Quantum Hoare Type Theory: Extended Abstract

Kartik Singhal John Reppy
University of Chicago
{ks, jhr}@cs.uchicago.edu

As quantum computers become real, it is high time we come up with effective techniques that help programmers write correct quantum programs. In classical computing, formal verification and sound static type systems prevent several classes of bugs from being introduced. There is a need for similar techniques in the quantum regime. Inspired by Hoare Type Theory [NMB08] in the classical paradigm, we propose Quantum Hoare Types by extending the Quantum IO Monad [AG09] by indexing it with pre- and postconditions that serve as program specifications. In this paper, we introduce Quantum Hoare Type Theory (QHTT), present its syntax and typing rules, and demonstrate its effectiveness with the help of examples.

QHTT has the potential to be a unified system for programming, specifying, and reasoning about quantum programs. This is a work in progress.[1]

1 Introduction

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 [HM18, HM19] as part of EPiQC[2] 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 better 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 Type Theory (HTT) for classical programming languages [NMB08], we extend the Quantum IO Monad interface [AG09] and propose Quantum Hoare Type Theory aimed at enabling both sound static type checking and formal verification of quantum programs.

The main idea is that instead of just indexing the QIO monad type with the type of computation result, we can also index it with preconditions and postconditions so as to integrate Hoare-style reasoning into the type system itself. The resulting Quantum Hoare Type, $\{P\}x{:}A\{Q\}$, specifies the preconditions, P, that hold on the quantum state before execution; the result, x, and its type, A; and, the postconditions, Q, that are true for the quantum state after successful execution. In this way the effectful quantum fragment of the program is effectively encapsulated inside the Quantum Hoare monad. Our theory, like HTT, allows the usual equational reasoning for the pure classical fragments of the program and uses syntax-directed type checking for generating strongest postconditions for the quantum fragment.

[1] An updated report is available [Sin20] at the time of this publication.

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

B. Valiron, S. Mansfield, P. Arrighi and P. Panangaden
(Eds.): Quantum Physics and Logic 2020 (QPL'20)
EPTCS 340, 2021, pp. 291–302, doi:10.4204/EPTCS.340.15

This paper is organized as follows. We discuss some related work and background in the next section, §2; specifically, Hoare Type Theory in the classical setting (§2.1) and then the Quantum IO Monad interface (§2.2). Then we introduce our contributions in merging these ideas together for specification and enforcement of useful semantic properties in section §3 along with its syntax (§3.1), typing rules (§3.2) and, some examples (§3.3). Finally, we conclude and share ideas for future work in the final section (§4).

2 Related Work and Background

Previous work, such as Proto-Quipper [Ros15, RS17] and QWire [PRZ17, RPZ18, Ran18], utilize a linear type system and dependent types to enforce a small subset of semantic properties, such as the no-cloning restriction 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 [DP06] and Quantum Hoare Logic [Yin12] in the spirit of Hoare [Hoa69] and Dijkstra [Dij76] 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 [NMB08] accomplishes exactly this goal. Our attempt is to merge these ideas for the quantum realm.

In the rest of this section, we provide background on the core ideas from existing literature that form the foundation for our work — Hoare Type Theory (HTT) and the Quantum IO Monad interface. We assume background in basics of quantum computing and refer the reader to Nielsen's excellent series of essays [MN19] for a first introduction and to the standard textbook [NC10] for advanced material.

2.1 Hoare Type Theory

A Hoare type, $\Delta.\Psi.\{P\}\ \mathtt{x} : A\ \{Q\}$, encodes preconditions and postconditions in the same spirit as Hoare triples to allow both specification and verification of effectful classical programs. It 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.' The contexts, Δ and Ψ, contain the variables and heap variables that may appear in both P and Q. In this presentation, we will omit the contexts in the Hoare type when they are unneeded. We show an example to demonstrate the expressiveness of a Hoare type — the `alloc` primitive from HTT [NMB08]:

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

This type specifies that `alloc` is a polymorphic function that takes as input a variable, x, of any simple type, α, that is executed in an empty heap (meaning it does not affect existing heap), returns a new location bound by a fresh variable, y, of type `nat` and initializes it with the supplied value x of type α.

Type checking in HTT involves generation of strongest postconditions at each step of the program. 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 can be undecidable. This second phase can be deferred to an automated theorem prover.

We provide more details when we introduce QHTT in section §3. An accessible introduction to Hoare Type Theory is available in the lecture notes by Perconti [Per12].

2.2 Quantum IO Monad

The Quantum IO (QIO) monad [AG09] is a purely functional interface for quantum programming that provides a separation between the unitary (reversible) and non-unitary (irreversible) fragments of quantum computation. It provides isolation of quantum effects inside a monad similar to what we need to provide in our Quantum Hoare monad.

QIO interface was developed as a library for the Haskell programming language but its design was influenced [Gre10] by the category **FQC** of finite quantum computations (where computations are interpreted as superoperators) as explored by the authors in previous work [GA08]. This is the most we will say about the relation of QIO monad to category theory in this paper.

Here we discuss the relevant bits from the QIO interface. QIO provides a type for referring to qubits, `Qbit`; a type for unitary operations, `U`; and, the `QIO` type operator which is a monad indexed by the result of the quantum computation. The primitive quantum operations defined using the QIO monad below (Haskell syntax) are:

```
mkQbit :: Bool → QIO Qbit       -- initialization
applyU :: U → QIO ()            -- apply a unitary to quantum state
measQbit :: Qbit → QIO Bool     -- measurement
```

Further, `U` is defined as a monoid with neutral element, `mempty`, and operation, `mappend`, that encodes reversible operations on quantum state. The core unitary primitives that we will need are:

```
rot :: Qbit → ((Bool, Bool) → C) → U
cond :: Qbit → (Bool → U) → U
```

`rot` takes a qubit and a two-by-two complex-valued unitary matrix (represented as a function from the matrix indices to $\mathbb{C}$) and lets one define arbitrary rotation on a single qubit. `rot` can then be used to define standard gates such as the Hadamard, H, and the Pauli gates, X, Y and Z. `cond` is the unitary conditional that is used to perform branching unitary operations. For the present paper, we use a simpler conditional, `ifQ`, that is defined as follows:

```
ifQ :: Qbit → U → U
ifQ q u = cond q (λx → if x then u else mempty)
```

That is, `ifQ` acts as the standard control operator in the quantum circuit model that runs its second argument based on the truth value of its first argument. Then, it is easy to define controlled gates such as CNOT with the expression `ifQ q1 (X q2)`.

Quantum state in the QIO interface is modeled as a normalized vector that stores pairs representing complex amplitudes associated with each basis state (represented as a map from Qbit to Bool types). The operations defined over the vector class ensure that the quantum state is kept normalized throughout.

QIO further ensures that the monoidal structure of unitary operations lets one run a unitary operation over the complete quantum state. We elide details here and refer the reader to [AG09] for more.

Types	A,B,C	::=	$\mathbf{1} \mid \mathbf{Bool} \mid \mathbf{Qbit} \mid \mathbf{U} \mid \mathbf{Pure} \mid A \otimes B \mid \Pi x{:}A.B \mid \Delta.\Psi.\{P\}x{:}A\{Q\}$
Assertions	P,Q,R	::=	$\top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \supset Q \mid \neg P \mid \exists x{:}A.P \mid \forall x{:}A.P \mid$ $\exists h{:}\mathtt{heap}.P \mid \forall h{:}\mathtt{heap}.P \mid \mathtt{Id}_A(M,N) \mid \mathtt{HId}(H,G) \mid$ $\mathtt{indom}(H,M)$
Q. Heaps	H,G	::=	$h \mid \mathtt{empty} \mid \mathtt{upd}(H,M,N)$
Elim terms	K,L	::=	$x \mid KM \mid M : A$
Intro terms	M,N,O	::=	$K \mid () \mid \lambda x.M \mid \mathtt{do}\,E \mid \mathtt{true} \mid \mathtt{false}$
Q. Commands	c	::=	$\mathtt{mkQbit}\,M \mid \mathtt{measQbit}\,M \mid \mathtt{applyU}\,M \mid \mathtt{if}\,M\,\mathtt{then}\,N\,\mathtt{else}\,O$
Computations	E,F	::=	$\mathtt{return}\,M \mid x \leftarrow K;E \mid x \Leftarrow c;E \mid x =_A M;E$
Variable context	Δ	::=	$\cdot \mid x : A,\Delta$
Q. Heap context	Ψ	::=	$\cdot \mid h,\Psi$
Assertion context	Γ	::=	$\cdot \mid P,\Gamma$

Figure 1: Syntax of Quantum Hoare Type Theory

3 Quantum Hoare Type Theory

In this section, we introduce the Quantum Hoare Type inspired by the QIO monad and its type theory (QHTT) by replacing the classical effectful portion of Hoare Type Theory with quantum effects. The core idea is to encapsulate any quantum effect inside a monadic Quantum Hoare type, $\Delta.\Psi.\{P\}x{:}A\{Q\}$, and formalize reasoning of quantum effects using strongest postconditions in a similar fashion as HTT in the classical setting.

The next two sections discuss the syntax and typing of QHTT. The last section show examples written in QHTT.

3.1 Syntax

Figure 1 shows the syntax of Quantum Hoare Type Theory. Our presentation closely follows that of HTT [NMB08].

Types We include primitive types for unit, booleans, qubits and unitary operations, and a type **Pure** for representing pure quantum state vectors (complex valued vectors in Hilbert space); these correspond to similar types in the QIO work. There are also type constructors for pairs, dependent functions and the Hoare type from HTT.

Assertions Apart from the usual first order logic, we have assertions for reasoning about propositional equality of terms, $\mathtt{Id}_A(M,N)$, and heaps, $\mathtt{HId}(H,G)$.

As done in HTT, some common convenience assertions can be defined using the base primitives. Particularly, `emp` denotes that the current heap, say h, is empty, that is, $\mathtt{HId}(h, \mathtt{empty})$. $M \mapsto N$ represents the only mapping in the (singleton) heap. $M \hookrightarrow N$ says that looking up M in the heap returns N which can be more explicitly written as $\mathtt{seleq}(H,M,N)$.

Quantum State We use the terms quantum state and quantum heaps interchangeably throughout. A quantum heap is a partial function from qubits (that can in turn be thought of as locations, represented using natural numbers) to at most one quantum state vector.

For the purpose of the type theory, heaps are functional, so that, for example, $\texttt{upd}(H, M, N)$ returns a new heap after updating heap H at location M with N.

Terms The terms are divided into introduction and elimination sorts for the purpose of bidirectional typechecking like in HTT.

The pure fragment allows higher-order functions while the impure (effectful) fragment is encapsulated within the monadic `do` E construct and supports a simple imperative quantum language inspired by QIO constructs. It is worth noting that `do` E represents suspended computation and is considered pure.

Quantum Commands These are the quantum-specific effectful commands from QIO for initialization, unitary application and measurement of qubits. We also support classical control using the `if` M `then` N `else` O construct similar to HTT.

Computations The computations are the usual monadic return and three sequencing operations that bind their result to a variable similar to HTT. The first, $x \leftarrow K$, executes a suspended computation (such as applying a function that includes a `do` E in its body); the second, $x \Leftarrow c$ executes a primitive command c; and the last, $x =_A M$, is just syntactic sugar for let-binding.

Contexts Contexts are modelled exactly the same way as in HTT. The only difference is that instead of classical heaps, we have quantum heaps.

3.2 Typing Rules

In this section, we provide some details about type checking in HTT and how it is different in the quantum case.

3.2.1 Judgments

We explain the main judgments used in HTT that are required to understand the typing rules in the next section. The type system is designed to be bidirectional and syntax directed and hence includes separate judgments for intro and elim terms. We are able to use them unchanged for QHTT. A recent survey provides accessible introduction to bidirectional typing [DK19].

The type checking process also involves generating canonical forms for each term. A canonical form in HTT means a beta-normal (containing no beta-redexes) and eta-long (all intro terms are eta-expanded) form. We elide much detail here.

As usual, elim terms synthesize types ($\Rightarrow$) and intro terms are checked against the given type ($\Leftarrow$). Canonical forms are always synthesized. For the pure fragment:

$\Delta \vdash K \Rightarrow A[N']$ Elim term K has type A and canonical form N'
$\Delta \vdash M \Leftarrow A[M']$ Intro term M has type A and canonical form M'

Note that the blue colored $\Leftarrow$ symbol we use in the typing rules lives in a different syntactic category from the $\Leftarrow$ symbol used in the syntax (fig. 1) that binds the output of primitive commands.

The judgments for computations involve synthesizing the strongest postconditions and checking whether a given postcondition applies:

$\Delta; P \vdash E \Rightarrow x : A.Q[E']$	Computation E with precondition P has *strongest* postcondition Q and returns value x of type A. Its canonical form is E'.
$\Delta; P \vdash E \Leftarrow x : A.Q[E']$	Computation E with precondition P has postcondition Q and returns value x of type A. Its canonical form is E'.

Even though, we have shown the full forms of these judgments above, we will omit the canonical forms while presenting typing rules in the next section as they crowd the rules and do not affect the insight to be gained from them.

Finally, the judgment $\Delta; \Psi; \Gamma_1 \Longrightarrow \Gamma_2$ encodes the sequent calculus for the assertion logic. Recall that Δ, Ψ, and Γ denote the variable context, the quantum heap context and the assertion context, respectively.

The type system includes rules for primitive effectful commands and those for structuring composition such as monadic unit and bind. We reuse most of the HTT rules except those for primitives for quantum effects that need to specify the strongest postconditions for each primitive, we discuss that next.

3.2.2 Strongest Postconditions

Here we show the strongest postconditions for the primitive quantum commands of QHTT encoded in their typing rules. Given a quantum command for initialization, unitary application or measurement, its strongest postcondition is an assertion that most precisely captures the relationship between the initial state and the modified state after the execution. These rules along with the `consq` (consequent) rule of HTT (that ensures weakening of the given strongest postcondition to an arbitrary postcondition) work together to ensure composition of verification.

We need to use the relational composition connective from HTT ($P \circ Q$) that captures how heap evolves with computation. It basically reads: Q holds of the current heap which is obtained after modification of a prior heap for which P holds.

We also use HTT's difference operator ($\multimap$) below that captures changes to only the interesting fragment of the heap without modifying the rest of the heap.

We can now state some typing rules:

Initialization $x \Leftarrow \texttt{mkQbit}\, M; E$

$$\frac{\Delta \vdash M \Leftarrow \textbf{Bool} \qquad \Delta, x : \textbf{Qbit}; P \circ (x \mapsto \texttt{state}\,(M)) \vdash E \Rightarrow y : B.Q}{\Delta; P \vdash x \Leftarrow \texttt{mkQbit}\, M; E \Rightarrow y : B.(\exists x : \textbf{Qbit}.Q)}$$

where x is fresh and `state` is a function that translates a classical representation of quantum state (such as boolean here) to the equivalent quantum state vector.

Initialization can only be performed for a term M that can be type checked as a **Bool**. Then, we look at the rest of the computation, E, in a context that adds x of type **Qbit** under a precondition that extends the previous precondition, P, with the strongest postcondition for initialization, that is, the newly bound variable x points to the representation of a new qubit state; this should synthesize the strongest postcondition (with respect to the expanded precondition in the context), Q, for E.

To avoid dangling variables, we need to existentially quantify x in the postcondition, Q, of the conclusion.

Measurement $x \Leftarrow \texttt{measQbit}\, M; E$

$$\frac{\Delta \vdash M \Leftarrow \textbf{Qbit} \qquad \Delta; \Psi; P \Longrightarrow (M \hookrightarrow -) \qquad \Delta, x : \textbf{Bool}; P \circ ((M \mapsto -) \multimap \texttt{emp}) \vdash E \Rightarrow y : B.Q}{\Delta; P \vdash x \Leftarrow \texttt{measQbit}\, M; E \Rightarrow y : B.(\exists x : \textbf{Bool}.Q)}$$

The measurement rule can be read in a similar way as the previous rule. But in measurement, we need to additionally prove that the verification condition (written as a sequent), the precondition P implies that the qubit M is allocated, holds in the heap context Ψ. The strongest postcondition here is that the fragment of heap that only refers to location M becomes empty.

As may be apparent, initialization is analogous to `alloc` and measurement to `dealloc` and lookup primitives of HTT. We are working out precise details for unitary application, but at a high-level, they involve ensuring that the given unitary term actually represents a unitary matrix and updating the quantum state with the result of unitary application. This involves incorporating the monoidal structure of unitaries into our theory (as is done in QIO). This step also involves generating verification conditions that cannot be checked during this first phase of typechecking. We are also considering more tractable alternatives such as the sum-over-paths action semantics [Amy18] based on the Feynman path integral formulation.[3]

3.3 Sample programs

In this section, we show some simple examples from the QIO work [AG09] translated in our language and annotated with their program specifications using their Quantum Hoare types. An observation is that translating the programs was a very simple process except for coming up with the right specifications for the programs. We provide some commentary on assertions specified for these programs.

At the end of this section, we show a sample verification using QHTT for Bell pair generation.

Hello Quantum World

```
hqw : {emp} r : Bool {emp ∧ Id(r, false)}
    = do q ⇐ mkQbit false;
         measQbit q
```

In this trivial program where we initialize a new qubit with `false` and then immediately measure it, we assert that the result is equal to `false`. The complete specification also implicitly says that the existing quantum state is not affected in any way as both the pre- and postcondition include the `emp` assertion.

Coin Toss

```
rnd : {emp} r : Bool {emp}
    = do q ⇐ mkQbit false;
         applyU (H q);
         measQbit q
```

In the case of quantum coin toss, the result can be in either of its two boolean values `false` or `true`, hence, we need not specify any special postcondition as type checking the result will be sufficient for correctness.

[3] We ended up taking a different approach in subsequent work [Sin20].

Testing Bell Pair

```
testBell : {emp} (a, b) : (Bool, Bool) {emp ∧ Id(a, b)}  -- a,b ∈ {true, false}
         = do qa ⇐ mkQbit false;
              applyU (H qa);
              qb ⇐ mkQbit false;
              applyU (ifQ qa (X qb));
              (measQbit qa, measQbit qb)
```

Here, we are asserting that the returned booleans a and b hold the same value, which is what we expect from the first Bell state prepared in this program.

Verification We would like to describe how QHTT lets us verify the correctness of the above program based on the specification provided as the type of the function. For illustration, we show the assertions annotated as comments as we step through the `testBell` program below. We have elided specific details about unitary application as they are still being worked out.

```
testBell : {emp} (a, b) : (Bool, Bool) {emp ∧ Id(a, b)}
              -- P0: emp
         = do qa ⇐ mkQbit false;
              -- P1: P0 ∘ (qa ↦ |0⟩)
              applyU (H qa);
              -- P2: P1 ∘ ((qa ↦ |0⟩) ⊸ (qa ↦ |+⟩))
              qb ⇐ mkQbit false;
              -- P3: P2 ∘ (qb ↦ |0⟩)
              applyU (ifQ qa (X qb));
              -- P4: P3 ∘ ((qa ↦ |+⟩, qb ↦ |0⟩) ⊸ (qa, qb) ↦ |Φ+⟩)
              (measQbit qa, measQbit qb)
              -- P5: P4 ∘ ((qa ↦ -) ⊸ emp) ∘ ((qb ↦ -) ⊸ emp)
```

In the assertion P2, we show that only the portion of the quantum state referred to by the `applyU` command is affected. Similarly, in P4 we see that only the relevant portion of the heap containing the control and target qubits used in the CNOT operation are affected. In this case, we also needed to combine the two qubit state as they are now entangled. The last assertion, P5, merely encodes the strongest postcondition for measurement for each of the two qubits.

Finally, the `consq` rule of HTT ensures that the assertion P5 leads to the specified postcondition in the type of the program.

We show some two more examples in the appendix for a modular version of Bell pair generation (Appendix A) that shows use of higher-order features of QHTT and another for teleportation protocol (Appendix B). Both of these involve specifying assertions over quantum state whose theory we still need to work out.

4 Conclusions and Future Work

In this paper, we have described our ongoing work on Quantum Hoare Type Theory, which modifies HTT for quantum computing using ideas from the Quantum IO monad work. This approach has the potential to be a unified system for programming, specification, and reasoning about quantum programs. This is active work in progress and more details will be available in a forthcoming technical report [Sin20].

There are several avenues of future work to explore:

Mechanization There are multiple implementations of HTT in Coq such as Ynot [NMS+08]. We are similarly working on mechanizing QHTT in Coq for higher assurance of the usefulness and soundness of our theory. This will also enable us to extract verified circuits in a lower level quantum language such as OpenQASM for execution on real machines.

Quantum Assertion Logic An obvious need is to come up with an assertion logic similar to Separation Logic [Rey02] for quantum computing so as to be able to reason about only the interesting portions of the quantum state while still ensuring correctness of non-local effects such as entanglement. Various Quantum Hoare Logics that exist [Yin12] currently do not support frame rules that provide Separation Logic its power.[4]

Linearity Peter Selinger and collaborators have recently proposed a linear dependent typed version of Proto-Quipper (dubbed Proto-Quipper-D) [Sel20, FKS20]. It is an interesting challenge to reconcile linearity in our theory based on their proposal.

Circuits as Arrows Further, Proto-Quipper treats quantum circuits as first class citizens of the language. We would like to explore modifying our theory to treat Quantum Hoare types as arrows instead of as monads as was suggested by Vizzotto et al [VAS06]. It makes sense from the perspective of sequential composition as arrows can have an arbitrary number of input/outputs as opposed to monads.

Behavioral Types 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 [JP09] such as those inspired by the various quantum resource theories or the Heisenberg representation of quantum mechanics [RSSL19, RSSL20]. This may help us provide more specific postconditions that quantum programmers expect to hold true of their programs.

Classical Effects Finally, it will be an interesting challenge to reconcile both classical and quantum effects together in a single grand unified theory for effects.

Acknowledgments

We thank Robert Rand and the anonymous reviewers for their feedback on a previous draft of this paper. This work is funded by EPiQC, an NSF Expedition in Computing, under grant CCF-1730449.

References

[AG09] T. Altenkirch & A.S. Green (2009): *The Quantum IO Monad*. In: *Semantic Techniques in Quantum Computation*, Cambridge University Press, pp. 173–205, doi:10.1017/CBO9781139193313.006. Available at `http://www.cs.nott.ac.uk/~psztxa/g5xnsc/chapter.pdf`.

[Amy18] M. Amy (2018): *Towards Large-scale Functional Verification of Universal Quantum Circuits*. In: *Proc. QPL '18*, pp. 1–21, doi:10.4204/EPTCS.287.1.

[Dij76] E.W. Dijkstra (1976): *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

[4] Since the time of this writing, we learned about Unruh's work [Unr19] that we use in subsequent work [Sin20].

[DK19] J. Dunfield & N. Krishnaswami (2019): *Bidirectional Typing*. Available at https://arxiv.org/abs/1908.05839. Submitted to ACM Computing Surveys.

[DP06] E. D'hondt & P. Panangaden (2006): *Quantum Weakest Preconditions*. *Math. Struct. Comput. Sci.* 16(3), pp. 429–451, doi:10.1017/S0960129506005251. Available at https://www.cs.mcgill.ca/~prakash/Pubs/weakest_mscs.pdf.

[FKS20] P. Fu, K. Kishida & P. Selinger (2020): *Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract*. In: *Proc. LICS '20*, p. 440–453, doi:10.1145/3373718.3394765.

[GA08] A.S. Green & T. Altenkirch (2008): *From Reversible to Irreversible Computations*. *Electron. Notes Theor. Comput. Sci.* 210, pp. 65–74, doi:10.1016/j.entcs.2008.04.018. Proc. QPL '06.

[Gre10] A.S. Green (2010): *Towards a formally verified functional quantum programming language*. Ph.D. thesis, University of Nottingham. Available at http://eprints.nottingham.ac.uk/11457/.

[HM18] Y. Huang & M. Martonosi (2018): *QDB: From Quantum Algorithms Towards Correct Quantum Programs*. In: *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '18)*, pp. 4:1–4:14, doi:10.4230/OASIcs.PLATEAU.2018.4.

[HM19] Y. Huang & M. Martonosi (2019): *Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs*. In: *Proc. ISCA '19*, pp. 541–553, doi:10.1145/3307650.3322213. Available at https://arxiv.org/abs/1905.09721.

[Hoa69] C.A.R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[JP09] P. Jorrand & S. Perdrix (2009): *Abstract Interpretation Techniques for Quantum Computation*. In: *Semantic Techniques in Quantum Computation*, Cambridge University Press, pp. 206–234, doi:10.1017/CBO9781139193313.007.

[MN19] A. Matuschak & M.A. Nielsen (2019): *Quantum Computing for the Very Curious (and other essays)*. Online. Available at https://quantum.country.

[NC10] M.A. Nielsen & I.L. Chuang (2010): *Quantum Computation and Quantum Information*, 10th anniversary edition. Cambridge University Press, doi:10.1017/CBO9780511976667.

[NMB08] A. Nanevski, G. Morrisett & L. Birkedal (2008): *Hoare Type Theory, Polymorphism and Separation*. *J. Funct. Program.* 18(5–6), pp. 865–911, doi:10.1017/S0956796808006953. Available at https://software.imdea.org/~aleks/htt/jfpsep07.pdf.

[NMS+08] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau & L. Birkedal (2008): *Ynot: Dependent Types for Imperative Programs*. In: *Proc. ICFP '08*, pp. 229–240, doi:10.1145/1411204.1411237. Available at https://software.imdea.org/~aleks/htt/ynot08.pdf.

[Per12] J.T. Perconti (2012): *Hoare Type Theory: Dependent Types for State*. Available at http://www.ccs.neu.edu/home/amal/course/7480-s12/HTT-notes.pdf.

[PRZ17] J. Paykin, R. Rand & S. Zdancewic (2017): *QWIRE: A Core Language for Quantum Circuits*. In: *Proc. POPL '17*, pp. 846–858, doi:10.1145/3009837.3009894. Available at https://jpaykin.github.io/papers/prz_qwire_2017.pdf.

[Ran18] R. Rand (2018): *Formally Verified Quantum Programming*. Ph.D. thesis, University of Pennsylvania. Available at https://repository.upenn.edu/edissertations/3175.

[Rey02] J.C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *Proc. LICS '02*, pp. 55–74, doi:10.1109/LICS.2002.1029817. Available at https://www.cs.cmu.edu/~jcr/seplogic.pdf.

[Ros15] N.J. Ross (2015): *Algebraic and Logical Methods in Quantum Computation*. Ph.D. thesis, Dalhousie University. Available at http://arxiv.org/abs/1510.02198.

[RPZ18] R. Rand, J. Paykin & S. Zdancewic (2018): *QWIRE Practice: Formal Verification of Quantum Circuits in Coq*. In: *Proc. QPL '17*, pp. 119–132, doi:10.4204/EPTCS.266.8.

[RS17] F. Rios & P. Selinger (2017): *A Categorical Model for a Quantum Circuit Description Language (Extended Abstract)*. In: *Proc. QPL '17*, pp. 164–178, doi:10.4204/EPTCS.266.11.

[RSSL19] R. Rand, A. Sundaram, K. Singhal & B. Lackey (2019): *A Type System for Quantum Resources*. Available at `http://ks.cs.uchicago.edu/publication/quantum-resource-types/`. Draft.

[RSSL20] R. Rand, A. Sundaram, K. Singhal & B. Lackey (2020): *Gottesman Types for Quantum Programs*. In: *Proc. QPL '20*, *Electronic Proceedings in Theoretical Computer Science* 340, pp. 279–290, doi:10.4204/EPTCS.340.14.

[Sel20] P. Selinger (2020): *Dependently Typed Quantum Programming in Proto-Quipper*. Available at `https://popl20.sigplan.org/details/planqc-2020-papers/15/`. Invited talk at PLanQC '20.

[Sin20] K. Singhal (2020): *Quantum Hoare Type Theory*. Technical Report, University of Chicago. Available at `https://arxiv.org/abs/2012.02154`. Master's paper.

[Unr19] D. Unruh (2019): *Quantum Hoare Logic with Ghost Variables*. In: *Proc. LICS '19*, pp. 1–13, doi:10.1109/LICS.2019.8785779. Available at `https://arxiv.org/abs/1902.00325`.

[VAS06] J. Vizzotto, T. Altenkirch & A. Sabry (2006): *Structuring Quantum Effects: Superoperators as Arrows*. *Math. Struct. Comput. Sci.* 16(3), pp. 453–468, doi:10.1017/S0960129506005287. Available at `https://arxiv.org/abs/quant-ph/0501151`.

[Yin12] M. Ying (2012): *Floyd–Hoare Logic for Quantum Programs*. *ACM Trans. Program. Lang. Syst.* 33(6):19, doi:10.1145/2049706.2049708.

A Modular Bell Pair

Writing a modular version of `testBell`.

A.1 Hadamard basis states

```
qplus : {emp} r : Qbit {Id(r, |+⟩)}
      = do q ⇐ mkQbit false;
           applyU (H q);
           return q

qminus : {emp} r : Qbit {Id(r, |-⟩)}
       = do q ⇐ mkQbit true;
            applyU (H q);
            return q
```

A.2 Creating entanglement

```
share : Π a : Qbit.
        {a ∈ {|+⟩, |-⟩}}   -- a ∈ {b, c} is short for Id(a,b) ∨ Id(a,c)
           b : Qbit
        {Id(a, b) ∧ a ∈ {|0⟩, |1⟩}}
      = λa.do b ⇐ mkQbit false;
              applyU (ifQ a (X b));
              return b
```

A.3 Bell pair

```
bell : {emp} (a, b) : (Qbit, Qbit) {Id(a, b) ∧ a ∈ {|0⟩, |1⟩}}
     = do qa ← qplus;
          qb ← share qa;
          return (qa, qb)
```

A.4 Testing modular Bell pair

```
testBell : {emp} (a, b) : (Bool, Bool) {emp ∧ Id(a, b)}
         = do (qa, qb) ← bell;
              (measQbit qa, measQbit qb)
```

B Teleportation

B.1 Alice's circuit

```
alice : Π a : Qbit. Π e : Qbit
        {(Id(a, -) ∧ entangled(e)}
           r : (Bool, Bool)
        {emp}
      = λa.λe.do applyU (ifQ a (X e));
                 applyU (H a);
                 (measQbit a, measQbit e)
```

B.2 Bob's circuit

```
bob : Π m1 m2 : Bool. Π e : Qbit
      {entangled(e)}
         r : Qbit
      {(Id(r, -)}
    = λm1.λm2.λe.do if m1 then applyU (Z e) else ();
                    if m2 then applyU (X e) else ();
                    return e
```

B.3 Teleport

```
teleport : Π q : Qbit. x : Pure.
           {Id(q, x)}
              r : Qbit
           {Id(r, x)}
         = λq.do (a, b) ← bell;
                 (m1, m2) ← alice q a;
                 tq ← bob m1 m2 b;
                 return tq
```