Formal Verification of Gottesman Semantics

Jacob Zweifler
University of Chicago
jzweifler@uchicago.edu

Robert Rand
University of Chicago
rand@uchicago.edu

1 Introduction

In “Gottesman Types for Quantum Programs” [6], Rand et al. introduced a type system, based on the Gottesman-Knill theorem [3] for characterizing quantum circuits that use the Clifford gate set. While this system is *implemented* in the Coq proof assistant (allowing easy typechecking), it is not *verified* using that tool: There are no mechanized proofs that the types guarantee a program’s behavior. Instead, a user must convince herself that the theorems in the paper are true, that the type system reflects those theorems, and that the type system is faithfully implemented in Coq.

This work aims to patch these holes by directly providing a semantics for Gottesman types in Coq, and proving the necessary theorems from Gottesman and Rand et al. inside the proof assistant. This guarantees the type system actually enforces the desired properties, including specifying the set of input and outputs and guaranteeing separability. In the process, we allow further extensions to the type system, provided that those extensions are consistent with our semantics. This allows us to verify typing rules that go beyond the limited expressivity of our current rules and the limitations of the Clifford gate set (see Section 5).

The (in-progress) formalization is available at:
`https://github.com/inQWIRE/Heisenberg-Foundations`

2 Background: Quantum Circuits and Gottesman Types

Currently in the world of quantum computing, almost all programs can be described as a series of gate applications to a certain number of bits. For example, we have the superdense algorithm that acts on two qubits and two classical bits in an attempt to send two bits of information from Alice to Bob. For small scale algorithms such as these, it is not too difficult to examine the effect of gate applications on various states. However, when applying a larger number of gates on systems of size 2^n for large n, it may become more challenging to assess what the action of the gate application will be. Thus, an important problem in quantum computing is figuring out efficient ways of determining the effects of a program on an input state.

Gottesman’s paper ascribes what we interpret as a *type system* to the Clifford set of gates. This allows us to say things like $U : A \to B$, when U has certain effects on A and B. An important tool that Gottesman uses is the fact that we can learn everything about a program U by studying what it does to matrices of the form $I^{\otimes n} \otimes g \otimes I^{\otimes m}$, where $g \in \{X, Z\}$ (the Pauli gates minus Y). Thus, we can consider a type system with only three ground types, I, X, and Z. The type system then consists of many rules, all of which we would expect from the underlying linear algebra. For example, we have that $U_1 : A \to B$ and $U_2 : B \to C$ implies that $U_1; U_2 : A \to C$. With these rules, we can build efficient algorithms for determining the overall action of a program.

Submitted to:
QPL 2021

3 Syntax and Semantics for Gottesman Types

3.1 A Stratified Syntax for Gottesman Types

The actual typing grammar we use is fairly straightforward and is the same type system used by Rand et al [6]. As mentioned, we only need to consider three ground types, I, X, and Z. Naturally, we also want the other usual operations such as matrix multiplication, tensor products, and scalar multiplication. Since we want to consider types of programs, not just of states, we also want function types. Additionally, we want some notion of subtyping, i.e., intersection types. While not necessary, stratifying the type system helps to ensure that types are well defined (although there can still be ill-formed types even in the stratified syntax). Thus, all of this gives us the following grammar for our type system:

$$\begin{aligned} V &:= T \mid V \to V \mid V \cap V \\ T &:= G(\otimes G)^* \\ G &:= I \mid X \mid Z \mid -G \mid iG \mid G * G \end{aligned}$$

We also define programs as follows:

$$P := H\ n \mid T\ n \mid S\ n \mid CNOT\ n\ m \mid P;P$$

Where $H\ n$ denotes applying the H gate to the n-th bit, etc. So for example, the following is a valid typing statement for the CZ gate:

$$(H\ 1);\ (CNOT\ 0\ 1); (H\ 1)\ :\ X \otimes I \to X \otimes Z$$

.

3.2 The Heisenberg Semantics

Gottesman's analysis of quantum programs relies on the Heisenberg representation of quantum operations: For a unitary program U, matrix N, and state $|\psi\rangle$, we have that $UN|\psi\rangle = UNU^\dagger U|\psi\rangle$. Thus, when U acts on the state, it turns the action of N into an action of $UNU^\dagger$. Gottesman uses this to then say that U has type $N \to UNU^\dagger$. Thus, if we let $A = N$ and $B = UNU^\dagger$, the *Heisenberg semantics* of $U : A \to B$ is simply $UA = BU$, describing commutation rules. Thus, typing rules in the Heisenberg semantics are fairly easy to check, as we simply just need to multiply matrices and check if the two sides are equivalent. Unfortunately, this semantics only types terms of the form $A \to B$ and has no interpretation for intersection types: It does not make sense to say that $U(A_1 \cap A_2) = BU$.

3.3 The Eigenvector Semantics

In contrast to the Heisenberg semantics, the eigenvector semantics has a more extensive set of typing rules. Typing in the eigenvector semantics is defined as follows: For some vector v, and unitary matrix A, we say that $v : A$ if v is an eigenstate of A. Then, $U : A \to B$ if U takes all eigenvectors of A to eigenvectors of B (we also require that the eigenvalue stays the same). Intersection types make a lot of sense in this paradigm. For example, $U : A \cap B \to C$ simply means that if v is an eigenvector of A and B, then Uv is an eigenvector of C. Some useful examples of intersection types are the Bell state $(X \otimes X) \cap (Z \otimes Z)$ and the separable state $(Z \otimes I) \cap (I \otimes X)$. In practice, however, it is much harder to check typing statements with this paradigm since computing eigenvectors is much more computationally difficult than simply multiplying matrices. Fortunately, we prove in Coq that if U, A, and B are unitary, then this is equivalent to $UA = BU$. Thus, as long as we consider unitary matrices, we can show that the two different semantics are equivalent. This fact proved very useful in our verification of the type system.

4 Implementation in Coq

A significant chunk of this work involved creating and verifying the eigenvector semantics. In order to do this, we used the matrix library from the $\mathcal{Q}$WIRE programming language [5] and proceeded to create a more general grammar than the one above (since we allowed ground types to be any n-by-n matrix). $\mathcal{Q}$WIRE's matrix library contained all the basic linear algebra we needed, except for the core notion of eigenvectors. We defined the properties of eigenvectors and eigenvalues then proceeded to prove a variety of lemmas, verifying all the necessary aspects of the type system. As mentioned, it was crucial to show that both semantics discussed above are equivalent in order to prove all the necessary lemmas. For example, with the Heisenberg semantics, it is very straightforward to show that $U : A \to A'$ and $U : B \to B'$ implies $U : AB \to A'B'$. Showing this statement with the eigenvector semantics, however, is more difficult.

We also developed a symbolic representation that was more straightforward to formulate and use since we simply needed to define the syntactic grammar given in the prior section. In order to give the symbolic representation any meaning, we made various translation functions between the symbolic representation and the eigenvector representation. We then defined typing based on the eigenvector representation. In other words, p has type T if the analogous typing statement is true in the eigenvector paradigm. Then, by showing that the translation functions preserved all the operations, we were able to bootstrap the lemmas proven about the eigenvector representation in order to prove analogous lemmas for the symbolic representation. In this way, we were able to give meaning to the syntax.

There are various key benefits of the symbolic representation over the eigenvector representation. The restricted nature of both the programs and basic gate types made it possible to prove various lemmas which were unprovable in the eigenvector representation. For example, since programs consist entirely of unitary gate application, it can be shown that every program is unitary. Thus, it is truly the case that both semantics discussed above are equivalent for all gates in the symbolic representation. Another important reason a simplified symbolic representation is useful is because it is much easier to write tactics which operate on the syntactic types rather than actual matrices. This is possible in the simpler representation as we were more easily able to separate syntax from semantics. In this way, we were able to give the symbolic representation a relatively simple syntax, while still being able to easily typecheck our circuits.

5 Future Work

In the near future, we hope to capture the notion of separability present in the Gottesman types paper [6] and extend the system with the typing rules for measurement present in ongoing work [7]. We also hope to type-check universal quantum programs by adding the missing typing rule $T : X \to \frac{1}{\sqrt{2}}(X+Y)$. This will force us to recon with linear combinations of types and the potential for exponential blowup.

We also hope to implement and verify the CHP algorithm [1] to more efficiently typecheck circuits, and possibly Gidney's recent alternative, Stim [2]. In terms of applications, we plan to connect this work to the existing *s*QIR [4] and $\mathcal{Q}$WIRE [5] libraries, allowing for efficient verification of Clifford circuits. And once we can type T gate applications, we hope to explore the range of programs that can be characterized or verified efficiently within this framework.

References

[1] Scott Aaronson & Daniel Gottesman (2004): *Improved simulation of stabilizer circuits*. Physical Review A 70(5), p. 052328.

[2] Craig Gidney (2021): *Stim: a fast stabilizer circuit simulator*. arXiv preprint arXiv:2103.02202.

[3] Daniel Gottesman (1998): *The Heisenberg representation of quantum computers*. arXiv preprint quant-ph/9807006.

[4] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu & Michael Hicks (2021): *A verified optimizer for quantum circuits*. Proceedings of the ACM on Programming Languages 5(POPL), pp. 1–29.

[5] Jennifer Paykin, Robert Rand & Steve Zdancewic (2017): *QWIRE: a core language for quantum circuits*. ACM SIGPLAN Notices 52(1), pp. 846–858.

[6] Robert Rand, Aarthi Sundaram, Kartik Singhal & Brad Lackey (2020): *Gottesman Types for Quantum Programs*. In: Proceedings 17th International Conference on Quantum Physics and Logic (QPL 2020). Available at `https://people.cs.uchicago.edu/~rand/qpl_2020.pdf`.

[7] Robert Rand, Aarthi Sundaram, Kartik Singhal & Brad Lackey (2021): *Static Analysis of Quantum Programs via Gottesman Types*. arXiv preprint arXiv:2101.08939.