.* 44, 1 (Jan. 2009), 264–276. doi:10.1145/1594834.1480915

[41] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

[42] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. *Proc. ACM Program. Lang.* 7, PLDI, Article 140 (June 2023), 25 pages. doi:10.1145/3591254

[43] Amanda Xu, Abtin Molavi, Swamit Tannu, and Aws Albarghouthi. 2025. Optimizing Quantum Circuits, Fast and Slow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 777–793. doi:10.1145/3669940.3707240

[44] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: superoptimization of Quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 625–640. doi:10.1145/3519939.3523433

[45] Ganxiang Yang, Paige Raun, Runzhou Tao, and Ronghui Gu. 2026. Equality Saturation for Quantum Circuit Optimization. *Proc. ACM Program. Lang.* 10, PLDI, Article 176 (June 2026), 26 pages. doi:10.1145/3808254

[46] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268. doi:10.48550/arXiv.2101.01332

[47] Ed Younis, Costin C. Iancu, Wim Lavrijsen, Marc Davis, and Ethan Smith. 2021. Berkeley Quantum Synthesis Toolkit (BQSKit) v1. [Computer Software] https://doi.org/10.11578/dc.20210603.2. doi:10.11578/dc.20210603.2

[48] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (June 2023), 25 pages. doi:10.1145/3591239

Received 2026-03-17; accepted 2026-08-06