Tracking Errors through Types in Quantum Programs

Kesha Hietala Robert Rand Michael Hicks

University of Maryland, College Park, USA
{kesha, rrand, mwh}@cs.umd.edu
December 9, 2019

Abstract

Errors will be prevalent in near-term quantum computing and need to be taken into account when designing quantum algorithms. It is thus reasonable to provide language-level support for reasoning about errors in quantum programs, in the style of existing tools for classical languages [1, 2, 3]. This extended abstract describes a simple type-based approach to reasoning about fault tolerance in quantum programs. Our implementation extends QWIRE [4] and is available at `https://github.com/inQWIRE/QWIRE/tree/error_wires`.

Background. In order to use quantum computers for meaningful computation, we need to protect computations against noise. One way to do this is to use quantum error correcting codes. In a $[\![n,\ k,\ 2t+1]\!]$ quantum error correcting code, k *logical* qubits are encoded using a block of n *physical* qubits, and errors on up to t physical qubits within a block can be corrected. We call t the *threshold* of the code. For simplicity, we will only consider the case where $k = 1$ (i.e. a block encodes one logical qubit – so we call it an *encoded qubit*). Encoded qubits are manipulated using *fault-tolerant* operations. An operation is fault-tolerant if, given a single fault in one block of the input, the operation will propagate at most one error to each block of the output.

The simplest example of a quantum error correcting code is the 3-qubit repetition code, which is the quantum equivalent of a classical repetition code. Recall that in the classical 3-bit repetition code, 0 is encoded as 000 and 1 is encoded as 111. Error correction is performed by taking a majority vote of the three bits (e.g. MAJ(0,0,1) = 0, so 001 will be corrected to 000). Equivalently, errors can be detected using two *parity checks* that indicate whether the first and second, and second and third, bits agree. This information can then be used to perform correction. For instance, if the first and second bits do not agree, but the second and third bits do, then the first bit must be in error.

In the quantum case, we can construct a code that corrects a single bit flip (i.e., an X error) by encoding the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ as $|\overline{\psi}\rangle = \alpha|000\rangle + \beta|111\rangle$. Error correction is performed using the results of *syndrome measurements*, which are similar to parity checks. An example of a fault-tolerant operation for this code is $X \otimes X \otimes X$, which performs the logical not operation. In the standard model of fault-tolerant computation, error correction is applied after every fault-tolerant gate. The cost of fault-tolerant computation can be reduced by only applying error correction where errors are most likely to have occurred [5, 6]. For more detail about quantum fault tolerance and error correcting codes, we recommend Gottesman [7] or Chapter 10 of Nielsen and Chuang [8].

Overview. In this paper we propose to abstract away the details of how error correction and fault tolerance are implemented, and instead assume the existence of a set of fault-tolerant operations that can introduce up to a fixed number of errors. We present a language for quantum programming where the `Qubit` type represents encoded qubits, and an error correction operation is available. The language's type system checks whether all potential errors in the program will be successfully corrected: Well-typed programs will be error free.

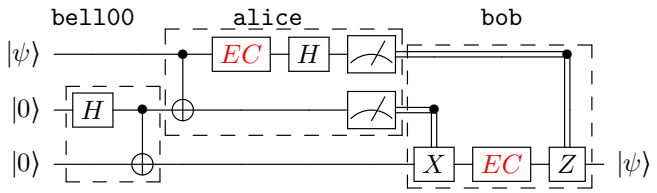

Figure 1: Teleportation circuit with included error correction.

Consider the circuit in Figure 1 (the corresponding program is shown in Figure 2). Assume our error-correcting code can correct up to three errors and that state preparation, measurement, and error correction are perfect, while unitary application may introduce up to one error. Our goal is to determine whether the error correction operations that we have added will successfully counteract potential errors. To do this, we track how errors propagate through the circuit and ensure that the number of accumulated errors in each encoded qubit does not exceed the threshold.

We can see that the circuit above is "correct" because there is no point at which the potential number of errors in an encoded qubit is above three. In the `bell00` sub-circuit, both qubits start with zero errors and the H gate introduces up to one error on the first qubit. Then the $CNOT$ gate propagates the error to both qubits and adds another potential error. In the `alice` sub-circuit, the first $CNOT$ gate now emits up to three errors on both qubits, so error correction must be applied to the first qubit before another gate can be safely applied. After the qubits are measured, their error disappears (error is not tracked for the classical bit type, which we assume is stored on classical hardware). In the `bob` sub-circuit, the input wire has up to two errors accumulated, and the X gate potentially adds a third, so error correction must be applied before the final Z gate. In the end, the output has up to one accumulated error.

Syntax, Semantics, and Typing. We implement error types in the linearly-typed QWIRE circuit language [4, 9], which is embedded in the Coq proof assistant [10]. We directly use QWIRE's syntax, which allows gate application, sequencing, and dynamic lifting (measurement and return of control to a classical computer). We add to QWIRE an error correction gate EC.

With a fixed error correcting code, we interpret QWIRE's `Qubit` type as an encoded qubit and assume that all gates are fault-tolerant. We associate with every gate a natural number n corresponding to the maximum number of errors that it can introduce.

To define fault tolerance we follow the presentation by Gottesman [7, Section 4.2]. First, fix a threshold t, which is the maximum number of errors that the code can correct. Then assume an *ideal decoder*, which takes an encoded state, corrects any errors, and returns an unencoded state (though not necessarily the "correct" state if too many errors have occurred). The basic requirements for a fault-tolerant operation are that it does not propagate too many errors and that it performs the desired ideal operation. In particular:

- A measurement operation is fault tolerant if it is equivalent to ideal decoding followed by ideal measurement, provided that the total number of errors in the incoming state and measurement operation is at most t.[1]

- A state preparation operation is fault tolerant if (i) the operation introduces at most t errors and (ii) state preparation followed by ideal decoding is equivalent to ideal state preparation.

- A unitary operation is fault tolerant if (i) the number of errors on every output after the operation is at most the sum of the errors introduced by the operation and the errors on all inputs *and* this quantity is at most t and (ii) unitary application followed by ideal decoding is equivalent to ideal decoding followed by the ideal unitary operation.

- An error correction operation is fault tolerant if (i) the output of error correction has at most t errors and (ii) error correction followed by ideal decoding is equivalent to performing ideal decoding, provided

that the sum of the errors in the input state and error correction operation is at most t.

Given the definitions above, we see that we can define the semantics of fault-tolerant programs in terms of the *ideal semantics* given in [4] so long as every "is at most t" condition is satisfied. If a program satisfies all "is at most t" conditions, then we will call it well-typed.

Implementation in $\mathcal{Q}$wire. $\mathcal{Q}$WIRE required relatively few modifications to support error tracking. The main change was adding additional parameters to some types to store error information. For example, we added a natural number parameter to the `Qubit` type representing the number of errors in a block, and a parameter to `Gate` specifying the number of errors introduced by that gate. We show part of the updated `Gate` definition below. `Gate k` W_1 W_2 describes a gate that takes input of type W_1 to output of type W_2 and introduces at most k errors.

```
Inductive Gate : ℕ → WType → WType → Set :=
  | U       : ∀ {W k} (u : Unitary k W),
                Gate k W (map_wtype W (k + (num_errs_wtype W)))
  | init₀   : Gate 0 One (Qubit 0)
  | meas    : ∀ {n}, Gate 0 (Qubit n) Bit
  | EC      : ∀ {n}, Gate 0 (Qubit n) (Qubit 0)
  ...
```

The most interesting rule in the definition above is the `U` rule for unitary gate application. In this rule, `num_errs_wtype` counts the total number of errors in a pattern (e.g. `Qubit` 1 ⊗ `Bit` ⊗ `Qubit` 2 has three errors) and `map_wtype` updates the error of every qubit in a pattern (e.g. we might update the previous example to be `Qubit` 3 ⊗ `Bit` ⊗ `Qubit` 3). These functions allow us to express that the error on every qubit after a unitary gate is at most the sum of the error introduced by the gate and the error on the input. In our definition, initialization, measurement, and error correction do not introduce any errors, and error correction produces a qubit with no errors. However, our definition could easily be updated to include errors in these operations. With our updated `Gate` type, the Coq type system is able to infer the number of errors on the outputs of a circuit, given errors for the inputs. This already provides the programmer with useful information. However, it does not enforce that errors do not get too large, which is our primary goal.

To address this goal, we extend $\mathcal{Q}$WIRE's linear type checking system (which is external to Coq's type system) to enforce that, for every gate application, the sum of the error introduced by the gate and the error on the input pattern does not exceed the specified threshold. Proofs about $\mathcal{Q}$WIRE well-typedness are (for the most part) automated, and adding automation for our new definition was easy because Coq has well-developed support for arithmetic and inequalities over natural numbers. Note that our typing rules augment $\mathcal{Q}$WIRE's existing linear type system, so programs that type check are guaranteed to use wires linearly and to satisfy our requirement of fault tolerance.

As a gentle test of our implementation, we have verified that we are able to type check the teleport example in Figure 2 using both Coq's built-in type system and $\mathcal{Q}$WIRE's type system. There the Coq type system guarantees that every function produces the specified number of errors, given the specified input errors, and `type_check` produces an automated proof that the error threshold is never passed.

Future Directions. This project is a work in progress with many avenues left to explore. Some interesting potential directions follow:

- We can extend our current type system with probabilities so that rather than checking whether all qubits have less than t errors, we check whether all qubits have less than t errors with probability at least p. This requires an error model that bounds the probability that each gate will introduce a certain number of errors. In this scenario, the type of a circuit might look like `Box` ([0,1]`Qubit` ⊗ [0,1]`Qubit`) ([0,0.996]`Qubit` ⊗ [1,0.998]`Qubit`), which says that given error-free inputs, the first output will

[1] When we say that two quantum operations are *equivalent* we mean that they are denoted by the same superoperator, and thus correspond to the same physical process.

```
Definition bell :
  Box One (Qubit 2 ⊗ Qubit 2) :=
  box () ⇒
    let a ← init₀ ();
    let b ← init₀ ();
    let a ← H a;
    let (a,b) ← CNOT (a,b);
    (a,b).

Definition alice :
  Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit) :=
  box qa ⇒
    let (q,a) ← CNOT qa;
    let q     ← EC q;
    let x     ← meas H q;
    let y     ← meas a;
    (x,y).
```

```
Definition bob :
  Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1) :=
  box (x,y,b) ⇒
    let (y,b)     ← bit_ctrl X (y,b);
    let b         ← EC b;
    let (x,b)     ← bit_ctrl Z (x,b);
    discard (x,y) ;
    b.

Definition teleport :
  Box (Qubit 0) (Qubit 1) :=
  box q ⇒
    (a,b) ← bell ();
    (x,y) ← alice  (q,a);
    bob (x,y,b).

Lemma teleport_WT : Typed_Box teleport 3.
Proof. type_check. Qed.
```

Figure 2: $\mathcal{Q}$WIRE teleport example.

have no accumulated errors with probability at least 0.996, and the second output will have at most one accumulated error with probability at least 0.998.

- We could also consider a type system where the qubit type represents physical qubits as opposed to encoded qubits, and error is tracked via *fidelity*. The idea is that each operation will increase the error of the system by some amount, and the analysis needs to prove that the final error is below some threshold ϵ. This is similar to the upper-bound analysis used in [11]. Note that in this case it doesn't make sense to talk about the probability of error on a single qubit, like a classical error tracking system would, because qubits may be entangled. Error becomes a "global" property.

  A downside of this approach is that it provides no direct support for error correction. From the perspective of the analysis, operations done to correct errors only introduce additional errors. (Note that [11] needed to reason about error correction directly using the semantics, outside of the their robustness logic.)

- Rather than assume the existence of fault tolerant operations, we could start from a basic error model (e.g. errors occur on independently on physical qubits with a fixed probability), define specific encoding and decoding operators, and then prove that the corresponding fault-tolerant operations satisfy our desired criteria. Because $\mathcal{Q}$WIRE is entirely within Coq, we could do this within a formally verified environment. We expect this to be doable, though challenging due to the complexity of quantum error correction and reasoning about complex matrices in Coq.

References

[1] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain⟨T⟩: A first-order type for uncertain data," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[2] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, 2013.

[3] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker, "Fault-tolerant typed assembly language," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[4] 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*, POPL '17, 2017.

[5] R. Majumdar, S. Basu, and S. Sur-Kolay, "A method to reduce resources for quantum error correction," in *Reversible Computation*, 2017.

[6] M. G. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz, "A fault tolerant, area efficient architecture for Shor's factoring algorithm," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[7] D. Gottesman, "An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation," *arXiv e-prints*, p. arXiv:0904.2557, Apr 2009.

[8] M. A. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[9] R. Rand, *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania, 2018.

[10] Coq Development Team, "The Coq proof assistant reference manual, version 8.9," 2019. Electronic resource, available from `https://coq.inria.fr/refman/`.

[11] S.-H. Hung, K. Hietala, S. Zhu, M. Ying, M. Hicks, and X. Wu, "Quantitative robustness analysis of quantum programs," in *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '19, 2019.