Formal Verification of Gottesman Semantics

Jacob Zweifler [1] Robert Rand [1]

[1]University of Chicago

The Gottesman Type System

When building a type system for our quantum programs, we need to include the following components:

- **Ground types** such as the identity gate and the X and Z rotation gates.
- **Gate operations** such as gate multiplication, gate tensoring, and scalar multiplication
- **Function types** in order to give programs types in addition to states
- **Intersection types** to give us a notion of subtyping

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:

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

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. We then interpret $P : V$ as being True iff P has type V. So for example, the following is a valid typing statement for the CZ gate:

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

.

Typing rules

We formalize the typing rules of Rand *et al.* [5]. A subset of the rules is listed below:

Ground-Type Rules:

$$\frac{}{H\ 0 : (\mathbf{X} \to \mathbf{Z}) \cap (\mathbf{Z} \to \mathbf{X})} \qquad \frac{}{CNOT\ 0\ 1 : (\mathbf{X} \otimes \mathbf{I} \to \mathbf{X} \otimes \mathbf{X})}$$

Tensor Rules:

$$\frac{g\ 0 : A \to A' \qquad |E| = m - 1}{g\ m : E \otimes A \otimes E' \to E \otimes A' \otimes E'}\ \otimes_0 \qquad \frac{g\ 0\ 1 : A \otimes B \to A' \otimes B'}{g\ 1\ 0 : B \otimes A \to B' \otimes A'}\ \otimes\text{-REV}$$

Arrow and Sequence Rules

$$\frac{p : A \to A' \qquad p : B \to B'}{p : (A * B) \to (A' * B')}\ \text{MUL} \qquad \frac{p : A \to A'}{p : iA \to iA'}\ \text{IM}$$

$$\frac{p_1 : A \to B \qquad p_2 : B \to C}{p_1; p_2 : A \to C}\ \text{CUT} \qquad p_1; (p_2; p_3) \equiv (p_1; p_2); p_3$$

Intersection Rules

$$\frac{g : A \qquad g : B}{g : A \cap B}\ \cap\text{-I} \qquad \frac{g : A \cap B}{g : A}\ \cap\text{-E}$$

$$\frac{g : (A \to B) \cap (A \to C)}{g : A \to (B \cap C)}\ \cap\text{-ARR} \qquad \frac{g : (A \to A') \cap (B \to B')}{g : (A \cap B) \to (A' \cap B')}\ \cap\text{-ARR-DIST}$$

Eigenvector Semantics

The eigenvector semantics is determined by the eigenstates of various gates and the effects of of programs on the eigenstates of these gates. There are thus two different notions of typing depending on whether we are considering states themselves or entire programs:

- **State Typing:** States are typed by the unitaries for which they are eigenvectors of. So if $|\psi\rangle$ is an eigenvector of A, we say that $|\psi\rangle$ has type A, or simply $|\psi\rangle : A$.
- **Program Typing:** Programs are typed based on where they send states. So a program p has type $A \to B$ if p sends all eigenstates of A to eigenstates of B (we also require that the eigenvalue stays the same). This is written $p : A \to B$.
- **Intersection Types:** One benefit of the eigenvector semantics is that it allows for program subtyping. Thus, we can say things like $p : T_1 \cap T_2$ to mean that p has type T_1 and T_2. Even more powerfully, we can consider when the input or output type of a program are intersections. Thus, the statement $p : A \cap B \to C$ means that p takes eigenstates of A and B to eigenstates of C, still requiring that the eigenvalue is the same.

Examples

1. For all states $|\psi\rangle$, we have that $|\psi\rangle : I$ (Top Rule)
2. $|0\rangle, |1\rangle : \mathbf{Z}$ and $|+\rangle, |-\rangle : \mathbf{X}$ (Basis States)
3. $|\Phi^+\rangle : (\mathbf{X} \otimes \mathbf{X}) \cap (\mathbf{Z} \otimes \mathbf{Z})$ (Bell Pair)
4. $|\psi\rangle \otimes |\phi\rangle : A \otimes B$ whenever $|\psi\rangle : A$ and $|\phi\rangle : B$ (Tensor Rule)
5. $H : (\mathbf{X} \to \mathbf{Z}) \cap (\mathbf{Z} \to \mathbf{X})$ (H Gate)
6. $CNOT : (\mathbf{X} \otimes \mathbf{I} \to \mathbf{X} \otimes \mathbf{X}) \cap (\mathbf{I} \otimes \mathbf{X} \to \mathbf{I} \otimes \mathbf{X}) \cap (\mathbf{Z} \otimes \mathbf{I} \to \mathbf{Z} \otimes \mathbf{I}) \cap (\mathbf{I} \otimes \mathbf{Z} \to \mathbf{Z} \otimes \mathbf{Z})$ (CNOT Gate)

Heisenberg Semantics

Gottesman's analysis [3] 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 operator N into the operator $UNU^\dagger$. Gottesman uses this to then say that U has type $N \to UNU^\dagger$. More generally, we get that U has type $A \to B$ if $UA = BU$.

It is a mathematical fact that when A, B, and U are unitary (as is always the case in quantum computing) we have that U sends eigenvectors of A to eigenvectors of B precisely when $UA = BU$ (as usual, we require that the eigenvalue is unchanged). Therefore, disregarding intersection types, the *Heisenberg semantics* exactly aligns with the eigenvector semantics. Considering intersection types, however, the *Heisenberg semantics* is less strong because it does not provide an interpretation for statements like $U(A_1 \cap A_2) = BU$.

Why Do We Need a Type System?

A type system may be useful for the following reasons:

- We can classify programs based on their type which is more general than looking at their specific effect
- Using the typing rules, we can build efficient algorithms for determining the overall action of a program
- It gives us a new way of thinking about quantum programs

Implementation in the Coq Proof Assistant

A significant chunk of this work involves creating and verifying the eigenvector semantics. In order to do this, we use a mathematical representation of states, gates, and programs. To keep the complicated linear algebra out of the way of program verification, we also developed a symbolic representation that was more straightforward to formulate and use. Then, we can deduce that p has type T when the analogous statement is true in the mathematical representation.

Mathematical Representation

We use the matrix library from the $\mathcal{Q}$wire programming language [4] and proceed to create a more general grammar than the one above (since we allow ground types to be any n-by-n matrix).

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.

Symbolic Representation

We define the syntactic grammar based on the grammar rules listed on the left. In order to give the symbolic representation any meaning, we use various translation functions between the representations. We then define typing based on the mathematical representation. In this way, we were able to give meaning to the syntax.

Future Possibilities

Given what we have so far, there is much room for moving forwards, both in terms of applications and expanding the type system:

1. We hope to capture the notion of separability present in the Gottesman types paper [5] and extend the system with the typing rules for measurement present in ongoing work [6].
2. 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.
3. In terms of applications, we plan to connect this work to the existing $\mathcal{Q}$wire [4] 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.
4. We also hope to implement and verify the CHP algorithm [1] to more efficiently typecheck circuits, and possibly Gidney's recent alternative, Stim [2].

References

- Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5):052328, 2004.
- Craig Gidney. Stim: a fast stabilizer circuit simulator. *arXiv preprint arXiv:2103.02202*, 2021.
- Daniel Gottesman. The heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*, 1998.
- Jennifer Paykin, Robert Rand, and Steve Zdancewic. Qwire: a core language for quantum circuits. *ACM SIGPLAN Notices*, 52(1):846–858, 2017.
- Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. Gottesman types for quantum programs. In *Proceedings 17th International Conference on Quantum Physics and Logic (QPL 2020)*, Jun 2020.
- Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. Static analysis of quantum programs via gottesman types. *arXiv preprint arXiv:2101.08939*, 2021.