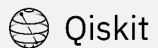


QAMP 2025

Team #35

Equivalence Checking Between
OpenQASM Programs in Lean



PRESENTERS

Kesar Valera
Kazi Muktadir Ahmed

MENTOR

Omar Shehab

PROGRAM

Qiskit Advocate
Mentorship Program 2025

QUANTUM COMPILERS LACK FORMAL GUARANTEES – WE BUILT A PROOF-BASED SOLUTION

The Problem

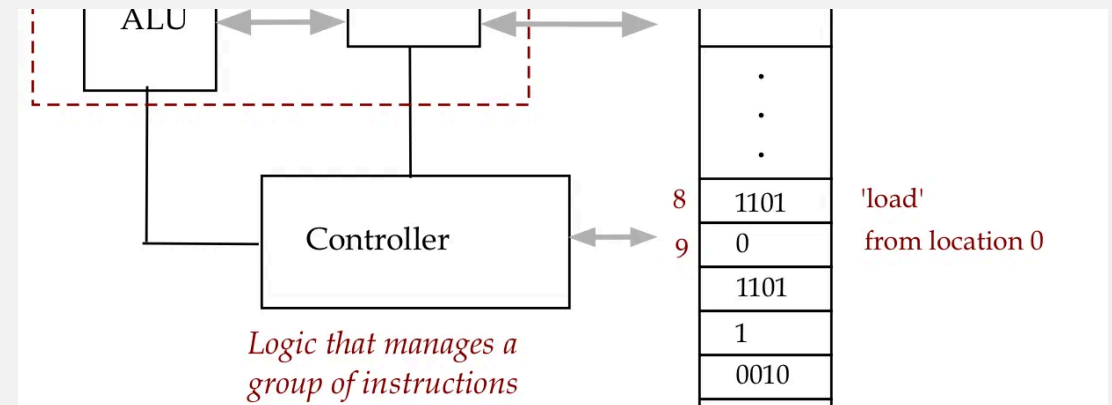
Quantum compilers rely on heuristics for circuit optimization, lacking mathematical proof of correctness.

Our Goal

Build a Lean 4-based framework to formally verify equivalence between OpenQASM programs.

The Outcome

Rigorous, proof-backed equivalence checking that guarantees two circuits compute the same quantum operation.



WHY THIS MATTERS

As quantum hardware scales, compiler bugs could lead to incorrect quantum computations with no way to detect them. Formal verification provides mathematical certainty.

DELIVERED A LEAN-POWERED CHECKER FOR 1 AND 2 QUBIT CIRCUIT EQUIVALENCE

MVP ACHIEVED

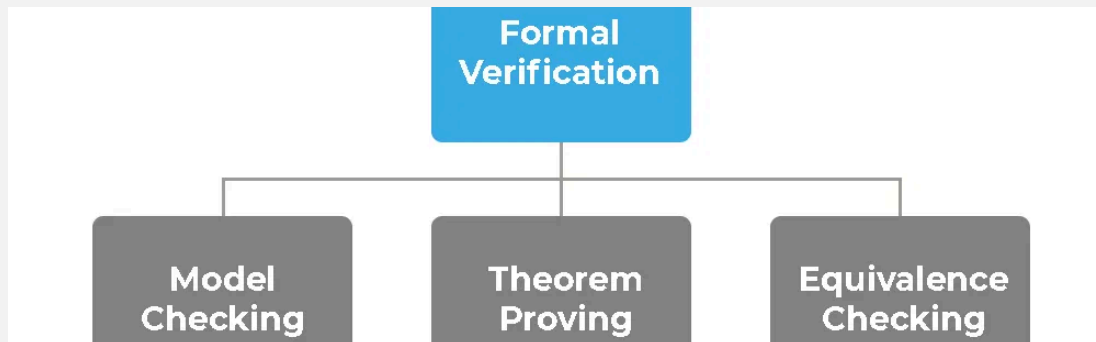
Lean-powered checker proving equivalence of 1 and 2 qubit circuits up to global phase.

Deliverables Completed

- Lean Intermediate Representation (IR) & semantics
- Automated equivalence checker for single-qubit
- Two-qubit circuit framework (CNOT, SWAP, CZ)
- Python CLI tool for OpenQASM to Lean conversion
- Technical documentation and analysis

Beyond MVP - Research Contribution

Identified and documented fundamental scalability limitations in Lean's kernel and profiled performance bottlenecks.



SINGLE-QUBIT GATES: IMPLEMENTATION & VERIFICATION

IMPLEMENTED GATES

Defined core gate set: I, H, X, Y, Z, S, T.

```
inductive SingleQubitGate | Z | X | Y | H | S | T
```

PROVEN IDENTITIES

Successfully verified key algebraic properties:

- $H^2 = I$ (Self-inverse)
- $X^2 = I, Y^2 = I, Z^2 = I$
- $S^2 = Z$ (Phase gate relation)

IMPLEMENTATION (LEAN 4)

```
5 inductive SingleQubitGate
6   | Z | X | Y | H | S | T
7   deriving Repr, DecidableEq
8
9   /-- A single-qubit circuit is a list of gates applied in order (head first). -/
10  abbrev SingleQubitCircuit := List SingleQubitGate
11
12  You 2 months ago | 1 author (Nou)
13  namespace SingleQubitGate
14
15  /-- Interpret a primitive gate as the corresponding unitary U[Qubit]. -/
16  noncomputable def toUnitary : SingleQubitGate → U[Qubit] :=
17    | Z => Qubit.Z
18    | X => Qubit.X
19    | Y => Qubit.Y
20    | H => Qubit.H
21    | S => Qubit.S
22    | T => Qubit.T
23
24  end SingleQubitGate
25
26  /-- Evaluate a circuit to a single unitary, left-to-right application. -/
27  noncomputable def evalCircuit (c : SingleQubitCircuit) : U[Qubit] :=
28    c.foldl (fun U g => SingleQubitGate.toUnitary g * U) (1 : U[Qubit])
29
30  /-- Boolean check: do two circuits have exactly the same 2x2 unitary matrix? -/
31  noncomputable def circuitsEqBool (c₁ c₂ : SingleQubitCircuit) : Bool :=
32    let U₁ := (evalCircuit c₁).val
33    let U₂ := (evalCircuit c₂).val
34    let e00 := decide (U₁ 0 0 = U₂ 0 0)
35    let e01 := decide (U₁ 0 1 = U₂ 0 1)
```

VERIFICATION OUTPUT

```
39 -- Examples: use circuitsEqBool directly with literal circuits
40 lemma hh_id_eq : circuitsEqBool [.H, .H] [] = true := by
41   unfold circuitsEqBool evalCircuit SingleQubitGate.toUnitary
42   simp
43
44 lemma ss_z_eq : circuitsEqBool [.S, .S] [.Z] = true := by
45   unfold circuitsEqBool evalCircuit SingleQubitGate.toUnitary
46   simp [Qubit.S_sq]
47
48 lemma hh_xx_eq : circuitsEqBool [.H, .H] [.X, .X] = true := by
49   unfold circuitsEqBool evalCircuit SingleQubitGate.toUnitary
50   simp [Qubit.H_sq, Qubit.X_sq]
51
```

Expected type
└ circuitsEqBool [SingleQubitGate.H, SingleQubitGate.H] [] = true

Messages (0/1)
▼ SingleQubitCircuitLearn400
Goals accomplished!

All Messages
No messages.

TWO-QUBIT FRAMEWORK: MAJOR MILESTONE

EXTENDED GATE SET

Implemented entangling gates and tensor products:

```
inductive TwoQubitGate | cnot | swap | cz
```

VERIFIED IDENTITIES

Proved complex 2-qubit equivalences:

- $\text{CNOT}^2 = I$ (Self-inverse)
- $\text{CZ}^2 = \text{CNOT}^4$ (Identity check)
- Tensor product lifting for 1-qubit gates

VERIFICATION OUTPUT (LEAN 4)

The screenshot displays the Lean 4 IDE interface. On the left, the source code for two lemmas is visible: `cnotTwiceld` (line 60) and `czTwice` (line 64). Both lemmas state that the circuit evaluation of a CNOT gate squared equals the identity, and a CZ gate squared equals a CNOT gate to the fourth power, respectively. The right pane shows the verification output, indicating that the goals for both lemmas have been successfully proven, with the message 'Goals accomplished!' at the bottom.

Screenshot showing successful proof of `cnotTwiceld` and `czTwice` lemmas.

BRIDGING PRACTICAL QUANTUM PROGRAMMING WITH FORMAL VERIFICATION

qasm_to_lean.py

OpenQASM 3.0 to Lean 4 Converter

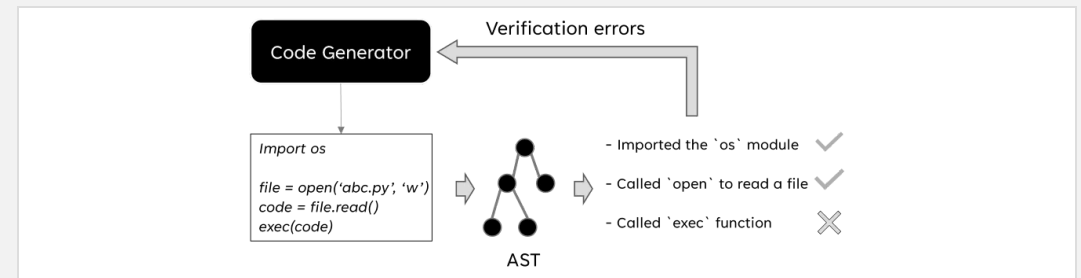
- Parses quantum circuits from .qasm files
- Generates Lean 4 equivalence proof code automatically
- Optional Lean compiler verification
- Zero external dependencies (pure Python)

CURRENT STATUS

Functional for single-qubit circuits; two-qubit support limited by kernel constraints.

VERIFICATION WORKFLOW

- 1 User provides two OpenQASM circuit files
- 2 Tool generates Lean code with circuit definitions
- 3 Lean compiler verifies the proof
- 4 User receives definitive answer



LEAN'S HEARTBEAT MECHANISM: A DETERMINISTIC RESOURCE LIMIT

WHAT IS A HEARTBEAT?

A heartbeat counts **small memory allocations** (in thousands). It provides a deterministic timeout metric independent of CPU speed or system load.

WHY IT EXISTS

- Prevents runaway computations
- Detects inefficient code early
- Resets before each command

THE ERROR ENCOUNTERED

```
(deterministic) timeout at 'whnf', maximum number  
of heartbeats (200000) has been reached
```

THE RESOURCE GAP

200,000

Default Limit

vs

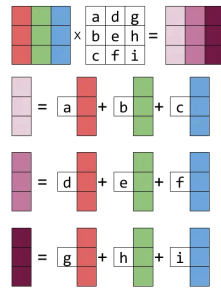
1,400,000

Required for 2-Qubit Check

THE REAL PROBLEM: CNOT UNFOLDING CASCADE

TIMEOUT ANATOMY (1.4M HEARTBEATS)

```
circuitsEqBool c1 c2 =  
  2 × evalCircuit (~1M)  
+ 16 × decide(ℂ eq) (~400k)  
-----  
TOTAL: ~1.4M heartbeats
```



THE UNFOLDING CASCADE

- ↓ TwoQubitGate.toUnitary .cnot
- ↓ Qubit.CNOT (Lean-QuantumInfo)
- ↓ controllize Qubit.X
- ↓ Matrix.control (1 ⊗ X)
- ↓ 16×16×16 pattern matches
- ↓ **500k+ heartbeats PER CIRCUIT**

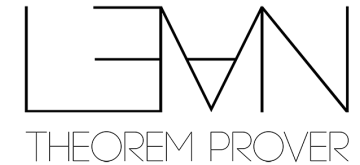
KEY INSIGHT

Each abstraction layer multiplies the kernel's normalization work exponentially.

ROOT CAUSE: PROOF-CARRYING ABSTRACTIONS

ABSTRACTION	PURPOSE	COMPUTATIONAL COST
$U[a]$	Type-safe unitary group	Extra <code>.val</code> extraction per op
<code>controllize</code>	Generic controlled gates	Recursive unfolding cascade
\otimes_u	Tensor products	16 entries computed recursively

Why This Happens: The kernel must fully normalize all abstractions to verify matrix equality. It cannot "trust" that `controllize X` equals CNOT—it must compute and compare all 16 matrix entries.



THE TRADE-OFF

- ✓ Beautiful for theorem proving
- ✗ Deadly for computational decidability

SCALING BREAKDOWN: SINGLE-QUBIT WORKS, TWO-QUBIT TIMEOUTS

CIRCUIT TYPE	MATRIX	HEARTBEATS	STATUS
Single-qubit	2×2	~200k	✓ Works
2-qubit CNOT	4×4	~1.4M	✗ Timeout



WORKAROUNDS TESTED

- maxHeartbeats 1000000 (Barely passes)
- norm_num [Qubit.CNOT] (Manual basis only)
- simp [controllize] (Still unfolds fully)

THE VERDICT

"Proofs scale, computation doesn't. Hybrid approach needed."

PATHS FORWARD: HYBRID VERIFICATION STRATEGY

1. MANUAL 4×4 MATRICES

Bypass abstractions for the decidable checker. Define matrices explicitly to avoid kernel unfolding overhead.

2. PROP-BASED EQUIVALENCE

Use $\forall \rho, \Phi_1 \rho = \Phi_2 \rho$ for proofs instead of boolean decidability. This scales better for theorem proving.

3. EXTERNAL VERIFIER

Compute results in SymPy/Z3 and import them into Lean as trusted axioms or certificates.



Qiskit

KEY INSIGHT

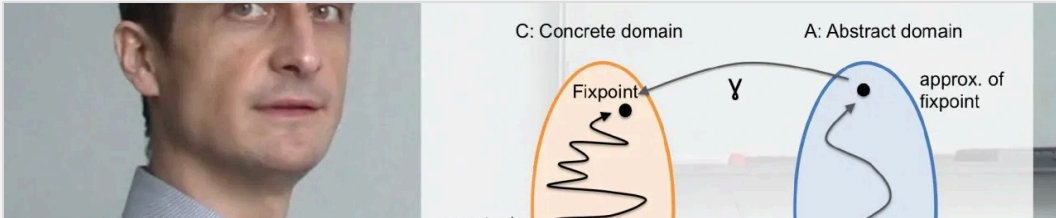
The heartbeat limit is working correctly—it's detecting that brute-force matrix computation is infeasible. The solution is to change strategy, not increase limits indefinitely.

PATHS FORWARD FOR SCALABLE QUANTUM CIRCUIT VERIFICATION IN LEAN

STRATEGIC SHIFT

From Matrix Calculation to Symbolic Reasoning

Move away from brute-force matrix evaluation for $n > 2$.
Use algebraic and symbolic techniques to avoid exponential state growth.



ALGEBRAIC REWRITE SYSTEMS

Build a rewrite engine using gate identities (e.g., $HZH = X$) to canonicalize circuits without matrix computation.

ZX-CALCULUS INTEGRATION

Formalize the ZX-calculus in Lean to reason about circuit equivalence graphically and compactly.

OPTIMIZED DATA STRUCTURES

Replace heavy tuple indexing with inductive types or QMDDs to cut kernel normalization overhead.

THANK YOU

Questions?



TEAM #35

Kesar Valera
Kazi Muktadir Ahmed

MENTOR

Omar Shehab

REPOSITORY

github.com/KesarEra/qamp-35-2025