Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs

CHARLES YUAN, MIT CSAIL, USA
CHRISTOPHER MCNALLY, MIT RLE, USA
MICHAEL CARBIN, MIT CSAIL, USA

Quantum programming languages enable developers to implement algorithms for quantum computers that promise computational breakthroughs in classically intractable tasks. Programming quantum computers requires awareness of *entanglement*, the phenomenon in which measurement outcomes of qubits are correlated. Entanglement can determine the correctness of algorithms and suitability of programming patterns.

In this work, we formalize *purity* as a central tool for automating reasoning about entanglement in quantum programs. A pure expression is one whose evaluation is unaffected by the measurement outcomes of qubits that it does not own, implying freedom from entanglement with any other expression in the computation.

We present Twist, the first language that features a type system for sound reasoning about purity. The type system enables the developer to identify pure expressions using type annotations. Twist also features purity assertion operators that state the absence of entanglement in the output of quantum gates. To soundly check these assertions, Twist uses a combination of static analysis and runtime verification.

We evaluate Twist's type system and analyses on a benchmark suite of quantum programs in simulation, demonstrating that Twist can express quantum algorithms, catch programming errors in them, and support programs that existing languages disallow, while incurring runtime verification overhead of less than 3.5%.

CCS Concepts: • **Computer systems organization** → *Quantum computing*; • **Theory of computation** → Denotational semantics; • **Software and its engineering** → *Formal language definitions*; *Language features*.

Additional Key Words and Phrases: quantum programming, entanglement, purity, type systems

ACM Reference Format:
Charles Yuan, Christopher McNally, and Michael Carbin. 2022. Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs. *Proc. ACM Program. Lang.* 6, POPL, Article 30 (January 2022), 32 pages. https://doi.org/10.1145/3498691

1 INTRODUCTION

Quantum programming languages [Altenkirch and Grattage 2005; Bichsel et al. 2020; Clairambault and de Visme 2019; Green et al. 2013; Paykin et al. 2017; Rand et al. 2019; Rennela and Staton 2017; Selinger 2004; Selinger and Valiron 2005; Svore et al. 2018; Wecker et al. 2014; Ying 2016] allow programmers to utilize the computational primitives enabled by the quantum computers of today and tomorrow. Algorithms for quantum computers offer computational breakthroughs in integer factorization [Shor 1997], search [Grover 1996], cryptographic and communication protocols [Bennett and Brassard 2014; Bennett et al. 1993], computational physics and chemistry [Childs et al. 2018; Kassal et al. 2011], and machine learning [Biamonte et al. 2017].

Authors' addresses: Charles Yuan, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, chenhuiy@csail.mit.edu; Christopher McNally, MIT RLE, 32 Vassar St, Cambridge, MA, 02139, USA, mcnallyc@mit.edu; Michael Carbin, MIT CSAIL, 32 Vassar St, Cambridge, MA, 02139, USA, mcarbin@csail.mit.edu.

2475-1421/2022/1-ART30
https://doi.org/10.1145/3498691

Quantum computation relies on the manipulation of *quantum states* consisting of *qubits*, the quantum analogs of classical data and bits. A quantum state exists in a *superposition*, a weighted sum over classical states. *Measurement* causes a superposition to assume a classical state, with probability derived mathematically from the weight ascribed to that state in the sum. In the standard QRAM [Knill 1996] model, computations execute on a classical computer with access to a quantum device that supports initializing and operating on quantum states.

1.1 Entanglement

Entanglement, the phenomenon of correlation between qubits,[1] is critical to the quantum computational advantage [Jozsa and Linden 2003]. Given a pair of entangled qubits, measuring one forces the other to assume a state consistent with the measurement. Thus, the measurement outcome of one qubit causes operations on the other to potentially yield different behavior.

In many instances, reasoning about entanglement is either necessary or beneficial. During debugging, determining whether two qubits are entangled at a particular point is a sanity check that an algorithm has been implemented correctly [Huang and Martonosi 2019]. Another application is in the handling of temporary qubits used by several algorithms[2] and programming patterns[3] that must be measured and deallocated so that physical qubits may be reused. If these temporaries remain entangled with the primary results of the computation, their measurement will cause the results to be incorrect [Bichsel et al. 2020; Green et al. 2013; Rand et al. 2019].

A further application is mitigation of information-leakage attacks on Shor's algorithm [Azuma 2017] and quantum bit-commitment schemes [Lo and Chau 1998] in which attackers introduce surreptitious entanglement that is rendered impossible if the sensitive state is verified to be not entangled. Yet another application is to compiler analyses that leverage descriptions of which qubits are entangled to optimize usage of resources such as qubits in programs [Häner et al. 2020].

Entanglement is thus a key to reason about the correctness of an algorithm, verify the suitability of a programming pattern, and empower compiler analyses. Though languages [Amy et al. 2017; Bichsel et al. 2020; Rand et al. 2019] have recently been developed to reason about quantum phenomena such as reversibility of computation, prior work has yet to facilitate sound reasoning about entanglement in quantum programs.

1.2 Purity

We introduce the concept of the *purity* of an expression, which enables reasoning about entanglement in quantum programs. Operationally, quantum expressions contain references to qubits, analogous to pointers to classical memory, and evaluate by executing operations known as *gates* on them. We say that an expression *owns* a qubit when the final value to which it evaluates, such as a tuple of qubits, contains a reference to that qubit.

An expression is *pure* if its evaluation is unaffected by the measurement outcome of any qubit it does not own, and *mixed* otherwise. Specifically, the qubits that a pure expression owns are only potentially entangled with each other and are *separable*, or free of entanglement, from those in the remainder of the program. Pure and mixed expressions coincide with the established quantum mechanical definitions of pure and mixed states [Nielsen and Chuang 2010] in that evaluating a pure expression results in its owned qubits constituting a pure sub-state of the program state.

[1] More precisely, the measurement outcomes of entangled quantum states have statistical correlations that cannot be explained by physical theories with local realism [Nielsen and Chuang 2010].
[2] Such as oracle functions in Grover's algorithm [Grover 1996] or modular multiplication in Shor's algorithm [Shor 1997].
[3] Such as conditionally executing statements depending on some predicate over qubits [Bichsel et al. 2020].

1.3 Specification and Verification of Purity

We present Twist, the first language that enables programmers to specify that an expression is pure and soundly verify that the specification holds. Twist provides a type system to specify that an expression is of pure type, purity assertions to declare the absence of entanglement, and a combination of static analysis and runtime verification to ensure that purity specifications hold.

Purity Types. The type system enables a programmer to specify the type of a qubit or tuple of qubits as either pure – unaffected by measurements of other qubits in the computation – or mixed – potentially affected by other qubits. The type system reasons about purity by tracking potential entanglement between qubits, conservatively identifying that any quantum gate operating on two qubits may entangle them. We introduce a type of *entangled pairs* of quantum data that are potentially entangled with each other, so that the outputs of multi-qubit quantum gates are of entangled tuple type. Elements projected from an entangled pair have mixed type to reflect their potential entanglement with each other.

Purity Assertions. Because the application of a quantum gate may in fact remove the entanglement of a qubit with others in the computation, the language provides constructs to assert the absence of entanglement relationships in the program. Twist provides two purity assertions: `cast`, which asserts that an expression is pure, and `split`, which asserts that the components of a pure entangled pair are separable from each other and hence pure. These two assertions work in concert to identify tuples of qubits that have no external entanglement, and then further assert the absence of entanglement between individual qubits within a tuple.

Purity Assertion Verification. Despite progress in statically characterizing the effect of gates on entanglement [Rand et al. 2021b], in a general quantum program, verifying that a qubit is pure is at least as hard as simulating the program [Gurvits 2003; Hayden et al. 2014]. Twist therefore relies on a combination of both static analysis and runtime verification to check purity assertions.

To verify the `cast` assertion, Twist first executes a conservative static analysis to determine whether the set of qubits that an expression owns is separable from all others in the program. To do so, the analysis identifies the qubits that ever share an entangled pair with any qubit in this set. If all such qubits are also in this set, then the analysis concludes the expression is pure.

To verify `split`, Twist tests at runtime whether two precise sets of qubits are separable from each other. These checks rely on a primitive that determines whether the runtime quantum state is separable or entangled. Similarly to the quantum assertions of Huang and Martonosi [2019]; Li et al. [2020]; Liu and Zhou [2021], we abort execution if the condition fails.

Separability Testing. Twist's purity assertions rely on a primitive that tests whether the runtime quantum state is separable. Harrow and Montanaro [2013]; Walborn et al. [2006] have proposed implementations of such a test in hardware, which remain a topic of active research. In this work, we implement Twist on a state vector-based quantum simulator, examples of which include Abraham et al. [2019]; Gheorghiu [2018]. We present a concrete implementation of separability testing in simulation based on the Schmidt decomposition [Schmidt 1907] of quantum states.

Summary. Together, Twist's purity types, purity assertions, and analysis techniques work to ensure the sound verification of purity specifications in quantum programs. The developer may use purity types to require pure expressions in computations, use purity assertion constructs to state conditions to be verified, and execute the static analysis and runtime verification on the assertions. The resulting guarantees of purity enable the programmer to debug algorithms, leverage idioms, and enjoy correctness guarantees in their programs.

1.4 Contributions

In this paper we present the following contributions:

- *Purity.* We present the novel definition of *pure* expressions in a quantum program, those that are unaffected by measurement outcomes of the remainder of the program. We formulate purity within the operational and denotational semantics of a functional quantum language.
- *Purity Types.* We present a type system that identifies pure expressions, and prove that in it, expressions of pure type are in fact pure.
- *Purity Assertions.* We present two types of purity assertions that state the absence of entanglement in the output of quantum gates: one stating that an expression is pure, and one stating that the two components of a pure entangled pair are individually pure.
- *Purity Assertion Verification.* We present a static analysis and runtime verifications for the purity assertions, such that programs passing verification satisfy their purity specifications.
- *Evaluation.* We implement Twist, a language featuring purity types and assertions, in quantum simulation. We show that Twist can express quantum algorithms and reject programming errors in them, that its runtime verification executes with overhead less than 3.5%, and that it can express semantically valid programs that existing languages disallow.

Our work introduces the powerful notion of purity to quantum programming, enabling sound reasoning for entanglement. Using Twist, developers can recognize quantum entanglement not as a cognitive burden but rather as a clarifying tool to understanding the correctness of their programs.

2 BACKGROUND ON QUANTUM COMPUTATION

The following is an overview of key concepts in quantum computation relevant to this work and our notational choices. Nielsen and Chuang [2010] provide a comprehensive reference.

2.1 Pure State Formalism

We first define a *pure quantum state* and the main formalism of quantum mechanics in this work.

Qubit. The basic unit of quantum information is the *qubit*, a linear combination $\gamma_0 |0\rangle + \gamma_1 |1\rangle$ known as a *superposition*, where $|0\rangle$ and $|1\rangle$[4] are *basis states* and $\gamma_0, \gamma_1 \in \mathbb{C}$ are *amplitudes* satisfying $|\gamma_0|^2 + |\gamma_1|^2 = 1$ describing relative weights of basis states. Examples of qubits include the classical zero bit $|0\rangle$, classical one bit $|1\rangle$, and the superposition states $(|0\rangle + |1\rangle)/\sqrt{2}$ and $(i\,|0\rangle - |1\rangle)/\sqrt{2}$.

Quantum State. A 2^n-dimensional *pure quantum state* $|\psi\rangle$ is a superposition over n-bit strings. For example, $(|00\rangle + |11\rangle)/\sqrt{2}$ is a quantum state over 2 qubits. Equivalently, we may represent any pure state as a *state vector*, a length-2^n vector of normalized complex amplitudes.[5]

Multiple qubits form a quantum state system by means of the tensor product $\otimes$. Thus, the state $|01\rangle$ is equal to the product $|0\rangle \otimes |1\rangle$. We use subscripts to denote the names of individual qubits or sets of qubits. For example, to denote a two-qubit system in which a qubit named α has state $|0\rangle$ and qubit β has state $|1\rangle$, we write $|0\rangle_\alpha \otimes |1\rangle_\beta$.[6]

We define the *empty state* $|\cdot\rangle$ to be the length-2^0 vector containing amplitude 1, which effectively describes a system of zero qubits. The tensor product of any $|\psi\rangle$ and $|\cdot\rangle$ is accordingly $|\psi\rangle$. The *domain* of a state, dom $|\psi\rangle$, is the set of qubit names it contains.

[4] The Dirac ket notation $|\cdot\rangle$ is customary in quantum mechanics. In our work, it simply denotes a quantum state.

[5] For example, the state vector corresponding to $(|00\rangle + |11\rangle)/\sqrt{2}$ is $[1/\sqrt{2}, 0, 0, 1/\sqrt{2}]^\top$, with the elements corresponding to amplitudes of $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ respectively.

[6] In our notation, we treat the tensor product as commutative, so that $|1\rangle_\beta \otimes |0\rangle_\alpha$ refers to the same state as $|0\rangle_\alpha \otimes |1\rangle_\beta$, with the understanding that qubits are actually stored in some canonical order in the state.

Unitary Operators. A 2^n-dimensional *unitary operator* is a linear operator on state vectors represented by an $2^n \times 2^n$ matrix U that preserves inner products and whose inverse is its Hermitian adjoint. We denote the state produced by a unitary operator acting on qubit α in state $|\psi\rangle$ by $U_\alpha |\psi\rangle$.[7] In this work, we use single-qubit quantum gates such as:

- X – bit-flip (NOT) gate, which maps $|0\rangle$ to $|1\rangle$ and vice versa;
- Z – phase-flip gate, which leaves $|0\rangle$ unchanged and maps $|1\rangle$ to $-|1\rangle$;
- H – Hadamard gate, which maps $|0\rangle$ to $(|0\rangle + |1\rangle)/\sqrt{2}$ and $|1\rangle$ to $(|0\rangle - |1\rangle)/\sqrt{2}$.

We also use two-qubit gates, such as controlled-NOT (CNOT), controlled-Z (CZ), and SWAP. The controlled gates perform NOT or Z on their target qubit if their control qubit is in state $|1\rangle$.

The SWAP gate swaps two qubits in a quantum state, and inserting SWAP gates enables us to rename qubits at will. We use $|\psi\rangle\,[\gamma/\alpha]$ to denote renaming qubit α to a new name γ in $|\psi\rangle$ by implicitly inserting SWAP gates. For example, $|\psi\rangle = |0\rangle_\alpha \otimes |1\rangle_\beta$ becomes $|\psi\rangle\,[\gamma/\alpha] = |1\rangle_\beta \otimes |0\rangle_\gamma$.

Measurement. A quantum *measurement* is a probabilistic operation over quantum states. When a qubit $\gamma_0 |0\rangle + \gamma_1 |1\rangle$ is measured,[8] the outcome is $|0\rangle$ with probability $|\gamma_0|^2$ and $|1\rangle$ with probability $|\gamma_1|^2$. Measuring a qubit within a larger quantum state will cause the entire state to probabilistically assume one of two outcomes. The outcome state after measurement is equal to the tensor product of the just-measured qubit in a basis state and the new state of the remainder of the system.

We denote the state produced by measurement of qubit α in state $|\psi\rangle$ as $M_\alpha |\psi\rangle$. To define measurement, we first rewrite the state into the form $|\psi\rangle = \gamma_0 |0\rangle_\alpha \otimes |\psi_0\rangle + \gamma_1 |1\rangle_\alpha \otimes |\psi_1\rangle$ where $|\psi_0\rangle$ and $|\psi_1\rangle$ are unique quantum states.[9] Then, with probability $|\gamma_0|^2$ we obtain $M_\alpha |\psi\rangle = |0\rangle_\alpha \otimes |\psi_0\rangle$, and with probability $|\gamma_1|^2$ we obtain $M_\alpha |\psi\rangle = |1\rangle_\alpha \otimes |\psi_1\rangle$.

Pure and Mixed States. Unlike the pure states defined so far, the result of a measurement is a classical probability distribution over pure states, known as a *mixed* state.

For example, $(|0\rangle + |1\rangle)/\sqrt{2}$ is a pure state, whereas the distribution of $|0\rangle$ with probability $1/2$ and $|1\rangle$ w.p. $1/2$ is a mixed state. Though measuring these two states immediately yields the same outcome distribution, they behave differently under unitary operators. For example, applying a Hadamard gate to $(|0\rangle + |1\rangle)/\sqrt{2}$ always produces $|0\rangle$. Applying Hadamard to the mixed state instead produces another mixed state that when measured yields either 0 or 1 with equal probability.[10]

Entanglement and Separability. A *bipartite* quantum state $|\psi\rangle$ is a state over the disjoint union of two qubit sets $A \sqcup B$. A bipartite state is *separable* if it can be written as a tensor product of two states over each set, $|\psi_A\rangle \otimes |\psi_B\rangle$, or *entangled* otherwise.

For example, the two-qubit state $(|00\rangle + |01\rangle + |10\rangle + |11\rangle)/2$ is separable because it is the product of two copies of $(|0\rangle + |1\rangle)/\sqrt{2}$. By contrast, the *Bell state* [Bell 1964] $(|00\rangle + |11\rangle)/\sqrt{2}$ is entangled because it cannot be written as the product of two single-qubit states.

Given the bipartite state $|\psi\rangle$, measuring the qubits of B will have different consequences for the remaining state of A, depending on whether $|\psi\rangle$ is separable. If $|\psi\rangle$ is separable, the measurement will leave A in a pure state, and if it is entangled, measurement will leave A in a mixed state. For example, measuring one of the qubits in the entangled Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$ results in the remaining qubit taking on a mixed state, $|0\rangle$ with probability $1/2$ and $|1\rangle$ with probability $1/2$.

[7] In this work, we will work with one- and two-qubit unitary operators on named qubits with the understanding that they will be padded to all n qubits in the system by tensor product with the identity matrix.

[8] We concern ourselves primarily with projective computational (i.e. 0/1) basis measurements, though our results can be generalized to other measurement forms.

[9] Scaling a quantum state by a coefficient $e^{i\theta}$ known as a global phase factor produces another state indistinguishable from the original by any measurement. Thus, we define equality and uniqueness of states to be up to global phase.

[10] In particular, this mixed state is $(|0\rangle + |1\rangle)/\sqrt{2}$ with probability $1/2$ and $(|0\rangle - |1\rangle)/\sqrt{2}$ with probability $1/2$.

Schmidt Decomposition. Given a bipartite pure state $|\psi\rangle$ over qubit sets $A \sqcup B$, we may compute its unique *Schmidt decomposition* [Schmidt 1907], $|\psi\rangle = \sum_j \lambda_j |\psi_{A_j}\rangle \otimes |\psi_{B_j}\rangle$ where $|\psi_{A_j}\rangle$ and $|\psi_{B_j}\rangle$ are states of A and B and λ_j are positive real *Schmidt coefficients* satisfying $\sum_j \lambda_j^2 = 1$. The Schmidt decomposition provides a criterion for separability – $|\psi\rangle$ is separable if and only if it has only one nonzero Schmidt coefficient.

2.2 Mixed State Formalism

We next describe mixed states, a more expressive alternative formalism for quantum computation. Mixed states model statistical ensembles of states arising over multiple program executions.

Density Matrix. Given a 2^n-dimensional state vector $|\psi\rangle$, we use $|\psi\rangle\langle\psi|$ to denote its outer product with itself, which is a $2^n \times 2^n$ matrix. A mixed state is mathematically represented as a *density matrix* ρ, a linear combination $\rho = \sum_j p_j |\psi_j\rangle\langle\psi_j|$ where each $p_j > 0$ and ρ is positive semidefinite. The *domain* of a density matrix, $\operatorname{dom}\rho$, is the set of qubits contained in each $|\psi_j\rangle$.

A density matrix is *normalized* if $\sum_j p_j = 1$ and $\operatorname{tr}\rho = 1$. *Partial density matrices* [Ying 2016] relax the conditions to $\sum_j p_j \leq 1$ and $\operatorname{tr}\rho \leq 1$ and can be added to form normalized density matrices.

Unitary Operators. A unitary quantum operator U applies to a density matrix ρ by matrix conjugation, so that the resulting matrix is $U\rho U^\dagger$ where $U^\dagger$ is the Hermitian adjoint of U.

Measurement. A quantum measurement is represented by a set of projections P_j corresponding to possible outcomes, where $\sum_j P_j = I$. Outcome P_j occurs with probability $\operatorname{tr}(\rho P_j)$, and results in normalized density matrix $P_j\rho P_j/\operatorname{tr}(\rho P_j)$. $P_j\rho P_j$ is its probability-weighted partial density matrix.

The principle of *deferred measurement* states that any computation that conditionally executes gates based on the measurement of a qubit produces the same mixed state as one that uses quantum conditioned gates and defers measurement until the end of the computation.[11]

Product States and Separability. We can construct a composite of mixed states using the tensor product $\rho_1 \otimes \rho_2$. We use the notation $\rho[\gamma/\alpha]$ to rename qubit α to γ in ρ by implicitly inserting SWAP gates. A mixed state ρ is *simply separable* if there exist ρ_1 and ρ_2 where $\rho = \rho_1 \otimes \rho_2$.

Partial Trace. The *partial trace* of ρ over A, $\operatorname{Tr}_A(\rho)$, is the unique linear operator satisfying:

$$\operatorname{Tr}_A(|\psi_A\rangle\langle\psi_A| \otimes |\psi_B\rangle\langle\psi_B|) = \operatorname{tr}(|\psi_A\rangle\langle\psi_A|)\, |\psi_B\rangle\langle\psi_B|$$

where $A = \operatorname{dom}|\psi_A\rangle$. In the special case where ρ is simply separable as $\rho_A \otimes \rho_B$ where $\operatorname{dom}\rho_A = A$ and $\operatorname{dom}\rho_B = B$, we have $\operatorname{Tr}_B(\rho) = \rho_A$ and $\operatorname{Tr}_A(\rho) = \rho_B$.

Purity. A mixed state ρ is *pure* if it is not an ensemble of more than one state: $\rho = |\psi\rangle\langle\psi|$ for some pure state $|\psi\rangle$. The *rank test* states that ρ is pure if and only if $\operatorname{tr}(\rho^2) = (\operatorname{tr}\rho)^2 = 1$. We similarly have that a partial density matrix $\rho = p\,|\psi\rangle\langle\psi|$ if and only if $\operatorname{tr}(\rho^2) = (\operatorname{tr}\rho)^2$ by linearity of trace.

3 EXAMPLE

We illustrate the value of purity and Twist with the protocol of *quantum teleportation* [Bennett et al. 1993], a demonstration of the power of entanglement and a building block for techniques such as gate teleportation [Gottesman and Chuang 1999]. The protocol transmits the information stored in one qubit to a receiver an arbitrary distance away by transferring only two classical bits of information. Though the original protocol uses measurement-conditioned gates to emphasize classical information exchange, we examine a variant [Kumar et al. 2020; Miller et al. 2011] that instead uses quantum-conditioned gates and defers all measurement to the end of the program.

[11] e.g. uses a CNOT gate rather than choosing whether to execute a NOT gate based on a classical measurement outcome.

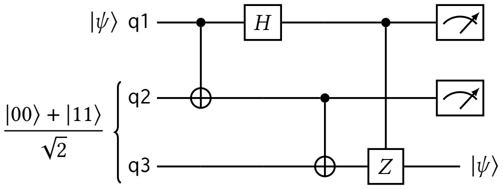

Fig. 1. Teleportation protocol with deferred measurement expressed as a quantum circuit. The inputs are q1, the qubit to be teleported, and a Bell pair of two qubits. The circuit applies conditional-NOT (CNOT) and conditional-Z (CZ) gates to q3. It measures two qubits and outputs one with the teleported state of q1.

```
1  fun teleport (q1 : qubit) : qubit =
2    let (q2 : qubit, q3 : qubit) = bell_pair () in
3    let (q1 : qubit, q2 : qubit) = CNOT (q1, q2) in
4    let q1 : qubit = H (q1) in
5    let (q2 : qubit, q3 : qubit) = CNOT (q2, q3) in
6    let (q1 : qubit, q3 : qubit) = CZ (q1, q3) in
7    let _ : bool * bool = measure (q1, q2) in q3
```

Fig. 2. Implementation of the teleportation circuit of Figure 1 as a quantum program.

Figure 1 presents a quantum circuit for the deferred-measurement variant of teleportation that accepts as input a qubit q1 to be teleported.[12] We assume a pair of qubits (q2, q3) exist in a Bell state (Section 2). The circuit entangles q1 and q2 by a CNOT gate, applies a Hadamard (H) gate to q1, and applies CNOT and CZ gates in succession on q3. Finally, the circuit measures q1 and q2 and outputs q3, which now contains the original state of q1.

Figure 2 presents a program implementing this circuit, as a function accepting a qubit q1 and returning the teleported output, where the helper `bell_pair` allocates a Bell pair.

3.1 Entanglement and Purity

The specification of the teleportation protocol is that the final state of q3 is the same as the initial state of q1. Thus, a central property that ensures the correctness of this program is that q3 is *pure* – free of entanglement with q1 and q2 and hence unaffected by the measurements of these qubits.

Consider instead if one were to replace the CZ gate on line 6 of Figure 2 with a different gate that causes q3 to remain entangled with q1 and q2, for example a CNOT gate. After replacing CZ with CNOT, q3 would be affected by the measurements on line 7.

Because q3 is now entangled with q1 at the time of measurement, the program no longer satisfies the specification that q3 assumes the original state of q1. Instead, if q1 initially has the state $(|0\rangle + |1\rangle)/\sqrt{2}$, then q3 will assume a different state:

$$\begin{cases} \frac{|0\rangle+|1\rangle}{\sqrt{2}} & \text{with probability } \frac{1}{2} \\ \frac{-|0\rangle+|1\rangle}{\sqrt{2}} & \text{with probability } \frac{1}{2} \end{cases}$$

This probability distribution over pure states is a mixed state (Section 2) that stems from different measurement outcomes of q1 and q2. By contrast, pure expressions must always evaluate to unique final states not dependent on the measurements of other qubits.

[12] For the purpose for demonstrating our results, understanding the rationale behind this circuit is not necessary; we refer the reader to Nielsen and Chuang [2010] for a detailed explanation.

```
 1 fun teleport (q1 : qubit<P>) : qubit<M> = (* mixed type *)
 2   let q23 : (qubit & qubit)<P> = bell_pair () in
 3   let (q2 : qubit<M>, q3 : qubit<M>) = q23 in
 4   let q12 : (qubit & qubit)<M> = CNOT (q1, q2) in
 5   let (q1 : qubit<M>, q2 : qubit<M>) = q12 in
 6   let q1 : qubit<M> = H (q1) in
 7   let (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
 8   let (q1 : qubit<M>, q3 : qubit<M>) = CZ (q1, q3) in
 9   let _ : bool * bool = measure (q1, q2) in q3
```

Fig. 3. Teleportation program in Twist, using purity types. The return type of the function is currently mixed.

```
 1 fun teleport (q1 : qubit<P>) : qubit<P> = (* pure type *)
 2   let q23 : (qubit & qubit)<P> = bell_pair () in
 3   let (q2 : qubit<M>, q3 : qubit<M>) = q23 in
 4   let (q1 : qubit<M>, q2 : qubit<M>) = CNOT (q1, q2) in
 5   let q1 : qubit<M> = H (q1) in
 6   let (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
 7   let (q1 : qubit<M>, q3 : qubit<M>) = CZ (q1, q3) in
 8   let q123 : ((qubit & qubit) & qubit)<M> = ((q1, q2), q3) in
 9   (* assert ((q1, q2), q3) is pure; check statically *)
10   let q123 : ((qubit & qubit) & qubit)<P> = cast<P>(q123) in
11   (* assert q3 is separable from (q1, q2); check dynamically *)
12   let (q12 : (qubit & qubit)<P>, q3 : qubit<P>) = split<P>(q123) in
13   let _ : bool * bool = measure (q12) in q3
```

Fig. 4. Teleportation example with purity types and assertions so that its return type is refined to be pure.

3.2 Purity Types

Figure 3 presents the teleportation program from Figure 2, but written in Twist with type annotations for purity. In a Twist program, every quantum expression is of a type that is either pure or mixed. Pure expressions are those that are unaffected by the measurement of other qubits whereas mixed expressions are those that may be affected.

We say that an expression *owns* a qubit if it evaluates to a value that contains a reference to the qubit. If a qubit owned by an expression is entangled with another unowned qubit, then the measurement outcome of the unowned qubit inevitably affects the state of the owned qubit. We describe entanglement in the type system using a type of *entangled pairs*, denoted by the type constructor &, such as on line 2 of Figure 3. An entangled pair stipulates that its two components potentially share entanglement.

Operations such as `bell_pair` (line 2) and `CNOT` (line 4) return entangled pairs, indicating that two qubits are potentially entangled. Projection from an entangled pair results in an expression of mixed type. For example, when the program projects `q2` and `q3` from the entangled pair `q23` on line 3, each qubit has mixed type because they may be entangled with each other. Finally, `q3` has mixed type according to the rules of the type system, denoting that this implementation may not satisfy its specification of returning a pure qubit.

3.3 Purity Assertions

Figure 4 presents an implementation of the `teleport` function that instead soundly returns a pure output by leveraging the purity assertions of Twist. The program performs two steps to verify that `q3` is not entangled with any other qubits in the program.

```
1 fun teleport (q1 : qubit<P>) : qubit<P> =
2   let q23 : (qubit & qubit)<P> = bell_pair () in
3   let (q2 : qubit<M>, q3 : qubit<M>) = q23 in
4   let (q1 : qubit<M>, q2 : qubit<M>) = CNOT (q1, q2) in
5   let q1 : qubit<M> = H (q1) in
6   let (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
7   let (q1 : qubit<M>, q3 : qubit<M>) = CZ (q1, q3) in
8   let _ : bool * bool = measure (q1, q2) in
9   cast<P>(q3) (* direct assertion of q3 as pure *)
```

Fig. 5. Teleportation program that asserts the purity of q3 with a single purity assertion on line 9. Checking the assertion that q3 is pure after the measurements of q1 and q2 requires reasoning on mixed states, which is inefficient in practice. Consequently, our static analysis rejects this program.

In the first step, the program forms an entangled pair containing all three qubits on line 8 and then on line 10 uses `cast<P>`, a *purifying-cast assertion* that states that this triple is pure and has no entanglement with the rest of the program.

In the second step, on line 12, the program uses `split<P>`, a *purifying-split assertion* that states that the two components of the triple, one containing q1 and q2, and the other q3, are not entangled with each other, confirming that they are both individually pure.

3.4 Purity Assertion Verification

Twist uses a static analysis to verify `cast`, whereas it verifies `split` at runtime.

Verifying `cast`. The static analysis determines that an expression is pure if it contains either zero or both components of every entangled pair created by the execution of the program. In Figure 4, Twist determines that q123 is not entangled with the rest of the program, because it contains both components of the pure entangled pair q23 created by `bell_pair` on line 2, along with the pure function argument q1, all of which never interact with any other qubits.

Verifying `split`. In contrast to using static analysis as with `cast`, Twist verifies `split` at runtime, because statically checking separability necessitates direct reasoning about the quantum semantics of gates, which to the best of our knowledge is at least as hard as simulation except for circuits constructed from restricted classes of gates [Hayden et al. 2014].

Specifically, Twist determines whether the two components of a pure entangled pair are entangled with each other via a runtime separability test. Our implementation of this test obtains the simulated state vector of the program at line 12, and determines whether it is separable along the components of the pair by means of the Schmidt decomposition (Section 2). Because q3 is in fact pure in this program, the assertion always succeeds.

Efficiency. We have designed `split` and `cast` to be used in concert to make their verification more efficient. Consider instead the program in Figure 5 that gives q3 pure type, but does not use `split` and only uses `cast` on line 9. While this program type checks and has a valid semantic meaning, the static analysis rejects this program by virtue of the fact that the operand of the `cast` (q3) does not contain all potentially entangled qubits (q1 and q2) as was the case in Figure 4.

The reason we have designed the static analysis to reject this program is because the measurements of q1 and q2 (line 8) imply that q3 could be in a mixed state, if q3 were in fact entangled with either q1 or q2. It is not possible to dynamically verify purity for a qubit within a mixed state by separability tests on the runtime quantum state alone. To perform this check, Twist would instead have to simulate the full density matrix of the program. However, density matrix simulation is

```
1 fun teleport (q1 : qubit<P>) : qubit<P> =
2   let (q2 : qubit<M>, q3 : qubit<M>) = bell_pair () in
3   let (q1 : qubit<M>, q2 : qubit<M>) = CNOT (q1, q2) in
4   let q1 : qubit<M> = H (q1) in
5   let (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
6   let (q1 : qubit<M>, q3 : qubit<M>) = CZ (q1, q3) in
7   let q123 : ((qubit & qubit) & qubit)<M> = ((q1, q2), q3) in
8   let (q12 : (qubit & qubit)<P>, q3 : qubit<P>) = split<P>(cast<P>(q123)) in
9   q3 (* safely discard pure expression q12 *)
```

Fig. 6. Concise teleportation program in Twist that implicitly discards a pure value q12 on line 9.

inefficient compared to state vector simulation. In a quantum simulator, density matrices have quadratic space overhead over state vectors, and standard simulators such as Abraham et al. [2019] support only half as many qubits in mixed-state simulation.[13]

By only performing runtime verification of `split`, whose argument is of pure type, Twist may exploit the available relative efficiency of testing separability of a runtime pure quantum state. Put together, Twist balances both static analysis to identify pure types where statically practical – discharging the `cast` operator in Figure 4 – as well as runtime verification to dynamically enforce purity where it too can be done practically – the `split` operator on a pure quantum state.

3.5 Discarding Pure Values

Twist's purity types enable us to write the teleportation program even more concisely than Figure 4. Line 13 of Figure 4 explicitly measures q12, a value the program no longer needs. Instead, in Figure 6 we remove this measurement operation and simply discard q12.

The reason that discarding unused qubits is a potential point of contention is the deferred measurement principle (Section 2). The principle states that discarding a qubit or allowing it to leave scope has the same effect as measuring it at its last point of use, meaning that discarding a value is always akin to measuring it. Measurement may in general affect the states of values elsewhere in the program, which would be an unintuitive consequence if it occurred when we implicitly discarded a variable.

However, the measurement outcomes of a pure expression cannot affect the states of qubits it does not own, and thus the outcome of the measurement of q12 cannot have any impact on the remaining computation. In general, Twist supports implicitly measuring a pure expression when it is discarded and goes out of scope.

Existing languages such as Paykin et al. [2017]; Selinger [2004] forbid programs from implicitly discarding quantum data. The language Silq [Bichsel et al. 2020] can automatically *uncompute* certain temporary qubits, restoring their value to zero and obviating the need to explicitly measure them. However, in Figure 4, Silq would not be able to automatically uncompute q12, because teleportation invokes the Hadamard gate, which has a phase component that Silq cannot support. Thus, safely writing the teleportation program in any existing quantum programming language requires the user to explicitly measure q12 after manually verifying that it is separable from q3. By contrast, Twist leverages the guarantee made by its purity type system that the discarded value cannot impact the remaining computation, resulting in a concise and safe program.

[13] In quantum mechanics, mixed states have more general definitions of entanglement and separability than pure states. We show in Appendix A that the special case of *simple separability* is appropriate for reasoning about purity in Twist. This is fortunate, as testing for more general separability is an open problem for non-trivially-small cases [Harrow and Montanaro 2013; Horodecki et al. 1996], and was proved by Gurvits [2003] to be **NP**-hard in the dimension of the density matrix.

4 μQ LANGUAGE

To formally define evaluation and purity in quantum programs, we present μQ, a small quantum language. μQ is a functional language featuring classical control and quantum data, in the style of the linear quantum λ-calculus of Selinger and Valiron [2005].

In this section, we present μQ's syntax and operational semantics. In Section 5, we develop its semantic properties, including purity, in terms of executions in the operational semantics. In Section 6, we present μQ's denotational semantics, which provides a more concise definition of purity useful to formulate Twist's purity assertions. In Section 7, we develop the language Twist by adding to μQ the purity assertions that reason about purity within the language itself.

4.1 Syntax

The syntax of μQ consists of types, expressions, and programs.

$$\begin{aligned}
\text{Type } \tau &::= \texttt{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \texttt{qubit} \\
\text{Expression } e &::= x \mid f \mid e_1(e_2) \mid (e_1, e_2) \mid \texttt{let } (x, y) = e_1 \texttt{ in } e_2 \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{T} \mid \texttt{F} \\
&\quad\mid \texttt{qinit } () \mid U(e) \mid U_2(e) \mid \texttt{measure}(e) \mid \texttt{ref}[\alpha] \\
\text{Program } m &::= \texttt{fun } f\ x = e; m \mid \texttt{fun main } () = e
\end{aligned}$$

μQ features the types τ of classical Booleans, pairs, functions, and qubits. Its expressions e include the classical constructs of variables x, functions f, applications, pairs, and `let`[14] and `if`-expressions. Boolean literals, which we denote with metavariable b, are T and F.

Other expressions interact with the quantum state. The operator `qinit` () creates a new qubit initialized to $|0\rangle$. The operator $U(e)$ applies a single-qubit unitary to qubit e, and $U_2(e)$ applies a two-qubit unitary gate to a pair e of two qubits.[15] The operator $\texttt{measure}(e)$ performs a quantum measurement of e, returning the classical outcome. The last operator $\texttt{ref}[\alpha]$ is a *qubit reference* that only appears in intermediate evaluations, as μQ programs do not expose concrete qubit names. Finally, every program m is a sequence of function declarations followed by a `main` function.

4.2 Type System

Figure 7 presents the judgment $\Gamma \vdash_\Delta e : \tau$ that assigns the expression e the type τ given the *context* Γ and the *qubit context* Δ. A context Γ maps variables to types, and a qubit context Δ is a set of allocated qubits. We define the *classical types* to be Booleans, functions, and pairs of classical types. Both contexts are linear, except for classical types, which may be freely duplicated or discarded.[16] To ensure that qubits are not duplicated, we use the disjoint set union $\sqcup$, defined only when its arguments are disjoint. We populate the initial context with types of function declarations using the judgment $\Gamma \vdash m$ ok (defined in Appendix D), stating that a program m is well-formed.

The typing rules for variables, functions, applications, pairs, and `let`-expressions are standard as in a linear λ-calculus. We define the *quantum types* to be qubits and pairs of quantum types. The rule for `if` imposes a condition that the type of the branches is a quantum type, which will simplify

[14] We also use $\texttt{let } x = e_1 \texttt{ in } e_2$ as syntactic sugar for $\texttt{let } (x, \_) = (e_1, \texttt{T}) \texttt{ in } e_2$ (where T is arbitrarily chosen).

[15] For simplicity, we do not represent larger gates, which can be decomposed into single- and two-qubit operators using constructions like Kitaev [1997].

[16] To prevent qubits from being duplicated or discarded due to the quantum no-cloning [Wootters and Zurek 1982] and no-deleting theorems [Pati and Braunstein 2000], the type system of μQ does not allow the structural rules of contraction or weakening for quantum data.

$$\textsc{T-Var}\ \frac{}{x : \tau \vdash_{\emptyset} x : \tau} \qquad \textsc{T-Fun}\ \frac{}{f : \tau \to \tau' \vdash_{\emptyset} f : \tau \to \tau'} \qquad \textsc{T-App}\ \frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} e_1(e_2) : \tau_2}$$

$$\textsc{T-Pair}\ \frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} (e_1, e_2) : \tau_1 \times \tau_2} \qquad \textsc{T-Let}\ \frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x : \tau_1, y : \tau_2 \vdash_{\Delta_2} e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} \texttt{let } (x, y) = e_1 \texttt{ in } e_2 : \tau}$$

$$\textsc{T-If}\ \frac{\begin{array}{c} \tau \text{ is a quantum type} \\ \Gamma_1 \vdash_{\Delta_1} e : \texttt{bool} \qquad \Gamma_2 \vdash_{\Delta_2} e_1 : \tau \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau \end{array}}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau} \qquad \textsc{T-Bool}\ \frac{}{\cdot \vdash_{\emptyset} b : \texttt{bool}} \qquad \textsc{T-Qinit}\ \frac{}{\cdot \vdash_{\emptyset} \texttt{qinit}\ () : \texttt{qubit}}$$

$$\textsc{T-U1}\ \frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}}{\Gamma \vdash_{\Delta} U(e) : \texttt{qubit}} \qquad \textsc{T-U2}\ \frac{\Gamma \vdash_{\Delta} e : \texttt{qubit} \times \texttt{qubit}}{\Gamma \vdash_{\Delta} U_2(e) : \texttt{qubit} \times \texttt{qubit}} \qquad \textsc{T-Measure}\ \frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}}{\Gamma \vdash_{\Delta} \texttt{measure}(e) : \texttt{bool}} \qquad \textsc{T-Ref}\ \frac{}{\cdot \vdash_{\{\alpha\}} \texttt{ref}[\alpha] : \texttt{qubit}}$$

Fig. 7. μQ type system.

our presentation. Booleans have Boolean type and `qinit` () has qubit type. Unitary operators operate on one qubit or a pair of qubits, and have the same type as their argument. Measurement of a qubit results in a classical Boolean type.[17] Qubit references have qubit type.

4.3 Operational Semantics

Figure 8 presents an operational semantics for μQ as a probabilistic transition system over *program states* – pairs of quantum states $|\psi\rangle$ and classical expressions e.[18] The semantics depend on particular measurement outcomes described by an *outcome map* O. An outcome map is a set of pairs (α, b), each meaning that qubit α was measured, and classical outcome b was observed for it.

Judgments. The value judgment v val states that functions, Boolean literals, qubit references, and pairs of values are values. The step judgment $|\psi\rangle; e \xmapsto{O}_p |\psi'\rangle; e'$ states that the expression e under state $|\psi\rangle$ steps to e' and new state $|\psi'\rangle$ after observing the measurement outcomes in O with probability p. The evaluation judgment $|\psi\rangle; e \xmapsto{O}{}^{*}_{p} |\psi'\rangle; v$ states that expression e under state $|\psi\rangle$ evaluates to a value v and state $|\psi'\rangle$ having observed outcomes O with probability p.

Functions. The semantics makes use of an unchanging global function context ϕ that maps function names f to definitions $\lambda x.e$. A program executes by collecting all function definitions into ϕ and then evaluating the body of the `main` function until it reaches a value.

Operators. Figure 8 presents a selection of the rules, with the remaining left-to-right call-by-value rules presented in Appendix D. The first two rules step application and `let`-expressions in the standard way, with no effect on $|\psi\rangle$. The `qinit` () operator adds a new qubit named α in state $|0\rangle$ to $|\psi\rangle$ and steps to a reference to α. A single-qubit unitary operator on a qubit reference to α steps to its argument and a new state with U applied on qubit α. Similarly, a two-qubit unitary operator applied on qubits α and β steps to its argument and a new state with U applied on α and β. An

[17] We also allow measurement of tuples as syntactic sugar for sequentially measuring each component.

[18] Any program state can take exactly one or two transitions, with probabilities adding to one. For a formal model of probability in our semantics, see the probabilistic reduction system introduced by Selinger and Valiron [2005].

$$\textsc{S-App}\quad \frac{\phi(f) = \lambda x.e \qquad e' \ \mathsf{val}}{|\psi\rangle; f(e') \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; [e'/x]e}
\qquad\qquad
\textsc{S-Let}\quad \frac{e_1 \ \mathsf{val} \qquad e_2 \ \mathsf{val}}{|\psi\rangle; \mathtt{let}\ (x, y) = (e_1, e_2)\ \mathtt{in}\ e' \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; [e_1, e_2/x, y]e'}$$

$$\textsc{S-Qinit}\quad \frac{\alpha \text{ fresh in } |\psi\rangle}{|\psi\rangle; \mathtt{qinit}\ () \overset{\emptyset}{\longmapsto}_1 |\psi\rangle \otimes |0\rangle_\alpha; \mathtt{ref}[\alpha]}
\qquad\qquad
\textsc{S-U1}\quad \frac{}{|\psi\rangle; U(\mathtt{ref}[\alpha]) \overset{\emptyset}{\longmapsto}_1 U_\alpha\, |\psi\rangle; \mathtt{ref}[\alpha]}$$

$$\textsc{S-U2}\quad \frac{}{|\psi\rangle; U_2(\mathtt{ref}[\alpha], \mathtt{ref}[\beta]) \overset{\emptyset}{\longmapsto}_1 U_{\alpha,\beta}\, |\psi\rangle; (\mathtt{ref}[\alpha], \mathtt{ref}[\beta])}$$

$$\textsc{S-IfT}\quad \frac{}{|\psi\rangle; \mathtt{if}\ \mathsf{T}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; e_1}
\qquad\qquad
\textsc{S-IfF}\quad \frac{}{|\psi\rangle; \mathtt{if}\ \mathsf{F}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; e_2}$$

$$\textsc{S-MeasureT}\quad \frac{M_\alpha\, |\psi\rangle = |1\rangle_\alpha \otimes |\psi'\rangle \text{ w.p. } p \qquad O = \{(\alpha, \mathsf{T})\}}{|\psi\rangle; \mathtt{measure}(\mathtt{ref}[\alpha]) \overset{O}{\longmapsto}_p |\psi'\rangle; \mathsf{T}}
\qquad\qquad
\textsc{S-MeasureF}\quad \frac{M_\alpha\, |\psi\rangle = |0\rangle_\alpha \otimes |\psi'\rangle \text{ w.p. } p \qquad O = \{(\alpha, \mathsf{F})\}}{|\psi\rangle; \mathtt{measure}(\mathtt{ref}[\alpha]) \overset{O}{\longmapsto}_p |\psi'\rangle; \mathsf{F}}$$

$$\textsc{E-Val}\quad \frac{v \ \mathsf{val}}{|\psi\rangle; v \overset{\emptyset}{\longmapsto}^*_1 |\psi\rangle; v}
\qquad\qquad
\textsc{E-Step}\quad \frac{|\psi\rangle; e \overset{O_1}{\longmapsto}_{p_1} |\psi'\rangle; e' \qquad |\psi'\rangle; e' \overset{O_2}{\longmapsto}^*_{p_2} |\psi''\rangle; v \qquad O' = O_1 \cup O_2}{|\psi\rangle; e \overset{O'}{\longmapsto}^*_{p_1 p_2} |\psi''\rangle; v}$$

Fig. 8. Selected μQ operational semantics rules. The full rules are given in Appendix D.

`if`-expression chooses a branch to step to depending on the condition. Measurement of a qubit has two probabilistic outcomes, classical true or false, with a new state under each outcome. The step rules for measurement transition to each outcome with its occurrence probability (Section 2).

Notation. We define the following new notations for judgments.

- $|\psi\rangle; e \ \longmapsto \cdot$ means that there exists a set I where for each $i \in I$, there exist $p_i > 0$, O_i, $|\psi_i\rangle$ and e_i where $|\psi\rangle; e \overset{O_i}{\longmapsto}_{p_i} |\psi_i\rangle; e_i$ and $\sum_i p_i = 1$.
- $|\psi\rangle; e \longmapsto |\psi'\rangle; e'$ means that there exist O and p where $|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'$.
- $|\psi\rangle; e \longmapsto^* |\psi'\rangle; e'$ means that there exist O and p where $|\psi\rangle; e \overset{O}{\longmapsto}^*_p |\psi'\rangle; e'$.

4.4 Type Safety

We now state the type safety properties of μQ. Progress states that a well-typed expression is a value or can step under a quantum state containing all qubits that the expression references.

Theorem 4.1 (Progress). *If* $\cdot \vdash_\Delta e : \tau$*, then* e *val or for all* $|\psi\rangle$ *where* $\Delta \subseteq \operatorname{dom} |\psi\rangle$*,* $|\psi\rangle; e \ \longmapsto \cdot$.

The proof is by induction on the derivation of $\cdot \vdash_\Delta e : \tau$. The preservation theorem states that a step preserves the type of the expression under the new qubit context.

Theorem 4.2 (Preservation). *If* $\Gamma \vdash_\Delta e : \tau$ *and* $\Delta \subseteq \operatorname{dom} |\psi\rangle$ *and* $|\psi\rangle; e \longmapsto |\psi'\rangle; e'$*, then we have* $\Gamma \vdash_{\Delta'} e' : \tau$ *where* $\Delta' \subseteq \operatorname{dom} |\psi'\rangle$.

The proof is by induction on the derivation of $|\psi\rangle; e \longmapsto |\psi'\rangle; e'$.

5 SEMANTIC PROPERTIES

In this section, we define the semantic property of purity using the operational semantics of μQ. Purity states that an expression executes independently of measurement outcomes of unowned qubits, or equivalently, the qubits owned by a pure expression are separable from those it does not own. Formally, purity states that there is only one possible final program state after evaluating an expression and measuring all qubits that it does not own.

5.1 Implicit Measurement

An expression e evaluates under a quantum state $|\psi\rangle$ to a value v and state $|\psi'\rangle$. The resulting state $|\psi'\rangle$ may contain qubits to which v does not refer. If these qubits are measured later in the program, then their measurement outcomes may affect the state of the qubits in v through entanglement.

To capture the effect of the eventual measurement of unowned qubits, we define a relation called *implicit measurement.* Given a state $|\psi\rangle$ and a value v, implicit measurement measures all qubits in $|\psi\rangle$ to which v does not refer. Define $\mathsf{Refs}(v)$ to be the sequence, or set when order is irrelevant, of qubit names referenced in v.

Definition 5.1 (Implicit Measurement). A state $|\psi\rangle$ *implicitly measures* modulo a value v to produce a new state $|\psi_v\rangle$ with probability p, written $|\psi\rangle; v \Downarrow_p |\psi_v\rangle$, when $M_A |\psi\rangle = |\psi_A\rangle \otimes |\psi_v\rangle$ with probability p where $A = \mathrm{dom}\,|\psi\rangle \setminus \mathsf{Refs}(v)$ and $\mathrm{dom}\,|\psi_v\rangle = \mathsf{Refs}(v)$.

We use notation $|\psi\rangle; v \Downarrow |\psi'\rangle$ to denote that there exists some p such that $|\psi\rangle; v \Downarrow_p |\psi'\rangle$. A value has a unique implicit measurement in $|\psi\rangle$ if and only if its referenced qubits are separable in $|\psi\rangle$.

5.2 Qubit Equivalence

We consider two program states $(|\psi_1\rangle, v_1)$ and $(|\psi_2\rangle, v_2)$ to be equivalent if they are equal up to consistent renaming of qubits. For example, we define

$$(|0\rangle_\alpha \otimes |1\rangle_\beta, (\mathtt{ref}[\alpha], \mathtt{ref}[\beta])) \equiv (|1\rangle_\gamma \otimes |0\rangle_\delta, (\mathtt{ref}[\delta], \mathtt{ref}[\gamma]))$$

because they can be substituted for each other in a program with no operational effect.

Definition 5.2 (Qubit Equivalence). $(|\psi_1\rangle, v_1) \equiv (|\psi_2\rangle, v_2)$ holds when $|\psi_2\rangle [\ell_1/\ell_2] = |\psi_1\rangle$ and $v_2[\ell_1/\ell_2] = v_1$ for two duplicate-free sequences of qubit names ℓ_1, ℓ_2 of equal length.

$\equiv$ is reflexive, symmetric, and transitive, making it an equivalence relation. We define the action of rewriting using qubit equivalence as replacing a value v_1 of quantum type with any qubit-equivalent value v_2 of the same type by also modifying the quantum state correspondingly.

5.3 Purity

The *purity* property states that a state and expression always evaluate and implicitly measure to a unique final state and value up to qubit equivalence.

Definition 5.3 (Purity). An expression e is *pure* under state $|\psi\rangle$, denoted $|\psi\rangle; e$ pure, when if $|\psi\rangle; e \mapsto^* |\psi_1\rangle; v_1$ and $|\psi_1\rangle; v_1 \Downarrow |\psi_1'\rangle$, and also $|\psi\rangle; e \mapsto^* |\psi_2\rangle; v_2$ and $|\psi_2\rangle; v_2 \Downarrow |\psi_2'\rangle$, then we have $(|\psi_1'\rangle, v_1) \equiv (|\psi_2'\rangle, v_2)$.

Purity asserts that under state $|\psi\rangle$, expression e evaluates to a unique value v and final state $|\psi'\rangle$ where v has a unique implicit measurement in $|\psi'\rangle$. This definition formalizes the intuition that the eventual measurement outcome of unowned qubits cannot affect the state of those it owns.

6 DENOTATIONAL SEMANTICS OF μQ

The operational definition of purity quantifies over all possible executions of a μQ program. In this section, we present the denotational semantics for μQ, which enables a more concise definition of purity based on the closed-form denotation of an expression. Denotational semantics reasons directly about an expression's effect on the distribution of the program state across all executions, using mixed states represented as density matrices to describe distributions over pure states.

Eliminating Nondeterminism. A μQ program executes nondeterministically in terms of both its quantum and classical state. An expression may evaluate to multiple distinct values when measurement outcomes influence classical control flow. Because this nondeterminism in the program's classical state complicates the development of a denotational semantics, we instead force all nondeterminism to occur in the quantum state.

Any expression of function type always evaluates to a unique value, by a simple induction argument over the operational semantics. For Booleans, we utilize the deferred measurement principle (Section 2) to interpret Booleans as deferred qubit measurements that are not resolved until the end of the program. For quantum types, we canonicalize values that differ only in the order of appearance of qubits. To do so, we use rewriting under qubit equivalence (Section 5) to dynamically rename qubits so that every value refers to the same qubits in the same order.

6.1 Semantics

Figure 9 presents the denotational semantics of μQ. The denotation of an expression e is a function from a context γ mapping variables to values and an input partial density matrix ρ to the final partial density matrix ρ' and value v to which e evaluates. The denotation of a program m adds functions to the initial γ and is always a normalized density matrix.

Basic Operators. The values of variables reside in the context γ. Values do not evaluate further. Pairs and `let`-bindings propagate the state through their evaluation. The denotation of function application is the application of f, the denotation of e_1, to the denotation of e_2. The denotation of `qinit` () adds a new qubit α in state $|0\rangle$ by tensor product with its density matrix form, which is an outer product $|0\rangle\langle 0|$. The denotation of unitary operators performs matrix conjugation (Section 2).

Measurement and Conditional Branches. The denotations of `if` and `measure` encode deferred measurement. A measurement has no immediate effect and simply evaluates its argument, where the notation `ref`$^\star[\alpha]$ differentiates a measured from an unmeasured qubit. An `if`-expression first determines whether its condition is a literal T or F and if so executes an appropriate branch.

Otherwise, the condition is a deferred measurement of a qubit α. The denotation computes the partial density matrices corresponding to each outcome using the matrix P_α projecting qubit α to $|1\rangle$ in ρ. It executes the corresponding branches to obtain matrices ρ_1 and ρ_2. It next corrects their dimensions using match_sizes, defined as taking tensor product with copies of $|0\rangle\langle 0|$ on the smaller matrix. This operation allocates the same number of qubits created by `qinit` to both branches.

The values v_1 and v_2 returned by the two branches are of the same type but may refer to different qubits. The denotation unifies v_1 and v_2 by renaming each qubit in v_2 and ρ_2' into the corresponding reference in v_1. It adds the two partial density matrices to weigh each outcome by its probability.

6.2 Semantics Equivalence

In this section, we show that the operational and denotational semantics are equivalent in terms of final program states under particular measurement outcomes. Let ϕ denote the initial context containing only top-level function declarations. Given an outcome map O and a value v, we define

$$
\begin{aligned}
\llbracket \mathtt{fun}\ f\ x = e; m \rrbracket_\gamma &= \text{let } \gamma' = \gamma[\lambda v.\, \llbracket e \rrbracket_{\gamma[v/x]} / f] \text{ in } \llbracket m \rrbracket_{\gamma'} \\
\llbracket \mathtt{fun\ main}\ () = e \rrbracket_\gamma &= \llbracket e \rrbracket_\gamma\, (|\cdot\rangle\langle\cdot|) \\
\\
\llbracket x \rrbracket_\gamma\, (\rho) &= (\rho, \gamma(x)) \\
\llbracket v \rrbracket_\gamma\, (\rho) &= (\rho, v) \text{ when } v \text{ val} \\
\llbracket (e_1, e_2) \rrbracket_\gamma\, (\rho) &= \text{let } \rho_1, v_1 = \llbracket e_1 \rrbracket_\gamma\, (\rho) \text{ in let } \rho_2, v_2 = \llbracket e_2 \rrbracket_\gamma\, (\rho_1) \text{ in } (\rho_2, (v_1, v_2)) \\
\llbracket \mathtt{let}\ (x, y) = e_1\ \mathtt{in}\ e_2 \rrbracket_\gamma\, (\rho) &= \text{let } \rho_1, (v_1, v_2) = \llbracket e_1 \rrbracket_\gamma\, (\rho) \text{ in } \llbracket e_2 \rrbracket_{\gamma[v_1, v_2/x, y]}\, (\rho_1) \\
\llbracket e_1(e_2) \rrbracket_\gamma\, (\rho) &= \text{let } \rho_1, f = \llbracket e_1 \rrbracket_\gamma\, (\rho) \text{ in let } \rho_2, v = \llbracket e_2 \rrbracket_\gamma\, (\rho_1) \text{ in } f(v)(\rho_2) \\
\llbracket \mathtt{qinit}\ () \rrbracket_\gamma\, (\rho) &= (\rho \otimes |0\rangle\langle 0|_\alpha, \mathtt{ref}[\alpha]) \text{ where } \alpha \text{ is fresh in } \rho \\
\llbracket U(e) \rrbracket_\gamma\, (\rho) &= \text{let } \rho', \mathtt{ref}[\alpha] = \llbracket e \rrbracket_\gamma\, (\rho) \text{ in } (U_\alpha \rho' U_\alpha^\dagger, \mathtt{ref}[\alpha]) \\
\llbracket U_2(e) \rrbracket_\gamma\, (\rho) &= \text{let } \rho', (\mathtt{ref}[\alpha], \mathtt{ref}[\beta]) = \llbracket e \rrbracket_\gamma\, (\rho) \text{ in } (U_{\alpha,\beta} \rho' U_{\alpha,\beta}^\dagger, (\mathtt{ref}[\alpha], \mathtt{ref}[\beta])) \\
\llbracket \mathtt{measure}(e) \rrbracket_\gamma\, (\rho) &= \text{let } \rho', \mathtt{ref}[\alpha] = \llbracket e \rrbracket_\gamma\, (\rho) \text{ in } (\rho', \mathtt{ref}^\star[\alpha]) \\
\llbracket \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rrbracket_\gamma\, (\rho) &= \text{let } \rho', v = \llbracket e \rrbracket_\gamma\, (\rho) \text{ in case } v \text{ of } \mathsf{T} \to \llbracket e_1 \rrbracket_\gamma\, (\rho') \mid \mathsf{F} \to \llbracket e_2 \rrbracket_\gamma\, (\rho') \mid \mathtt{ref}^\star[\alpha] \to \\
& \text{let } P_\alpha = |1\rangle\langle 1|_\alpha \otimes I_{\operatorname{dom}\rho \setminus \{\alpha\}} \text{ in} \\
& \text{let } \rho_1, v_1 = \llbracket e_1 \rrbracket_\gamma\, (P_\alpha \rho P_\alpha) \text{ and } \rho_2, v_2 = \llbracket e_2 \rrbracket_\gamma\, ((I_{\operatorname{dom}\rho} - P_\alpha)\rho(I_{\operatorname{dom}\rho} - P_\alpha)) \text{ in} \\
& \text{let } \rho_1', \rho_2' = \mathsf{match\_sizes}(\rho_1, \rho_2) \text{ in } (\rho_1' + \rho_2'[\mathsf{Refs}(v_1)/\mathsf{Refs}(v_2)], v_1)
\end{aligned}
$$

Fig. 9. μQ denotational semantics.

$\mathsf{apply}(O, v)$ to be the value syntactically identical to v except that every instance of $\mathtt{ref}^\star[\alpha]$ is replaced with b where $(\alpha, b) \in O$. In the opposite direction, we define $\mathsf{defer}(O)$ to be the tensor product of all outcomes in O expressed as quantum states:

$$\mathsf{defer}(O) = \bigotimes \{\text{if } b = \mathsf{T} \text{ then } |1\rangle_\alpha \text{ else } |0\rangle_\alpha \mid (\alpha, b) \in O\}$$

The following theorem states that the denotation of an expression captures every execution up to qubit equivalence. ρ contains all final $|\psi_i\rangle$ and outcomes O_i and some number of unused padding qubits, and deferred and operational measurement outcomes align in the final value v.

Theorem 6.1. *Given $|\psi\rangle$ and e, let the multiset $S = [(p_i, |\psi_i\rangle, v_i, O_i) \mid |\psi\rangle; e \overset{O_i}{\longmapsto}{}^{*}_{p_i} |\psi_i\rangle; v_i]$. Then,*

$$\llbracket e \rrbracket_\phi\, (|\psi\rangle\langle\psi|) = (\rho, v) \textit{ where } \rho = \textstyle\sum_{(p_i, |\psi_i\rangle, v_i, O_i) \in S} p_i\, |\psi_i'\rangle\langle\psi_i'| \textit{ and } (|\psi_i'\rangle, v) \equiv (|\psi_i''\rangle, v')$$
$$\textit{where } |\psi_i\rangle \otimes \textit{defer}(O_i) \otimes |0\rangle^* = |\psi_i''\rangle \textit{ and } v_i = \textit{apply}(O_i, v').$$

The proof is by induction on the derivations of $|\psi\rangle; e \overset{O_i}{\longmapsto}{}^{*}_{p_i} |\psi_i\rangle; v_i$. Most operators in the denotational semantics follow a left-to-right eager evaluation of the arguments and return the same value as the step rule in the operational semantics. The interesting cases are for unitary operators, `if`-expressions, and measurement, where we appeal to the correspondence between the state vector and density matrix models of quantum mechanics as well as the principle of deferred measurement.

6.3 Purity

The following corollary provides an equivalent definition of purity using denotational semantics.

Corollary 6.2. $|\psi\rangle; e$ pure *holds if and only if* $\llbracket e \rrbracket_\phi\, (|\psi\rangle\langle\psi|) = (\rho \otimes \rho', v)$ *where* $\operatorname{dom}\rho = \textit{Refs}(v)$ *and ρ is a pure state, i.e. $\rho = |\psi'\rangle\langle\psi'|$ for some $|\psi'\rangle$.*

Thus, computing the denotation of e under $|\psi\rangle$ gives us a direct way of testing whether $|\psi\rangle; e$ pure. We leverage this fact in the following sections to define the *purity assertion* operators of Twist.

7 TWIST LANGUAGE

In this section, we present the formal core of the Twist language. Twist extends μQ with a *purity type* system that specifies which expressions are pure. We present the two *purity assertion* operators that specify and check purity in a quantum program.

We present both a denotational and operational semantics of Twist. Denotationally, we define purity assertions using mathematical *separability conditions* on mixed-state denotations. Operationally, we implement them using a *separability test* primitive on runtime pure states.

7.1 Syntax

To develop Twist, we augment the syntax of μQ as follows, where new syntax is in black:[19]

$$
\begin{aligned}
\text{Purity } \pi &::= \mathbf{P} \mid \mathbf{M} \\
\text{Quantum type } \varrho &::= \texttt{qubit} \mid \varrho_1 \,\&\, \varrho_2 \\
\text{Type } \tau &::= \texttt{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \varrho^\pi \\
\text{Quantum value } q &::= \texttt{ref}[\alpha] \mid [q_1, q_2] \\
\text{Expression } e &::= x \mid f \mid e_1(e_2) \mid (e_1, e_2) \mid \texttt{let } (x, y) = e_1 \texttt{ in } e_2 \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{T} \mid \texttt{F} \\
&\mid \texttt{qinit } () \mid U(e) \mid U_2(e) \mid \texttt{measure}(e) \mid q^\pi \\
&\mid \texttt{entangle}_\pi(e) \mid \texttt{split}_\pi(e) \mid \texttt{cast}_\pi(e)
\end{aligned}
$$

We introduce purity annotations π, either pure, **P**, or mixed, **M**. We also introduce a sort of *quantum types* ϱ,[20] either a `qubit` or an *entangled pair* $\varrho_1 \,\&\, \varrho_2$ of two quantum types. They describe *quantum values* q, either a reference to a qubit or an entangled pair $[q_1, q_2]$ of two quantum values.

The new forms of expressions manipulate purity and have no effect on the quantum state. A purity-annotated quantum value q^π is an expression that only arises in intermediate evaluations. The $\texttt{entangle}_\pi(e)$ operator constructs an entangled pair from its argument, the $\texttt{split}_\pi(e)$ operator destructs an entangled pair into two components, and $\texttt{cast}_\pi(e)$ modifies the purity of an expression.

7.2 Type System

Figure 10 presents the type system of Twist, extending the type system of μQ to the new operators. The new judgment $\vdash_\Delta q : \varrho$ assigns the quantum value q the quantum type ϱ under the context Δ. A reference to qubit α has qubit type under a context only containing α, and an entangled pair has entangled pair type if its components have quantum types.

We next modify the typing judgment $\Gamma \vdash_\Delta e : \tau$, showing only the rules that change, with the remainder in Appendix D. The rule for `if` requires its branches to have the same quantum type and now returns a mixed version of that type. The `qinit` $()$ operator returns a pure qubit. Unitary operators operate on a qubit or entangled pair of any purity and have the same purity as their argument. The $\texttt{measure}(e)$ operator accepts a qubit of any purity and returns a Boolean. A purity-annotated quantum value q^π has type ϱ^π if q has quantum type ϱ.

Entangled Pairs and Purity Assertions. The purpose of the entangled pair type is to denote values that may have been entangled by a two-qubit unitary operator. The $\texttt{entangle}_\pi$ operator creates an entangled pair of purity π from an ordinary pair of quantum values of that same purity.

[19] We describe in Appendix C additional syntactic features of Twist, such as implicit measurement of discarded pure values.

[20] Lowercase Greek letter qoppa.

$$\textsc{Q-Ref}\quad \frac{}{\vdash_{\{\alpha\}} \texttt{ref}[\alpha] : \texttt{qubit}} \qquad \textsc{Q-Pair}\quad \frac{\vdash_{\Delta_1} q_1 : \varrho_1 \qquad \vdash_{\Delta_2} q_2 : \varrho_2}{\vdash_{\Delta_1 \sqcup \Delta_2} [q_1, q_2] : \varrho_1 \,\&\, \varrho_2}$$

$$\textsc{T-If}\quad \frac{\Gamma_1 \vdash_{\Delta_1} e : \texttt{bool} \qquad \Gamma_2 \vdash_{\Delta_2} e_1 : \varrho^{\pi} \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \varrho^{\pi}}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \varrho^{\mathbf{M}}}$$

$$\textsc{T-Qinit}\quad \frac{}{\cdot \vdash_{\emptyset} \texttt{qinit}\ () : \texttt{qubit}^{\mathbf{P}}} \qquad \textsc{T-U1}\quad \frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}^{\pi}}{\Gamma \vdash_{\Delta} U(e) : \texttt{qubit}^{\pi}} \qquad \textsc{T-U2}\quad \frac{\Gamma \vdash_{\Delta} e : (\texttt{qubit} \,\&\, \texttt{qubit})^{\pi}}{\Gamma \vdash_{\Delta} U_2(e) : (\texttt{qubit} \,\&\, \texttt{qubit})^{\pi}}$$

$$\textsc{T-Measure}\quad \frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}^{\pi}}{\Gamma \vdash_{\Delta} \texttt{measure}(e) : \texttt{bool}} \qquad \textsc{T-Qval}\quad \frac{\vdash_{\Delta} q : \varrho}{\cdot \vdash_{\Delta} q^{\pi} : \varrho^{\pi}} \qquad \textsc{T-Entangle}\quad \frac{\Gamma \vdash_{\Delta} e : {\varrho_1}^{\pi} \times {\varrho_2}^{\pi}}{\Gamma \vdash_{\Delta} \texttt{entangle}_{\pi}(e) : (\varrho_1 \,\&\, \varrho_2)^{\pi}}$$

$$\textsc{T-Split}\quad \frac{\Gamma \vdash_{\Delta} e : (\varrho_1 \,\&\, \varrho_2)^{\pi}}{\Gamma \vdash_{\Delta} \texttt{split}_{\pi}(e) : {\varrho_1}^{\pi} \times {\varrho_2}^{\pi}} \qquad \textsc{T-Cast}\quad \frac{\Gamma \vdash_{\Delta} e : \varrho^{\pi'}}{\Gamma \vdash_{\Delta} \texttt{cast}_{\pi}(e) : \varrho^{\pi}}$$

Fig. 10. Twist purity type system. Only rules changed from Figure 7 shown; full rules in Appendix D.

When a program extracts the two components of an entangled pair of any purity, conservatively the type system assumes they became entangled and now constitute mixed states, and assigns them mixed type. The operator $\texttt{split}_{\mathbf{M}}(e)$ destructs a mixed entangled pair into two mixed components, and $\texttt{cast}_{\mathbf{M}}(e)$ sets the purity of any expression to mixed.

Two *purity assertion* operators obtain pure types for quantum expressions. The *purifying-split* operator $\texttt{split}_{\mathbf{P}}$ destructs a pure entangled pair into two pure components, and the *purifying-cast* operator $\texttt{cast}_{\mathbf{P}}$ sets the purity of any expression to pure. To ensure that expressions of pure type are actually pure, the semantics of the purity assertions impose conditions on their usage.

7.3 Denotational Semantics

Figure 11 presents the denotational semantics of Twist, extending the semantics of μQ to the new operators. We define $(\!| v |\!)$ to strip purity annotations and replace entangled pairs with ordinary pairs in v, so that the denotation of a value v is $(\!| v |\!)$. Operators $\texttt{entangle}_{\pi}$, $\texttt{cast}_{\mathbf{M}}$, and $\texttt{split}_{\mathbf{M}}$ only manipulate purity annotations, and so their denotation is simply the denotation of their argument.

$\texttt{split}_{\mathbf{P}}$ and $\texttt{cast}_{\mathbf{P}}$ evaluate their arguments and then assert a condition about the resulting partial density matrix. If the condition holds, the operator leaves the state unchanged. Otherwise, its denotation is the special element $\bot$ corresponding to the program aborting at runtime.

Separability Conditions. Given a partial density matrix ρ and a partition of its domain, a *separability condition* states that ρ is simply separable into sub-states whose domains are the qubit sets of the partition. The $\texttt{split}_{\mathbf{P}}$ operator asserts that ρ is simply separable into, i.e. is the product of three sub-states, where two are pure and correspond to q_1 and q_2. The definition of $\texttt{cast}_{\mathbf{P}}$ is similar, asserting that the state has a pure sub-state corresponding to q.

Path Sensitivity. A purity assertion under one branch of an `if`-expression checks that an expression is pure across states satisfying that branch of the `if`-condition. The typing rule for `if` ensures that a pure value inside a branch will be considered mixed at the end of the `if`-expression. For example, suppose x and y are two qubits in a Bell pair. Each $\texttt{cast}_{\mathbf{P}}$ within

$$\texttt{if measure}(x)\ \texttt{then}\ \texttt{cast}_{\mathbf{P}}(y)\ \texttt{else}\ \texttt{cast}_{\mathbf{P}}(y) : \texttt{qubit}^{\mathbf{M}}$$

$$
\begin{aligned}
\llparenthesis\, v \,\rrparenthesis &= v \text{ when } v \text{ is } f,\, b, \text{ or } \mathtt{ref}[\alpha] \\
\llparenthesis\, q^{\pi} \,\rrparenthesis &= \llparenthesis\, q \,\rrparenthesis \\
\llparenthesis\, [q_1, q_2] \,\rrparenthesis &= (\llparenthesis\, q_1 \,\rrparenthesis, \llparenthesis\, q_2 \,\rrparenthesis)
\end{aligned}
$$

$$
\begin{aligned}
\llbracket v \rrbracket_{\gamma}(\rho) &= (\rho, \llparenthesis\, v \,\rrparenthesis) \text{ when } v \text{ val} \\
\llbracket \mathtt{entangle}_{\pi}(e) \rrbracket = \llbracket \mathtt{cast}_{\mathbf{M}}(e) \rrbracket &= \llbracket \mathtt{split}_{\mathbf{M}}(e) \rrbracket = \llbracket e \rrbracket \\
\llbracket \mathtt{split}_{\mathbf{P}}(e) \rrbracket_{\gamma}(\rho) &= \llbracket e \rrbracket_{\gamma}(\rho) \;\text{ if }\; \llbracket e \rrbracket_{\gamma}(\rho) = (\rho_1 \otimes \rho_2 \otimes \rho_0, (q_1, q_2)) \text{ else } \bot \\
&\quad \text{where } \operatorname{dom} \rho_1 = \mathrm{Refs}(q_1) \text{ and } \operatorname{dom} \rho_2 = \mathrm{Refs}(q_2) \\
&\quad \text{and } \rho_1 \text{ and } \rho_2 \text{ are pure} \\
\llbracket \mathtt{cast}_{\mathbf{P}}(e) \rrbracket_{\gamma}(\rho) &= \llbracket e \rrbracket_{\gamma}(\rho) \;\text{ if }\; \llbracket e \rrbracket_{\gamma}(\rho) = (\rho_1 \otimes \rho_0, q) \text{ else } \bot \\
&\quad \text{where } \operatorname{dom} \rho_1 = \mathrm{Refs}(q) \text{ and } \rho_1 \text{ is pure}
\end{aligned}
$$

Fig. 11. Selected Twist denotational semantics. Only rules changed from Figure 9 shown.

S-IfT

$$\frac{}{|\psi\rangle; \mathtt{if}\ \mathsf{T}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \xmapsto{\emptyset}_1 |\psi\rangle; \mathtt{cast}_{\mathbf{M}}(e_1)}$$

S-IfF

$$\frac{}{|\psi\rangle; \mathtt{if}\ \mathsf{F}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \xmapsto{\emptyset}_1 |\psi\rangle; \mathtt{cast}_{\mathbf{M}}(e_2)}$$

S-Entangle

$$\frac{}{|\psi\rangle; \mathtt{entangle}_{\pi}({q_1}^{\pi}, {q_2}^{\pi}) \xmapsto{\emptyset}_1 |\psi\rangle; [q_1, q_2]^{\pi}}$$

S-SplitMixed

$$\frac{}{|\psi\rangle; \mathtt{split}_{\mathbf{M}}([q_1, q_2]^{\mathbf{M}}) \xmapsto{\emptyset}_1 |\psi\rangle; ({q_1}^{\mathbf{M}}, {q_2}^{\mathbf{M}})}$$

S-SplitPure

$$\frac{|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_0\rangle \qquad \operatorname{dom} |\psi_1\rangle = \mathrm{Refs}(q_1) \qquad \operatorname{dom} |\psi_2\rangle = \mathrm{Refs}(q_2)}{|\psi\rangle; \mathtt{split}_{\mathbf{P}}([q_1, q_2]^{\mathbf{P}}) \xmapsto{\emptyset}_1 |\psi\rangle; ({q_1}^{\mathbf{P}}, {q_2}^{\mathbf{P}})}$$

S-Cast

$$\frac{(\textit{unchecked})}{|\psi\rangle; \mathtt{cast}_{\pi}(q^{\pi'}) \xmapsto{\emptyset}_1 |\psi\rangle; q^{\pi}}$$

Fig. 12. Selected Twist operational semantics. The full rules are given in Appendix D. Rule (S-Cast) is unsound because purity cannot be verified using $|\psi\rangle$ on one execution alone. Combining this semantics with the static analysis in Section 8 enables building a sound interpreter.

is valid because if x was measured to be $|0\rangle$, y must also be in the pure state $|0\rangle$, and measuring $|1\rangle$ for x would likewise yield $|1\rangle$ for y. Nevertheless, the expression overall is mixed – the type of `if` is always mixed because the type system cannot assume any particular outcome for x.

Verifying Separability Conditions. Soundly verifying Twist's purity assertions requires verifying their separability conditions. If desired, a mixed-state quantum simulator can be used for this purpose. The simulator determines whether a density matrix is simply separable by taking its partial trace and executing the rank test (Section 2). Though this approach supports simulating all well-typed Twist programs, it is tied to a computationally inefficient mixed-state simulator.

7.4 Operational Semantics

The operational semantics of Twist manipulates a pure quantum state over an individual program execution. The semantics verifies separability conditions concretely using a *separability test* primitive that determines whether the runtime state is separable or entangled.

Figure 12 presents the updated step judgment, and in Appendix D, we modify the value judgment to state that q^{π} val. For `if`-expressions, because a different evaluation of the condition could have resulted in taking the other branch, the result may depend on the measurement outcome of some unspecified qubit. Thus, the rule for `if` casts its output to mixed. The $\mathtt{entangle}_{\pi}$ operator creates an entangled pair, and $\mathtt{split}_{\mathbf{M}}$ destructs an entangled pair into an ordinary pair containing its components annotated as mixed. We next define semantics for $\mathtt{split}_{\mathbf{P}}$ and $\mathtt{cast}_{\mathbf{P}}$.

Separability Tests. Given the pure state $|\psi\rangle$ and a partition of its domain, a *separability test* determines whether $|\psi\rangle$ is separable into pure sub-states whose domains are the sets in the partition.

Verifying `split`. The `split` operator soundly verifies purity using a separability test on the runtime quantum state $|\psi\rangle$. Sound verification is possible because the typing rule for $\texttt{split}_\mathbf{P}(e)$ guarantees that e is pure. The expression $\texttt{split}_\mathbf{P}(e)$ evaluates under state $|\psi\rangle$ by first evaluating e to a unique value $[q_1, q_2]^\mathbf{P}$ and state $|\psi'\rangle$. Purity guarantees that no qubit in $[q_1, q_2]$ is entangled with the rest of $|\psi'\rangle$. Thus, $|\psi'\rangle = |\psi_{12}\rangle \otimes |\psi_0\rangle$ where $\operatorname{dom} |\psi_{12}\rangle = \mathsf{Refs}([q_1, q_2])$, and furthermore $|\psi_{12}\rangle$ is identical across all executions. To verify the purity of q_1 and q_2, the premises of the $\texttt{split}_\mathbf{P}$ step rule test whether $|\psi_{12}\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$ where $\operatorname{dom} |\psi_1\rangle = \mathsf{Refs}(q_1)$ and $\operatorname{dom} |\psi_2\rangle = \mathsf{Refs}(q_2)$.

Verifying `cast`. By contrast, the state of the current execution alone cannot indicate whether a $\texttt{cast}_\mathbf{P}$ is sound. For example, suppose that x and y are in a Bell pair, and consider the expression:

$$(\texttt{measure}(x), \texttt{cast}_\mathbf{P}(y)) : \texttt{bool} \times \texttt{qubit}^\mathbf{P}$$

This expression returns a qubit y that is in a mixed state because y was entangled with x before x was measured, and hence the $\texttt{cast}_\mathbf{P}$ assertion is invalid. However, on any particular execution of the program, y appears to take on a pure state of either $|0\rangle$ or $|1\rangle$. Upon reaching the $\texttt{cast}_\mathbf{P}$, it is not possible to determine that y is mixed, because the probabilistic branch has already occurred.

Thus, the operational semantics for $\texttt{cast}_\mathbf{P}$ cannot precisely verify its separability condition and is necessarily unsound or incomplete. Figure 12 presents an operational semantics in which $\texttt{cast}_\mathbf{P}$ is unsound and does not verify purity. Instead, we present in Section 8 a sound static analysis that guarantees that all uses of $\texttt{cast}_\mathbf{P}$ occur on pure expressions. Combining this analysis with the operational semantics allows Twist to verify a large class of programs featuring purity assertions using only the runtime pure state of the program.

Executing Separability Tests. The operational semantics uses a separability test primitive that operates on the runtime pure quantum state. Though this work does not focus on the hardware implementation of this primitive, we overview in Appendix B a proposed implementation based on work by Harrow and Montanaro [2013]. In a pure-state simulator, Twist verifies $\texttt{split}_\mathbf{P}$ using the Schmidt decomposition (Section 2). Twist computes the Schmidt coefficients of a state vector by interpreting it as a matrix and taking its singular value decomposition using well-studied algorithms [Golub and Reinsch 1970]. Then, Twist checks that there is only one nonzero coefficient.

7.5 Type Safety

We now state the type safety properties for Twist. We introduce a judgment to state that an attempt to step a $\texttt{split}_\mathbf{P}$ aborts the program if its separability condition fails, denoted $|\psi\rangle; e \not\mapsto_{\texttt{split}}$ and fully defined in Appendix D. The progress theorem states that well-typed closed expressions are values, can step, or will fail at runtime due to failing a $\texttt{split}_\mathbf{P}$ separability condition:

Theorem 7.1 (Progress). *If* $\cdot \vdash_\Delta e : \tau$, *then either* e *val or for all* $|\psi\rangle$ *where* $\Delta \subseteq \operatorname{dom} |\psi\rangle$, *we have either* $|\psi\rangle; e \ \mapsto \ \cdot$ *or* $|\psi\rangle; e \not\mapsto_{\texttt{split}}$.

The proof is given in Appendix F.1.1 and is analogous to Theorem 4.1, extended to the operators in Twist. The preservation theorem is identical to Theorem 4.2 and proved analogously.

7.6 Semantics Equivalence

The denotational semantics constrain valid operational executions in the presence of the $\texttt{cast}_\mathbf{P}$ operator. The following theorem modifies Theorem 6.1 by stating that if the denotation of a program is a valid density matrix, then it agrees with the operational semantics:

Theorem 7.2. *Let the multiset* $S = [(p_i, |\psi_i\rangle, v_i, O_i) \mid |\psi\rangle; e \xmapsto{O_i}{}^{*}_{p_i} |\psi_i\rangle; v_i]$. *Then, if*

$$[\![e]\!]_\phi\,(|\psi\rangle\!\langle\psi|) = (\rho, v) \text{ then } \rho = \textstyle\sum_{(p_i,|\psi_i\rangle,v_i,O_i)\in S} p_i\,|\psi_i'\rangle\!\langle\psi_i'| \text{ and } (|\psi_i'\rangle, v) \equiv (|\psi_i''\rangle, v')$$
$$\text{where } |\psi_i\rangle \otimes \mathit{defer}(O_i) \otimes |0\rangle^* = |\psi_i''\rangle \text{ and } v_i = \mathit{apply}(O_i, v').$$

The proof of this theorem adds cases for $\texttt{entangle}_\pi$, $\texttt{cast}_{\mathbf{M}}$, and $\texttt{split}_{\mathbf{M}}$, which evaluate to values with the same denotation as their argument, as well as $\texttt{split}_{\mathbf{P}}$, whose operational and denotational separability conditions align, and $\texttt{cast}_{\mathbf{P}}$ when its separability condition holds.

Similarly, we weaken Corollary 6.2 to state that the denotation having a pure sub-state is a sufficient condition for purity. Definition 5.3 for purity holds vacuously when the program aborts at runtime and there is no execution to a final value. While the denotation of an illegal assertion is always $\perp$, corresponding to the program aborting at runtime, the operational semantics does not verify the separability condition of $\texttt{cast}_{\mathbf{P}}$ and may not abort.

7.7 Purity Soundness

We now prove that under the operational semantics, the purity type system excluding the $\texttt{cast}_{\mathbf{P}}$ operator is sound. In this section, we assume expressions contain no instances of $\texttt{cast}_{\mathbf{P}}$. We first establish a relationship stating that quantum states respect pure annotations in expressions:

Definition 7.3 (Compatibility). An expression e is *compatible* with quantum state $|\psi\rangle$, denoted $|\psi\rangle \vDash e$, if for every $q^{\mathbf{P}}$ appearing within e, we have $|\psi\rangle; q^{\mathbf{P}}$ pure.

For example, the expression $\texttt{ref}[\alpha]^{\mathbf{P}}$ of pure type is only pure if qubit α is separable from the rest of the system in $|\psi\rangle$. We maintain the compatibility property through a preservation theorem for Twist that augments Theorem 4.2:

Theorem 7.4 (Preservation). *If* $\Gamma \vdash_\Delta e : \tau$ *and* $\Delta \subseteq \operatorname{dom}|\psi\rangle$ *and* $|\psi\rangle \vDash e$ *and* $|\psi\rangle; e \mapsto |\psi'\rangle; e'$, *then* $\Gamma \vdash_{\Delta'} e' : \tau$ *where* $\Delta' \subseteq \operatorname{dom}|\psi'\rangle$ *and* $|\psi'\rangle \vDash e'$.

The proof is by induction on the derivation of $|\psi\rangle; e \mapsto |\psi'\rangle; e'$ and given in Appendix F.1.2. The main soundness theorem states that an expression with pure type is pure. Assuming the expression satisfies runtime verification for $\texttt{split}_{\mathbf{P}}$, it evaluates to a unique final value and state.

Theorem 7.5 (Purity Soundness). *If* $\cdot \vdash_\Delta e : ϙ^{\mathbf{P}}$, $\Delta \subseteq \operatorname{dom}|\psi\rangle$, *and* $|\psi\rangle \vDash e$, *then* $|\psi\rangle; e$ pure.

The proof is by logical relations. For the relation, we define purity at a type τ, lifting purity to function types by stating that they take pure inputs to pure outputs and to pairs by stating that their components are pure. We give the full proof in Appendix F.1.3, strengthening the theorem to open terms using a substitution of free variables for pure expressions, and then proceeding by induction on the derivation of $\Gamma \vdash_\Delta e : \tau$ using the strengthened inductive hypothesis.

8 STATIC ANALYSIS FOR PURITY

In this section, we present a static analysis guaranteeing that all uses of $\texttt{cast}_{\mathbf{P}}$ operators are sound, obviating the need to verify them using mixed-state simulation. This analysis verifies that the qubits owned by an expression are separable from all others in the system, including those that were measured. The analysis is sound and conservative, relying on the fact that qubit α may only become entangled with qubit β by entering the same entangled pair as β or a qubit γ that is entangled with β. Thus, the analysis tracks possibly-entangled qubits by a variant of data-dependence analysis.

A-If

$$\frac{\Gamma_1 \vdash_A e : \mathsf{bool} \qquad \Gamma_2 \vdash_A e_1 : \varrho^f \qquad \Gamma_2 \vdash_A e_2 : \varrho^g}{\Gamma_1, \Gamma_2 \vdash_A \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \varrho^{\mathbf{M}}}$$

A-Entangle

$$\frac{\Gamma \vdash_A e : {\varrho_1}^f \times {\varrho_2}^g \qquad h = \mathsf{Combine}(f, g)}{\Gamma \vdash_A \mathtt{entangle}_\pi(e) : (\varrho_1 \mathbin{\&} \varrho_2)^h}$$

A-SplitMixed

$$\frac{\Gamma \vdash_A e : (\varrho_1 \mathbin{\&} \varrho_2)^f \qquad j\ \mathsf{fresh} \qquad g = \mathsf{Split}(f, j)}{\Gamma \vdash_A \mathtt{split}_{\mathbf{M}}(e) : {\varrho_1}^g \times {\varrho_2}^g}$$

A-CastMixed

$$\frac{\Gamma \vdash_A e : \varrho^f}{\Gamma \vdash_A \mathtt{cast}_{\mathbf{M}}(e) : \varrho^f}$$

A-CastPure

$$\frac{\Gamma \vdash_A e : \varrho^{\mathbf{P}}}{\Gamma \vdash_A \mathtt{cast}_{\mathbf{P}}(e) : \varrho^{\mathbf{P}}}$$

Fig. 13. Selected rules of analysis type system. The full rules are given in Appendix E.

8.1 Tracking Split Entangled Pairs

If a pure entangled pair is split into two components e_1 and e_2, any expression containing only e_1 or e_2 is potentially mixed. For an expression to be pure it must contain, for every $\mathtt{split}_{\mathbf{M}}$ in the program, either zero or both of its components. Based on this observation, the analysis tracks for each expression the fraction of each $\mathtt{split}_{\mathbf{M}}$ in the program it contains.

Our approach is similar to fractional permissions [Boyland 2003] in that we associate each type with a fractional quantity that is divided upon destructing a type into constituents. Specifically, we associate each expression with one fraction per entangled pair created by the program, representing the components of the pair. We modify the type system from Section 7 to generalize purities **P** and **M** to a data structure that we call a *history*.

Definition 8.1. A *history* is a linear combination $\sum_i c_i x_i$ where each $c_i \in \mathbb{Q}$, $0 \le c_i < 1$ and each x_i is a symbol distinguishable from x_j when $i \ne j$.

The analysis assigns each instance of $\mathtt{split}_{\mathbf{M}}$ in the program a unique index i and associates it with the symbol x_i. An expression's history specifies the fraction of each $\mathtt{split}_{\mathbf{M}}$ it contains.

Split and Combine. We define two operations to manipulate histories. Letting histories $f = \sum_{i=0}^N a_i x_i$ and $g = \sum_{i=0}^N b_i x_i$, and a fresh index $j > N$, define:

$$\mathsf{Split}(f, j) = \tfrac{1}{2}\left(f + x_j\right) \quad \text{and} \quad \mathsf{Combine}(f, g) = \textstyle\sum_{i=0}^N \mathrm{frac}\left(a_i + b_i\right) x_i$$

The $\mathsf{Split}(f, j)$ operator adds a new term x_j to f and halves every coefficient, representing the components that each hold half of the parent. The $\mathsf{Combine}(f, g)$ operator adds two histories term-wise, taking the fractional part of each coefficient so that fractions adding to one cancel. The definition performs this cancellation because if an expression contains either zero or both components of every $\mathtt{split}_{\mathbf{M}}$ in the program, it must be pure.

Pure Expressions. The history containing zero terms contains no fractional component of any $\mathtt{split}_{\mathbf{M}}$ in the program, meaning it represents a pure expression. We denote this history **P** for sake of continuity. As an example, suppose $[q_1, q_2]$ has history $f = \frac{1}{2}x_1 + \frac{3}{4}x_2$. Applying $\mathtt{split}_{\mathbf{M}}$ produces two expressions with history $\mathsf{Split}(f, 3) = \frac{1}{4}x_1 + \frac{3}{8}x_2 + \frac{1}{2}x_3$. Neither expression can become pure unless combined with the other to cancel the x_3 term. Now suppose that value q_3 has $g = \frac{1}{2}x_1 + \frac{1}{4}x_2$. Then $[[q_1, q_2], q_3]$ has history $\mathsf{Combine}(f, g) = \mathbf{P}$ and is pure. Thus, $[[q_1, q_2], q_3]$ has no outside entanglements because it cannot contain a fraction of any $\mathtt{split}_{\mathbf{M}}$ in the program.

Analysis Type System. Formally, the analysis accepts a program by assigning it a type under a modified type system with purities replaced by histories. Figure 13 presents the typing judgment $\Gamma \vdash_A e : \tau$ used by the analysis. Nearly all of its defining rules are derived directly from the original type system, and only the shown typing rules are modified substantially.

An `if`-expression obtains a special history $\mathbf{M}$ where $\mathsf{Split}(\mathbf{M}, j) = \mathbf{M}$ and $\mathsf{Combine}(f, \mathbf{M}) = \mathbf{M}$ for any history f. The reason is that the static analysis cannot know which branch the `if` takes. The rule for $\mathtt{entangle}_\pi$ invokes $\mathsf{Combine}(f, g)$ on its argument, disregarding the annotation π. The rule for $\mathtt{split}_\mathbf{M}$ invokes $\mathsf{Split}(f, j)$ on its arguments to compute the type of the result. The $\mathtt{cast}_\mathbf{M}$ operator has the same history as its argument, and the $\mathtt{cast}_\mathbf{P}$ operator is only valid when the analysis knows its argument to be pure.

8.2 Purity Soundness

The following theorem states that a well-typed program that passes the analysis will satisfy its purity specification at runtime. An expression of pure type is pure, and assuming it satisfies runtime verification for $\mathtt{split}_\mathbf{P}$, it evaluates to a unique final value and state.

Theorem 8.2 (Purity Soundness). *If* $\cdot \vdash_A e : \varrho^\mathbf{P}$, *then* $|\cdot\rangle; e$ pure.

We give the full proof in Appendix F.2.1, proceeding by induction on $\cdot \vdash_A e : \varrho^\mathbf{P}$ to show that an expression with pure history does not own any qubit that is entangled with any unowned qubit. A program passing the static analysis may only invoke $\mathtt{cast}_\mathbf{P}$ on expressions of history $\mathbf{P}$, which are never the results of `if`-expressions or entangled with unowned qubits, and hence are pure.

9 EVALUATION

We now implement the type checker, static analysis, and interpreter for Twist,[21] and use them to analyze and execute a set of benchmark programs and answer the following research questions:

RQ1. Is Twist expressive enough to permit writing standard quantum algorithms?
RQ2. Does Twist reject programs that contain bugs caused by violating purity specifications?
RQ3. How does Twist's runtime performance compare between pure- and mixed-state simulators?
RQ4. What is the runtime overhead of Twist's purity assertions in simulation?
RQ5. Is Twist expressive enough to permit programs that existing languages disallow?

9.1 Implementation

We implemented the interpreter in OCaml, using Quantum++ [Gheorghiu 2018], a state-of-the-art C++ quantum simulator. We perform measurements and unitary gates by invoking Quantum++ functions. The implementation adds support for three-qubit Toffoli (CCNOT) and Fredkin (CSWAP) gates and arbitrary (controlled) phase rotation gates. We implemented purity assertions using both pure- and mixed-state simulation (Section 7), which Quantum++ natively supports.

9.2 Methodology

For RQ1, we wrote a set of benchmark programs, described in the next section, and annotated each with purity specifications. For RQ2, we modified several programs to introduce a small bug, for example deleting a vital unitary gate or using an incorrect gate, such that the program would yield incorrect results. We then executed the type checker, static analysis, and both pure- and mixed-state runtime verification on the programs. We did not execute later analysis passes on ill-typed programs, nor did we execute the pure-state simulator when the static analysis failed.

For RQ3 and RQ4, we wrote a family of programs invoking runtime verification on an increasing number of qubits, described in the next section. We executed them using both the pure- and mixed-state simulators and measured their execution time, specifically the portion of time spent

[21] The implementation is available at https://www.github.com/psg-mit/twist-popl22.

```
fun teleport (q1 : qubit<P>) : qubit<P> =
  let (q2 : qubit<M>, q3 : qubit<M>) = bell_pair () in
  let (q1 : qubit<M>, q2 : qubit<M>) = CNOT (q1, q2) in
  let q1 : qubit<M> = H (q1) in
  let q3 = if measure (q2) then X (q3) else q3 in
  let q3 = if measure (q1) then Z (q3) else q3 in
  cast<P>(q3)
```

Fig. 14. *Teleport-Measure* benchmark, a variant of *Teleport-Deferred* that uses classical `if`-expressions.

performing runtime verification. We executed all benchmarks on a MacBook Pro with 2.4GHz 8-core Intel Core i9 processor and 64 GB of RAM. We invoked optimization level -O3 and enabled OpenMP (used by Quantum++). All reported timings are the average of 10 executions.

For RQ5, we compare the results of Twist's analyses on the benchmarks with the type system of Silq [Bichsel et al. 2020], a recent quantum programming language. We chose to compare against Silq because of its claim to express high-level constructs while preventing unintuitive or physically unrealizable behavior, and because it claims to subsume features of languages such as Green et al. [2013]; Paykin et al. [2017]; Svore et al. [2018]. Because our benchmarks extensively utilize purity annotations, which Silq does not support, we did not translate them to Silq, but instead reasoned about whether its type system would accept an equivalent program.

9.3 Benchmark Programs

We implemented a range of programs featuring entangled states, including well-known quantum algorithms. These benchmarks include *Teleport-Deferred*, the example from Section 3 of deferred-measurement teleportation; a classical AND oracle function; a faulty substitution of a Bell state by a Greenberger-Horne-Zeilinger state [Greenberger et al. 1989]; Deutsch, Deutsch-Jozsa [Deutsch 1992], and Grover's [Grover 1996] algorithms; a quantum Fourier transform [Coppersmith 1994]; and Shor's nine-qubit error-correcting code [Calderbank and Shor 1996]. For the benchmarks *AndOracle*, *Deutsch*, *DeutschJozsa*, *Grover*, and *ShorCode*, we also implemented erroneous variants that violate their purity specifications and yield incorrect results.

The *Teleport-Measure* benchmark (Figure 14) is a variant of *Teleport-Deferred* that does not use quantum conditional gates. Instead, the program measures `q1` and `q2` and uses the classical outcomes to conditionally execute gates on `q3` via `if`-expressions followed by a purifying cast.

To study the performance of runtime verification on increasingly complex programs, we implemented a benchmark *ModMul(n)*, inspired by Huang and Martonosi [2019], which implements modular multiplication for a number of qubits n ranging from 4 to 22, with a different version of the program for each input size. The programs use purity assertions to verify that after a conditional modular multiplication followed by its inverse, the condition qubit is separable from the multiplicand. We also implemented an erroneous variant *ModMul(n)-NotInverse* where the inverse operation is incorrect, resulting in the condition qubit remaining entangled.

Full descriptions of all benchmark programs may be found in Appendix G, and their full sources are provided in Appendix H. The sources utilize syntax extensions to Twist described in Appendix C.

9.4 RQ1 and RQ2: Analysis Results

We list in Table 1 the analysis outputs for each benchmark compared to ground-truth knowledge. Detailed descriptions of the analysis results may be found in Appendix G.

Table 1. Evaluation results on benchmark programs. The column "valid" denotes the ground truth of whether the program is valid under its purity specification, and the following three columns state whether the program passed the type check, static analysis, and runtime verification respectively. Pure- and mixed-state simulations agreed in all instances where both were run. If type checking failed, the later analyses were not executed.
* For tests that failed the static analysis, only the mixed-state simulator was executed.
† For *Teleport-Measure*, the static analysis was sound but imprecise (overly conservative).

		purity specifications				Twist
Name	# qubits	valid	types	static	dynamic	correct
Teleport-Deferred	3	✓	✓	✓	✓	✓
└ Teleport-NoCZ	3	✗	✓	✓	✗	✓
└ Teleport-Measure	3	✓	✓	✗	✓*	–†
AndOracle	3	✓	✓	✓	✓	✓
└ AndOracle-NotUncomputed	3	✗	✗	N/A	N/A	✓
Bell-GHZ	3	✗	✗	N/A	N/A	✓
Deutsch	2	✓	✓	✓	✓	✓
└ Deutsch-BadResultBasis	2	✗	✓	✓	✗	✓
DeutschJozsa	3	✓	✓	✓	✓	✓
└ DeutschJozsa-MixedInit	3	✗	✓	✗	✗*	✓
Grover	4	✓	✓	✓	✓	✓
└ Grover-BadOracle	4	✗	✓	✗	✗*	✓
QFT	3	✓	✓	✓	✓	✓
ShorCode	9	✓	✓	✓	✓	✓
└ ShorCode-Drop	9	✗	✗	N/A	N/A	✓
ModMul(n)	4–22	✓	✓	✓	✓	✓
└ ModMul(n)-NotInverse	4–22	✗	✓	✓	✗	✓

Research Question 1. We can express quantum algorithms such as *Deutsch*, *DeutschJozsa*, *Grover*, *QFT*, and *ShorCode*, and Twist correctly determines that they satisfy their purity specification.

Research Question 2. The type checker correctly rejects three of the benchmarks, *AndOracle-NotUncomputed*, *Bell-GHZ*, and *ShorCode-Drop*, which use mixed expressions in contexts that require pure expressions. The static analysis correctly rejects two of the benchmarks, *DeutschJozsa-MixedInit*, and *Grover-BadOracle*, that inappropriately use the purifying-cast operator to coerce a mixed expression into a pure one. Runtime verification correctly rejects three of the remaining benchmarks, *Teleport-NoCZ*, *Deutsch-BadResultBasis*, and *ModMul(n)-NotInverse*, due to failing the separability condition for the purifying-split operator.

For *Teleport-Measure*, the static analysis is imprecise. The static analysis rejects the final purifying-cast assertion and does not permit the result to be annotated as pure, even though it is pure. As a result, we must use the mixed-state simulator to determine that the program is valid.

9.5 RQ3 and RQ4: Timing Results

We display in Figure 15 the runtime performance of the *ModMul(n)* family of programs. All static analyses terminated within 50 ms and are not counted as part of runtime.

Research Question 3. Mixed-state simulation rapidly becomes impractical to execute compared to pure-state simulation, taking longer on 11 qubits than the pure-state simulator did on 22 qubits.

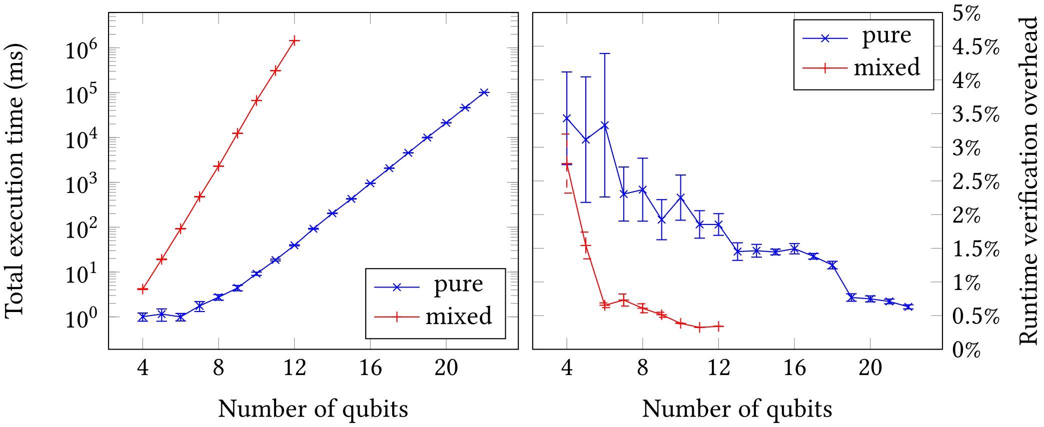

Fig. 15. Execution performance of *ModMul(n)* programs in Twist. The first plot depicts total execution time as the number of qubits n increases, and the second depicts the percentage of total execution time spent on runtime verification. Error bars display the standard error of the mean of ten runs.

Research Question 4. In pure-state simulation, runtime verification is not a large performance burden. On a program with 4 qubits, the relative overhead of runtime verification is the largest, at 3.5% of total runtime. As the number of gates and qubits increases, the relative overhead of runtime verification diminishes, and is approximately 0.5% for 22-qubit programs.

Mixed-state simulation is similar, with runtime verification overhead being about 3% for 4-qubit programs and approximately 0.3% for 12-qubit programs. Though the relative overhead is ostensibly lower for mixed than pure states, the baseline is much slower for mixed-state simulation.

The reason why the relative overhead decreases with more qubits in this application is that the larger test cases require more gates to encode multiplication over more qubits. However, each program needs only one purity assertion to enforce its purity specification.

9.6 RQ5: Comparison to Silq

We compare Twist with Silq [Bichsel et al. 2020], a recent language that claims to enable high-level programming. Silq supports type annotations expressing freedom from effects such as mutation and measurement, letting it automatically uncompute certain temporary qubits that exit scope.

Unlike Silq, Twist's type system detects programs that violate user-defined purity specifications. Also, Twist permits discarding pure values, allowing the developer to write programs that discard temporary values that are separable from the ongoing computation.

Silq rejects the *Teleport-Deferred*, *Deutsch*, and *ShorCode* benchmarks, which discard a temporary expression that has become separable at the end of the computation. Though Silq supports automatically uncomputing certain qubits that exit scope, it does so only for qubits that are introduced within the same scope and not subjected to unsupported gates with phase-level effects. In these benchmarks, Silq cannot determine whether an arbitrary expression, such as the input or output of a function, is separable and safe to discard, and requires the developer to manually measure it. However, Twist accepts these programs, because it can determine that measuring a pure expression cannot affect the ongoing computation, and thus permits implicitly measuring it.

10 EXTENSIONS AND LIMITATIONS

Language Features. We overview in Appendix C the higher-level syntactic features that we have already implemented in Twist to enable writing more concise programs. Adding additional features such as arrays, loops, quantum conditional blocks, and automatic adjoints may further improve ease of use. Currently, the user must annotate purities in types, and type inference would make programming faster and more concise, especially if the developer could use it as a tool to immediately see the inferred purity specification of their programs.

Classical Control. Classical control affects the precision of our static analysis, as seen in the *Teleport-Measure* example. Programs that use classical control currently require the mixed-state approach of purity assertion verification, which is inefficient. Further work may improve the precision of the static analysis on classical control and enable execution on a pure-state simulator.

Hardware Execution. In this work, we operate in an idealized model of quantum computation and describe how the purity assertions of Twist can be implemented in simulation using the primitive of separability testing. In quantum hardware, determining whether a state is separable is a form of quantum state tomography [Vogel and Risken 1989], where ascertaining properties of an unknown quantum state generally requires many copies of the state to obtain high accuracy. In Appendix B, we describe a procedure to determine with high probability whether a pure quantum state is separable, based on Harrow and Montanaro [2013]. Additionally, existing runtime quantum assertion frameworks such as Huang and Martonosi [2019]; Li et al. [2020]; Liu and Zhou [2021] support some form of separability testing which Twist could leverage. These methods indicate that separability testing is potentially achievable natively in quantum hardware.

Simulation Accuracy. Twist's runtime verification as implemented relies on numeric floating-point arithmetic in the quantum simulator, which may introduce the possibility of imprecision error. Our attempts to exploit imprecision to cause separability tests to pass on an entangled state were unsuccessful, but it is possible that very slightly entangled states or excessive error accumulation in the simulator could lead to unsoundness, which would require further effort to mitigate.

11 RELATED WORK

Entanglement Reasoning. Honda [2015]; Perdrix [2005, 2008]; Prost and Zerrari [2009] present logical systems for reasoning about entanglement in quantum programs based on type systems and abstract interpretation. These frameworks track fine-grained entanglement between specific qubits, making them limited in scale and unable to handle more complex programs such as teleportation.

Rand et al. [2021a,b] propose a type system establishing circumstances under which gate outputs are separable, based on their Heisenberg representation. Unlike ours, it does not guarantee purity, and can only determine separability in specific bases. However, this direction may enable better static checking for separability conditions, increasing the utility of purity specifications.

Researchers have developed compilers that perform reasoning about entanglement in a quantum circuit. For example, ScaffCC [JavadiAbhari et al. 2014] provides a disentanglement check warning the user when possibly entangled qubits exit scope. Unlike Twist, this check is purely syntactic and cannot be refined by semantic knowledge about the program. Häner et al. [2020] leverage entanglement annotations to optimize circuit gate count, but do not ensure that the annotations are sound. By contrast, we proved that pure-typed expressions are in fact pure.

Frameworks for quantum runtime assertions support reasoning about entangled and separable states, such as Huang and Martonosi [2019] who use statistical hypothesis testing on measurement outcomes of repeated program executions, Li et al. [2020] who check predicates using projection operators, and Liu and Zhou [2021] who propose quantum circuits implementing approximate assertions. Though these tools do not support sound reasoning for whole programs, their techniques may be useful to implement runtime verification in Twist.

Quantum Program Verification. Separation logic [Reynolds 2002] is a well-studied formalism for reasoning about memory aliasing in classical programs, and researchers have extended it into the probabilistic [Barthe et al. 2020] and quantum [Zhou et al. 2021] realms, generalizing the notion of separation to probabilistic independence and quantum separability respectively. They have also adapted classical techniques such as model checking [Gay et al. 2008], abstract interpretation [Yu and Palsberg 2021], and Hoare logic [Singhal 2020; Unruh 2019], and developed relational proof strategies [Barthe et al. 2019] for quantum programs. Verification tools have enabled applications such as analysis of robustness against error [Hung et al. 2019; Tao et al. 2021] and mechanized soundness proofs for optimizations [Hietala et al. 2021; Shi et al. 2020].

Twist's pure annotation can be construed as separable conjunction in separation logic, a connection worthy of additional study. However, an advantage of our type system is that it is directly usable during programming – purity is a first-class construct, and the type checker and analyses automate reasoning about purity. The programmer may simply write a program with annotations and execute it to be confident that it satisfies the specification. Our automated reasoning does not require the programmer to understand a proof framework external to the language.

Twist's static analysis relies on manipulations of fractions in a similar way as fractional permissions [Boyland 2003], originally introduced to reason about mutable effects and which has been extended to separation logic [Bornat et al. 2005], symbolic rather than concrete quantities [Heule et al. 2011], and applications in memory management [Suenaga and Kobayashi 2009]. To our knowledge, we are the first to use fractional permissions-style reasoning for quantum entanglement.

Ancilla Correctness. One application of purity guarantees in a quantum program is the correctness of temporary qubits, known as ancillas, and verifying uncomputation of ancillas. Silq [Bichsel et al. 2020] automatically inserts sound uncomputation for ancillas or rejects programs not supporting uncomputation. Paradis et al. [2021] provide a general scheme to synthesize uncomputation for circuits, and Rand et al. [2019] provide a mechanized system for proving ancilla correctness. In general, Twist benefits from the presence of automatic or provable uncomputation, because it can trust the correctness of the uncomputation and elide the runtime verification.

In turn, systems with automatic uncomputation benefit from soundness guarantees of purity types and convenience of implicitly discarding pure values statically known to not require uncomputation. We demonstrated that Twist can detect incorrect programs that these languages cannot. In addition, Twist does not require distinguishing particular qubits as ancillas, and its runtime verification can perform reasoning that they cannot, for example recognizing gates like Hadamard as self-inverse.

12 CONCLUSION

Quantum computing presents unique challenges to programmers who must reason about phenomena such as entanglement that have no analog in the classical world, and if improperly addressed can result in unintuitive bugs. So far, quantum programming languages have sought to ensure valid semantics under the laws of quantum mechanics, but have not developed comprehensive means of understanding entanglement. The result is that the developer must manually determine whether their computations are affected by measurement outcomes of seemingly unrelated qubits.

In this work, we introduce Twist, the first language with sound reasoning for purity, the property of an expression that states its evaluation is unaffected by measurement outcomes of unowned qubits. We present language constructs to assert purity of expressions and verifications for these assertions. Twist enjoys a soundness guarantee stating that in programs that pass its verifications, every expression of pure type is in fact pure and free from entanglement.

To our knowledge, this work is the first to define the powerful notion of purity, which enables sound reasoning about entanglement in quantum programs. We hope this work paves the way to languages featuring abstractions that align with the complex and unintuitive phenomena inherent in quantum computing, allowing classical programmers to reap its computational benefits.

ACKNOWLEDGMENTS

We would like to thank Alex Renda, Eric Atkinson, Jesse Michel, Tian Jin, Alex Lew, Sara Achour, Robert Rand, Yipeng Huang, Axel Feldmann, and Rohan Yadav, who all provided feedback on drafts of this paper. This work was supported in part by the MIT-IBM Watson AI Lab, the Intel Probabilistic Computing Center, and the Sloan Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Héctor Abraham et al. 2019. Qiskit: An Open-source Framework for Quantum Computing.

T. Altenkirch and J. Grattage. 2005. A Functional Quantum Programming Language. In *IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1109/LICS.2005.1

Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *Computer Aided Verification*.

Hiroo Azuma. 2017. An entangling-probe attack on Shor's algorithm for factorization. *Journal of Modern Optics* 65, 4 (Nov 2017). https://doi.org/10.1080/09500340.2017.1397221

Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A Probabilistic Separation Logic. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3371123

Gilles Barthe, Justin Hsu, Mingsheng Ying, Nengkun Yu, and Li Zhou. 2019. Relational Proofs for Quantum Programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3371089

J. S. Bell. 1964. On the Einstein Podolsky Rosen paradox. *Physics* 1 (Nov 1964). Issue 3. https://doi.org/10.1103/PhysicsPhysiqueFizika.1.195

Charles H. Bennett and Gilles Brassard. 2014. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science* 560 (2014). https://doi.org/10.1016/j.tcs.2014.05.025

Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.* 70 (Mar 1993). Issue 13. https://doi.org/10.1103/PhysRevLett.70.1895

Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (Sep 2017). https://doi.org/10.1038/nature23474

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3386007

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Symposium on Principles of Programming Languages*. https://doi.org/10.1145/1047659.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *International Symposium on Static Analysis*. https://doi.org/10.1007/3-540-44898-5_4

Harry Buhrman, Richard Cleve, John Watrous, and Ronald de Wolf. 2001. Quantum Fingerprinting. *Phys. Rev. Lett.* 87 (Sep 2001). Issue 16. https://doi.org/10.1103/PhysRevLett.87.167902

A. R. Calderbank and Peter W. Shor. 1996. Good quantum error-correcting codes exist. *Physical Review A* 54, 2 (Aug 1996). https://doi.org/10.1103/PhysRevA.54.1098

Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences* 115, 38 (2018). https://doi.org/10.1073/pnas.1801723115

Pierre Clairambault and Marc de Visme. 2019. Full Abstraction for the Quantum Lambda-Calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3371131

D. Coppersmith. 1994. An approximate Fourier transform useful in quantum factoring.

David Deutsch. 1992. Rapid Solution of Problems by Quantum Computation. *Proceedings of the Royal Society of London: Mathematical and Physical Sciences* 439, 1907 (1992). https://doi.org/10.1098/rspa.1992.0167

Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. 2008. QMC: A Model Checker for Quantum Systems. In *International Conference on Computer Aided Verification*. https://doi.org/10.1007/978-3-540-70545-1_51

Vlad Gheorghiu. 2018. Quantum++: A modern C++ quantum computing library. *PLOS ONE* 13, 12 (Dec 2018). https://doi.org/10.1371/journal.pone.0208073

Olivier Giraud. 2007. Distribution of Bipartite Entanglement for Random Pure States. *Journal of Physics A: Mathematical and Theoretical* 40, 11 (Feb. 2007). https://doi.org/10.1088/1751-8113/40/11/014

G. H. Golub and C. Reinsch. 1970. Singular Value Decomposition and Least Squares Solutions. *Numer. Math.* 14, 5 (April 1970). https://doi.org/10.1007/BF02163027

Daniel Gottesman and Isaac L. Chuang. 1999. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature* 402, 6760 (Nov 1999). https://doi.org/10.1038/46503

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2491956.2462177

Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going Beyond Bell's Theorem*. Springer Netherlands, Dordrecht. https://doi.org/10.1007/978-94-017-0849-4_10

Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *ACM Symposium on Theory of Computing*. https://doi.org/10.1145/237814.237866

Leonid Gurvits. 2003. Classical deterministic complexity of Edmonds' problem and Quantum Entanglement. In *ACM Symposium on Theory of Computing*. https://doi.org/10.1145/780542.780545

A. Haar. 1933. Der Massbegriff in der Theorie der Kontinuierlichen Gruppen. *Annals of Mathematics* 34 (1933). https://doi.org/10.2307/1968346

Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2020. Assertion-based Optimization of Quantum Programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3428201

Aram W. Harrow and Ashley Montanaro. 2013. Testing Product States, Quantum Merlin-Arthur Games and Tensor Optimization. *J. ACM* 60, 1, Article 3 (Feb. 2013). https://doi.org/10.1145/2432622.2432625

Patrick Hayden, Kevin Milner, and Mark M. Wilde. 2014. Two-Message Quantum Interactive Proofs and the Quantum Separability Problem. *Quantum Info. Comput.* 14, 5-6 (April 2014). https://doi.org/10.5555/2638661.2638663

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2011. Fractional Permissions without the Fractions. In *Workshop on Formal Techniques for Java-Like Programs*. https://doi.org/10.1145/2076674.2076675

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3434318

K. Honda. 2015. Analysis of Quantum Entanglement in Quantum Programs using Stabilizer Formalism. In *QPL*. https://doi.org/10.4204/EPTCS.195.19

Michał Horodecki, Paweł Horodecki, and Ryszard Horodecki. 1996. Separability of mixed states: necessary and sufficient conditions. *Physics Letters A* 223, 1-2 (Nov 1996). https://doi.org/10.1016/S0375-9601(96)00706-2

Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. In *International Symposium on Computer Architecture*. https://doi.org/10.1145/3307650.3322213

Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. 2019. Quantitative Robustness Analysis of Quantum Programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3290344

Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *ACM Conference on Computing Frontiers*. https://doi.org/10.1145/2597917.2597939

Richard Jozsa and Noah Linden. 2003. On the role of entanglement in quantum-computational speed-up. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 459, 2036 (Aug 2003). https://doi.org/10.1098/rspa.2002.1097

Ivan Kassal, James D. Whitfield, Alejandro Perdomo-Ortiz, Man-Hong Yung, and Alán Aspuru-Guzik. 2011. Simulating Chemistry Using Quantum Computers. *Annual Review of Physical Chemistry* 62, 1 (2011). https://doi.org/10.1146/annurev-physchem-032210-103512

A. Yu. Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (dec 1997). https://doi.org/10.1070/RM1997V052N06ABEH002155

E Knill. 1996. Conventions for quantum pseudocode.

Abhijeet Kumar, Saeed Haddadi, Mohammad Pourkarimi, Bikash Behera, and Prasanta Panigrahi. 2020. Experimental realization of controlled quantum teleportation of arbitrary qubit states via cluster states. *Scientific Reports* 10 (08 2020). https://doi.org/10.1038/s41598-020-70446-8

Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3428218

Ji Liu and Huiyang Zhou. 2021. Systematic Approaches for Precise and Approximate Quantum State Runtime Assertion. In *IEEE International Symposium on High-Performance Computer Architecture*. https://doi.org/10.1109/HPCA51647.2021.00025

Hoi-Kwong Lo and H.F. Chau. 1998. Why quantum bit commitment and ideal quantum coin tossing are impossible. *Physica D: Nonlinear Phenomena* 120, 1-2 (Sep 1998). https://doi.org/10.1016/S0167-2789(98)00053-0

Igor L. Markov and Mehdi Saeedi. 2012. Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation. *Quantum Info. Comput.* 12, 5-6 (May 2012).

Warner A. Miller, Grigoriy Kreymerman, Christopher Tison, P. Alsing, and Jonathan McDonald. 2011. Quantum computing in a piece of glass. In *SPIE - The International Society for Optical Engineering*. https://doi.org/10.1117/12.883332

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

Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: Synthesizing Uncomputation in Quantum Circuits. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3453483.3454040

A. Pati and S. Braunstein. 2000. Impossibility of deleting an unknown quantum state. *Nature* 404 (2000). https://doi.org/10.1038/404130b0

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3009837.3009894

Simon Perdrix. 2005. Quantum Patterns and Types for Entanglement and Separability. In *International Workshop on Quantum Programming Languages*. https://doi.org/10.1016/j.entcs.2006.12.015

Simon Perdrix. 2008. Quantum Entanglement Analysis Based on Abstract Interpretation. In *International Symposium on Static Analysis*. https://doi.org/10.1007/978-3-540-69166-2_18

Frédéric Prost and Chaouki Zerrari. 2009. Reasoning about Entanglement and Separability in Quantum Higher-Order Functions. *Unconventional Computation* (2009). https://doi.org/10.1007/978-3-642-03745-0_25

Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: Reasoning about Reversible Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (Jan 2019). https://doi.org/10.4204/EPTCS.287.17

Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. 2021a. Extending Gottesman Types Beyond the Clifford Group. In *Workshop on Programming Languages for Quantum Computing*.

Robert Rand, Aarthi Sundaram, Kartik Singhal, and Brad Lackey. 2021b. Static Analysis of Quantum Programs via Gottesman Types. arXiv:2101.08939 [quant-ph]

Mathys Rennela and Sam Staton. 2017. Classical Control, Quantum Circuits and Linear Logic in Enriched Category Theory. In *Conference on Mathematical Foundations of Programming Semantics*. https://doi.org/10.23638/LMCS-16(1:30)2020

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1109/LICS.2002.1029817

E. Schmidt. 1907. Zur Theorie der linearen und nichtlinearen Integralgleichungen. *Math. Ann.* 63 (1907). https://doi.org/10.1007/BF01449770

Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14 (08 2004). https://doi.org/10.1017/S0960129504004256

Peter Selinger and Benoît Valiron. 2005. A Lambda Calculus for Quantum Computation with Classical Control. *Typed Lambda Calculi and Applications* (2005). https://doi.org/10.1017/S0960129506005238

Yunong Shi, Runzhou Tao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2020. CertiQ: A Mostly-automated Verification of a Realistic Quantum Compiler. arXiv:1908.08963 [quant-ph]

Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct 1997). https://doi.org/10.1137/S0097539795293172

Kartik Singhal. 2020. Quantum Hoare Type Theory. arXiv:2012.02154 [cs.PL]

Kohei Suenaga and Naoki Kobayashi. 2009. Fractional Ownerships for Safe Memory Deallocation. In *Asian Symposium on Programming Languages and Systems*. https://doi.org/10.1007/978-3-642-10672-9_11

Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Real World Domain Specific Languages Workshop*. https://doi.org/10.1145/3183895.3183901

Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T. Chong, and Ronghui Gu. 2021. Gleipnir: Toward Practical Error Analysis for Quantum Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3453483.3454029

Dominique Unruh. 2019. Quantum Hoare Logic with Ghost Variables. In *IEEE Symposium on Logic in Computer Science*. https://doi.org/10.1109/LICS.2019.8785779

K Vogel and H Risken. 1989. Determination of quasiprobability distributions in terms of probability distributions for the rotated quadrature phase. *Physical Review A* 40, 5 (September 1989). https://doi.org/10.1103/PhysRevA.40.2847

S. P. Walborn, P. H. Souto Ribeiro, L. Davidovich, F. Mintert, and A. Buchleitner. 2006. Experimental Determination of Entanglement with a Single Measurement. *Nature* 440, 7087 (April 2006). https://doi.org/10.1038/nature04627

Dave Wecker, Krysta M. Svore, and Krysta M. Svore. 2014. LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing. (February 2014).

W. Wootters and W. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299 (1982). https://doi.org/10.1038/299802a0

Mingsheng Ying. 2016. *Foundations of Quantum Programming* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nengkun Yu and Jens Palsberg. 2021. Quantum Abstract Interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3453483.3454061

Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. A Quantum Interpretation of Bunched Logic amp; Quantum Separation Logic. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS52264.2021.9470673