Hoare Types for Quantum Programming Languages

POPL 2020 *Student Research Competition* Extended Abstract

KARTIK SINGHAL, University of Chicago, USA

1 PROBLEM AND MOTIVATION

It is difficult to reason about the correctness of quantum programs. Sound static type systems help prevent a huge class of bugs from occurring but since the realm of quantum programming is still new there is not a lot of consensus on what kind of types make the most sense. Further, it is unclear how much they help programmers reason about the semantic properties associated with the quantum algorithms that they are implementing.

Recent work [5, 6] as part of EPiQC[1] has identified several classes of bugs in quantum programs and proposed approaches to tackle them. The technique that holds the most promise is assertion checking using preconditions and postconditions. But assertions are usually checked dynamically during runtime, which can be wasteful of precious quantum computing or simulation resources.

As programming languages researchers, we think it will be more useful to encode such assertions into a static type system both for formal verification and to aid the programmers in writing correct programs from the start. Inspired by the use of Hoare triples in the verification of imperative programs and building on the idea of Hoare Types in classical programming languages [9], we propose Quantum Hoare Types aimed at enabling both sound static type checking and formal verification of quantum programs.

2 BACKGROUND AND RELATED WORK

2.1 Quantum Computing

The basic unit of information in quantum computing is a qubit, that can be represented using a unit vector in a 2-dimensional complex vector space. Using Dirac's bra-ket notation, we can write an arbitrary qubit, $|\psi\rangle$, as a linear combination of the computational basis vectors: $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $|0\rangle = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and $|1\rangle = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$, and the two constants are complex amplitudes that are normalized using the constraint $|\alpha|^2 + |\beta|^2 = 1$. This means that when measured the probability of the qubit being in the state 0 is $|\alpha|^2$ and that for the state 1 is $|\beta|^2$. To work with multiple qubits, we need to take their tensor product, denoted by $\otimes$. Computation is performed by the application of unitary quantum gates, such as Hadamard (H) or controlled NOT (CX), that can be represented as square matrices. Mathematically, a gate application is equivalent to multiplying the matrix representing the gate with the state vector of the qubit(s).

This basic unit of information is not the only reason why quantum computing is fundamentally different from classical computing—certain quantum mechanical phenomenon are highly counter-intuitive! One such phenomenon is *quantum entanglement* which, at its most basic level, involves a correlation between two qubits such that when one is measured the outcome obtained necessarily influences the measurement outcome of the other qubit (even if the two qubits are far apart). We will see how this can be utilized for computation in the quantum teleportation algorithm that we describe and program in §3.1. We refer the reader to the standard textbook [10] or Nielsen's excellent series of essays [8] for more background.

[1] EPiQC: Enabling Practical-Scale Quantum Computation: https://epiqc.cs.uchicago.edu

Author's address: Kartik Singhal, Department of Computer Science, University of Chicago, USA, ks@cs.uchicago.edu.

2.2 Other Attempts

Previous work such as Proto-Quipper [7, 15, 16] and $\mathcal{Q}$WIRE [11–13] utilize a linear type system and dependent types to enforce a small subset of semantic properties, such as the no-cloning theorem and whether a unitary gate is of the right dimension. These advances in quantum type systems, although helpful, still fall short in encoding and enforcing even more useful properties that one would like to be able to express for the purpose of verification.

Our approach builds upon previous work in reasoning about quantum programs such as Quantum Weakest Preconditions [1] and Quantum Hoare Logic [18] in the spirit of Hoare [4] and Dijkstra [2] but attempts to bring those reasoning techniques into the type system. The hope is that programmers will be able to encode some of the semantic properties that they expect of their programs as specifications in their code and type checking will ensure correctness of some of those properties. In the classical setting, Hoare Type Theory (HTT) [9] accomplishes exactly this goal. Here, we attempt to merge these ideas for the quantum realm.

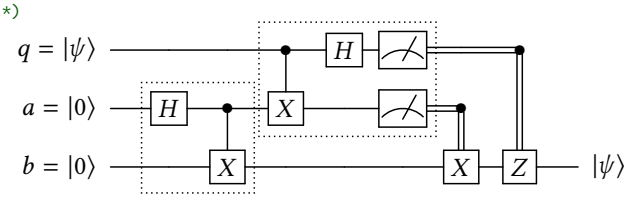
3 RESULTS AND CONTRIBUTIONS

In this section we describe Quantum Hoare Types using an example of programming quantum teleportation protocol. We start with the background necessary to understand teleportation and then present how Quantum Hoare Types help us write and verify the correctness of the protocol.

3.1 Example: Quantum Teleportation

Teleportation is a simple quantum algorithm that is used to communicate a quantum state using two classical bits and a shared entangled pair of qubits. Alice is trying to send a message encoded in the qubit to Bob. There are three broad steps in the protocol: 1) preparation, ie, entanglement creation between two qubits to be shared between Alice and Bob in advance; 2) sending the message by Alice; and, 3) reconstruction of the message by Bob. Program and circuit follow:

```
teleport : Π|ψ⟩:qbit.{q ↦qbit |ψ⟩ ∧ a ↦qbit |0⟩ ∧ b ↦qbit |0⟩}
      c:Circuit(qbit⊗qbit⊗qbit, qbit) {b ↦qbit |ψ⟩} =
  box (q, a, b) ⇒
    (* 1. Bell state preparation for Alice and Bob *)
    a       ← H a;
    (a, b)  ← CX (a, b);
    (* 2. Bell measurement by Alice *)
    (q, a)  ← CX (q, a);
    q       ← H q;
    x       ← meas q;
    y       ← meas a;
    (* 3. Correction by Bob *)
    (y, b)  ← CX (y, b);
    (x, b)  ← CZ (x, b);
    output b
```



Here we show a function `teleport` that implements the quantum teleportation circuit. It returns one qubit and takes three qubits as arguments: one qubit `q` that could be in any unknown state $|\psi\rangle$ which Alice is trying to send to Bob; and two qubits, `a` and `b`, initially in the state $|0\rangle$. The function first entangles the latter two qubits into a Bell state. Then, Alice performs a Bell measurement of her two qubits `q` and `a` and sends the resulting classical bits (`x` and `y`) to Bob. Finally, Bob applies corrective measurement using the quantum gates X and Z controlled by the classical bits that he received from Alice. The result is that Bob's qubit is now in the same state as the initial state of the message qubit `q`. The specification that the function must meet is written using a Hoare type, which we describe next.

3.2 Types

A Hoare type encodes preconditions and postconditions in the same spirit as Hoare triples. The type shown on the left below can be read as 'for a stateful computation executed in a heap that satisfies precondition P, return a value of type A in a heap that satisfies postcondition Q.' We show an example, the `alloc` primitive from HTT [9], to demonstrate the expressiveness of a Hoare type:

$$\{P\}\ \mathtt{x} : A\ \{Q\} \qquad\qquad \mathtt{alloc} : \forall\alpha.\Pi x : \alpha.\{\mathtt{emp}\}\ y : \mathtt{nat}\ \{y \mapsto_\alpha x\}$$

In order to extend HTT to support quantum circuits, we augment its syntax with that of QWIRE [11, 12]. QWIRE is a core quantum circuit language that can cooperate with an arbitrary classical host language. This requires minimal support from the host language to interface with the circuit language. In our case, HTT acts as the classical host language for QWIRE.

QWIRE's circuit language has a wire type W that can be a unit, a bit, a qubit or a tuple of wires. The host language (HTT) is augmented with additional types to support interaction between the two languages and to treat circuits as data:

$$W ::= 1 \mid \mathtt{bit} \mid \mathtt{qbit} \mid W1 \otimes W2 \qquad\qquad A ::= \ldots \mid \mathtt{Unit} \mid \mathtt{Bool} \mid A \times A \mid \mathtt{Circuit}(W1, W2)$$

3.3 Verification

Here we describe how HTT lets us verify the correctness of teleportation based on the specification provided as the type of the function. Type checking in HTT involves generation of strongest postconditions at each step of the program with the initial precondition as input. Verification is a two-step process: the first phase does basic type checking and verification condition generation which is decidable, the second phase needs to show the validity of the generated verification conditions, which is undecidable. This second phase can be deferred to an automated theorem prover. For illustration, we show the assertions as we step through step 1 of teleportation. The final state here is that qubits `a` and `b` are entangled into the first Bell state, $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

```
(* P0 (given): {q ↦qbit |ψ⟩ ∧ a ↦qbit |0⟩ ∧ b ↦qbit |0⟩} *)
a      ← H a;
(* P1: {q ↦qbit |ψ⟩ ∧ a ↦qbit 1/√2(|0⟩+|1⟩) ∧ b ↦qbit |0⟩} *)
(a, b) ← CX (a, b);
(* P2: {q ↦qbit |ψ⟩ ∧ (a,b) ↦qbit |ϕ+⟩ *)
```

4 FURTHER WORK

This approach has the potential to be a unified system for programming, specification and reasoning about quantum programs. We need to overcome challenges such as compactly specifying properties of quantum state when dealing with more than a few qubits. We also need to maintain the syntax directed nature of HTT's bidirectional typechecking while incorporating quantum wire types.

We are working on programming other interesting quantum algorithms that can serve as a litmus test for this technique, specifically, a cryptographic protocol called Quantum Key Distribution and the well-known Grover's search algorithm. These examples involve some form of classical control or iteration that makes them non-trivial for our technique.

We also look to explore how our types will evolve when we start incorporating higher-order constructs in our quantum language such as those demonstrated in languages like Quantum λ-calculus [17] and Quipper [3].

Another venue for exploration is to incorporate more precise types that can distinguish between qubits in pure classical state vs. those in superposition vs. those in entanglement inspired by the various quantum resource theories (see [14] and references therein).

REFERENCES

[1] Ellie D'hondt and Prakash Panangaden. 2006. Quantum Weakest Preconditions. *Mathematical Structures in Computer Science* 16, 3 (June 2006), 429–451. https://doi.org/10/c46pd2

[2] Edsger W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, USA.

[3] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 333–342. https://doi.org/10/gf8t6q

[4] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10/b2q64c

[5] Yipeng Huang and Margaret Martonosi. 2018. QDB: From Quantum Algorithms Towards Correct Quantum Programs. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:14. https://doi.org/10/gf8t75

[6] Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA 2019)*. ACM, New York, NY, USA, 541–553. https://doi.org/10/gf8t77

[7] Mohamed Yousri Mahmoud and Amy P. Felty. 2019. Formalization of Metatheory of the Quipper Quantum Programming Language in a Linear Logic. *Journal of Automated Reasoning* 63, 4 (Dec. 2019), 967–1002. https://doi.org/10/ggb9c6

[8] Andy Matuschak and Michael A. Nielsen. 2019. Quantum Computing for the Very Curious (and other essays). https://quantum.country

[9] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming* 18, 5-6 (Sept. 2008), 865–911. https://doi.org/10/bsf5mn

[10] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, Cambridge, UK.

[11] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 846–858. https://doi.org/10/gf8t6s

[12] Robert Rand. 2018. *Formally Verified Quantum Programming.* PhD Thesis. University of Pennsylvania, Philadelphia, PA, USA. https://repository.upenn.edu/edissertations/3175

[13] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic (QPL 2017)*. Open Publishing Association, Waterloo, NSW, Australia, 119–132. https://doi.org/10/gf8skv

[14] Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. 2019. A Type System for Quantum Resources. (Oct. 2019). http://ks.cs.uchicago.edu/publication/quantum-resource-types/ (Extended Abstract, in submission).

[15] Francisco Rios and Peter Selinger. 2017. A Categorical Model for a Quantum Circuit Description Language (Extended Abstract). In *Proceedings 14th International Conference on Quantum Physics and Logic (QPL 2017)*. Open Publishing Association, Waterloo, NSW, Australia, 164–178. https://doi.org/10/gf8skw

[16] Neil J. Ross. 2015. *Algebraic and Logical Methods in Quantum Computation.* PhD Thesis. Dalhousie University, Halifax, Canada. http://arxiv.org/abs/1510.02198

[17] Peter Selinger and Benoît Valiron. 2006. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science* 16, 3 (June 2006), 527–552. https://doi.org/10/cb9gnr

[18] Mingsheng Ying. 2012. Floyd–hoare Logic for Quantum Programs. *ACM Trans. Program. Lang. Syst.* 33, 6 (Jan. 2012), 19:1–19:49. https://doi.org/10/fxjg9k