April 25, 2024

CS 345H: QEC Formal Verification

Neil Allavarpu
neilallavarpu@utexas.edu

Michelle Ding
michelle.ding01@utexas.edu

Aadarsh Narayan
aadarsh271.k.narayan@utexas.edu

Contents

1. Abstract 3
2. Introduction 3
    2.1. Overview of QEC 3
    2.2. Overview of Syntax & Semantics 3
3. Implementation 4
    3.1. Operational Semantics of QECV-Lang 4
    3.2. Denotational Semantics of Stabilizer Expressions 6
4. Conclusion and Future Work 6
    4.1. Notes on Compilation 6
5. References 6

1. Abstract

Quantum Error Correction (QEC) is necessary for fault-tolerant quantum circuits, and a formalization of these codes is especially important in the face of large-scale quantum computing. In this paper, we present a formal verification of QEC semantics in `CoQ` based on the main paper [1] that uses stabilizers as a predicate. We draw from the `SQIR` and `QuantumLib` codebases to provide the necessary mathematical and quantum foundations to do so.

2. Introduction

2.1. Overview of QEC

From the survey paper [2], there are three major types of errors that arise in quantum computing. Measurement errors are addressed by running an algorithm repeatedly to exponentially decrease their error probability. Hardware errors introduce bit errors or local phase changes in quantum circuits. Software bugs have been detected by encoding assertions/invariants and model checking in the circuit or using QHL.

We formalize the error-correcting codes for the second type of error. We utilize the `SQIR` [3] and `QuantumLib` [4] codebases to hash out the operational semantics in `CoQ`. This codebase was chosen for various reasons. Looking into the open-source code shows that despite its recent development, it has already verified algorithms such as Deutch-Josza, Grover's, Phase Estimation and properties of Shor's algorithm. In addition, `SQIR` has clean notation reflecting a quantum setting that transfers directly to `CoQ` proofs.

In the rest of this paper, we first discuss the result of researching existing models of quantum error correction. We then explain the syntax and semantics of our model, and formalize it in `CoQ`. We import the utility methods from `SQIR` and `QuantumLib` to provide the background mathematical structures and theorems for our selected model. We walk through the design of our implementation, then discuss potential future work such as application to concrete examples as a demonstration.

2.2. Overview of Syntax & Semantics

We adapt our Quantum Error Correction `CoQ` Protocol from the following paper [1].

Quantum circuits can be designed from gates and qubits. Operations include initialization of qubits, measurement of qubits, and gate operations on qubits. The paper abstracts quantum circuits with all of their complexity into a simple, *IMP*-like syntax, called *QECV-Lang*. Like *IMP*, *QECV-Lang* has commands for variable assignment, sequences of commands, if conditionals, while loops, and "skip" for termination. In order to actually apply *IMP* to Quantum programs, we replace the notion of variables with qubit states. To apply *IMP* to Quantum Error Correction, we also add in a new type of variable called "stabilizers", because their result depends on the state of the qubit we desire to correct.

```
Inductive stabilizer : unitary -> Prop :=
| eigenstate: forall ψ A, stabilizer_pair ψ A -> stabilizer A.
```

Listing 1: Stabilizer formalism proposed in main paper

This syntax then comes with operational semantics largely similar to *IMP*, with the change that qubits are modified by a Unitary rule (representing unitary evolution). This way, we model the fact that qubits cannot be naively cloned. Stabilizers, on the other hand are treated as normal variables, under the assumption that a real quantum circuit can use arbitrarily pre-designed unitary gates. As a result, stabilizers are given an Assignment rule. Finally, in order to model both the results of error correction and the requirement of measurement to gain classical information from a quantum system, the if and while commands are modified to include both stabilization and measurement in their evaluation.

Through the combination of this syntax and modified semantics, we gain a sufficiently powerful DSL to model QEC. The focus of our implementation was on this syntax and semantics, but the paper also proceeded to derive Hoare logic for QEC as well.

3. Implementation

We implement the operational semantics formalization of *QECV-Lang*, and opted to implement the evaluation of stabilizers using denotational semantics.

```
Inductive step :
QECV_Lang * (qubit_state * stabilizer_map) ->
QECV_Lang * (qubit_state * stabilizer_map) ->
Prop
```

Listing 2: State of a program

The state of the program is represented as a $2n \times 2n$ density matrix over all n qubits; this will capture the state of all n qubits at once. The state also contains a map of stabilizer variables to stabilizer values (as strings of Pauli Matrices, or alternatively unitary matrices), akin to how we used a map of variables to naturals in *IMP* to represent values of variables.

3.1. Operational Semantics of QECV-Lang

The bulk of our work is done on operational semantics, based on the main paper [1].

$$\text{(Skip)}\quad \frac{}{\langle \textbf{skip}, (\rho,\sigma)\rangle \to \langle E, (\rho,\sigma)\rangle}$$

$$\text{(Initialization)}\quad \frac{}{\langle q := |0\rangle, (\rho,\sigma)\rangle \to \langle E, (\rho_0^q,\sigma)\rangle}$$

$$\text{(Unitary)}\quad \frac{}{\langle \bar{q} := U[\overline{q}], (\rho,\sigma)\rangle \to \langle E, (U\rho U^\dagger,\sigma)\rangle}$$

$$\text{(Sequence E)}\quad \frac{}{\langle E; \text{Prog}_2, (\rho,\sigma)\rangle \to \langle \text{Prog}_2, (\rho,\sigma)\rangle}$$

$$\text{(Stabilizer exp)}\quad \frac{}{\langle c\cdot s, \sigma\rangle \to c\cdot s} \qquad \frac{}{\langle \pm\int, \sigma\rangle \to \pm\sigma(\int)}\ ,\ c \in \{1,-1,i,-i\}$$

$$\text{(Assignment)}\quad \frac{\langle s_e^u, \sigma\rangle \to c\cdot s,}{\langle \int := s_e^u, (\rho,\sigma)\rangle \to \langle E, (\rho, \sigma[c\cdot s/\int])\rangle}$$

$$\text{(Sequence)}\quad \frac{\langle \text{Prog}_1, (\rho,\sigma)\rangle \to \langle \text{Prog}_1', (\rho',\sigma')\rangle}{\langle \text{Prog}_1; \text{Prog}_2, (\rho,\sigma)\rangle \to \langle \text{Prog}_1'; \text{Prog}_2, (\rho',\sigma')\rangle}$$

$$\text{(If -1)}\quad \frac{}{\langle \textbf{if } M[\int,\bar{q}] \textbf{ then } \text{Prog}_1 \textbf{ else } \text{Prog}_0 \textbf{ end}, (\rho,\sigma)\rangle \to \langle \text{Prog}_0, (M_0\rho M_0^\dagger, \sigma[-\sigma(\int)/\int])\rangle}\ ,\ M_0 = \frac{I-\int}{2}$$

$$\text{(If 1)}\quad \frac{}{\langle \textbf{if } M[\int,\bar{q}] \textbf{ then } \text{Prog}_1 \textbf{ else } \text{Prog}_0 \textbf{ end}, (\rho,\sigma)\rangle \to \langle \text{Prog}_1, (M_1\rho M_1^\dagger, \sigma)\rangle}\ ,\ M_1 = \frac{I+\int}{2}$$

$$\text{(While -1)}\quad \frac{}{\langle \textbf{while } M[\int,\bar{q}] \textbf{ do } \text{Prog}_1 \textbf{ done}, (\rho,\sigma)\rangle \to \langle E, (M_0\rho M_0^\dagger, \sigma[-\sigma(\int)/\int])\rangle}$$

$$\text{(While 1)}\quad \frac{}{\langle \textbf{while } M[\int,\bar{q}] \textbf{ do } \text{Prog}_1 \textbf{ done}, (\rho,\sigma)\rangle \to \langle \text{Prog}_1; \textbf{while } M[\int,\bar{q}] \textbf{ do } \text{Prog}_1 \textbf{ done}, (M_1\rho M_1^\dagger, \sigma)\rangle}$$

Figure 1: Original semantic rules defined in *QECV-Lang*.

Here is an overview of how we implemented these rules in `CoQ`.

- Instead of having a separate empty program E, we instead reuse **skip** as the empty/terminal program. This does not affect the overall results of execution, but simply reduces one unnecessary rule.
- To initialize a qubit to a fixed value, we trace out the given qubit in the density mtarix to replace its state with the zero state, and update the qubit state matrix.
- To apply a unitary transformation, we perform multiplication of the current state ρ and the unitary matrix U to arrive at the new state ρ': $\rho' := U\rho U^\dagger$.
- To assign a stabilizer variable to a new value given an expression for the new value, we use the denotational semantics of the stabilizer expression language (below) to evaluate the expression given the current state of the program, and then update the stabilizer map to contain this new value for the given variable; this is very similar to the variable assignment rules of *IMP*.
- The rule for evaluation of a **Seq** of two instructions is identical to the evaluation rule for **Seq** in *IMP*: we either step the first subprogram once if it can step, and if it is **skip** we reduce the **Seq** to just the second subprogram and continue.
- The condition for a conditional statement (**if** and **while**) is a measurement of a stabilizer variable on the current state of the program to see if the qubitstate is in the $+1$ or -1 eigenstate of the stabilizer variable (which is some unitary matrix). The evaluation of **if** and **while** statements after resolving the condition matches the evaluation of their counterparts in *IMP* almost exactly.
  - If the program is in the $+1$ eigenstate, then no correction is required and the program proceeds down the "true" branch.
  - If the program is in the -1 eigenstate, then an error correction is applied to restore the qubit state and then the program proceeds down the "false" branch.
  - With **if** statements, the "true" branch corresponds to reducing to the first subprogram, and the "false" branch corresponds to reducing to the second program.
  - With **while** statements, the "true" branch converts the current program to a sequence of the body, followed by the **while** statement again, and the "false" branch reduces to **skip** immediately.

3.2. Denotational Semantics of Stabilizer Expressions

A stabilizer expression in this language consists of a few variants:

- An expression can be a fixed unitary (a string of Pauli matrices)
- A phase shift of another expression (multiplication by i, -1, or $-i$)
- A lookup of a stabilizer variable.

To evaluate these expressions to a stabilizer value, which is a final unitary, we use denotational semantics with the following cases: Given a stabilizer expression and a current stabilizer variable map,

- A fixed unitary expression evaluates to itself
- A phase shift of another expression evaluates to the denotation of that expression (recursively evaluated) multiplied by the phase shift of the expression
- A lookup of a stabilizer variable corresponds to a map lookup in the stabilizer map.

4. Conclusion and Future Work

In the future, we would like to include examples of how our algorithm works. The simplest encoding would be a three-bit repetition code, which contains one bit flip error to be detected and corrected by our system.

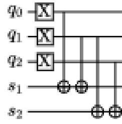

Figure 2: X error on qubit q_1

Currently, our system only supports two-dimensional qubits. Extending this to multidimensional qubits would require a second parameter and slightly longer proofs.

4.1. Notes on Compilation

Our setup runs on `CoQ` version `8.16`, `voqc` (SQIR) version `4.13.1` and `QuantumLib` version `1.1.0`. For `SQIR` and `QuantumLib` dependencies, we follow the instructions given in `README.md` [3], [4] . Our repository can be found here.

5. References

[1] A. Wu, G. Li, H. Zhang, G. G. Guerreschi, Y. Xie, and Y. Ding, “QECV: Quantum Error Correction Verification.” 2021.

[2] M. Lewis, S. Soudjani, and P. Zuliani, “Formal Verification of Quantum Programs: Theory, Tools, and Challenges,” *ACM Transactions on Quantum Computing*, vol. 5, no. 1, Dec. 2023, doi: 10.1145/3624483.

[3] SQIR Developers, “SQIR.” [Online]. Available: https://github.com/inQWIRE/SQIR

[4] The INQWIRE Developers, “INQWIRE QuantumLib.” [Online]. Available: https://github.com/inQWIRE/QuantumLib