Exact Inference for Quantum Circuits: A Testing Oracle for Quantum Software Stacks

Kanguk Lee
School of Computing
KAIST
Daejeon, South Korea
p51lee@kaist.ac.kr

Jaemin Hong
Institute of Information and Electronics
KAIST
Daejeon, South Korea
jaemin.hong@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

***Abstract*—Quantum software stacks (QSSs), which provide quantum circuit transformers and simulators, enable circuit transformations and the execution of circuits on classical computers. Despite their importance, they have not been effectively tested yet, leaving the correctness in question. The main obstacle to testing is the absence of a testing oracle, which checks the semantics-preservation of circuit transformations and the correctness of simulation results. While previous studies have employed differential and metamorphic testing to circumvent the necessity for an oracle, they have detected very few non-crash bugs. In this work, we address this gap by introducing QASMInfer, an *exact inference* system for quantum circuits, which computes the probability distribution of possible circuit outcomes. By supporting circuits written in OpenQASM, the de facto standard quantum assembly language used by most QSSs, QASMInfer acts as a unified testing oracle for multiple QSSs. Our design of QASMInfer achieves three key goals: (1) support for *dynamic circuits*, an important class of quantum circuits, (2) efficiency, and (3) reliability. For efficiency, we introduce two optimizations and an efficient matrix representation. For reliability, we prove *physical consistency*, ensuring that QASMInfer's inference results adhere to the physical principles of quantum computing. To simplify the proof, we introduce OpenQASMCore, a core language for OpenQASM, and perform exact inference for OpenQASM by desugaring it to OpenQASMCore. Our implementation and proof are fully mechanized in the Coq proof assistant. Testing six real-world QSSs using QASMInfer revealed 31 bugs, including 20 non-crash bugs, demonstrating QASMInfer's effectiveness as a testing oracle.**

***Index Terms*—quantum software stack, quantum circuit, testing oracle, exact inference**

I. Introduction

Quantum computing is attracting significant attention from academia and industry for its potential to outperform classical computing by enabling faster algorithms. A dominant model for quantum programming is the *quantum circuit*, a sequence of quantum operations (gates) applied to quantum bits (qubits). To manage the complexity of building these circuits, developers rely on *quantum software stacks* (QSSs) like Qiskit [1] and Cirq [2]. A typical QSS has several key components:

- A *programming interface*, such as a Python API, to construct quantum circuits as an internal representation.
- *Transformers* that optimize or modify circuits to ensure compatibility with specific quantum hardware while preserving their semantics.
- *Simulators* that execute quantum circuits on classical computers.
- *Hardware mapping* that maps its internal circuit representation onto quantum hardware for actual execution.

While transformers and simulators are significant since they are heavily relied on in quantum program development, their correctness is debatable. A recent study has reported that QSSs are buggy [3], yet testing has not been effectively conducted. The primary obstacle to testing is the absence of a testing oracle, which checks the semantics-preservation of circuit transformations and the correctness of simulation results.

Previous attempts to fill this gap have used existing testing techniques. QDiff [4] and QuteFuzz [5] employ differential testing by comparing the outcomes for the same circuit across different QSSs. MorphQ [6] applies metamorphic testing by comparing the results of feeding semantically equivalent circuits to the same QSS. Although these methods have identified several crash bugs—four by QDiff, 13 by QuteFuzz, and 13 by MorphQ—their effectiveness is limited: they have detected only a few non-crash bugs—four by QuteFuzz—which silently produce incorrect results.

In this work, we present QASMInfer, an *exact inference* system for quantum circuits to address the need for a testing oracle. Exact inference, a concept from probabilistic programming, stands for calculating the probability distribution of a random variable. QASMInfer computes the probability distribution of the possible output states of a quantum circuit, whose behavior is nondeterministic due to the nature of quantum mechanics. By determining the behavior of quantum circuits, QASMInfer serves as a testing oracle for circuit transformers and simulators. Testing a transformer involves comparing the probability distributions of circuits before and after the transformation. Testing a simulator involves running it on a certain circuit multiple times and checking whether the output states adhere to the circuit's probability distribution using a statistical method.

QASMInfer specifically performs exact inference on circuits written in OpenQASM, the de facto standard quantum assembly language. It fully supports OpenQASM 2, widely adopted by current QSSs, and partially supports OpenQASM 3, the latest version, omitting features incompatible with OpenQASM 2, such as unbounded loops. Most of QSSs support conversion

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00203

between OpenQASM code and their circuit representations. Once we have circuits in OpenQASM, we can test multiple QSSs by converting the circuits into each QSS's representation and feeding them to transformers and simulators. This makes QASMInfer a unified testing oracle applicable across various QSSs.

Our design of QASMInfer achieves three key goals. First, it supports *dynamic circuits* [7]. For *static circuits*, which never change control flow based on measurements, exact inference is straightforward, as probability distributions are derived by the addition and multiplication of the matrices corresponding to circuits' gates. However, interesting applications [8] involve dynamic circuits, where measurements alter control flow and probability distributions cannot be calculated by simple matrix operations. As many QSSs support dynamic circuits, QASMInfer must support them to achieve high testing coverage. To enable exact inference for dynamic circuits, we use *branching* [9], which creates multiple matrices corresponding to possible outcomes at each measurement.

Second, QASMInfer is efficient. Improving efficiency not only reduces testing time but also enhances effectiveness. Exact inference for quantum circuits on classical computers incurs time and space complexities growing exponentially with the number of qubits and measurements. This implies that, for example, reducing memory consumption by half enables the exact inference on a circuit with an additional qubit within the same memory budget, thus diversifying testing inputs and improving the effectiveness of testing. To achieve efficient inference, we introduce two optimizations that reduce the number of matrices created during branching and propose an efficient representation of the matrices used for exact inference.

Third, QASMInfer serves as a reliable testing oracle by ensuring *physical consistency*, i.e., that the computed probability distributions align with the physical principles governing the behavior of quantum circuits. These principles are modeled with matrices by physicists, and the model's constraints are stated through five postulates [10], to which we prove QASMInfer adheres. To simplify the proof, we introduce OpenQASMCore, a core language for OpenQASM. QASMInfer desugars OpenQASM into OpenQASMCore and performs exact inference on this core language. Our proof establishes the physical consistency of the exact inference for OpenQASMCore, which implies the same for OpenQASM. Both our implementation and proof are fully mechanized in the Coq proof assistant and available online [11].

To assess the effectiveness of QASMInfer as a testing oracle, we tested six real-world QSSs: Qiskit, Cirq, TKET [12], staq [13], PyQuil [14], and Braket [15]. Qiskit, Cirq, and TKET provide both transformers and simulators, but TKET's simulators are proprietary, so we tested only the transformers; staq provides only transformers, and PyQuil and Braket provide only simulators. For testing inputs, we generated 50,000 random OpenQASM programs containing code patterns that potentially trigger optimizations during transformer testing. Our testing found no bugs in QASMInfer, advocating its reliability, and identified 31 bugs across the QSSs—11 in Qiskit, 10 in Cirq, seven in TKET, two in staq, and one in Braket. Of these, 20 were non-crash bugs, underscoring the testing's effectiveness in comparison to the previous approaches. We reported these bugs, and 27 have been confirmed by the developers so far, all being classified as previously unknown bugs except for four.

$$
\begin{array}{lll}
\text{Bit value} & b \in \{0,1\} & \text{Real number} \quad r \in \mathbb{R} \\
\text{Classical bit} & c \in \mathbb{N} & \text{Qubit} \quad q \in \mathbb{N} \\
\text{Statement} & s ::= \mathsf{nop} \mid s; s \mid \mathsf{if}\ c \rightsquigarrow b\ \mathsf{then}\ s \mid \mathsf{U}(r,r,r)\ q & \\
& \quad \mid \mathsf{CX}\ q\ q \mid c := \mathsf{Measure}\ q \mid \mathsf{Reset}\ q &
\end{array}
$$

Fig. 1: Syntax of OpenQASMCore

Overall, our contributions are as follows:

- We introduce OpenQASMCore, a core language for OpenQASM (§II).
- We propose an exact inference algorithm for OpenQASMCore, with two optimizations and an efficient matrix representation, and prove its physical consistency (§III).
- We enable exact inference for OpenQASM by introducing desugaring from OpenQASM to OpenQASMCore (§IV).
- We show the effectiveness of our optimizations and matrix representation in reducing inference times and validate the utility of QASMInfer as a testing oracle by identifying 31 bugs in real-world QSSs (§V).

We also discuss related work (§VI) and conclude the paper (§VII).

II. Core Language: OpenQASMCore

In this section, we define OpenQASMCore's syntax (§II-A) and provide example circuits to introduce quantum computing concepts and clarify the exact inference process (§II-B).

A. Syntax

Fig. 1 defines the syntax of OpenQASMCore. The metavariables are: b (bit values 0 or 1), r (real numbers), c (classical bits) and q (qubits), where both c and q range over natural numbers. A program (i.e., circuit) uses a finite number of classical bits (n) and qubits (m), numbered from 0 to $n-1$ and 0 to $m-1$, respectively. The metavariable s ranges over statements, and a program consists of a single statement.

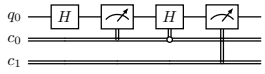
The language provides seven kinds of statements: three classical and four quantum-specific. Classical statements are nop (no-op), $s_1; s_2$ (sequential execution), and $\mathsf{if}\ c \rightsquigarrow b\ \mathsf{then}\ s$ (conditional on classical bit c). Quantum-specific statements are $\mathsf{U}(r_1, r_2, r_3)\ q$ (rotation gate), $\mathsf{CX}\ q_1\ q_2$ (controlled NOT gate), $c := \mathsf{Measure}\ q$ (measurement), and $\mathsf{Reset}\ q$ (resetting qubit to $|0\rangle$). Further details on quantum-specific statements are in §II-B.

B. Examples

We now provide example quantum circuits to introduce basic quantum computing concepts, as commonly discussed in the physics literature [16]. We first explain a single-qubit circuit using both state vectors (§II-B1) and density matrices

```
1 | U(π/2, 0, π) 0;
2 | 0 := Measure 0;
3 | if 0 ⇝ 0 then U(π/2, 0, π) 0;
4 | 1 := Measure 0
```

(a) OpenQASMCore



(b) Diagram

Fig. 2: An example single-qubit dynamic quantum circuit

(§II-B2). While state vectors offer an intuitive understanding, QASMInfer uses density matrices to enable optimizations (§III-B). Next, we extend these concepts to a multi-qubit circuit (§II-B3).

1) Single-qubit circuit (with state vector): A quantum bit, or *qubit*, exists in a *superposition* $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers satisfying $|\alpha|^2 + |\beta|^2 = 1$. Measurement of a qubit yields 0 or 1, with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. The *state vector* $|\psi\rangle$ is represented as:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \quad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \quad |\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix}.$$

Fig. 2a shows an example quantum circuit in OpenQASMCore, and Fig. 2b illustrates its diagram. It is a dynamic circuit, as control flow (line 3) depends on a prior measurement outcome (line 2). The circuit uses one qubit, q_0, and two classical bits, c_0 and c_1 with initial in states $|0\rangle$ and 0.

a) Line 1: We apply the $\mathsf{U}(\frac{\pi}{2}, 0, \pi)$ gate, i.e., the *Hadamard gate* (H), to qubit q_0, creating a superposition. This operation updates $|0\rangle$ through matrix multiplication:

$$H|0\rangle = \tfrac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \tfrac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}|1\rangle.$$

b) Line 2: We measure q_0 and assign the outcome to c_0, which results in two branches [9]: one where $c_0 = 0$ and q_0 collapses to $|0\rangle$ and another where $c_0 = 1$ and q_0 collapses to $|1\rangle$, each with probability $1/2$, each with probability $1/2$. We represent the distribution of the possible states by enumerating the branches, which we call *branch states*. Each branch state is a tuple of a state vector, classical bits, and the branch probability: $(\begin{pmatrix}1\\0\end{pmatrix}, [0,0], \frac{1}{2})$ and $(\begin{pmatrix}0\\1\end{pmatrix}, [1,0], \frac{1}{2})$.

c) Line 3: If $c_0 = 0$, the Hadamard gate is applied to q_0. In the first branch $(\begin{pmatrix}1\\0\end{pmatrix}, [0,0], \frac{1}{2})$, c_0 is 0, so the state becomes $(\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}, [0,0], \frac{1}{2})$. In the second branch $(\begin{pmatrix}0\\1\end{pmatrix}, [1,0], \frac{1}{2})$, c_0 is 1, so the state remains unchanged. Thus, the updated probability distribution is $(\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}, [0,0], \frac{1}{2})$ and $(\begin{pmatrix}0\\1\end{pmatrix}, [1,0], \frac{1}{2})$.

d) Line 4: We measure q_0 and assign the outcome to c_1. In the first branch, the measurement yields 0 or 1 with equal probability, splitting into two sub-branches: $(\begin{pmatrix}1\\0\end{pmatrix}, [0,0], \frac{1}{4})$ and $(\begin{pmatrix}0\\1\end{pmatrix}, [0,1], \frac{1}{4})$. In the second branch, the measurement always yields 1, updating c_1 to give $(\begin{pmatrix}0\\1\end{pmatrix}, [1,1], \frac{1}{2})$. Thus, we have $(\begin{pmatrix}1\\0\end{pmatrix}, [0,0], \frac{1}{4})$, $(\begin{pmatrix}0\\1\end{pmatrix}, [0,1], \frac{1}{4})$, and $(\begin{pmatrix}0\\1\end{pmatrix}, [1,1], \frac{1}{2})$.

2) Single-qubit circuit (with density matrix): A *density matrix* represents a qubit's state as a matrix, formed by a convex combination of outer products of state vectors. For example, a density matrix with a single outer product can be expressed as:

$$\rho = \begin{pmatrix}\alpha\\\beta\end{pmatrix}\begin{pmatrix}\alpha\\\beta\end{pmatrix}^\dagger = \begin{pmatrix}\alpha\\\beta\end{pmatrix}\begin{pmatrix}\alpha^* & \beta^*\end{pmatrix} = \begin{pmatrix}\alpha\alpha^* & \alpha\beta^*\\\beta\alpha^* & \beta\beta^*\end{pmatrix}$$

where α^* is the complex conjugate of α, and $M^\dagger$ is the conjugate transpose of a matrix M. The diagonal elements represent the probabilities of measurement outcomes, as $\alpha\alpha^* = |\alpha|^2$. We now explain the exact inference of Fig. 2's circuit using density matrices. The initial density matrix is $\begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}1&0\end{pmatrix} = \begin{pmatrix}1&0\\0&0\end{pmatrix}$.

a) Line 1: We apply the Hadamard gate to q_0, updating the density matrix via $U\rho U^\dagger$ where $U = \frac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix}$ (Hadamard gate) and ρ is the current density matrix:

$$\tfrac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot\begin{pmatrix}1&0\\0&0\end{pmatrix}\cdot\tfrac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix} = \tfrac{1}{2}\begin{pmatrix}1&1\\1&1\end{pmatrix}.$$

b) Line 2: We measure q_0 and assign the outcome to c_0. Each measurement outcome has a corresponding matrix: $\begin{pmatrix}1&0\\0&0\end{pmatrix}$ for 0 and $\begin{pmatrix}0&0\\0&1\end{pmatrix}$ for 1. The probability of an outcome is $P = \mathrm{tr}(\rho M)$, and the updated density matrix is $\frac{1}{P}M\rho M$, where M is the matrix for the outcome and ρ is the current density matrix. For outcome 0, we compute the probability

$$\mathrm{tr}(\tfrac{1}{2}\begin{pmatrix}1&1\\1&1\end{pmatrix}\begin{pmatrix}1&0\\0&0\end{pmatrix}) = \mathrm{tr}(\tfrac{1}{2}\begin{pmatrix}1&0\\1&0\end{pmatrix}) = \tfrac{1}{2},$$

and the updated density matrix becomes

$$\tfrac{1}{1/2}\cdot\begin{pmatrix}1&0\\0&0\end{pmatrix}\cdot\tfrac{1}{2}\begin{pmatrix}1&1\\1&1\end{pmatrix}\cdot\begin{pmatrix}1&0\\0&0\end{pmatrix} = \begin{pmatrix}1&0\\0&0\end{pmatrix}.$$

Similarly, for outcome 1, the probability is also $\frac{1}{2}$ and the updated density matrix is $\begin{pmatrix}0&0\\0&1\end{pmatrix}$. Thus, we get the probability distribution consisting of $(\begin{pmatrix}1&0\\0&0\end{pmatrix}, [0,0], \frac{1}{2})$ and $(\begin{pmatrix}0&0\\0&1\end{pmatrix}, [1,0], \frac{1}{2})$, which is consistent with the result obtained using the state vector: $\begin{pmatrix}1&0\\0&0\end{pmatrix} = \begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}1&0\end{pmatrix}$ and $\begin{pmatrix}0&0\\0&1\end{pmatrix} = \begin{pmatrix}0\\1\end{pmatrix}\begin{pmatrix}0&1\end{pmatrix}$.

In fact, we can avoid splitting branches to handle measurements by using a *mixed state*, which is a convex combination of density matrices. The two density matrices can be combined as: $\frac{1}{2}\begin{pmatrix}1&0\\0&0\end{pmatrix} + \frac{1}{2}\begin{pmatrix}0&0\\0&1\end{pmatrix} = \frac{1}{2}\begin{pmatrix}1&0\\0&1\end{pmatrix}$. While this may seem efficient, it is impractical for exact inference because mixed states do not represent classical bits. Representing classical bits within mixed states requires promoting them to qubits. Unfortunately, an n-qubit system is represented by a density matrix of size $2^n \times 2^n$. This causes matrix size to grow exponentially with base 4 in the number of classical bits.

For this reason, QASMInfer splits branches to handle measurements. However, when multiple density matrices share the same classical bits, mixed states can avoid branching without increasing the size of the density matrix. The optimizations in §III-B leverage this idea.

c) Lines 3–4: We end up with $(\begin{pmatrix}1&0\\0&0\end{pmatrix}, [0,0], \frac{1}{4})$, $(\begin{pmatrix}0&0\\0&1\end{pmatrix}, [0,1], \frac{1}{4})$, and $(\begin{pmatrix}0&0\\0&1\end{pmatrix}, [1,1], \frac{1}{2})$.

3) Multi-qubit circuit: A 2-qubit system with outcomes 00, 01, 10, or 11 and probabilities $|\alpha|^2$, $|\beta|^2$, $|\gamma|^2$, and $|\delta|^2$ is described by the following state vector and density matrix:

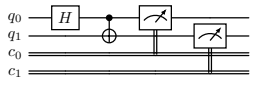
$$\begin{aligned}|\psi\rangle &= \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle\\ &= \alpha\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \gamma\begin{pmatrix}0\\0\\1\\0\end{pmatrix} + \delta\begin{pmatrix}0\\0\\0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\\\gamma\\\delta\end{pmatrix}\\ \rho &= \begin{pmatrix}\alpha\\\beta\\\gamma\\\delta\end{pmatrix}\begin{pmatrix}\alpha\\\beta\\\gamma\\\delta\end{pmatrix}^\dagger\end{aligned}$$

Fig. 3 shows a 2-qubit quantum circuit with qubits q_0 and q_1, and classical bits c_0 and c_1. Both qubits are initially set to $|0\rangle$, giving the initial state vector and density matrix:

$$|00\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \qquad \begin{pmatrix}1\\0\\0\\0\end{pmatrix}\begin{pmatrix}1\\0\\0\\0\end{pmatrix}^\dagger = \begin{pmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}.$$

```
1 | U(π/2, 0, π) 0;
2 | CX 0 1;
3 | 0 := Measure 0;
4 | 1 := Measure 1
```

(a) OpenQASMCore



(b) Diagram

Fig. 3: An example multi-qubit quantum circuit

a) Line 1: We apply the Hadamard gate to q_0. Its 2×2 matrix is extended to a 4×4 matrix by taking the tensor product with an identity matrix:

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & -1\end{pmatrix}.$$

Using this matrix, the updated density matrix is:

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & -1\end{pmatrix} \cdot \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix} \cdot \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & -1\end{pmatrix} = \frac{1}{2}\begin{pmatrix}1 & 0 & 1 & 0\\ 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0\\ 0 & 0 & 0 & 0\end{pmatrix}.$$

The diagonal elements indicate measurement outcomes of 00 or 10, each with a probability of $1/2$.

b) Line 2: We apply the CNOT gate to q_0 (*control*) and q_1 (*target*), which flips the target if the control is 1, creating *entanglement*. Its 4×4 matrix is $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$. Applying CNOT gate updates the density matrix as:

$$\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix} \cdot \frac{1}{2}\begin{pmatrix}1 & 0 & 1 & 0\\ 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0\\ 0 & 0 & 0 & 0\end{pmatrix} \cdot \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix} = \frac{1}{2}\begin{pmatrix}1 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 1 & 0 & 0 & 1\end{pmatrix}.$$

The diagonal elements indicate that the outcomes can be 00 or 11, each with probability $1/2$.

c) Line 3: We measure q_0 and assign the outcome to c_0. The measurement matrices for outcomes 0 and 1 from the measurement of q_0 are obtained using the tensor product:

$$\begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix} \qquad \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}.$$

As in the single-qubit case, we can compute the probability of outcome 0 and 1, resulting in the probability distribution consisting of $(\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix}, [0, 0], \frac{1}{2})$ and $(\begin{pmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}, [1, 0], \frac{1}{2})$.

d) Line 4: We measure q_1 and assign the outcome to c_1. The matrices for outcomes 0 and 1 are:

$$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0\end{pmatrix} \qquad \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \otimes \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Using these, we find that the outcome is always 0 in the first branch and 1 in the second branch. Therefore, we end up with $(\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix}, [0, 0], \frac{1}{2})$ and $(\begin{pmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}, [1, 1], \frac{1}{2})$.

III. Exact Inference and Its Physical Consistency

This section presents an exact inference algorithm for OpenQASMCore (§III-A), two optimization techniques (§III-B), an efficient matrix representation (§III-C), and the physical consistency proof of the exact inference process (§III-D). Our implementation and proof are fully mechanized in Coq [11].

A. Exact Inference

Fig. 4 shows the definitions related to the exact inference for OpenQASMCore. The metavariables are as follows: ρ (density matrices of size $2^n \times 2^n$), σ (classical states mapping bits to values), P (branch probabilities between 0 and 1), and Σ (branch states as triples of ρ, σ, and P). The functions *gate*, *prob*, and *msr* compute the updated density matrix after a gate, the probability of a measurement outcome, and the updated density matrix after a measurement, respectively. Each takes a density matrix and a gate/outcome matrix, requiring both to be of the same size. The function *qbts* returns the number of qubits in a given density matrix.

Density matrix $\rho \in \bigcup_{n\in\mathbb{N}} \mathcal{M}_{2^n,2^n}(\mathbb{C})$	$gate(\rho, U) = U\rho U^\dagger$
Classical state $\sigma \in \bigcup_{n\in\mathbb{N}} \mathbb{Z}_n \to \{0,1\}$	$prob(\rho, M) = \mathrm{tr}(\rho M)$
Probability $P \in \mathbb{R}_{[0,1]}$	$msr(\rho, M) = \frac{1}{prob(\rho,M)} M\rho M$
State $\Sigma ::= (\rho, \sigma, P)$	$qbts(\rho) = n$ if $\rho \in \mathcal{M}_{2^n,2^n}(\mathbb{C})$

Fig. 4: Definitions related to exact inference

Fig. 5 shows the rules defining the exact inference algorithm for OpenQASMCore in the form of $\boxed{\overline{\Sigma} \vdash s \Rightarrow \overline{\Sigma}}$. Following Vytiniotis [17] and Coughlin [18]'s notation, an overline represents zero or more repetitions of the enclosed material, with metavariables subscripted and separated by commas. Thus, $\overline{\Sigma} \vdash s \Rightarrow \overline{\Sigma'}$ denotes $\Sigma_1, \Sigma_2, \ldots, \Sigma_n \vdash s \Rightarrow \Sigma'_1, \Sigma'_2, \ldots, \Sigma'_m$, indicating that the branch states Σ_1, Σ_2, …, Σ_n are updated to Σ'_1, Σ'_2, …, Σ'_m after s. Most rules handle a single branch and are lifted to multiple branches by [Eval-Many], which applies s to each branch and collects the results.

The exact inference of classical statements are conventional, as defined by rules [Eval-Nop], [Eval-Seq], [Eval-If-False], and [Eval-If-True].

Rule [Eval-Rotate] defines the exact inference of a rotation gate, which updates only the density matrix. The matrix U_{n,q,r_1,r_2,r_3} is detailed in Fig. 6. It generalizes the 2×2 rotation matrix U_{r_1,r_2,r_3}, the action of $\mathsf{U}(r_1, r_2, r_3)$ on a single qubit, to an n-qubit system targeting qubit q. This is done via the function $gen_{n,q}$, which lifts a 2×2 matrix to a $2^n \times 2^n$ matrix acting on the q-th qubit (indexed from zero). Here, I_m represents the $2^m \times 2^m$ identity matrix.

Rule [Eval-CNot] defines the exact inference of the CNOT gate, which also updates only the density matrix. Fig. 7 defines CX_{n,q_1,q_2}, the matrix that flips the q_2-th qubit using the q_1-th qubit as control in an n-qubit system. X flips a single qubit, and $X_{n,q}$ flips the q-th qubit in an n-qubit system. The second case of CX_{n,q_1,q_2}'s definition implies $CX_{n,0,q} = (\begin{array}{c|c} I_{n-1} & 0 \\ \hline 0 & X_{n-1,q-1} \end{array})$, flipping the q-th qubit using the 0-th as control. The third case similarly defines $CX_{n,q,0}$, and the fourth case inductively define CX_{n,q_1,q_2} for nonzero q_1 and q_2, using $CX_{n,0,q}$ and $CX_{n,q,0}$ as base cases.

Rules [Eval-Meas], [Eval-Meas-0], and [Eval-Meas-1] define the exact inference of a measurement statement. Rule [Eval-Meas] splits the branch when both 0 and 1 are possible outcomes, while [Eval-Meas-0] and [Eval-Meas-1] handle cases with a single possible outcome. The matrices $Proj_{n,q,0}$ and $Proj_{n,q,1}$ are generalized from $Proj_0$ and $Proj_1$ using tensor products, as shown in Fig. 6.

Rule [Eval-Reset] defines the exact inference of a reset statement, collapsing a qubit to $|0\rangle$. To achieve this, we introduce a fresh classical bit for the measurement result. If

$$\boxed{\overline{\Sigma} \vdash s \Rightarrow \overline{\Sigma}} \qquad \frac{\overline{\Sigma \vdash s \Rightarrow \overline{\Sigma'}}}{\overline{\Sigma} \vdash s \Rightarrow \overline{\overline{\Sigma'}}} \text{ [Eval-Many]}$$

$$\Sigma \vdash \mathsf{nop} \Rightarrow \Sigma \text{ [Eval-Nop]} \qquad \frac{\Sigma \vdash s_1 \Rightarrow \overline{\Sigma'} \qquad \overline{\Sigma'} \vdash s_2 \Rightarrow \overline{\Sigma''}}{\Sigma \vdash s_1; s_2 \Rightarrow \overline{\Sigma''}} \text{ [Eval-Seq]}$$

$$\frac{\Sigma = (\rho, \sigma, P) \qquad \sigma(c) \neq b}{\Sigma \vdash \mathsf{if}\ c \rightsquigarrow b\ \mathsf{then}\ s \Rightarrow \Sigma} \text{ [Eval-If-False]}$$

$$\frac{\Sigma = (\rho, \sigma, P) \qquad \sigma(c) = b \qquad \Sigma \vdash s \Rightarrow \overline{\Sigma'}}{\Sigma \vdash \mathsf{if}\ c \rightsquigarrow b\ \mathsf{then}\ s \Rightarrow \overline{\Sigma'}} \text{ [Eval-If-True]}$$

$$\frac{qbts(\rho) = n}{(\rho, \sigma, P) \vdash \mathsf{U}(r_1, r_2, r_3)\ q \Rightarrow (gate(\rho, U_{n,q,r_1,r_2,r_3}), \sigma, P)} \text{ [Eval-Rotate]}$$

$$\frac{qbts(\rho) = n}{(\rho, \sigma, P) \vdash \mathsf{CX}\ q_1\ q_2 \Rightarrow (gate(\rho, CX_{n,q_1,q_2}), \sigma, P)} \text{ [Eval-Cnot]}$$

$$\frac{\begin{array}{c} qbts(\rho) = n \qquad prob(\rho, Proj_{n,q,0}) \neq 0 \qquad prob(\rho, Proj_{n,q,1}) \neq 0 \\ \Sigma_0 = (msr(\rho, Proj_{n,q,0}), \sigma[c \mapsto 0], P \times prob(\rho, Proj_{n,q,0})) \\ \Sigma_1 = (msr(\rho, Proj_{n,q,1}), \sigma[c \mapsto 1], P \times prob(\rho, Proj_{n,q,1})) \end{array}}{(\rho, \sigma, P) \vdash c := \mathsf{Measure}\ q \Rightarrow \Sigma_0, \Sigma_1} \text{ [Eval-Meas]}$$

$$\frac{\begin{array}{c} qbts(\rho) = n \qquad prob(\rho, Proj_{n,q,1}) = 0 \\ \Sigma_0 = (msr(\rho, Proj_{n,q,0}), \sigma[c \mapsto 0], P \times prob(\rho, Proj_{n,q,0})) \end{array}}{(\rho, \sigma, P) \vdash c := \mathsf{Measure}\ q \Rightarrow \Sigma_0} \text{ [Eval-Meas-0]}$$

$$\frac{\begin{array}{c} qbts(\rho) = n \qquad prob(\rho, Proj_{n,q,0}) = 0 \\ \Sigma_1 = (msr(\rho, Proj_{n,q,1}), \sigma[c \mapsto 1], P \times prob(\rho, Proj_{n,q,1})) \end{array}}{(\rho, \sigma, P) \vdash c := \mathsf{Measure}\ q \Rightarrow \Sigma_1} \text{ [Eval-Meas-1]}$$

$$\frac{\begin{array}{c} \Sigma = (\rho, \sigma, P) \qquad c_{reset} \notin \mathsf{dom}(\sigma) \\ \Sigma \vdash c_{reset} := \mathsf{Measure}\ q; \mathsf{if}\ c_{reset} \rightsquigarrow 1\ \mathsf{then}\ \mathsf{U}(\pi, 0, \pi)\ q \Rightarrow \overline{\Sigma'} \end{array}}{\Sigma \vdash \mathsf{Reset}\ q \Rightarrow \overline{\Sigma'}} \text{ [Eval-Reset]}$$

Fig. 5: Exact inference for OpenQASMCore

$$U_{r_1,r_2,r_3} = \begin{pmatrix} e^{-i(r_2+r_3)/2}\cos(r_1/2) & -e^{-i(r_2-r_3)/2}\sin(r_1/2) \\ e^{i(r_2-r_3)/2}\sin(r_1/2) & e^{i(r_2+r_3)/2}\cos(r_1/2) \end{pmatrix}$$

$$gen_{n,q}(M) = I_q \otimes M \otimes I_{n-q-1} \quad U_{n,q,r_1,r_2,r_3} = gen_{n,q}(U_{r_1,r_2,r_3})$$

$$Proj_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad Proj_{n,q,0} = gen_{n,q}(Proj_0)$$
$$Proj_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad Proj_{n,q,1} = gen_{n,q}(Proj_1)$$

Fig. 6: Generalization of 2×2 matrices

the measurement outcome is 0, no action is needed; if 1, we apply the $\mathsf{U}(\pi, 0, \pi)$ gate to flip it.

The exact inference of a program starts with a probability distribution consisting of a single state where every qubit is $|0\rangle$ and every bit is 0, with a probability of 1.

B. Optimizations

We introduce two optimizations for the exact inference algorithm: *branch unification* and *superoperator replacement*. These reduce branch numbers by leveraging the concept of mixed states. To avoid enlarging density matrices with classical bits, mixed states are used only when different density matrices have the same classical bits. Fig. 8 shows the corresponding inference rules for these optimizations.

a) Branch Unification: Rule [Eval-Unify] defines the branch unification optimization, applicable to any statement evaluation. If two branches share identical classical states, we merge their quantum states into a single density matrix while maintaining the identical classical state. This rule guarantees that the number of branches during the exact inference of a circuit with n classical bits is upper-bounded by 2^n, as the system can result in a maximum of 2^n different classical states.

b) Superoperator Replacement: Rule [Eval-Reset-Opt] replaces [Eval-Reset] to handle reset statements more efficiently. When a classical bit is used only for a conditional

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad X_{n,q} = gen_{n,q}(X)$$

$$CX_{n,q_1,q_2} = \begin{cases} I_n & \text{if } q_1 = q_2 = 0 \\ Proj_0 \otimes I_{n-1} + Proj_1 \otimes X_{n-1,q_2-1} & \text{if } q_1 = 0 \wedge q_2 > 0 \\ X \otimes Proj_{n-1,q_1-1,0} + I_1 \otimes Proj_{n-1,q_1-1,1} & \text{if } q_1 > 0 \wedge q_2 = 0 \\ I_1 \otimes CX_{n-1,q_1-1,q_2-1} & \text{if } q_1 > 0 \wedge q_2 > 0 \end{cases}$$
$$(0 \leq q_1 < n, 0 \leq q_2 < n)$$

Fig. 7: Generalization of CNOT

$$\frac{\begin{array}{c} \overline{\Sigma} \vdash s \Rightarrow \Sigma'_0, \ldots, \Sigma'_i, \ldots, \Sigma'_j, \ldots, \Sigma'_n \\ \Sigma'_i = (\rho_i, \sigma, P_i) \qquad \Sigma'_j = (\rho_j, \sigma, P_j) \qquad \rho_{ij} = \frac{P_i}{P_i+P_j}\rho_i + \frac{P_j}{P_i+P_j}\rho_j \end{array}}{\overline{\Sigma} \vdash s \Rightarrow \Sigma'_0, \ldots, \Sigma'_{i-1}, \Sigma'_{i+1}, \ldots, \Sigma'_{j-1}, \Sigma'_{j+1}, \ldots, \Sigma'_n, (\rho_{ij}, \sigma, P_i + P_j)} \text{ [Eval-Unify]}$$

$$\frac{\begin{array}{c} qbts(\rho) = n \\ \rho_0 = msr(\rho, Proj_{n,q,0}) \qquad p_0 = prob(\rho, Proj_{n,q,0}) \qquad \rho_1 = msr(\rho, Proj_{n,q,1}) \\ p_1 = prob(\rho, Proj_{n,q,1}) \qquad \rho' = p_0\rho_0 + p_1 gate(\rho_1, U_{\pi,0,\pi}) \end{array}}{(\rho, \sigma, P) \vdash \mathsf{Reset}\ q \Rightarrow (\rho', \sigma, P)} \text{ [Eval-Reset-Opt]}$$

Fig. 8: Optimized exact inference

operation without affecting further computation or the result, it can be removed from the state. Rule [Eval-Reset] assigns a measurement outcome from a qubit to a classical bit, which is used only for conditionally flipping the qubit and not referenced thereafter. Thus, Rule [Eval-Reset-Opt] removes the classical bit from the state and uses a mixed state, transforming the density matrix into a new one by applying a *superoperator*, which directly maps one density matrix to another.

C. Matrix Representation in Coq

For efficient exact inference, QASMInfer requires an efficient matrix representation in Coq. The Coq community has proposed five approaches: dependent lists, dependent pairs, dependent records, functions indexed by natural numbers, and functions indexed by finite elements [19]. Each approach, however, possess disadvantages in terms of complexity and efficiency. We discuss these issues and propose an alternative representation.

The first three approaches use two-dimensional arrays, which complicate definitions and proofs as constructing new matrices involves computing and positioning each element. Since Coq implements arrays as cons lists, this approach complicates proving the physical consistency of QASMInfer, making it less desirable.

The last two approaches represent matrices as index-to-value functions, reducing the complexity associated with two-dimensional arrays. Definitions and proofs are simplified as it is only necessary to specify the computation of each element's value without the need to place these values in a matrix structure. Due to their simplicity, prior research in quantum programming [20], [21] uses the functional representations for Coq mechanizations. However, a significant drawback of these approaches is the inefficiency of execution. Accessing an element involves computing its value based on the matrix's construction, rather than retrieving a stored value. For instance, accessing an element in the product of m matrices of size $n \times n$ requires $O(n^m)$ time complexity. This inefficiency makes them unsuitable for our goal of efficient exact inference.

To address these concerns, we propose an inductive definition of matrices in Coq as follows:

```
Inductive Matrix: nat -> Type :=
  | bas_mat: Complex -> Matrix O
  | rec_mat: forall {n: nat},
             Matrix n -> Matrix n ->
             Matrix n -> Matrix n -> Matrix (S n).
```

Here, `Matrix` n denotes a $2^n \times 2^n$ matrix built inductively from four submatrices. This inductive approach simplifies definitions and proofs without explicitly computing each element's value. For instance, the tensor product is defined as follows:

```
Fixpoint tprod {m n: nat} (A: Matrix m) (B: Matrix n):
     Matrix (m + n) :=
  match A with
  | bas_mat c => c .* B
  | rec_mat A1 A2 A3 A4 =>
      rec_mat (tprod A1 B) (tprod A2 B)
              (tprod A3 B) (tprod A4 B)
  end.
```

This representation is more efficient: accessing an element only involves retrieving a stored value in $O(\log n)$ time, compared to the $O(n)$ for array-based representations, and $O(n^m)$ for functional representations. Thus, the inductive representation enables efficient execution while preserving simplicity in definitions and proofs. Note that our approach is tailored for quantum programming, limited to $2^n \times 2^n$ matrices, which fit density matrices but not arbitrary matrices.

D. Physical Consistency

We introduce the postulates of quantum computing and prove that the optimized exact inference of QASMInfer adheres to the postulates. Scherer [10] summarizes the constraints of the physical model regarding quantum computing within five postulates. To ensure consistency with the terminologies in this paper, we rephrase the original postulates by presenting three postulates. The exact quote of the original postulates and the pencil-and-paper proofs outlining the key ideas behind the mechanized proofs are available in the companion report [22].

The first postulate describes the operation of gates:

Postulate 1 (Gate)**.** *The change of a density matrix from ρ to ρ' by a gate is described by $\rho' = U\rho U^\dagger$ where U is a unitary matrix, i.e., $U^\dagger U = I$.*

We prove that the exact inference of each kind of gate statements in OpenQASMCore satisfies the postulate:

Theorem 2 (Rotation Gate)**.** *Suppose that the following holds: $(\rho, \sigma, P) \vdash \mathsf{U}(r_1, r_2, r_3)\ q \Rightarrow (\rho', \sigma', P')$. Then, there exists a unitary matrix U such that $\rho' = U\rho U^\dagger$.*

Theorem 3 (Cnot Gate)**.** *Suppose that the following holds: $(\rho, \sigma, P) \vdash \mathsf{CX}\ q_1\ q_2 \Rightarrow (\rho', \sigma', P')$. Then, there exists a unitary matrix U such that $\rho' = U\rho U^\dagger$.*

The second postulate describes the results of measurements:

Postulate 4 (Measurement)**.** *For a quantum system described by a density matrix ρ, a physical observable to be measured is represented by a self-adjoint matrix A, whose dimension is the same as ρ. Each measurement outcome corresponds to λ, an eigenvalue of A. The probability of the outcome corresponding to λ is $tr(\rho M)$ where M is the projection onto the eigenspace of λ. The change of a density matrix from ρ to ρ' by the measurement is described by $\rho' = \frac{1}{tr(\rho M)} M\rho M$.*

Identifier	x	$\in$	Id	
Length/Index	n	$\in$	$\mathbb{N}$	
Program	p	$::=$	$\overline{d}; \overline{S}$	
Declaration	d	$::=$	$\mathsf{Creg}\ x_c[n]$	Classical register
		$\mid$	$\mathsf{Qreg}\ x_q[n]$	Quantum register
		$\mid$	$\mathsf{Gate}\ x_g(\overline{x_r})\ \overline{x_q} = \overline{g}$	Custom gate
Gate operation	g	$::=$	$\mathsf{U}(e, e, e)\ a_q$	Rotation
		$\mid$	$\mathsf{CX}\ a_q\ a_q$	Cnot
		$\mid$	$x_g(\overline{e})\ \overline{a_q}$	Custom gate
Expression	e	$::=$	r	Real number
		$\mid$	x_r	Real number parameter
		$\mid$	$e + e \mid \cdots$	Math operations
Argument	a	$::=$	$x[n]$	Quantum/classical bit
		$\mid$	x	Register
Statement	S	$::=$	Q	Quantum operation
		$\mid$	$\mathsf{if}\ x_c \rightsquigarrow n\ \mathsf{then}\ Q$	Conditional
Quantum operation	Q	$::=$	g	Gate operation
		$\mid$	$a_c := \mathsf{Measure}\ a_q$	Measurement
		$\mid$	$\mathsf{Reset}\ a_q$	Reset

Fig. 9: Syntax of OpenQASM

The exact inference of the measurement follows the postulate:

Theorem 5 (Measurement)**.** *Suppose that the following holds: $(\rho, \sigma, P) \vdash c := \mathsf{Measure}\ q \Rightarrow \overline{\Sigma'}$. Then, $\exists A. \forall \Sigma' \in \{\overline{\Sigma'}\}. \exists \lambda$ such that the followings hold: (1) A is self-adjoint; (2) λ is an eigenvalue of A; (3) M is projection onto the eigenspace of λ; (4) $w' = (\rho', \sigma', P')$; (5) $\rho' = \frac{1}{tr(\rho M)} M\rho M$; (6) $\frac{P'}{P} = tr(\rho M)$.*

The last postulate describes the density matrix properties:

Postulate 6 (Density Matrix)**.** *A density matrix ρ satisfies the following: (1) ρ is self-adjoint: $\rho^\dagger = \rho$; (2) ρ is positive semi-definite: $\rho \geq 0$, i.e., $z^\dagger \rho z \geq 0$ for every column vector z; (3) $tr(\rho) = 1$.*

We show that for every OpenQASMCore program, if the exact inference starts with a valid density matrix, then it ends with valid density matrices.

Theorem 7 (Density Matrix)**.** *Suppose that the following holds: $(\rho, \sigma, P) \vdash s \Rightarrow \overline{\Sigma'}$. If ρ is self-adjoint and positive semi-definite and $tr(\rho) = 1$, then for every $(\rho', \sigma', P') \in \{\overline{\Sigma'}\}$, ρ' is self-adjoint and positive semi-definite and $tr(\rho') = 1$.*

The initial state of the exact inference is a state where every qubit is $|0\rangle$, is trivially a valid density matrix. Therefore, exact inference of every OpenQASMCore program terminates with valid density matrices.

IV. Desugaring OpenQASM to OpenQASMCore

In this section, we present the syntax of OpenQASM (§IV-A) and define desugaring from OpenQASM to OpenQASMCore (§IV-B). This enables us to indirectly perform the exact inference for OpenQASM while ensuring physical consistency. The desugarer is implemented in Coq as a part of QASMInfer and publicly available [11].

A. Syntax of OpenQASM

The syntax of OpenQASM is outlined in Fig. 9. Metavariables include x (identifiers for classical/quantum registers, custom gates, and parameters: x_c, x_q, x_g, x_r), n (natural

numbers), and p (programs as sequences of declarations and statements). A declaration d defines a classical register $\mathsf{Creg}\ x_c[n]$, a quantum register $\mathsf{Qreg}\ x_q[n]$, or a custom gate $\mathsf{Gate}\ x_g(\overline{x_r})\ \overline{x_q} = \overline{g}$, where x_g is defined using parameters $\overline{x_r}$ and qubits $\overline{x_q}$ with body $\overline{g}$. Custom gates can reference built-in or earlier gates but cannot be recursive. Gate operations include rotations $\mathsf{U}(e_1, e_2, e_3)\ a_q$, CNOTs $\mathsf{CX}\ a_q\ a'_q$, and custom gates $x_g(\overline{e})\ \overline{a_q}$. Rotations take three real-valued expressions and a quantum argument a_q. Arguments a may be a register x or a bit $x[n]$, with a_q and a_c for quantum and classical arguments. Register-level gates apply elementwise; for example, $\mathsf{CX}\ x\ y$ means $\mathsf{CX}\ x[i]\ y[i]$, and $\mathsf{CX}\ x[0]\ y$ means $\mathsf{CX}\ x[0]\ y[i]$. The involved registers must have matching lengths. A statement S is either a quantum operation Q or a conditional $\mathsf{if}\ x_c \rightsquigarrow n\ \mathsf{then}\ Q$, which compares the binary number stored in a classical register x_c (least significant bit at index 0) to n. A quantum operation Q is either a gate g, a measurement $a_c := \mathsf{Measure}\ a_q$ (storing the result in a classical bit or register), or a reset $\mathsf{Reset}\ a_q$ (sets target qubits to $|0\rangle$).

In fact, for the sake of presentation, the syntax defined in Fig. 9 omits a few OpenQASM features:

- We omit OpenQASM 3 features incompatible with OpenQASM 2, such as unbounded loops, because QASMInfer does not support them.
- We omit *include* statements, which are used to include code from another file in the current code. They can be replaced with proper code through preprocessing.
- We omit *opaque gate* declarations, which declare gates that can be physically implemented but cannot be described in OpenQASM. It is impossible to determine the behavior of opaque gates, and they do not accord with our goal of performing exact inference.
- We omit *barrier* statements, which only affects optimization but not the result of exact inference, can be simply replaced by nop.
- We require all declarations to appear before statements, unlike OpenQASM's interleaved form. This can be easily reordered through preprocessing.

B. Desugaring

The desugaring process from OpenQASM to OpenQASMCore involves following five steps:

1) Unrolling operations: We unroll each quantum operation with register arguments into a sequence of single-qubit operations, similar to loop unrolling. For instance, $\mathsf{CX}\ x\ y$ is desugared as $\mathsf{CX}\ x[0]\ y[0]; \mathsf{CX}\ x[1]\ y[1]$, assuming both x and y have lengths of 2. OpenQASM requires registers to have equal lengths; otherwise, the desugaring fails.

2) Inlining custom gates: We inline custom gates to eliminate their definitions and usages because OpenQASMCore lacks custom gates. The inlining is straightforward as custom gates are non-recursive. If a reference to an undefined custom gate is encountered, the inlining fails.

3) Decomposing conditionals: We desugar conditional statements that compare against a natural number, as OpenQASMCore only supports bit comparisons. We decompose each conditional into nested conditionals that compare individual bits of the register with the corresponding binary digit of the number. For instance, $\mathsf{if}\ x \rightsquigarrow 6\ \mathsf{then}\ \cdots$ is desugared as $\mathsf{if}\ x[2] \rightsquigarrow 1\ \mathsf{then}\ \mathsf{if}\ x[1] \rightsquigarrow 1\ \mathsf{then}\ \mathsf{if}\ x[0] \rightsquigarrow 0\ \mathsf{then}\ \cdots$.

4) Representing qubits and classical bits with natural numbers: We represent qubits and classical bits using natural numbers instead of identifiers. Conceptually, we line up all quantum registers and substitute each qubit access with the corresponding index within the line; this process is repeated for classical registers.

5) Evaluating expressions: We evaluate and substitute every expression used as an argument for a rotation gate with the actual real number. This substitution is possible because all parameters have been replaced by constants and operations through inlining, leaving expressions with no variables.

V. Evaluation

We evaluate our approach with four research questions:

- **RQ1. Effectiveness of optimizations**: How significantly do our optimizations and matrix representation reduce inference times? (§V-A)
- **RQ2. Effectiveness as a testing oracle** (§V-B)
  - **RQ2.1.** How many real-world bugs can be detected using QASMInfer as a testing oracle? (§V-B1)
  - **RQ2.2.** How does QASMInfer compare with prior work with respect to bug detection? (§V-B2)
- **RQ3. Impact of simulator runs on bug detection**: How does the number of simulator runs affect the bug detection capability of testing? (§V-C)
- **RQ4. Time cost per step**: How much time does each step of the testing process take? (§V-D)

We extracted OCaml code from the QASMInfer Coq mechanization and ran all experiments on a MacBook Pro (M1 Max, 32 GB RAM). The evaluation environment is available at [23].

A. RQ1: Effectiveness of Optimizations

We evaluate the effectiveness of our optimizations and matrix representation in reducing inference times by comparing five settings: (1) the functional representation of matrices without the optimizations, the inductive representation of matrices (2) without the optimizations, (3) with branch unification, (4) with superoperator replacement, and (5) with both. The optimizations particularly decrease the inference times of dynamic circuits, which include mid-circuit measurements, by reducing the creation of new branches during measurements. Due to the scarcity of dynamic circuits in available OpenQASM benchmark suites [24]–[26], we generated 100 random dynamic circuits for evaluation. Each circuit contains five qubits and average 62.8 random operations, where each operation is either a reset, a measurement, or a gate chosen from Qiskit's built-in gate set, consisting of 52 different gates. Each operation has a 50% chance of being conditional, and each circuit contains

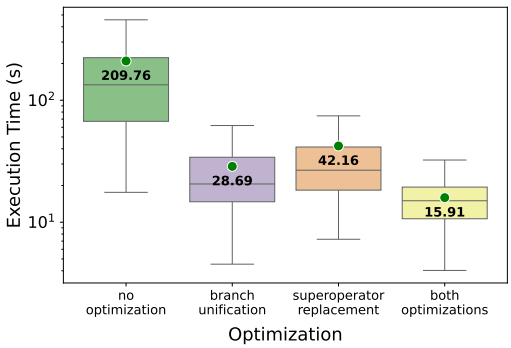


Fig. 10: Inference times for different settings (log scale)

5.7 measurements and 4.9 resets on average. We performed exact inference on each circuit with a one-hour timeout.

Results show that the inductive matrix representation and optimizations significantly reduce inference time. Under setting 1, no circuit completed, showing the inefficiency of functional matrices. Fig. 10 shows inference times for settings 2–5; average values (green dots) are annotated. Branch unification reduces time by 86% on average; superoperator replacement, 80%; and both together, 92%. Branch unification helps especially when circuits have more measurements than classical bits, and superoperator replacement is highly beneficial for frequent resets, avoiding exponential branching.

B. RQ2: Effectiveness as a Testing Oracle

We evaluate the effectiveness of QASMInfer as a testing oracle by applying the testing procedure using QASMInfer as a testing oracle to six real-world QSSs (§V-B1) and comparing it with prior works by testing the same QSSs (§V-B2). Each QSS uses an internal representation (e.g., `QuantumCircuit` in Qiskit and `cirq.Circuit` in Cirq) and supports OpenQASM import/export, enabling testing without code modification.

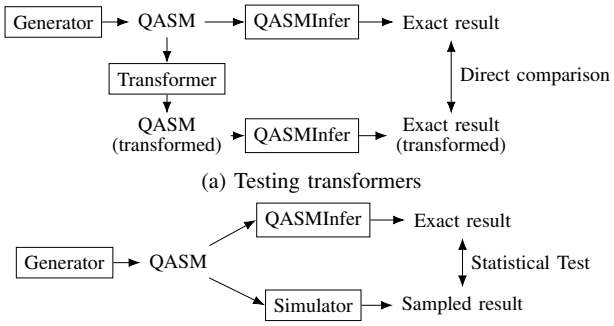
For testing inputs, we generated 50,000 random circuits: 40,000 static and 10,000 dynamic. This scale was chosen to complete generation within a few hours. More static circuits were generated because optimizations in transformers primarily target static circuits. Transformers or simulators that do not support dynamic circuits were tested only with static inputs.

The circuit generation process inserts gate sequences likely to trigger optimizations during transformer testing, producing circuits with an average of 108.5 operations. Specifically, we instantiate gate sequence patterns from Table I with randomly chosen gates and qubits, and interleave them with other gates and measurements until the circuit reaches the target size. For dynamic circuits, each insertion is made classically conditional with a 50% probability. The circuits used in testing have 4-5 qubits, balancing circuit size and execution time to maximize throughput, although QASMInfer can handle circuits with over 12 qubits in the evaluation environment. Our inspection of optimization passes in QSSs revealed that even three-qubit circuits suffice to trigger most optimizations.

We now describe our testing procedure. Fig. 11 shows the workflows for testing transformers (Fig. 11a) and simulators (Fig. 11b). For transformers, we import generated

TABLE I: Gate sequences inserted to generated circuits

Gate Sequence	Optimization Description
Clifford/linear gates	Replaced by a single gate.
Self-inverse gates	Removed in pairs
G, commutants of G, and G^{-1}	G and G^{-1} removed.
CNOT gates	Removed in pairs.
2-qubit gates	Rewritten via Weyl decomposition.
Gates handled by Hoare logic	Rewritten via Hoare logic.
Commuting 1 and 2-qubit gates	Optimized via commutation.
SWAP gate and measurement	SWAP removed.
Diagonal gate and measurement	Diagonal gate removed.
Reset on a qubit in $\|0\rangle$ state	Reset removed.
Measurement and reset	Reset removed.



(a) Testing transformers

(b) Testing simulators

Fig. 11: Workflow of RQ2

OpenQASM circuits to the QSS's internal representation, apply the transformer, and export the transformed circuit back to OpenQASM. We then perform exact inference on both the original and transformed circuits and directly compare their probability distributions. Since QASMInfer yields exact probabilities, we use direct numerical comparison, avoiding the statistical testing that is used in existing methods [4], [6]. This is a significant advantage, as statistical tests inherently produce false positives at a rate determined by the significance level (e.g., 0.01). As the number of test cases increases, the absolute number of false positives grows, requiring additional manual inspection. In contrast, QASMInfer performs direct numerical comparison based on exact probability distributions, avoiding this issue.

For simulators, we import and execute generated OpenQASM circuits on the simulator 1,024 times, while also performing exact inference on the same circuit using QASMInfer. We follow Qiskit's default of 1,024 runs and confirmed its sufficiency experimentally (§V-C). We then apply the chi-square goodness-of-fit test [27] to evaluate whether the simulator's sampled result is consistent with the exact probability distribution. A p-value below 0.01 indicates a significant divergence between the simulator and QASMInfer, suggesting incorrect simulation results. Although simulator testing uses statistical tests, QASMInfer compares observed simulator outcomes to an exact probability distribution (one-sample test), unlike existing two-sample methods. This enables bug detection with fewer simulator runs (§V-C).

1) RQ2.1: Overall Bug Detection Results: We applied the testing procedure to the six QSSs (detailed in Table II), including four Qiskit, two Cirq, and two staq versions. All

TABLE II: Tested quantum software stacks

Name	Versions	# Sim.	# Trans.
§Qiskit [1]	0.25.2, 0.45.0, 1.0.2, 2.2.1	12	19
§Cirq [2]	1.3.0, 1.6.1	1	18
TKET [12]	1.26.0	‡0	37
staq [13]	3.3, 3.5	0	4
PyQuil [14]	4.8.0	†1	0
Braket [15]	1.76.0	†1	0

§They support OpenQASM 3.
‡Simulators are proprietary and thus not tested.
†They support only static circuits.

support OpenQASM 2; Qiskit and Cirq additionally support OpenQASM 3 import and export, and were therefore also tested with OpenQASM 3. Our results demonstrate the effectiveness of QASMInfer as a testing oracle. As summarized in Table III, we identified 31 bugs across five of the six tested QSSs: 11 in Qiskit, 10 in Cirq, seven in TKET, two in staq, and one in Braket. The fourth and fifth columns indicate whether each bug was previously unknown and detectable only with dynamic circuits. 27 bugs were confirmed by the developers, all except four of which were previously unknown. 15 are solely triggered by dynamic circuits, highlighting their critical role in testing. The sixth column of the table categorizes the source of each bug as simulation, transformation, import, or export. These correspond to errors in simulators, transformers, OpenQASM-to-internal conversion, and internal-to-OpenQASM conversion, respectively. The seventh column categorizes each bug as semantics, validity, or crash. Semantics refers to incorrect behavior, i.e., simulators producing wrong distributions or transformers not preserving semantics. Validity refers to the export process yielding invalid OpenQASM code that cannot be fed into QASMInfer, e.g., due to syntax errors. Crash refers to the software crashes. Overall, our approach proved highly effective in identifying bugs: we detected 20 non-crash bugs missed by prior work, including nine semantics bugs, the kind QASMInfer excels at detecting. No bugs were found in QASMInfer, signifying its reliability.

Now, we discuss some examples. Bug 2 resides in the import process of Qiskit v0.25.2, where OpenQASM conditionals are dropped in the internal representation. For instance, 'if $\mathsf{c} \rightsquigarrow 1$ then CX q[0] q[1]' is erroneously converted to 'CX q[0] q[1],' making most dynamic circuits execute incorrectly.

Bug 4 resides in `HoareOptimizer` transformer of Qiskit v0.45.0, leading to incorrect optimizations by mishandling conditions. Consider a circuit that applies a Hadamard gate once or twice depending on the value of classical register `c`:

$$\mathsf{U}(\pi/2, 0, \pi)\ \mathsf{q}[0];\ \mathsf{if}\ \mathsf{c} \rightsquigarrow 1\ \mathsf{then}\ \mathsf{U}(\pi/2, 0, \pi)\ \mathsf{q}[0];\ \mathsf{d} := \mathsf{Measure}\ \mathsf{q}$$

The transformer incorrectly optimizes this to $\mathsf{c} := \mathsf{Measure}\ \mathsf{q}$, removing the Hadamard gates. Although a pair of Hadamard gates cancel as they are self-inverse, this optimization wrongly ignores that the second gate is conditional. In this specific example, where `c` equals 0, only one Hadamard gate should be applied. QASMInfer detects this by showing a difference in the probability distributions of the output, confirming the bug: the original circuit gives a 50% chance for each output, whereas the optimized one gives 100% chance of 0.

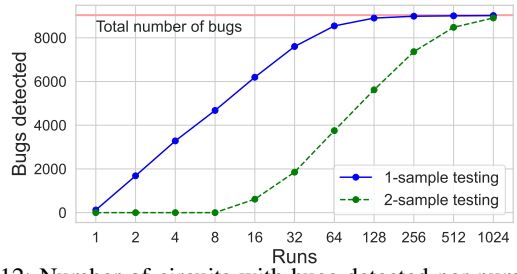


Fig. 12: Number of circuits with bugs detected per number of simulator runs (x-axis is log scale)

Bug 13 resides in the export process of Cirq, generating OpenQASM code with a classical register of incorrect size. The following example uses a classical register of size 2 for measurement results:

$$\mathsf{c}[0] := \mathsf{Measure}\ \mathsf{q}[0];\ \mathsf{c}[0] := \mathsf{Measure}\ \mathsf{q}[0];\ \mathsf{c}[1] := \mathsf{Measure}\ \mathsf{q}[1]$$

Cirq exports this as $\mathsf{Creg}\ \mathsf{c}[1]$, mistakenly declaring size 1. QASMInfer detects the bug by failing to desugar such code.

2) RQ2.2: Comparison with Prior Work: To further validate the effectiveness of QASMInfer as a testing oracle, we compared QASMInfer with QDiff, MorphQ, and QuteFuzz. We applied QASMInfer to the same QSSs studied in each work: Qiskit for QDiff and MorphQ, and Qiskit, Cirq, and TKET for QuteFuzz. For comparison with MorphQ, we used the same version it tested; for QDiff and QuteFuzz, we used the latest QSS versions available at the time of their experiments, as they did not specify which versions they tested. Note that QDiff also tested Cirq and PyQuil, but we excluded them because, at the time of their experiments, those did not properly support OpenQASM. All comparison experiments were run for 48 hours, following the original studies.

The comparison experiments reveal the strengths of QASMInfer. Compared to QDiff and MorphQ, QASMInfer detected substantially more bugs, as summarized in Table IV. Notably, it detected non-crash bugs that both previous approaches entirely missed, highlighting the advantage of exact inference as a testing oracle. In addition, compared to QuteFuzz, QASMInfer detected the same or a greater number of bugs in each of the three QSSs, as shown in Table V.

C. RQ3: Impact of Simulator Runs on Bug Detection

We investigate the likelihood of bug detection with varying numbers of simulator runs. As described in §V-B, each circuit is executed on a simulator 1,024 times and compared against the correct probability distribution using statistical tests. Fewer simulator runs generally leads to higher p-values, reducing bug detection rates. To evaluate this impact, we conducted a case study. With 1,024 runs, Qiskit v0.25.2's simulator yielded a p-value below 0.01 for 9,037 out of 50,000 circuits due to Bug 2. We varied the number of runs to determine how many among these 9,037 circuits lead to a p-value below 0.01.

The blue line in Fig. 12 depicts how many circuits reveal the bug at each simulator run count. Even with 128 runs, over 9,000 circuits can expose the bug. This suggests that simulator runs can be reduced without significantly compromising bug

TABLE III: Real-world bugs found by our approach

ID	QSS	Status	New	Dyn	Source	Kind	Description
1	Qiskit v0.25.2	fixed	no	no	transform	semantics	wrong commutation analysis
2	Qiskit v0.25.2	fixed	yes	yes	import	semantics	losing classical conditionals
3	Qiskit v0.45.0	fixed	yes	no	transform	semantics	incorrectly optimizing SWAP and measurement
4	Qiskit v0.45.0	fixed	yes	yes	transform	semantics	Hoare optimizer ignoring classical conditions
5	Qiskit v0.45.0, 1.0.2, 2.2.1	confirmed	yes	no	transform	semantics	Hoare optimizer misusing gate cancellation
6	Qiskit v1.0.2,	fixed	yes	no	import	crash	crash due to missing gate definitions
7	Qiskit v1.0.2, 2.2.1	confirmed	yes	no	transform	crash	crash while optimizing swap before measurement
8	Qiskit v2.2.1	confirmed	no	no	transform	crash	Hoare optimizer removing the same gate twice
9	Qiskit v2.2.1	reported	yes	no	transform	crash	Hoare optimizer trying to look up a deleted gate
10	Qiskit v2.2.1	reported	yes	no	transform	validity	generating empty unnamed gates
11	Qiskit v2.2.1	fixed	yes	yes	transform	semantics	incorrect commutation check for controlled gates
12	Cirq v1.3.0, 1.6.1	fixed	yes	no	transform	semantics	incorrectly merging Pauli gates to measurements
13	Cirq v1.3.0	fixed	yes	no	export	validity	invalid classical register size
14	Cirq v1.3.0	fixed	yes	no	export	validity	wrong real number syntax
15	Cirq v1.6.1	confirmed	yes	yes	transform	crash	crash while adding dynamical decoupling
16	Cirq v1.6.1	fixed	yes	no	import	crash	rejecting OpenQASM 3 identifiers
17	Cirq v1.6.1	confirmed	yes	yes	import	validity	incorrect ordering of control keys
18	Cirq v1.6.1	fixed	yes	yes	export	crash	crash on classical register conditions
19	Cirq v1.6.1	confirmed	no	yes	transform	semantics	incorrect commutation relation of measurements
20	Cirq v1.6.1	unsupported	yes	yes	import	crash	not initializing classical bits of OpenQASM
21	Cirq v1.6.1	fixed	yes	yes	import	semantics	incorrect reordering of controlled gates
22	TKET	fixed	yes	yes	import	semantics	losing classical conditionals
23	TKET	fixed	yes	yes	export	validity	misplaced classical conditionals
24	TKET	fixed	yes	no	export	validity	undefined gates
25	TKET	fixed	yes	no	transform	crash	crash while optimizing measurements of same qubit
26	TKET	fixed	yes	yes	transform	crash	crash during optimization
27	TKET	fixed	yes	yes	transform	crash	crash during optimization
28	TKET	confirmed	yes	no	export	validity	wrong gate syntax
29	staq v3.3	fixed	yes	yes	transform	semantics	losing conditionals
30	staq v3.5	confirmed	no	yes	transform	validity	not inlining conditioned gates
31	Braket	reported	-	no	simulate	semantics	wrong simulation results

TABLE IV: Comparison with QDiff and MorphQ

	QDiff	QASMInfer	MorphQ	QASMInfer
Non-Crash	0	2	0	7
Crash	0	6	8 †(13)	7 †(30)
Total	0	8	8 †(13)	14 †(37)

†Parenthetical counts follow MorphQ's reporting style of separately counting each variation of "missing or duplicate gate definition" bug.

TABLE V: Comparison with QuteFuzz

	Qiskit v1.3.0		Cirq v1.4.1		TKET v1.31.0	
Non-Crash	2	3	1	1	1	1
Crash	4	6	2	2	7	11
Total	6	9	3	3	8	12

Each pair of columns compares QuteFuzz (left) and QASMInfer (right).

detection capabilities, improving efficiency in contexts like continuous integration.

To further explore the benefits of QASMInfer as an oracle, we repeated the experiment using a differential/metamorphic testing approach, akin to QDIFF [4] and MorphQ [6]. QDIFF compares the same circuit on two simulators, while MorphQ compares semantically equivalent circuits on the same simulator. Both rely on two-sample tests, such as the Kolmogorov-Smirnov test (K-S test), to determine if two output distributions match. In contrast, our approach employs a one-sample test against the exact expected probability distribution. Mimicking differential/metamorphic testing, we executed each circuit in Qiskit v0.25.2 with and without Bug 2 and compared the outcomes using the K-S test, varying simulator run counts.

The green line in Fig. 12 illustrates the number of circuits

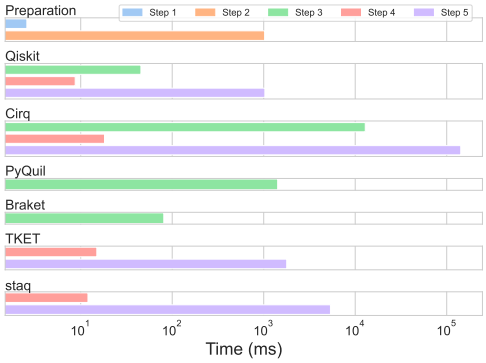


Fig. 13: Average time spent per step (log scale)

where the bug is detected via differential/metamorphic testing, which consistently identifies fewer bugs than our one-sample method across all run counts. These results underscore the advantages of employing exact inference in testing.

D. RQ4: Time Cost per Step

We investigate the time spent by each step of testing: (1) circuit generation, (2) exact inference, (3) simulator execution, (1,024 runs) (4) circuit transformation, and (5) exact inference of a transformed circuit. Simulator testing involves steps 1-3; transformer testing includes steps 1, 2, 4, and 5. This analysis identifies bottlenecks and potential optimizations.

Fig. 13 depicts the average time spent by each step. For Qiskit, step 2 dominates simulator testing (1,023.4 ms), while steps 1 and 3 take 2.6 ms and 45.5 ms, respectively, totaling 1,071.5 ms. In transformer testing, step 5 similarly dominates

(1,029.8 ms), while step 4 takes just 8.7 ms, for a total of 2,064.5 ms. By pre-generating input circuits, steps 1 and 2 can be performed in advance, reducing times to 45.5 ms (simulator) and 1,038.5 ms (transformer).

We now examine the results from other QSSs. Step 3 is significantly longer than in Qiskit: 12,869.3 ms in Cirq, 1,421.4 ms in PyQuil, and 80.9 ms in Braket, suggesting variations in simulator performance. Conversely, step 4 shows similar times across QSSs: 18.2 ms in Cirq, 15.7 ms in TKET, and 12.0 ms in staq. However, step 5 is much longer than in Qiskit, at 142,406.9 ms in Cirq, 1,784.5 ms in TKET, and 5,362.4 ms in staq. This reflects differing transformer strategies; while Qiskit focuses on optimizations that maintain circuit sizes, other QSSs' transformers may expand circuits to comply with gate constraints.

E. Threats to Validity

A potential threat to external validity is generalizability. QASMInfer does not support OpenQASM 3 features that are incompatible with OpenQASM 2, such as unbounded loops. While such features are rarely used in practice yet, future work is needed to evaluate QASMInfer on broader language features. Additionally, the scalability of QASMInfer is limited by the exponential growth of quantum states, as is common with executing quantum circuits on classical computers. While this prevents its application to very large circuits, our evaluation demonstrates that its performance is sufficient for its primary goal of testing transformers and simulators, where circuits of a size capable of revealing complex bugs are still manageable.

A threat to internal validity arises from randomized circuit generation. Although it aims to trigger common optimizations, rare edge-case bugs may be missed. The size and diversity of our 50,000-circuit test set partially mitigate this concern.

For construct validity, the main concern is QASMInfer's correctness. The inference engine is formally verified, but desugaring is not, as it is trivial. In theory, false positives can occur, but we have not observed them in practice.

VI. Related Work

a) Testing Oracles for Language Implementations: For traditional languages, *executable formal semantics* have served as testing oracles for implementations such as interpreters and compilers. The JavaScript semantics was defined manually in the K framework [28] and automatically extracted [29] from the language specifications [30] to test JavaScript engines [31]. The WebAssembly semantics was defined in Isabelle/HOL as a fuzzing oracle for interpreters [32] and mechanized using SpecTec [33], which automatically generates an interpreter.

b) Quantum Programming Languages: Researchers have proposed high-level quantum programming languages as complements to quantum assembly languages, such as OpenQASM, the main focus of this work. Early efforts integrate quantum computation into classical paradigms. QCL [34] and qGCL [35] introduce imperative features, while quantum lambda calculus [36], QML [37], and Quipper [38] focus on functional approaches. More recent languages address specific challenges. Silq [39] simplifies *uncomputation*; Twist [40] uses types to reasons about *purity*; Tower [41] supports random-access memory for pointer-based data structures; Qunity [42] unifies quantum and classical constructs; and Feng and Ying [43] propose a Hoare logic for quantum programs, describing states of dynamic circuits.

A few studies deal with quantum assembly languages. VOQC [21], built on SQIR in Coq, is a verified quantum circuit optimizer compatible with OpenQASM. QWIRE [20] is a quantum assembly language that can be manipulated via a classical host language, with semantics defined by density matrices. Its type system ensures trace preservation, aligning with the last condition specified by Postulate 6. However, QWIRE does not enforce the other properties required by the postulates, whereas this work proves the physical consistency.

c) Exact Probabilistic Inference: Exact probabilistic inference computes precise probability distributions in probabilistic models, crucial for reliable decision-making. Early methods generalize variable elimination, leading to probabilistic programming languages that leverage graph representations. For example, IBAL [44] and Fun [45] compile programs into factor graphs for efficient inference. Dice [46] and BernoulliProb [47] employ decision diagrams, as in PRISM [48]. Recent approaches employ sum-product networks (SPNs) [49]: SPPL [50] compiles to SPNs, and FSPN [51] extends SPNs for recursion. PERPL [52] compiles unbounded recursion to polynomial systems, solving them for least fixed points numerically. Tools like PSI [53], λPSI [54], and ProbZelus [55] provide symbolic exact inference over probabilistic expressions.

VII. Conclusion

In this work, we present QASMInfer, an exact inference system for OpenQASM to address the need for a testing oracle for QSSs. QASMInfer desugars OpenQASM into OpenQASMCore and performs exact inference on OpenQASMCore. We introduce two optimizations—branch unification and superoperator replacement—and an inductive matrix representation in Coq, significantly reducing inference times. We also prove the physical consistency of QASMInfer to ensure that its inference results align with quantum mechanical principles. Our evaluation demonstrates QASMInfer's effectiveness as a testing oracle by identifying 31 bugs in six real-world QSSs. As future work, we plan to extend QASMInfer to higher-level quantum languages and improve its scalability. A concrete next step is supporting omitted features like unbounded loops.

Acknowledgment

This work was partly supported by the National Research Foundation of Korea (NRF) (2022R1A2C2003660 and 2021R1A5A1021944), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (2024-00337703), and Samsung Electronics Co., Ltd.

REFERENCES

[1] IBM, "Qiskit," https://qiskit.org/, 2024.

[2] Google, "Cirq," https://quantumai.google/cirq, 2024.

[3] M. Paltenghi and M. Pradel, "Bugs in quantum computing platforms: an empirical study," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527330

[4] J. Wang, Q. Zhang, G. H. Xu, and M. Kim, "QDiff: Differential testing of quantum software stacks," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 692–704. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678792

[5] I. Iwumbwe, B. Z. Liu, and J. Wickerson, "QuteFuzz: Fuzzing quantum compilers using randomly generated circuits with control flow and sub-circuits," in *PlanQC '25*, 2025, https://github.com/QuteFuzz/QuteFuzz.

[6] M. Paltenghi and M. Pradel, "MorphQ: Metamorphic testing of the Qiskit quantum computing platform," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. New York, NY, USA: Association for Computing Machinery, 2023.

[7] T. Alexander, L. Bishop, A. Cross, J. Gambetta, A. Javadi-Abhari, B. Johnson, and J. Smolin, "A new OpenQASM for a new era of dynamic circuits," https://medium.com/qiskit/a-new-openqasm-for-a-new-era-of-dynamic-circuits-87f031cac49, 2020.

[8] M. Dobšíček, G. Johansson, V. Shumeiko, and G. Wendin, "Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark," *Physical Review A*, vol. 76, no. 3, sep 2007. [Online]. Available: https://doi.org/10.1103%2Fphysreva.76.030306

[9] P. Selinger, "Towards a quantum programming language," *Mathematical Structures in Computer Science*, vol. 14, no. 4, p. 527–586, 2004.

[10] W. Scherer, *Mathematics of Quantum Computing*, 1st ed., ser. Physics and Astronomy. Springer Cham, 2019.

[11] K. Lee, J. Hong, and S. Ryu, "Exact inference for quantum circuits: A testing oracle for quantum software stacks (artifact)," https://github.com/kaist-plrg/QASMInfer, 2025.

[12] Quantinuum, "Tket," https://tket.quantinuum.com/, 2024.

[13] softwareQinc, "staq," https://github.com/softwareQinc/staq, 2024.

[14] Rigetti, "Pyquil," https://github.com/rigetti/pyquil, 2024.

[15] Amazon, "Braket," https://aws.amazon.com/braket/, 2024.

[16] M. Hayashi, S. Ishizaka, A. Kawachi, G. Kimura, and T. Ogawa, *Introduction to quantum information science*. Springer, 2014.

[17] D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén, "HALO: Haskell to logic through denotational semantics," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 431–442. [Online]. Available: https://doi.org/10.1145/2429069.2429121

[18] D. Coughlin and B.-Y. E. Chang, "Fissile type analysis: Modular checking of almost everywhere invariants," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 73–85. [Online]. Available: https://doi.org/10.1145/2535838.2535855

[19] Z. Shi and G. Chen, "Integration of multiple formal matrix models in coq," in *Dependable Software Engineering. Theories, Tools, and Applications*, W. Dong and J.-P. Talpin, Eds. Cham: Springer Nature Switzerland, 2022, pp. 169–186.

[20] J. Paykin, R. Rand, and S. Zdancewic, "QWIRE: A core language for quantum circuits," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 846–858. [Online]. Available: https://doi.org/10.1145/3009837.3009894

[21] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks, "A verified optimizer for quantum circuits," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. [Online]. Available: https://doi.org/10.1145/3434318

[22] K. Lee, J. Hong, and S. Ryu, "Exact inference for quantum circuits: A testing oracle for quantum software stacks (companion report)," Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17239918

[23] ——, "Exact inference for quantum circuits: A testing oracle for quantum software stacks (evaluation)," https://hub.docker.com/r/qasminfer/qasminfer, 2025.

[24] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017.

[25] A. Li, S. Stein, S. Krishnamoorthy, and J. Ang, "QASMBench: A low-level quantum benchmark suite for NISQ evaluation and simulation," *ACM Transactions on Quantum Computing*, vol. 4, no. 2, feb 2023. [Online]. Available: https://doi.org/10.1145/3550488

[26] T. Tomesh, P. Gokhale, V. Omole, G. S. Ravi, K. N. Smith, J. Viszlai, X.-C. Wu, N. Hardavellas, M. R. Martonosi, and F. T. Chong, "Supermarq: A scalable quantum benchmark suite," 2022.

[27] K. Pearson, "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

[28] D. Park, A. Stefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 346–356. [Online]. Available: https://doi.org/10.1145/2737924.2737991

[29] J. Park, J. Park, S. An, and S. Ryu, "JISET: JavaScript IR-based semantics extraction toolchain," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 647–658. [Online]. Available: https://doi.org/10.1145/3324884.3416632

[30] ECMA International. (2020) ECMA-262, 11th edition, ECMAScript ®2020 Language Specification. [Online]. Available: https://262.ecma-international.org/11.0

[31] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "JEST: N+1-version differential testing of both JavaScript engines and specification," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 13–24. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00015

[32] C. Watt, M. Trela, P. Lammich, and F. Märkl, "Wasmref-isabelle: A verified monadic interpreter and industrial fuzzing oracle for webassembly," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591224

[33] D. Youn, W. Shin, J. Lee, S. Ryu, J. Breitner, P. Gardner, S. Lindley, M. Pretnar, X. Rao, C. Watt, and A. Rossberg, "Bringing the webassembly standard up to speed with spectec," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656440

[34] B. Ömer, *Structured quantum programming*, 2003.

[35] J. W. Sanders and P. Zuliani, "Quantum programming," in *Mathematics of Program Construction*, R. Backhouse and J. N. Oliveira, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 80–99.

[36] P. Maymin, "Extending the lambda calculus to express randomized and quantumized algorithms," 1997.

[37] T. Altenkirch and J. Grattage, "A functional quantum programming language," in *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, 2005, pp. 249–258.

[38] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 333–342. [Online]. Available: https://doi.org/10.1145/2491956.2462177

[39] B. Bichsel, M. Baader, T. Gehr, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 286–300. [Online]. Available: https://doi.org/10.1145/3385412.3386007

[40] C. Yuan, C. McNally, and M. Carbin, "Twist: Sound reasoning for purity and entanglement in quantum programs," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498691

[41] C. Yuan and M. Carbin, "Tower: Data structures in quantum superposition," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563297

[42] F. Voichick, L. Li, R. Rand, and M. Hicks, "Qunity: A unified language for quantum and classical computing," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, jan 2023. [Online]. Available: https://doi.org/10.1145/3571225

[43] Y. Feng and M. Ying, "Quantum hoare logic with classical variables," *ACM Transactions on Quantum Computing*, vol. 2, no. 4, dec 2021. [Online]. Available: https://doi.org/10.1145/3456877

[44] A. Pfeffer, "Ibal: A probabilistic rational programming language," in *International Joint Conference on Artificial Intelligence*, 2001. [Online]. Available: https://dl.acm.org/doi/10.5555/1642090.1642189

[45] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael, "Measure transformer semantics for bayesian machine learning," in *European Symposium on Programming*, 2011. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-19718-5_5

[46] S. Holtzen, G. V. den Broeck, and T. D. Millstein, "Scaling exact inference for discrete probabilistic programs," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1 – 31, 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3428208

[47] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström, "Bayesian inference using data flow analysis," in *ESEC/FSE 2013*, 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2491411.2491423

[48] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*, 2011. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-22110-1_47

[49] H. Poon and P. M. Domingos, "Sum-product networks: A new deep architecture," *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pp. 689–690, 2011. [Online]. Available: https://ieeexplore.ieee.org/document/6130310?reason=concurrency

[50] F. A. Saad, M. C. Rinard, and V. K. Mansinghka, "Sppl: probabilistic programming with fast exact symbolic inference," *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3453483.3454078

[51] A. Stuhlmüller and N. D. Goodman, "A dynamic programming algorithm for inference in recursive probabilistic programs," *ArXiv*, vol. abs/1206.3555, 2012. [Online]. Available: https://arxiv.org/abs/1206.3555

[52] D. Chiang, C. McDonald, and C. chieh Shan, "Exact recursive probabilistic programming," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 665 – 695, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3586050

[53] T. Gehr, S. Misailovic, and M. T. Vechev, "Psi: Exact symbolic inference for probabilistic programs," in *International Conference on Computer Aided Verification*, 2016. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-41528-4_4

[54] T. Gehr, S. Steffen, and M. T. Vechev, "λpsi: exact inference for higher-order probabilistic programs," *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3385412.3386006

[55] E. H. Atkinson, C. Yuan, G. Baudart, L. Mandel, and M. Carbin, "Semi-symbolic inference for efficient streaming probabilistic programming," *Proceedings of the ACM on Programming Languages*, vol. 6, pp. 1668 – 1696, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3563347