Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs

by

Chenhui Yuan

B.S., Carnegie Mellon University (2019)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 11, 2022

Certified by .....
Michael Carbin
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Twist: Sound Reasoning for Purity and Entanglement in Quantum Programs

by

Chenhui Yuan

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

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, I 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.

I 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.

I 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 several languages disallow, while incurring runtime verification overhead of less than 3.5%.

Thesis Supervisor: Michael Carbin
Title: Associate Professor of Electrical Engineering and Computer Science

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Quantum programming languages (Svore et al., 2018; Green et al., 2013; Paykin et al., 2017; Rand et al., 2019; Wecker et al., 2014; Ying, 2016; Altenkirch and Grattage, 2005; Selinger, 2004; Selinger and Valiron, 2005; Bichsel et al., 2020; Clairambault and de Visme, 2019; Rennela and Staton, 2017) 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), and computational physics and chemistry (Childs et al., 2018; Kassal et al., 2011).

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

[1] More precisely, measurement outcomes of entangled quantum states have statistical correlations that cannot be explained by physical theories with local realism (Nielsen and Chuang, 2010).

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; Rand et al., 2019; Green et al., 2013).

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 (Bichsel et al., 2020; Rand et al., 2019; Amy et al., 2017) 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 programs.

1.2 Purity

I 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 execute operations known as *gates* on them. 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.

1.3 Specification and Verification of Purity

I 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.

[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).

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. I 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), execution is aborted 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, I implement Twist on a state vector-based quantum simulator, examples of which include Abraham et al. (2019); Gheorghiu (2018). I 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 I present the following contributions:

- *Purity.* I present the novel definition of *pure* expressions in a quantum program, those that are unaffected by measurement outcomes of the remainder of the program. I formulate purity within the operational and denotational semantics of a functional quantum language.
- *Purity Types.* I present a type system that identifies pure expressions, and prove that in it, expressions of pure type are in fact pure.
- *Purity Assertions.* I 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.* I present a static analysis and runtime verifications for the purity assertions, such that programs passing verification satisfy their purity specifications.
- *Evaluation.* I implement Twist, a language featuring purity types and assertions, in quantum simulation. I 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.

This 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.

Chapter 2

Background on Quantum Computation

This chapter overviews key concepts in quantum computation relevant to this work and its notational choices. Nielsen and Chuang (2010) provide a comprehensive reference.

2.1 Pure State Formalism

This section defines 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 $|1\rangle$[1] 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, one may represent any pure state as a *state vector*, a length-2^n vector of normalized complex amplitudes.[2]

Multiple qubits form a quantum state system by means of the tensor product $\otimes$. Thus, the state $|01\rangle$ is equal to the product $|0\rangle \otimes |1\rangle$. Subscripts denote the names of individual qubits or sets of qubits. For example, $|0\rangle_\alpha \otimes |1\rangle_\beta$ denotes a two-qubit system in which a qubit named α has state $|0\rangle$ and qubit β has state $|1\rangle$.[3]

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

[2] 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.

[3] In this work, the tensor product is treated 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.

The *empty state* $|\cdot\rangle$ is defined to be the length-2^0 vector containing amplitude 1, which 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, $\mathrm{dom}\,|\psi\rangle$, is the set of qubit names it contains.

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. The state produced by a unitary operator acting on qubit α in state $|\psi\rangle$ is denoted $U_\alpha\,|\psi\rangle$.[4] Single-qubit quantum *gates* include:

- 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 $(|0\rangle - |1\rangle)/\sqrt{2}$.

Two-qubit quantum gates include 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 renaming qubits at will. $|\psi\rangle\,[\gamma/\alpha]$ denotes 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,[5] 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.

The state produced by measurement of qubit α in state $|\psi\rangle$ is denoted $M_\alpha\,|\psi\rangle$. To define measurement, the state is first rewritten 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.[6] Then, with probability $|\gamma_0|^2$ the outcome $M_\alpha\,|\psi\rangle = |0\rangle_\alpha \otimes |\psi_0\rangle$ is obtained, and with probability $|\gamma_1|^2$ the outcome $M_\alpha\,|\psi\rangle = |1\rangle_\alpha \otimes |\psi_1\rangle$ is obtained.

Pure and Mixed States Unlike pure states, 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

[4]This work uses 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.

[5]This work is concerned primarily with projective computational (i.e. 0/1) basis measurements, though its results can be generalized to other measurement forms.

[6]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, equality and uniqueness of states are defined up to global phase.

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.[7]

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 $1/2$.

Schmidt Decomposition Given a bipartite pure state $|\psi\rangle$ over qubit sets $A \sqcup B$, one 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

This section describes 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 $|\psi\rangle$, $|\psi\rangle\langle\psi|$ denotes 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.

[7] 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 $1/2$.

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 $\mathrm{tr}\,(\rho P_j)$, and results in normalized density matrix $P_j\rho P_j/\mathrm{tr}\,(\rho P_j)$. $P_j\rho P_j$ 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.[8]

Product States and Separability One can construct a composite of mixed states using the tensor product $\rho_1 \otimes \rho_2$. The notation $\rho[\gamma/\alpha]$ renames 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, $\mathrm{Tr}_A(\rho)$, is the unique linear operator satisfying:

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

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

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 $\mathrm{tr}\,(\rho^2) = (\mathrm{tr}\,\rho)^2 = 1$. It is similarly the case that a partial density matrix $\rho = p\,|\psi\rangle\langle\psi|$ if and only if $\mathrm{tr}\,(\rho^2) = (\mathrm{tr}\,\rho)^2$ by linearity of trace.

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

Chapter 3

Example

This chapter illustrates 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, here a variant (Miller et al., 2011; Kumar et al., 2020) is examined that instead uses quantum-conditioned gates and defers all measurement to the end of the program.

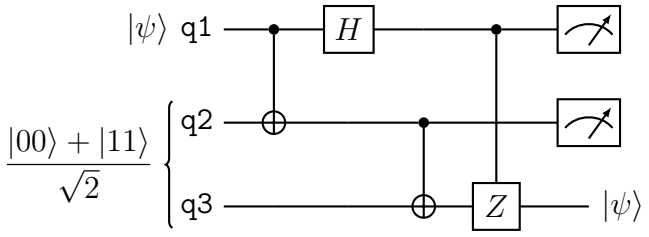

Figure 3-1: Teleportation protocol with deferred measurement expressed as a quantum circuit. The inputs are `q1`, the qubit to be teleported, and a Bell pair of two qubits. The circuit applies conditional-NOT (CNOT) and conditional-Z (CZ) gates to `q3`. It measures two qubits and outputs one with the teleported state of `q1`.

Figure 3-1 presents a quantum circuit for the deferred-measurement variant of teleportation that accepts as input a qubit `q1` to be teleported.[1] A pair of qubits (`q2`, `q3`) are assumed to exist in a Bell state (Chapter 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 3-2 presents a program implementing this circuit. It accepts a qubit `q1` and returns the teleported output, where the helper `bell_pair` allocates a Bell pair.

[1]For the purpose for demonstrating this work's results, understanding the rationale behind this circuit is not necessary; the reader is referred to Nielsen and Chuang (2010) for a detailed explanation.

```
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
```

Figure 3-2: Implementation of the teleportation circuit of Figure 3-1 as a quantum program.

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 if one were to replace the CZ gate on line 6 of Figure 3-2 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 (Chapter 2) that stems from different measurement outcomes of `q1` and `q2`. By contrast, pure expressions always evaluate to unique final states not dependent on measurements of other qubits.

3.2 Purity Types

Figure 3-3 presents the program from Figure 3-2, 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.

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. Entanglement is described in the type system using a type of *entangled pairs*, denoted by the type constructor `&`, such as on line 2 of Figure 3-3. An entangled pair stipulates that its two components potentially share entanglement.

```
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
```

Figure 3-3: 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
```

Figure 3-4: Teleportation example with purity types and assertions so that its return type is refined to be pure.

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 3-4 presents an implementation of `teleport` 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.

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. This process confirms that the two qubits 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 3-4, 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 the author's 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. Twist's 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 (Chapter 2). Because `q3` is in fact pure in this program, the assertion always succeeds at runtime.

Efficiency The `split` and `cast` operators are designed to be used in concert to make their verification more efficient. Consider instead the program in Figure 3-5 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 3-4.

The reason the static analysis is designed 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 inefficient compared

```
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 *)
```

Figure 3-5: 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, the static analysis rejects this program.

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.[2]

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 3-4 – 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 writing the teleportation program even more concisely than Figure 3-4. Line 13 of Figure 3-4 explicitly measures `q12`, a value the program no longer needs. Instead, the program in Figure 3-6 removes this measurement operation and simply discards `q12`.

The reason that discarding unused qubits is a potential point of contention is the deferred measurement principle (Chapter 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 a program implicitly discards a variable.

[2]In quantum mechanics, mixed states have more general definitions of entanglement and separability than pure states. Chapter 11 shows 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 (Horodecki et al., 1996; Harrow and Montanaro, 2013), and was proved by Gurvits (2003) to be **NP**-hard in the dimension of the density matrix.

```
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>) =
9     split<P>(cast<P>(q123)) in
10  q3 (* safely discard pure expression q12 *)
```

Figure 3-6: Concise teleportation program in Twist that implicitly discards a pure value `q12` on line 10.

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, given a direct translation that eliminates the purity annotations from the program in Figure 3-4, Silq will not uncompute `q12` unless the user modifies the program. The reason why Silq does not automatically uncompute `q12` when it drops from scope is that Silq does not automatically uncompute qubits affected by gates with phase-level effects such as Hadamard, as is the case for `q12` in `teleport`.

To write this example in Silq, the developer may either 1) manually invoke a `measure` operator on `q12`, 2) manually invoke a `forget` operator on `q12`, or 3) apply the `qfree` annotation to the sequence of gates that produced it. In the first two cases, neither `measure` nor `forget` guarantees that its argument is separable from the remaining computation, meaning that invoking either operator may in general leave the computation in a mixed state if the developer does not accurately reason about the purity of the computation. With the third strategy, applying the `qfree` annotation invokes the requirement of the Silq type system that `qfree` functions do not use gates such as Hadamard that have phase effects, disqualifying teleportation and programs such as the efficient quantum adder of Draper (2000). Silq could in principle be extended with an unsafe cast to `qfree` that would permit this example and other programs with phase effects, with the proviso that the developer must themselves accurately reason about the safety of such casts in general.

By contrast, Twist leverages the guarantee made by its purity type system that the discarded value cannot impact the remaining computation. The result is that a Twist developer may implement `teleport` as a concise and safe program.

Chapter 4

μQ Language

To formally define evaluation and purity in quantum programs, this chapter presents μQ, a small quantum language. μQ features classical control and quantum data in the style of the linear quantum λ-calculus of Selinger and Valiron (2005).

This chapter presents μQ's syntax and operational semantics. Chapter 5 develops its semantic properties, including purity, in terms of executions in the operational semantics. Chapter 6 presents μQ's denotational semantics, which provides a more concise definition of purity useful to formulate Twist's purity assertions. Chapter 7 develops 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.

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

μQ features the types τ of Booleans, pairs, functions, and qubits. Its expressions e include the classical constructs of variables x, functions f, applications, pairs, and `let`[1] and `if`-expressions. Boolean literals, denoted 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.[2] The operator `measure`(e) performs a quantum measurement of e, returning the classical

[1] `let` $x = e_1$ `in` e_2 is also used as syntactic sugar for `let` $(x, \_) = (e_1, \mathsf{T})$ `in` e_2 (where T is arbitrarily chosen).

[2] For simplicity, the language does not represent larger gates, which can be decomposed into single- and two-qubit operators using constructions such as Kitaev (1997).

$$\frac{\Gamma, x : \tau \vdash_\emptyset e : \tau' \qquad \Gamma, f : \tau \to \tau' \vdash m \ \mathsf{ok}}{\Gamma \vdash \texttt{fun}\ f\ x = e; m \ \mathsf{ok}} \qquad \frac{\Gamma \vdash_\emptyset e : \tau}{\Gamma \vdash \texttt{fun main}\ () = e \ \mathsf{ok}}$$

Figure 4-1: Definition of well-formed programs in μQ.

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 4-2 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.

The *classical types* are defined to be Booleans, functions, and pairs of classical types. Both contexts are linear, except for classical types, which may be freely duplicated or discarded.[3] To ensure that qubits are not duplicated, the disjoint set union $\sqcup$, defined only when its arguments are disjoint, is used.

The initial context is populated with types of function declarations using the judgment $\Gamma \vdash m \ \mathsf{ok}$ (defined in Figure 4-1), 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. The *quantum types* are defined 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 the 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.[4] Qubit references have qubit type.

[3]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.

[4]Measurement of tuples is also allowed as syntactic sugar for sequentially measuring each component.

T-Var

$$\frac{}{x : \tau \vdash_{\emptyset} x : \tau}$$

T-Fun

$$\frac{}{f : \tau \to \tau' \vdash_{\emptyset} f : \tau \to \tau'}$$

T-App

$$\frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} e_1(e_2) : \tau_2}$$

T-Pair

$$\frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} (e_1, e_2) : \tau_1 \times \tau_2}$$

T-Let

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

T-If

$$\frac{\tau \text{ is a quantum type} \qquad \Gamma_1 \vdash_{\Delta_1} e : \texttt{bool} \qquad \Gamma_2 \vdash_{\Delta_2} e_1 : \tau \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

T-Bool

$$\frac{}{\cdot \vdash_{\emptyset} b : \texttt{bool}}$$

T-Qinit

$$\frac{}{\cdot \vdash_{\emptyset} \texttt{qinit}\ () : \texttt{qubit}}$$

T-U1

$$\frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}}{\Gamma \vdash_{\Delta} U(e) : \texttt{qubit}}$$

T-U2

$$\frac{\Gamma \vdash_{\Delta} e : \texttt{qubit} \times \texttt{qubit}}{\Gamma \vdash_{\Delta} U_2(e) : \texttt{qubit} \times \texttt{qubit}}$$

T-Measure

$$\frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}}{\Gamma \vdash_{\Delta} \texttt{measure}(e) : \texttt{bool}}$$

T-Ref

$$\frac{}{\cdot \vdash_{\{\alpha\}} \texttt{ref}[\alpha] : \texttt{qubit}}$$

Figure 4-2: μQ type system.

4.3 Operational Semantics

Figure 4-3 presents an operational semantics for μQ as a probabilistic transition system over *program states* – pairs of quantum states $|\psi\rangle$ and classical expressions e.[5] 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 4-3 presents the left-to-right call-by-value semantics of μQ. 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 `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 (Chapter 2).

Notation The following new notations are defined for judgments.

- $|\psi\rangle; e \;\mapsto \cdot$ 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 $|\psi\rangle; e \xmapsto{O_i}_{p_i} |\psi_i\rangle; e_i$ and $\sum_i p_i = 1$.
- $|\psi\rangle; e \mapsto |\psi'\rangle; e'$ means that there exist O and p where $|\psi\rangle; e \xmapsto{O}_p |\psi'\rangle; e'$.
- $|\psi\rangle; e \mapsto^* |\psi'\rangle; e'$ means that there exist O and p where $|\psi\rangle; e \xmapsto{O}{}^*_p |\psi'\rangle; e'$.

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

V-Fun
$$\frac{}{f \text{ val}}$$

V-Bool
$$\frac{}{b \text{ val}}$$

V-Qref
$$\frac{}{\texttt{ref}[\alpha] \text{ val}}$$

V-Pair
$$\frac{e_1 \text{ val} \qquad e_2 \text{ val}}{(e_1, e_2) \text{ val}}$$

S-App
$$\frac{\phi(f) = \lambda x.e \qquad e' \text{ val}}{|\psi\rangle; f(e') \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; [e'/x]e}$$

S-AppL
$$\frac{|\psi\rangle; e_1 \overset{O}{\longmapsto}_p |\psi'\rangle; e_1'}{|\psi\rangle; e_1(e_2) \overset{O}{\longmapsto}_p |\psi'\rangle; e_1'(e_2)}$$

S-AppR
$$\frac{e_1 \text{ val} \qquad |\psi\rangle; e_2 \overset{O}{\longmapsto}_p |\psi'\rangle; e_2'}{|\psi\rangle; e_1(e_2) \overset{O}{\longmapsto}_p |\psi'\rangle; e_1(e_2')}$$

S-PairL
$$\frac{|\psi\rangle; e_1 \overset{O}{\longmapsto}_p |\psi'\rangle; e_1'}{|\psi\rangle; (e_1, e_2) \overset{O}{\longmapsto}_p |\psi'\rangle; (e_1', e_2)}$$

S-PairR
$$\frac{e_1 \text{ val} \qquad |\psi\rangle; e_2 \overset{O}{\longmapsto}_p |\psi'\rangle; e_2'}{|\psi\rangle; (e_1, e_2) \overset{O}{\longmapsto}_p |\psi'\rangle; (e_1, e_2')}$$

S-Let
$$\frac{e_1 \text{ val} \qquad e_2 \text{ val}}{|\psi\rangle; \texttt{let } (x, y) = (e_1, e_2) \texttt{ in } e' \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; [e_1, e_2/x, y]e'}$$

S-LetS
$$\frac{|\psi\rangle; e_1 \overset{O}{\longmapsto}_p |\psi'\rangle; e_1'}{|\psi\rangle; \texttt{let } (x, y) = e_1 \texttt{ in } e_2 \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{let } (x, y) = e_1' \texttt{ in } e_2}$$

E-Val
$$\frac{v \text{ val}}{|\psi\rangle; v \overset{\emptyset}{\longmapsto}_1^* |\psi\rangle; v}$$

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

S-Qinit
$$\frac{\alpha \text{ fresh in } |\psi\rangle}{|\psi\rangle; \texttt{qinit } () \overset{\emptyset}{\longmapsto}_1 |\psi\rangle \otimes |0\rangle_\alpha; \texttt{ref}[\alpha]}$$

S-U1
$$\frac{}{|\psi\rangle; U(\texttt{ref}[\alpha]) \overset{\emptyset}{\longmapsto}_1 U_\alpha |\psi\rangle; \texttt{ref}[\alpha]}$$

S-U1S
$$\frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; U(e) \overset{O}{\longmapsto}_p |\psi'\rangle; U(e')}$$

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

S-U2S
$$\frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; U_2(e) \overset{O}{\longmapsto}_p |\psi'\rangle; U_2(e')}$$

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

S-MeasureF
$$\frac{M_\alpha |\psi\rangle = |0\rangle_\alpha \otimes |\psi'\rangle \text{ w.p. } p \qquad O = \{(\alpha, \mathsf{F})\}}{|\psi\rangle; \texttt{measure}(\texttt{ref}[\alpha]) \overset{O}{\longmapsto}_p |\psi'\rangle; \mathsf{F}}$$

S-MeasureS
$$\frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{measure}(e) \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{measure}(e')}$$

S-IfT
$$\frac{}{|\psi\rangle; \texttt{if } \mathsf{T} \texttt{ then } e_1 \texttt{ else } e_2 \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; e_1}$$

S-IfF
$$\frac{}{|\psi\rangle; \texttt{if } \mathsf{F} \texttt{ then } e_1 \texttt{ else } e_2 \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; e_2}$$

S-IfS
$$\frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{if } e' \texttt{ then } e_1 \texttt{ else } e_2}$$

Figure 4-3: μQ operational semantics rules.

4.4 Type Safety

The type safety properties of μQ are now stated. 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 1 (Progress)**.** *If* $\cdot \vdash_\Delta e : \tau$*, then* e val *or for all* $|\psi\rangle$ *where* $\Delta \subseteq \operatorname{dom} |\psi\rangle$*,* $|\psi\rangle; e \mapsto \cdot$*.*

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

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

Chapter 5

Semantic Properties

This chapter defines 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, a relation is defined 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 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 $A = \operatorname{dom}|\psi\rangle \setminus \mathsf{Refs}(v)$ and $\operatorname{dom}|\psi_v\rangle = \mathsf{Refs}(v)$.*

The notation $|\psi\rangle; v \Downarrow |\psi'\rangle$ denotes 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

Two program states $(|\psi_1\rangle, v_1)$ and $(|\psi_2\rangle, v_2)$ are considered equivalent if they are equal up to consistent renaming of qubits. For example, it is defined that

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

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

Definition 2 (Qubit Equivalence)**.** $(|\psi_1\rangle, v_1) \equiv (|\psi_2\rangle, v_2)$ *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. The action of rewriting using qubit equivalence is defined 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 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* $(|\psi_1'\rangle, v_1) \equiv (|\psi_2'\rangle, v_2)$*.*

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.

Chapter 6

Denotational Semantics of μQ

The operational definition of purity quantifies over all possible executions of a μQ program. This chapter presents 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, the semantics instead forces 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, the semantics utilizes the deferred measurement principle (Chapter 2) to interpret Booleans as deferred qubit measurements that are not resolved until the end of the program. For quantum types, it canonicalizes values that differ only in the order of appearance of qubits. To do so, it uses rewriting under qubit equivalence (Chapter 5) to dynamically rename qubits so that every value refers to the same qubits in the same order.

6.1 Semantics

Figure 6-1 presents the denotational semantics of the μQ language. 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 (Chapter 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 $\texttt{ref}^\star[\alpha]$ 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 $\mathsf{match\_sizes}$, 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

This chapter shows 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, define $\mathsf{apply}(O, v)$ to be the value syntactically identical to v except that every instance of $\texttt{ref}^\star[\alpha]$ is replaced with b where $(\alpha, b) \in O$. In the opposite direction, 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.

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

Figure 6-1: μQ denotational semantics.

Theorem 3. *Given* $|\psi\rangle$ *and* e*, let 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, it is the case that*

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

The proof is by induction on the derivations of $|\psi\rangle; e \xmapsto{O_i}{}^*_{p_i} |\psi_i\rangle; v_i$. 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, which 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 equivalently defines purity using denotational semantics.

Corollary 1. $|\psi\rangle; e$ pure *holds if and only if* $[\![e]\!]_\phi(|\psi\rangle\langle\psi|) = (\rho \otimes \rho', v)$ *where* $\operatorname{dom}\rho = \mathsf{Refs}(v)$ *and* ρ *is a pure state, i.e.* $\rho = |\psi'\rangle\langle\psi'|$ *for some* $|\psi'\rangle$*.*

Thus, computing the denotation of e under $|\psi\rangle$ provides a direct way of testing whether $|\psi\rangle; e$ pure. This fact is leveraged in the following sections to define the *purity assertion* operators of Twist.

Chapter 7

Twist Language

This chapter presents the formal core of the Twist language. Twist extends μQ with a *purity type* system that specifies which expressions are pure. It presents the two *purity assertion* operators that specify and check purity in a quantum program.

Both a denotational and operational semantics of Twist are presented. Denotationally, purity assertions are defined using mathematical *separability conditions* on mixed-state denotations. Operationally, they are implemented using a *separability test* primitive on runtime pure states.

7.1 Syntax

Twist augments the syntax of μQ as follows, where new syntax is in black:[1]

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

Introduced are purity annotations π, either pure, $\mathbf{P}$, or mixed, $\mathbf{M}$. Also introduced is a sort of *quantum types* ϱ,[2] either a `qubit` or an *entangled pair* $\varrho_1 \,\&\, \varrho_2$ of two quantum types. They denote *quantum values* q, either a reference to a qubit or an entangled pair $[q_1, q_2]$ of two quantum values.

[1] Chapter 9 describes additional syntactic features of Twist, such as implicit measurement of discarded pure values.

[2] Lowercase Greek letter qoppa.

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 7-1 presents the type system of Twist, extending the type system of μQ to the new operators. The new judgment $\vdash_\Delta q : \mathfrak{o}$ assigns the quantum value q the quantum type $\mathfrak{o}$ 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.

Next, the typing judgment $\Gamma \vdash_\Delta e : \tau$ is modified. 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 $\mathfrak{o}^\pi$ if q has quantum type $\mathfrak{o}$.

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.

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 7-2 presents the denotational semantics of Twist, extending the semantics of μQ to the new operators. 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.

Q-Ref
$$\frac{}{\vdash_{\{\alpha\}} \texttt{ref}[\alpha] : \texttt{qubit}}$$

Q-Pair
$$\frac{\vdash_{\Delta_1} q_1 : \varrho_1 \qquad \vdash_{\Delta_2} q_2 : \varrho_2}{\vdash_{\Delta_1 \sqcup \Delta_2} [q_1, q_2] : \varrho_1 \,\&\, \varrho_2}$$

T-Var
$$\frac{}{x : \tau \vdash_{\emptyset} x : \tau}$$

T-Fun
$$\frac{}{f : \tau \to \tau' \vdash_{\emptyset} f : \tau \to \tau'}$$

T-App
$$\frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} e_1(e_2) : \tau_2}$$

T-Pair
$$\frac{\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1 \qquad \Gamma_2 \vdash_{\Delta_2} e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_{\Delta_1 \sqcup \Delta_2} (e_1, e_2) : \tau_1 \times \tau_2}$$

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

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

T-Bool
$$\frac{}{\cdot \vdash_{\emptyset} b : \texttt{bool}}$$

T-Qinit
$$\frac{}{\cdot \vdash_{\emptyset} \texttt{qinit}\ () : \texttt{qubit}^{\mathbf{P}}}$$

T-U1
$$\frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}^{\pi}}{\Gamma \vdash_{\Delta} U(e) : \texttt{qubit}^{\pi}}$$

T-U2
$$\frac{\Gamma \vdash_{\Delta} e : (\texttt{qubit} \,\&\, \texttt{qubit})^{\pi}}{\Gamma \vdash_{\Delta} U_2(e) : (\texttt{qubit} \,\&\, \texttt{qubit})^{\pi}}$$

T-Measure
$$\frac{\Gamma \vdash_{\Delta} e : \texttt{qubit}^{\pi}}{\Gamma \vdash_{\Delta} \texttt{measure}(e) : \texttt{bool}}$$

T-Qval
$$\frac{\vdash_{\Delta} q : \varrho}{\cdot \vdash_{\Delta} q^{\pi} : \varrho^{\pi}}$$

T-Entangle
$$\frac{\Gamma \vdash_{\Delta} e : {\varrho_1}^{\pi} \times {\varrho_2}^{\pi}}{\Gamma \vdash_{\Delta} \texttt{entangle}_{\pi}(e) : (\varrho_1 \,\&\, \varrho_2)^{\pi}}$$

T-Split
$$\frac{\Gamma \vdash_{\Delta} e : (\varrho_1 \,\&\, \varrho_2)^{\pi}}{\Gamma \vdash_{\Delta} \texttt{split}_{\pi}(e) : {\varrho_1}^{\pi} \times {\varrho_2}^{\pi}}$$

T-Cast
$$\frac{\Gamma \vdash_{\Delta} e : \varrho^{\pi'}}{\Gamma \vdash_{\Delta} \texttt{cast}_{\pi}(e) : \varrho^{\pi}}$$

Figure 7-1: Twist purity type system.

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

$$
\begin{aligned}
[\![v]\!]_{\gamma}(\rho) &= (\rho, (\!|\, v \,|\!)) \text{ when } v \ \mathsf{val} \\
[\![\texttt{entangle}_{\pi}(e)]\!] &= [\![\texttt{cast}_{\mathbf{M}}(e)]\!] = [\![\texttt{split}_{\mathbf{M}}(e)]\!] = [\![e]\!] \\
[\![\texttt{split}_{\mathbf{P}}(e)]\!]_{\gamma}(\rho) &= [\![e]\!]_{\gamma}(\rho) \text{ if } [\![e]\!]_{\gamma}(\rho) = (\rho_1 \otimes \rho_2 \otimes \rho_0, (q_1, q_2)) \text{ else } \bot \\
&\quad \text{where } \operatorname{dom}\rho_1 = \mathsf{Refs}(q_1) \text{ and } \operatorname{dom}\rho_2 = \mathsf{Refs}(q_2) \\
&\quad \text{and } \rho_1 \text{ and } \rho_2 \text{ are pure} \\
[\![\texttt{cast}_{\mathbf{P}}(e)]\!]_{\gamma}(\rho) &= [\![e]\!]_{\gamma}(\rho) \text{ if } [\![e]\!]_{\gamma}(\rho) = (\rho_1 \otimes \rho_0, q) \text{ else } \bot \\
&\quad \text{where } \operatorname{dom}\rho_1 = \mathsf{Refs}(q) \text{ and } \rho_1 \text{ is pure}
\end{aligned}
$$

Figure 7-2: Selected Twist denotational semantics. Only changes from Figure 6-1 shown.

$\texttt{split}_{\mathbf{P}}$ 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.

Classical Control 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

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

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 (Chapter 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. It verifies separability conditions concretely using a *separability test* primitive that determines whether the runtime state is separable or entangled.

Figure 7-3 presents the updated step and value judgments. The value judgment is modified 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 $\texttt{entangle}_{\pi}$ operator creates an entangled pair, and $\texttt{split}_{\mathbf{M}}$ destructs an entangled pair into an ordinary pair containing its components annotated as mixed. Next, semantics are defined for $\texttt{split}_{\mathbf{P}}$ and $\texttt{cast}_{\mathbf{P}}$.

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:

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

$$\text{V-Fun}\ \frac{}{f\ \mathsf{val}} \qquad \text{V-Bool}\ \frac{}{b\ \mathsf{val}} \qquad \text{V-Qval}\ \frac{}{q^{\pi}\ \mathsf{val}} \qquad \text{V-Pair}\ \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{(e_1, e_2)\ \mathsf{val}}$$

$$\text{S-QInit}\ \frac{\alpha \text{ fresh in } |\psi\rangle}{|\psi\rangle; \texttt{qinit}\ () \overset{\emptyset}{\longmapsto}_1 |\psi\rangle \otimes |0\rangle_\alpha; \texttt{ref}[\alpha]^\pi} \qquad \text{S-U1}\ \frac{}{|\psi\rangle; U(\texttt{ref}[\alpha]^\pi) \overset{\emptyset}{\longmapsto}_1 U_\alpha |\psi\rangle; \texttt{ref}[\alpha]^\pi}$$

$$\text{S-U1S}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; U(e) \overset{O}{\longmapsto}_p |\psi'\rangle; U(e')} \qquad \text{S-U2}\ \frac{}{|\psi\rangle; U_2([\texttt{ref}[\alpha], \texttt{ref}[\beta]]^\pi) \overset{\emptyset}{\longmapsto}_1 U_{\alpha,\beta} |\psi\rangle; [\texttt{ref}[\alpha], \texttt{ref}[\beta]]^\pi}$$

$$\text{S-U2S}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; U_2(e) \overset{O}{\longmapsto}_p |\psi'\rangle; U_2(e')} \qquad \text{S-MeasureT}\ \frac{M_\alpha |\psi\rangle = |1\rangle_\alpha \otimes |\psi'\rangle \text{ w.p. } p \qquad O = \{(\alpha, \mathsf{T})\}}{|\psi\rangle; \texttt{measure}(\texttt{ref}[\alpha]^\pi) \overset{O}{\longmapsto}_p |\psi'\rangle; \mathsf{T}}$$

$$\text{S-MeasureF}\ \frac{M_\alpha |\psi\rangle = |0\rangle_\alpha \otimes |\psi'\rangle \text{ w.p. } p \qquad O = \{(\alpha, \mathsf{F})\}}{|\psi\rangle; \texttt{measure}(\texttt{ref}[\alpha]^\pi) \overset{O}{\longmapsto}_p |\psi'\rangle; \mathsf{F}} \qquad \text{S-MeasureS}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{measure}(e) \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{measure}(e')}$$

$$\text{S-IfT}\ \frac{}{|\psi\rangle; \texttt{if}\ \mathsf{T}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; \texttt{cast}_{\mathbf{M}}(e_1)} \qquad \text{S-IfF}\ \frac{}{|\psi\rangle; \texttt{if}\ \mathsf{F}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; \texttt{cast}_{\mathbf{M}}(e_2)}$$

$$\text{S-IfS}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\text{S-Entangle}\ \frac{}{|\psi\rangle; \texttt{entangle}_\pi({q_1}^\pi, {q_2}^\pi) \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; [q_1, q_2]^\pi} \qquad \text{S-EntangleS}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{entangle}_\pi(e) \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{entangle}_\pi(e')}$$

$$\text{S-SplitS}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{split}_\pi(e) \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{split}_\pi(e')} \qquad \text{S-SplitMixed}\ \frac{}{|\psi\rangle; \texttt{split}_{\mathbf{M}}([q_1, q_2]^{\mathbf{M}}) \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; ({q_1}^{\mathbf{M}}, {q_2}^{\mathbf{M}})}$$

$$\text{S-SplitPure}\ \frac{|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_0\rangle \qquad \dom |\psi_1\rangle = \mathsf{Refs}(q_1) \qquad \dom |\psi_2\rangle = \mathsf{Refs}(q_2)}{|\psi\rangle; \texttt{split}_{\mathbf{P}}([q_1, q_2]^{\mathbf{P}}) \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; ({q_1}^{\mathbf{P}}, {q_2}^{\mathbf{P}})} \qquad \text{S-Cast}\ \frac{(\textit{unchecked})}{|\psi\rangle; \texttt{cast}_\pi(q^{\pi'}) \overset{\emptyset}{\longmapsto}_1 |\psi\rangle; q^\pi}$$

$$\text{S-CastS}\ \frac{|\psi\rangle; e \overset{O}{\longmapsto}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{cast}_\pi(e) \overset{O}{\longmapsto}_p |\psi'\rangle; \texttt{cast}_\pi(e')}$$

Figure 7-3: Twist operational semantics. 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 Chapter 8 enables building a sound interpreter.

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 7-3 presents an operational semantics in which $\texttt{cast}_{\mathbf{P}}$ is unsound and does not verify purity. Instead, Chapter 8 presents 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, Chapter 10 overviews 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 (Chapter 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

Next are the type safety properties for Twist. A judgment is introduced to state that an attempt to step a $\texttt{split}_{\mathbf{P}}$ aborts the program if its separability condition fails, denoted $|\psi\rangle; e \not\mapsto_{\texttt{split}}$. 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 4 (Progress)**.** *If* $\cdot \vdash_{\Delta} e : \tau$*, then either* e val *or for all* $|\psi\rangle$ *where* $\Delta \subseteq \operatorname{dom} |\psi\rangle$*, either* $|\psi\rangle; e \mapsto \cdot$ *or* $|\psi\rangle; e \not\mapsto_{\texttt{split}}$.

The proof is given in Appendix A.1.1 and is analogous to Theorem 1, extended to the operators in Twist. The preservation theorem is identical to Theorem 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 3 by stating that the denotation of a valid program agrees with the operational semantics:

Theorem 5. *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, 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, Corollary 1 is weakened to state that the denotation having a pure substate is a sufficient condition for purity. Definition 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 $\bot$, 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

It is now proved that under the operational semantics, the purity type system excluding the $\texttt{cast}_\mathbf{P}$ operator is sound. In this chapter, it is assumed that expressions contain no instances of $\texttt{cast}_\mathbf{P}$. First, a relationship is established stating that quantum states respect pure annotations in expressions:

Definition 4 (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*,* $|\psi\rangle; q^\mathbf{P}$ 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 $|\psi\rangle$. The compatibility property is maintained through a preservation theorem for Twist that augments Theorem 2:

Theorem 6 (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 A.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 (Purity Soundness)**.** *If* $\cdot \vdash_\Delta e : \mathfrak{o}^\mathbf{P}$*,* $\Delta \subseteq \operatorname{dom}|\psi\rangle$*, and* $|\psi\rangle \vDash e$*, then it is the case that* $|\psi\rangle; e$ pure*.*

The proof is by logical relations. For the relation, 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. The full proof is given in Appendix A.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.

Chapter 8

Static Analysis for Purity

This chapter presents 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.

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 $\texttt{split}_{\mathbf{M}}$ in the program, either zero or both of its components. Based on this observation, the analysis tracks for each expression the fraction of each $\texttt{split}_{\mathbf{M}}$ in the program it contains.

The approach in this analysis is similar to fractional permissions (Boyland, 2003) in that it associates each type with a fractional quantity that is divided upon destructing a type into constituents. Specifically, it associates each expression with one fraction per entangled pair created by the program, representing the components of the pair. The type system from Chapter 7 is modified to generalize purities **P** and **M** to a data structure called a *history*.

Definition 5. *A* history *is a linear combination* $\sum_i c_i x_i$ *where each* $c_i \in \mathbb{Q}$, $0 \leq c_i < 1$ *and each* x_i *is a symbol distinguishable from* x_j *when* $i \neq j$.

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

Split and Combine Two operations are defined 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:

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

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 of this operator performs this cancellation because if an expression contains either zero or both components of every $\texttt{split}_{\mathbf{M}}$ in the program, it must be pure.

Pure Expressions The history containing zero terms contains no fractional component of any $\texttt{split}_{\mathbf{M}}$ in the program, meaning it represents a pure expression. This history is denoted $\mathbf{P}$ 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 $\texttt{split}_{\mathbf{M}}$ 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 $\texttt{split}_{\mathbf{M}}$ 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 8-1 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.

An `if`-expression obtains a special history $\mathbf{M}$ defined such that $\mathsf{Split}(\mathbf{M}, j) = \mathbf{M}$ 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 $\texttt{entangle}_{\pi}$ invokes $\mathsf{Combine}(f, g)$ on its argument, disregarding the annotation π. The rule for $\texttt{split}_{\mathbf{M}}$ invokes $\mathsf{Split}(f, j)$ on its arguments to compute the type of the result. The $\texttt{cast}_{\mathbf{M}}$ operator has the same history as its argument, and the $\texttt{cast}_{\mathbf{P}}$ 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 $\texttt{split}_{\mathbf{P}}$, it evaluates to a unique final value and state.

Theorem 8 (Purity Soundness)**.** *If* $\cdot \vdash_A e : \varrho^{\mathbf{P}}$*, then* $|\cdot\rangle; e$ pure*.*

A-Var
$$\frac{}{x : \tau \vdash_A x : \tau}$$

A-Fun
$$\frac{\Gamma, x : \tau_1 \vdash_A e : \tau_2}{\Gamma \vdash_A \lambda x.e : \tau_1 \to \tau_2}$$

A-App
$$\frac{\Gamma_1 \vdash_A e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash_A e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_A e_1(e_2) : \tau_2}$$

A-Pair
$$\frac{\Gamma_1 \vdash_A e_1 : \tau_1 \qquad \Gamma_2 \vdash_A e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_A (e_1, e_2) : \tau_1 \times \tau_2}$$

A-Let
$$\frac{\Gamma_2 \vdash_A e : \tau_1 \times \tau_2 \qquad \Gamma_1, x : \tau_1, y : \tau_2 \vdash_A e' : \tau}{\Gamma_1, \Gamma_2 \vdash_A \texttt{let}\ (x, y) = e\ \texttt{in}\ e' : \tau}$$

A-If
$$\frac{\Gamma_1 \vdash_A e : \texttt{bool} \quad \Gamma_2 \vdash_A e_1 : \varrho^f \quad \Gamma_2 \vdash_A e_2 : \varrho^g}{\Gamma_1, \Gamma_2 \vdash_A \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \varrho^{\mathbf{M}}}$$

A-Bool
$$\frac{}{\cdot \vdash_A b : \texttt{bool}}$$

A-Qinit
$$\frac{}{\cdot \vdash_A \texttt{qinit}\ () : \texttt{qubit}^{\mathbf{P}}}$$

A-U1
$$\frac{\Gamma \vdash_A e : \texttt{qubit}^f}{\Gamma \vdash_A U(e) : \texttt{qubit}^f}$$

A-U2
$$\frac{\Gamma \vdash_A e : (\texttt{qubit} \,\&\, \texttt{qubit})^f}{\Gamma \vdash_A U_2(e) : (\texttt{qubit} \,\&\, \texttt{qubit})^f}$$

A-Measure
$$\frac{\Gamma \vdash_A e : \texttt{qubit}^f}{\Gamma \vdash_A \texttt{measure}(e) : \texttt{bool}}$$

A-Entangle
$$\frac{\Gamma \vdash_A e : {\varrho_1}^f \times {\varrho_2}^g \quad h = \mathsf{Combine}(f, g)}{\Gamma \vdash_A \texttt{entangle}_\pi(e) : (\varrho_1 \,\&\, \varrho_2)^h}$$

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

A-SplitPure
$$\frac{\Gamma \vdash_A e : (\varrho_1 \,\&\, \varrho_2)^{\mathbf{P}}}{\Gamma \vdash_A \texttt{split}_{\mathbf{P}}(e) : {\varrho_1}^{\mathbf{P}} \times {\varrho_2}^{\mathbf{P}}}$$

A-CastMixed
$$\frac{\Gamma \vdash_A e : \varrho^f}{\Gamma \vdash_A \texttt{cast}_{\mathbf{M}}(e) : \varrho^f}$$

A-CastPure
$$\frac{\Gamma \vdash_A e : \varrho^{\mathbf{P}}}{\Gamma \vdash_A \texttt{cast}_{\mathbf{P}}(e) : \varrho^{\mathbf{P}}}$$

Figure 8-1: Rules of analysis type system.

The full proof is given in Appendix A.2.1, proceeding by induction on $\cdot \vdash_A e : \mathtt{Q}^{\mathbf{P}}$ 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.

Chapter 9

Additional Syntactic Features of Twist

This chapter presents syntactic features of Twist that enable more concise programs.

9.1 Affine Pure Types

As discussed, Twist allows pure expressions to be discarded and implicitly measured. Formally, a syntactic translation is performed to insert measurements of unused variables of type $\mathfrak{o}^{\mathbf{P}}$ and discard the result of the measurement. Specifically, it translates functions and `let`-bindings:

$$\lambda x.e \rightsquigarrow \lambda x.\texttt{let } \_ = \texttt{measure}(x) \texttt{ in } e$$
$$\texttt{let } (x, y) = e' \texttt{ in } e \rightsquigarrow \texttt{let } (x, y) = e' \texttt{ in let } \_ = \texttt{measure}(x) \texttt{ in } e$$

when x has type $\mathfrak{o}^{\mathbf{P}}$ and does not appear in e, and likewise for y if necessary. It is irrelevant when the implicit measurement takes place, and here it is done before evaluating e. To discard a pair containing only pure and classical types, its contents are recursively measured and discarded.

9.2 Inferring Conversion Operators

Converting between ordinary and entangled product types, as well as between different purities, requires appropriate `entangle`, `split`, and `cast` operators. Instead of forcing the user to write these operators, Twist exposes a generalized `let`-expression using type annotations from which the language can automatically insert appropriate operators. The generalized `let`-expression binds an expression to a *pattern* p which is either a type-annotated variable or a pair of patterns:

$$\textsf{Pattern } p ::= x : \tau \mid (p_1, p_2)$$

This expression is a derived form that recursively translates into core constructs:

$$\texttt{let } (p_1, p_2) = e \texttt{ in } e' \rightsquigarrow \texttt{let } (x, y) = e \texttt{ in let } p_1 = x \texttt{ in let } p_2 = y \texttt{ in } e'$$

The procedure to infer conversion operators is type-directed. For every `let`-expression binding the expression e to a pattern p, it synthesizes the type τ of e and the type τ' that p expects to bind. It then follows rules to transform e by inserting conversions so that its type becomes τ':

- If $\tau = \tau'$, do nothing to e.
- If $\tau = \varrho^{\mathbf{P}}$ and $\tau' = \varrho^{\mathbf{M}}$, replace with $\texttt{cast}_{\mathbf{M}}(e)$.
- If $\tau = \varrho^{\mathbf{M}}$ and $\tau' = \varrho^{\mathbf{P}}$, replace with $\texttt{cast}_{\mathbf{P}}(e)$.
- If $\tau = (\varrho_1 \,\&\, \varrho_2)^{\pi}$ and $\tau' = {\varrho_1}^{\pi} \times {\varrho_2}^{\pi}$, replace with $\texttt{split}_{\pi}(e)$.
- If $\tau = {\varrho_1}^{\pi} \times {\varrho_2}^{\pi}$ and $\tau' = (\varrho_1 \,\&\, \varrho_2)^{\pi}$, replace with $\texttt{let}\ (x, y) = e\ \texttt{in}\ \texttt{entangle}_{\pi}(x, y)$.
- In other cases, recursively descend into τ and τ' and apply the above rules.

As an example, a program that takes an entangled pair of qubits as input and performs a phase flip conditioned on the first qubit requires a number of conversions:

```
fun f (qs : (qubit & qubit)<P>) : (qubit & qubit)<P> =
  let (q1 : qubit<M>, q2 : qubit<M>) = split<M>(cast<M>(qs)) in
  let q1 : qubit<M> = Z (q1) in
  cast<P>(entangle<M>(q1, q2))
```

Conversion operator inference allows expressing the program much more concisely as:

```
fun f (qs : (qubit & qubit)<P>) : (qubit & qubit)<P> =
    let (q1 : qubit<M>, q2 : qubit<M>) = qs in
    let q1 = Z (q1) in
    let out : (qubit & qubit)<P> = (q1, q2) in out
```

9.3 Polymorphism over Purity

The fact that functions must provide specific purities in their types can result in code duplication with pure and mixed variants. Thus, the language supports generic purity parameters where functions instantiate at a given purity at call site. Polymorphism allows more concise programs that more accurately describe the effect of functions on purity, and also allows the static analysis of Chapter 8 to be more precise. The syntax of purities is extended to allow *variables*:

$$\pi ::= \mathbf{P} \mid \mathbf{M} \mid {'\alpha}$$

It is required that every purity variable be introduced exactly once in the argument of the function in which it is used. The syntax does not permit $\texttt{split}_{\pi}(e)$ for generic π because its operation fundamentally differs for $\mathbf{P}$ or $\mathbf{M}$,[1] but generics are permitted in `entangle` and `cast`, which are statically checked.

[1] One may cast to $\mathbf{M}$ to invoke $\texttt{split}_{\mathbf{M}}$, then later cast back.

To check a function with generic purity, the static analysis instantiates for each purity variable a unique index i and associated history x_i, guaranteeing that it cannot be conflated with any other variable. The type system instantiates a function of polymorphic type when it is applied to an argument of specific type.

Generic purities increase the precision of the static analysis. Consider the program:

```
fun f (q : qubit<M>) : qubit<M> = q
fun g (q : qubit<P>) : qubit<P> = cast<P>(f (cast<M>(q)))
```

This program does not pass the static analysis because the function `f` returns a qubit annotated as mixed rather than pure. However, `f` is over-specified to only take mixed qubits to mixed qubits, and simply inlining its definition into `g` results in a well-typed program. An interprocedural static analysis might realize this fact, but a simpler solution is to allow `f` to be polymorphic in the purity of `q`. The argument to `f` can be annotated with the generic purity `'p`, which `g` then instantiates with **P**:

```
fun f (q : qubit<'p>) : qubit<'p> = q
fun g (q : qubit<P>) : qubit<P> = f (q)
```

This program passes the static analysis and clearly expresses the effect of `f` on purity.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 10

Separability Testing in Hardware

This chapter briefly presents an implementation of separability testing on a hardware quantum computer. It utilizes concepts from the density matrix formalism of quantum mechanics, including reduced density matrices and the quantity of purity. The reader is referred to Nielsen and Chuang (2010) for the technical background to this chapter.

Adapting the following procedure to Twist requires re-executing a program to produce new copies of a state that is being subject to a separability test, as separability testing is a form of quantum state tomography (Vogel and Risken, 1989) that in general requires multiple copies of a state to characterize it.

The *SWAP test* (Buhrman et al., 2001) is a separability-testing scheme that consumes two copies of a d-qubit quantum state $|\psi\rangle$, and in $O(\log d)$ time accepts separable states with probability 1, using $O(1)$ extra qubits. Given a state $|\psi\rangle$, it is divided into subsystems A and B of dimensions d_A and d_B with reduced density matrices ρ_A and ρ_B.

First, two copies $|\psi\rangle \otimes |\psi\rangle$ are produced of the state under test. The A subsystems of the two states are swapped, conditioned on an ancilla qubit, and the same procedure is applied to the B subsystems. If a subsystem is a single qubit, this amounts to a single Fredkin gate; in general, it requires $\log d$ Fredkin gates. Finally, a measurement is performed to detect the phase acquired by the ancilla qubits.

The SWAP test accepts with probability $p = \frac{1}{2} + \frac{1}{2}\mathfrak{p}$, where $\mathfrak{p} = \mathrm{tr}(\rho_A^2) = \mathrm{tr}(\rho_B^2)$ is the subsystem purity. During the swaps, the ancilla qubits acquire a phase depending on the purity of the respective subsystems, which are equal for a bipartite state. The phase can be detected by an interferometric measurement, yielding the expression for p. More precisely, define the *distance* ϵ between $|\psi\rangle$ and the nearest separable state as:

$$\epsilon = 1 - \max\left\{|\langle\psi|\varphi\rangle|^2 : |\varphi\rangle \text{ is a separable state}\right\}$$

This testing scheme then yields a false positive rate that depends on ϵ, such that the test accepts with probability $1 - \alpha\epsilon \leq p \leq 1 - \beta\epsilon$, where $\alpha = 2$ and $\beta = 11/512$ (Harrow and Montanaro, 2013). The test has been realized experimentally (Walborn et al., 2006). However, these bounds given assume perfect operation of the quantum hardware; accounting for the cost of imperfect gates in near-term hardware remains an open problem.

To bound ϵ above with high probability, one could run n repetitions of the SWAP test, registering observations $\mathbf{X} = (X_1, X_2, \ldots, X_n)$. If $|\psi\rangle$ is a separable state, and hardware operations are perfect, these will result in $X_i = 1$ for all i. To control the distance parameter ϵ, the test can be expressed as a parameter estimation problem, and bound the probability that the actual distance is greater than some fixed ϵ^* conditioned on the observations by Markov's inequality:

$$\Pr[\epsilon > \epsilon^* \mid \mathbf{X}] \leq \frac{\mathbb{E}[\epsilon \mid \mathbf{X}]}{\epsilon^*}$$

where the conditional expectation term may be obtained from the known quantity $\Pr[\mathbf{X} \mid \epsilon] = p^n$ and a prior distribution over ϵ given by a Haar-uniform distribution (Haar, 1933) over states. This quantity could equally well be computed in terms of the purity $\mathfrak{p}$, rather than ϵ, and closed-form expressions for the corresponding distribution are known (Giraud, 2007). In this way, one can achieve any desired level of confidence in the separability of $|\psi\rangle$, by setting the parameter ϵ^* and adjusting the number of iterations n to upper-bound the probability of a false negative.

Chapter 11

Entanglement for Pure and Mixed States

This work focuses on the analysis of entanglement and separability in the sense of pure states rather than mixed states. However, from the perspective of quantum mechanics, the notions of separability for pure states and mixed states are not equivalent, and it is conceivable that one would want to analyze quantum programs using the mixed-state definition of separability instead.

This chapter briefly illustrates differences between the two formalisms and argue that the pure-state definition, known as *simple separability* when applied to density matrices, is appropriate for reasoning about purity in quantum programs. Thus, the separability tests powering Twist's purity assertions are precise and avoid the extra complexity of general separability of mixed states.

11.1 Definitions of Entanglement: Concurring Case

This section presents example programs that are analyzed using the frameworks of pure- and mixed-state entanglement. First is a case where the pure- and mixed-state definitions align.

Figure 11-1 presents Twist functions that produce the four maximally entangled *Bell states* (Bell, 1964), the first of which is used throughout examples in this work:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \qquad |\Phi^-\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}$$
$$|\Psi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \qquad |\Psi^-\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}$$

In Twist, pure quantum values have the property that even when the overall program is in a mixed state, the qubits owned by a pure value constitute a pure substate of the program state. For example, consider the program in Figure 11-2, which produces a qubit `x` in state $|1\rangle$ and two qubits `y` and `z` in the Bell state $|\Phi^+\rangle$. When `z` is measured, the whole program enters a mixed state: $|1\rangle_{\mathtt{x}} \otimes |0\rangle_{\mathtt{y}}$ with probability 1/2

```
fun phi_plus () : (qubit & qubit)<P> =
  CNOT (H (qinit ()), qinit ())
fun phi_minus () : (qubit & qubit)<P> =
  CNOT (H (X (qinit ())), qinit ())
fun flip_second (xy : (qubit & qubit)<P>) : (qubit & qubit)<P> =
  let (x : qubit<M>, y : qubit<M>) = xy in
  let (xy : (qubit & qubit)<P>) = (x, X (y)) in xy
fun psi_plus () : (qubit & qubit)<P> = flip_second (phi_plus ())
fun psi_minus () : (qubit & qubit)<P> = flip_second (phi_minus ())
```

Figure 11-1: Definitions of the four Bell states. The listing uses Twist syntax extensions described in Chapter 9.

```
let x : qubit<P> = X (qinit ()) in
let (y : qubit<M>, z : qubit<M>) = phi_plus () in
let _ = measure (z) in (x, y)
```

Figure 11-2: A quantum program that produces a mixed state with a sub-state for the qubit `x` that is pure.

and $|1\rangle_{\mathtt{x}} \otimes |1\rangle_{\mathtt{y}}$ with probability $1/2$. Nevertheless, `x` is pure, as evidenced by the fact that in either branch, the program state is separable into some sub-state for `y` and a sub-state for `x` which is always the same: $|1\rangle_{\mathtt{x}}$.

The program state and this property of the pure expression `x` can be equivalently expressed using the mixed state formalism. Mathematically, the density matrix corresponding to the program state immediately after the measurement of `z` is:

$$\begin{aligned}\rho &= \frac{|\psi_0\rangle\langle\psi_0| + |\psi_1\rangle\langle\psi_1|}{2} \text{ where } |\psi_0\rangle = |1\rangle_{\mathtt{x}} \otimes |0\rangle_{\mathtt{y}}, |\psi_1\rangle = |1\rangle_{\mathtt{x}} \otimes |1\rangle_{\mathtt{y}} \\ &= |1\rangle\langle 1|_{\mathtt{x}} \otimes \frac{|0\rangle\langle 0|_{\mathtt{y}} + |1\rangle\langle 1|_{\mathtt{y}}}{2}\end{aligned}$$

The fact that the density matrix ρ is separable into density matrices for `x` and `y`, where the factor $|1\rangle\langle 1|_{\mathtt{x}}$ is a pure density matrix, indicates in this specific example that the expression `x` is pure. Here, the pure- and mixed-state notions of separability coincide.

11.2 Definitions of Entanglement: Contrasting Case

Next is a case where the mixed-state definition of entanglement yields a different conclusion from the pure-state definition. Figure 11-3 presents a program that produces, uniformly randomly, one of the four Bell states. The function `random_bell` returns an entangled pair of qubits whose state is one of these four with $1/4$ probability each.

```
fun random_bool () : bool = measure (H (qinit ()))
fun random_bell () : (qubit & qubit)<M> =
  if random_bool ()
  then if random_bool () then phi_plus () else phi_minus ()
  else if random_bool () then psi_plus () else psi_minus ()
```

Figure 11-3: A quantum program that produces one of the four Bell states at random.

The result is not a pure expression, because the final state is a mixed state and each individual execution depends on the measurement outcomes of the intermediate qubits used by `random_bool` as a source of randomness.

In the output of `random_bell`, both qubits of the pair are also mixed when considered in isolation because they are entangled with each other. If, for example, one qubit is extracted (and the other is measured), there is no sound manner in Twist to coerce it to a qubit of pure type. The static and dynamic verifications will reject attempts using the purifying-cast and split operators because on any execution, regardless of which of the four Bell states is produced, the two qubits are entangled.

However, remarkably, the mixed state corresponding to the output of `random_bell` is separable by the mixed state definition. Mathematically:

$$\begin{aligned}\rho &= \frac{|\Phi^+\rangle\langle\Phi^+| + |\Phi^-\rangle\langle\Phi^-| + |\Psi^+\rangle\langle\Psi^+| + |\Psi^-\rangle\langle\Psi^-|}{4} \\ &= \begin{pmatrix} 1/4 & 0 & 0 & 0 \\ 0 & 1/4 & 0 & 0 \\ 0 & 0 & 1/4 & 0 \\ 0 & 0 & 0 & 1/4 \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \otimes \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix} \\ &= \frac{|0\rangle\langle 0| + |1\rangle\langle 1|}{2} \otimes \frac{|0\rangle\langle 0| + |1\rangle\langle 1|}{2}\end{aligned}$$

Thus, quantum mechanically, the uniform mixture of the four Bell states is indistinguishable (by any measurement process) from two independent copies of one single qubit in a uniform mixture of $|0\rangle$ and $|1\rangle$. The mixed-state definition of separability thus recovers a fact that the pure-state definition, which always concludes that `x` and `y` are not separable, cannot.

11.3 Sufficiency of Pure-State Entanglement

As the example demonstrates, there are situations where using the mixed-state formalism provides more fine-grained information about how the program state factors into independent sub-states. However, for the purposes of Twist, knowing that the program state is separable into two mixed states does not always refine reasoning about the purity of expressions. After all, it only matters whether there is a pure sub-state in the program, not a mixed one, and ultimately, Twist is concerned not with separability but with purity.

Effectively, the mixed-state definition of separability is too general for Twist. Mixed states can be separable in ways that do not contribute to purity reasoning:

- The mixed state is separable, but the components are not pure states, and thus do not provide information about purity of sub-expressions (`random_bell`). In this case, the sub-expressions are individually in mixed states.
- The mixed state is separable, but only as a convex combination of tensor products. For example, decomposing $\rho = \frac{\rho_1 \otimes \rho_2 + \rho_3 \otimes \rho_4}{2}$ also does not provide information about purity of sub-expressions. In this case, the sub-expressions may be classically correlated.

Only if a density matrix ρ is *simply separable* into $\rho_1 \otimes \rho_2$ where one factor is a pure state does separability guarantee that a sub-expression is pure in the sense of Twist. Simple separability is stronger than separability for mixed states because it guarantees that there exists no classical correlation between the two sub-states.

However, the information of simple separability can be recovered through pure-state reasoning. After all, assuming without loss of generality that $\rho_1 = |\phi\rangle\langle\phi|$ and $\rho_2 = \sum_j p_j \, |\psi_j\rangle\langle\psi_j|$,

$$\rho = |\phi\rangle\langle\phi| \otimes \sum_j p_j \, |\psi_j\rangle\langle\psi_j| = \sum_j p_j \left(|\phi\rangle \otimes |\psi_j\rangle\right) \left(\langle\phi| \otimes \langle\psi_j|\right)$$

which is equivalent to stating that all executions of the program yield states that share a common factor $|\phi\rangle$, and can be checked by pure-state separability tests alone.

Chapter 12

Evaluation

This chapter discusses the implementation of the type checker, static analysis, and interpreter for Twist.[1] The implementation was used 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 that violate 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?

12.1 Implementation

The interpreter is implemented in OCaml, using Quantum++ (Gheorghiu, 2018), a state-of-the-art C++ quantum simulator. It performs 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. Purity assertions are implemented using both pure- and mixed-state simulation (Chapter 7), which Quantum++ natively supports.

12.2 Methodology

For RQ1, a set of benchmark programs, described in the next chapter, are written and each annotated with purity specifications. For RQ2, several programs are modified 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. The type checker, static

[1]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)
```

Figure 12-1: *Teleport-Measure* benchmark, a variant of *Teleport-Deferred* that uses classical `if`-expressions.

analysis, and both pure- and mixed-state runtime verification are then executed on the programs. Later analysis passes are not executed on ill-typed programs, nor the pure-state simulator when the static analysis fails.

For RQ3 and RQ4, a family of programs are written invoking runtime verification on an increasing number of qubits, described in the next chapter. They are executed using both the pure- and mixed-state simulators and their execution time measured, specifically the portion of time spent performing runtime verification. All benchmarks are executed on a MacBook Pro with 2.4GHz 8-core Intel Core i9 processor and 64 GB of RAM. Optimization level -O3 is invoked and OpenMP (used by Quantum++) is enabled. All reported timings are the average of 10 executions.

For RQ5, the results of Twist's analyses on the benchmarks are compared with the type system of Silq (Bichsel et al., 2020), a recent quantum programming language.

12.3 Benchmark Programs

A range of programs featuring entangled states are implemented, including well-known quantum algorithms. These benchmarks include *Teleport-Deferred*, the example from Chapter 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*, erroneous variants are also implemented that violate their purity specifications and yield incorrect results.

The *Teleport-Measure* benchmark (Figure 12-1) 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, a benchmark *ModMul(n)* is implemented, 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. Also implemented is 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 below, and their full sources are provided in Appendix B. The sources utilize syntax extensions to Twist described in Chapter 9.

Teleport-Deferred This program implements the deferred-measurement teleportation from Chapter 3. Also implemented is an erroneous variant, *Teleport-NoCZ*, which replaces the CZ gate with a CNOT and results in the final qubit actually being entangled, and a third variant *Teleport-Measure* that measures the ancillas and uses classical rather than quantum conditioning.

AndOracle This program inverts the phase of the state conditioned on two qubits, as may be seen in an oracle for Grover's algorithm. Also implemented is an erroneous variant *AndOracle-NotUncomputed* that does not correctly uncompute the ancilla.

Bell-GHZ This program attempts to illegally substitute a sub-state of a Greenberger-Horne-Zeilinger state (Greenberger et al., 1989) for a Bell state, by dropping an entangled ancilla.

Deutsch This program implements Deutsch's algorithm (Deutsch, 1992), which determines whether a black-box function $f : \{0,1\} \to \{0,1\}$ is the constant function. Also implemented is an erroneous variant *Deutsch-BadResultBasis*, which omits an essential Hadamard gate, causing the result qubit to be in the incorrect basis, and attempts to drop an entangled ancilla.

DeutschJozsa This program implements the Deutsch-Jozsa algorithm (Deutsch, 1992), a generalization of Deutsch's algorithm, which determines whether a black-box function $f : \{0,1\}^n \to \{0,1\}$ is constant or balanced (returns 1 for exactly half of the domain). Also implemented is an erroneous variant *DeutschJozsa-MixedInit*, which uses an incorrect initial state of $(|00\rangle + |11\rangle)/\sqrt{2}$ rather than $(|00\rangle + |01\rangle + |10\rangle + |11\rangle)/2$ and will produce an incorrect result. This variant uses a purifying cast to try to force the algorithm to accept this incorrect state.

Grover This program implements Grover's search algorithm (Grover, 1996) on a two-qubit, four-element database, locating a distinguished element in cell $|11\rangle$. Also implemented is an erroneous variant *Grover-BadOracle*, which uses *AndOracle-NotUncomputed* and drops an entangled ancilla. It uses a purifying cast to try to force the algorithm to accept this incorrect oracle.

QFT This program implements the quantum Fourier transform (Coppersmith, 1994), a building block for Simon's algorithm and Shor's algorithm, on three qubits. Its purity specification is that its output has the same purity annotation as its input.

ShorCode This program implements encoding and decoding operations for Shor's error correcting code (Calderbank and Shor, 1996), which uses nine physical qubits to correct an arbitrary single error on one qubit. The program also implements a phase flip error channel and runs error correction on a qubit subject to phase-flip error. The purity specifications on the encoding and decoding operations allows the program to discard the extra parity bits that are separable from the decoded data. Also implemented is an erroneous variant *ShorCode-Drop*, which is the result of a programmer error that shadows a function argument that is not known to be pure.

ModMul(n) The scheme of Markov and Saeedi (2012) describes quantum circuits for modular multiplication that generalize a predictable structural pattern for arbitrary n, with the number of gates linear in n.

For each n, conditional multiplication mod $2^n - 1$ by some k and also by k^{-1} are implemented. Applying the first operation on a n-qubit register and a condition qubit entangles the register and qubit. Then, applying the inverse operation must disentangle the register and condition. Thus, the program uses a purifying-split operator to verify that the condition qubit is pure at program termination. Also implemented is an erroneous variant *ModMul(n)-NotInverse* where multiplication by k^{-1} is defective, resulting in a register and condition that are still entangled.

12.4 RQ1 and RQ2: Analysis Results

Table 12.1 lists the analysis outputs for each benchmark compared to ground-truth knowledge about the program. Detailed descriptions of the analysis results follow.

Research Question 1 A Twist program 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, it is necessary to use the mixed-state simulator to determine that the program is valid.

Teleport-Deferred Twist determines that the teleportation circuit satisfies its purity specification, taking a pure qubit to another pure qubit. It also determines that the erroneous variant that substitutes a CZ gate does not correctly separate the output qubit from the temporaries, meaning that measuring the temporaries causes the qubit to enter a mixed state.

AndOracle Twist determines that the oracle correctly uncomputes ancillas and yields a pure output. It also determines that the variant that does not uncompute an ancilla may yield a mixed output, violating its specification.

Bell-GHZ Twist determines that the program is ill-typed due to its attempt to directly return mixed qubits in a function whose output is specified to be pure.

Deutsch Twist determines that the algorithm satisfies its specification, including the fact that an ancilla is separable from the output and can be safely dropped. It also determines that the erroneous variant results in an entangled ancilla and rejects the attempt to drop it at runtime.

DeutschJozsa Twist determines that the algorithm satisfies its specification. For the erroneous variant with a defective initial state, it determines at the use site of the state that it has an incorrect purity and the attempt to directly cast the type of the state fails the static analysis.

Grover Twist determines that the algorithm satisfies its specification. For the erroneous variant with a defective oracle, the attempt to directly cast the result of the oracle fails the static analysis.

QFT Twist determines that the algorithm satisfies its purity specification, taking pure inputs to pure outputs. The result indicates that the output is correct and lacks entanglement with any other qubit in the system.

ShorCode Twist determines that the algorithm satisfies its purity specification, including the fact that at the end of the decoding process, the extra check bits may be safely discarded without disrupting the decoded data. For the erroneous variant with a programming error, the type checker detects that a value not known to be pure cannot be shadowed.

Table 12.1: 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	✗	✓	✓	✗	✓

ModMul(n) Twist determines that the algorithm satisfies its purity specification that the condition bit is separable from the output, implying that the inverse operation was implemented correctly. For the erroneous variant with incorrect inverse, the runtime verification fails, indicating the condition bit is still entangled and that the algorithm is incorrect.

12.5 RQ3 and RQ4: Timing Results

Figure 12-2 displays 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.

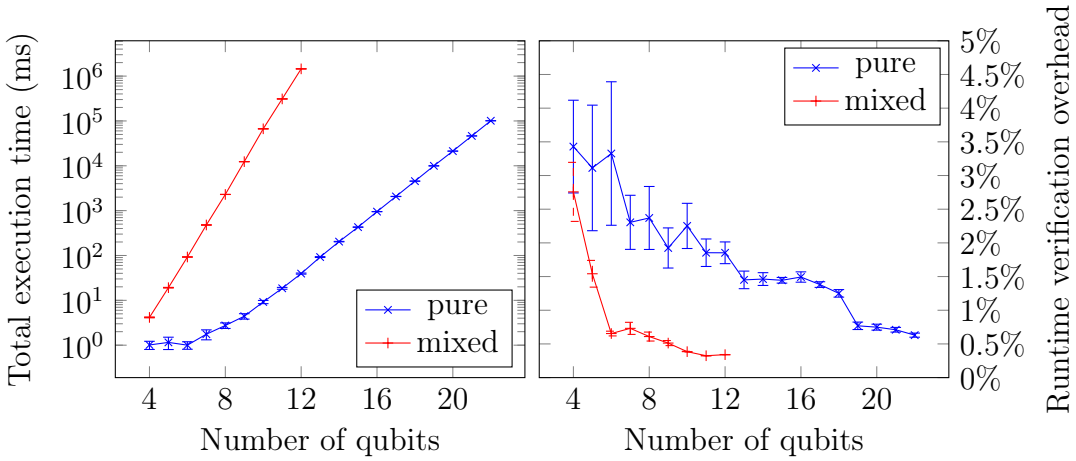

Figure 12-2: Execution performance of *ModMul(n)* programs in Twist. The first plot depicts total execution time as the number of qubits n increases, and the second depicts the percentage of total execution time spent on runtime verification. Error bars display the standard error of the mean of ten runs.

Research Question 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.

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, increasing total runtime. By contrast, each program still needs only one purity assertion to enforce its purity specification.

12.6 RQ5: Comparison to Silq

This section compares Twist to Silq (Bichsel et al., 2020), a recent language that enables high-level programming through type annotations expressing freedom from effects such as mutation and measurement, letting it automatically uncompute certain temporary qubits that exit scope.

Silq was chosen because it is designed to express high-level constructs while preventing unintuitive or physically unrealizable behavior, and because it subsumes features of languages such as Paykin et al. (2017); Green et al. (2013); Svore et al. (2018). Because the benchmarks extensively utilize Twist's annotations, it was evaluated whether Silq's type system accepted a direct translation of the program without any Twist-specific features while still allocating and discarding the same qubits.

Silq rejects a direct translation of the *Teleport-Deferred*, *Deutsch*, and *ShorCode* benchmarks, which, as written in Twist, each discard a temporary expression that has become separable at the end of the computation without measuring its value. Though Silq supports automatically uncomputing certain qubits when they exit scope, it does so only for qubits that are introduced within the same scope by an operation that Silq's type system deems to be `qfree`, i.e. one that does not use unsupported gates with phase-level effects such as Hadamard. 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 via uncomputation.

To translate these benchmarks to Silq, the developer may either 1) manually invoke a `measure` operator on the expression being discarded, 2) manually invoke a `forget` operator on the expression, or 3) apply the `qfree` annotation to the operation that produced it. In the first two cases, neither `measure` nor `forget` guarantees that the expression is separable from the remaining computation, meaning that invoking either operator may in general leave the computation in a mixed state if the developer does not accurately reason about the purity of the computation. In the third case, the `qfree` annotation invokes the requirement of the Silq type system that disqualifies the use of phase-level effects within a `qfree` operation to compute the expression, for example the efficient quantum adder of Draper (2000). As with `forget`, Silq could in principle be extended with an unsafe cast to `qfree` that would permit this implementation, with the proviso that the developer must themselves accurately reason about the safety of such casts in general.

By contrast, Twist soundly accepts these programs when annotated with purity specification, because it can determine that measuring a pure expression cannot affect the ongoing computation, and thus permits implicitly measuring it.

Chapter 13

Extensions and Limitations

Language Features Chapter 9 overview the higher-level syntactic features that are 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 the 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 This work operates 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. Chapter 10 describes 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. 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.

Chapter 14

Related Work

Entanglement Reasoning Perdrix (2005, 2008); Prost and Zerrari (2009); Honda (2015) 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. (2021b,a) 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, in Twist, all 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 Twist's 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. The language's 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 the author's knowledge, this work is 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. This work demonstrates 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.

Chapter 15

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, I 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. I 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 the author's knowledge, this work is the first to define the powerful notion of purity, which enables sound reasoning about entanglement in quantum programs. It is hoped that 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.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Proofs of Semantic Properties

This chapter provides proofs of progress, preservation, and purity soundness theorems for Twist.

A.1 Twist Language

A.1.1 Progress (Theorem 4)

First, the canonical forms lemmas for values are stated:

Lemma 1. *If* $\Gamma \vdash_\Delta e : \tau$ *and* e val*, then:*

- *If* τ *is* ϱ^π*, then* e *is* q^π *where* $\vdash_\Delta q : \varrho$*.*
- *If* τ *is* `bool`*, then* e *is one of* T *or* F*.*
- *If* τ *is* $\tau_1 \times \tau_2$*, then* e *is* (e_1, e_2) *where* e_1 val *and* e_2 val*.*
- *If* τ *is* $\tau_1 \to \tau_2$*, then* e *is* f *and* $\phi(f) = \lambda x.e$*.*

Proof. By inversion of e val. □

Lemma 2. *If* $\vdash_\Delta q : \varrho$*, then if* ϱ *is* `qubit` *then* q *is* $\texttt{ref}[\alpha]$ *for some* α *and if* ϱ *is* $[\varrho_1, \varrho_2]$ *then* q *is* $[q_1, q_2]$ *for some* q_1, q_2*.*

Proof. By inversion of $\vdash_\Delta q : \varrho$. □

Next, the main theorem is proved:

Proof. Proceed by induction on the derivation of $\cdot \vdash_\Delta e : \tau$. T-Var does not apply.

In cases T-Abs, T-Qval, T-Bool it is the case that e val.

In cases T-App, T-Pair, by IH either $|\psi\rangle; e_1 \mapsto \cdot$ or e_1 val. In the former case, apply S-AppL or S-PairL to obtain $|\psi\rangle; e \mapsto \cdot$. In the latter case, by IH either $|\psi\rangle; e_2 \mapsto \cdot$ or e_2 val. In the former case, apply S-AppR or S-PairR and in the latter case apply S-App or V-Pair.

In cases T-Let, T-U1, T-U2, and T-Entangle, by IH either $|\psi\rangle; e \mapsto \cdot$ or e val. In the former case, apply S-LetS, S-U1S, S-U2S, or S-EntangleS. In the latter case, apply S-Let, S-U1, S-U2, or S-Entangle.

In case T-Qinit, apply S-Qinit. In case T-If, by IH either $|\psi\rangle; e \mapsto \cdot$ or e val. In the former case, apply S-IfS. In the latter case, apply S-IfT or S-IfF.

In case T-Measure, by IH either $|\psi\rangle; e \mapsto \cdot$ or e val. In the former case, apply S-MeasureS. In the latter case, apply one of S-MeasureT or S-MeasureF, whose probabilities resulting from a two-outcome quantum measurement add to 1.

In case T-Split, by IH it is the case that $|\psi\rangle; e \mapsto \cdot$ or e val. In the former case, apply S-SplitS. In the latter case, either apply S-SplitMixed, or if the premises of S-SplitPure are true, apply S-SplitPure, otherwise apply S-SplitFail.

In case T-Cast, by IH it is the case that $|\psi\rangle; e \mapsto \cdot$ or e val. In the former case, apply S-CastS. In the latter case, apply S-Cast. □

A.1.2 Preservation (Theorem 6)

The following uses $|\psi'\rangle; q$ pure as shorthand for $|\psi'\rangle; q^{\mathbf{P}}$ pure. First, a helpful lemma is stated. Its proof follows from the fact that purity of quantum values is equivalent to separability, and separable qubits remain separable under irrelevant measurement or unitary operations:

Lemma 3. *If $|\psi\rangle; q$ pure and $|\psi'\rangle$ is any sequence of M_A or U_A applied to $|\psi\rangle$ where A contains no qubits owned by q, then $|\psi'\rangle; q$ pure.*

Next, the main theorem is proved:

Proof. Proceed by induction on the derivation of $|\psi\rangle; e \mapsto |\psi'\rangle; e'$. In each case, to show that $|\psi'\rangle \vDash e'$, it is shown that every pure annotation inside a term that the step introduces is pure. If the step does not introduce pure annotations and does not have any effect on $|\psi\rangle$, then $|\psi\rangle \vDash e$ immediately implies $|\psi'\rangle \vDash e'$.

In case S-App, by inversion of $\Gamma_1, \Gamma_2 \vdash_{\Delta_1,\Delta_2} (\lambda x.e)(e') : \tau$ it is the case that $\Gamma_1, x : \tau' \vdash_{\Delta_1} e : \tau$ and $\Gamma_2 \vdash_{\Delta_2} e' : \tau'$. Thus it is the case that $\Gamma_1, \Gamma_2 \vdash_{\Delta_1,\Delta_2} [e'/x]e : \tau$.

In case S-Let, by inversion of typing it is the case that $\Gamma, x : \tau_1, y : \tau_2 \vdash_{\Delta} e' : \tau$ and $\Gamma_1 \vdash_{\Delta_1} e_1 : \tau_1$ and $\Gamma_2 \vdash_{\Delta_2} e_2 : \tau_2$. Thus it is the case that $\Gamma, \Gamma_1, \Gamma_2 \vdash_{\Delta,\Delta_1,\Delta_2} [e_1, e_2/x, y]e : \tau$.

In case S-Qinit, the introduced term $\texttt{ref}[\alpha]^{\mathbf{P}}$ has type $\texttt{qubit}^{\mathbf{P}}$ and is separable and thus pure in $|\psi\rangle \otimes |0\rangle_{\alpha}$.

In case S-SplitMixed, there are no introduced pure annotations. By inversion of $\Gamma \vdash_{\Delta_1,\Delta_2} \texttt{split}_{\mathbf{M}}([q_1, q_2]^{\mathbf{M}}) : (\varrho_1 \,\&\, \varrho_2)^{\mathbf{M}}$, it is the case that $\vdash_{\Delta_1} q_1 : \varrho_1$ and $\vdash_{\Delta_2} q_2 : \varrho_2$, meaning that $\Gamma \vdash_{\Delta_1,\Delta_2} ({q_1}^{\mathbf{M}}, {q_2}^{\mathbf{M}}) : (\varrho_1 \,\&\, \varrho_2)^{\mathbf{M}}$.

In case S-Entangle, by inversion of $\Gamma \vdash_{\Delta_1,\Delta_2} \texttt{entangle}_{\pi}({q_1}^{\pi}, {q_2}^{\pi}) : (\varrho_1 \,\&\, \varrho_2)^{\pi}$ it is the case that $\vdash_{\Delta_1} q_1 : \varrho_1$ and $\vdash_{\Delta_2} q_2 : \varrho_2$, meaning $\Gamma \vdash_{\Delta_1,\Delta_2} [q_1, q_2]^{\pi} : (\varrho_1 \,\&\, \varrho_2)^{\pi}$. If π is $\mathbf{M}$ then no pure annotations are introduced. If π is $\mathbf{P}$, by the IH it is the case that $|\psi\rangle; q_1$ pure and $|\psi\rangle; q_2$ pure, meaning $|\psi\rangle; [q_1, q_2]$ pure and it is the case that $|\psi\rangle \vDash [q_1, q_2]^{\mathbf{P}}$.

In case S-Cast, where π is $\mathbf{M}$, no pure annotations are introduced. By inversion of $\Gamma \vdash_\Delta \mathtt{cast}_{\mathbf{M}}(q^\pi) : \varrho^{\mathbf{M}}$ it is the case that $\vdash_\Delta q : \varrho$, meaning $\Gamma \vdash_\Delta q^{\mathbf{M}} : \varrho^{\mathbf{M}}$.

In case S-U1, by inversion of $\Gamma \vdash_\Delta U(\mathtt{ref}[\alpha]^\pi) : \varrho^\pi$ it is the case that $\vdash_\Delta \mathtt{ref}[\alpha] : \varrho$, meaning $\Gamma \vdash_\Delta \mathtt{ref}[\alpha]^\pi : \varrho^\pi$. If π is $\mathbf{M}$ no pure annotations are introduced. If π is $\mathbf{P}$, then by the IH it is the case that $|\psi\rangle; q_1$ pure. Because the unitary operator only acts on α, it is the case that $U_\alpha |\psi\rangle; q_1$ pure meaning $U_\alpha |\psi\rangle \vDash \mathtt{ref}[\alpha]^{\mathbf{P}}$.

In case S-U2, the same reasoning applies as for S-U1, except that if π is pure then by the IH it is the case that $|\psi\rangle; [\mathtt{ref}[\alpha], \mathtt{ref}[\beta]]$ pure and the unitary operator only acts on α, β, meaning it is the case that $U_{\alpha,\beta} |\psi\rangle; [\mathtt{ref}[\alpha], \mathtt{ref}[\beta]]$ pure and $U_{\alpha,\beta} |\psi\rangle \vDash [\mathtt{ref}[\alpha], \mathtt{ref}[\beta]]^{\mathbf{P}}$.

In cases S-IfT and S-IfF, by inversion of $\Gamma \vdash_\Delta \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \varrho^{\mathbf{M}}$, where e is T or F, it is the case that $\Gamma \vdash_\Delta e_1 : \varrho^\pi$ and $\Gamma \vdash_\Delta e_2 : \varrho^\pi$, meaning $\Gamma \vdash_\Delta \mathtt{cast}_{\mathbf{M}}(e_1) : \varrho^{\mathbf{M}}$ and $\Gamma \vdash_\Delta \mathtt{cast}_{\mathbf{M}}(e_2) : \varrho^{\mathbf{M}}$.

In cases S-MeasureT and S-MeasureF, no pure annotations are introduced. By inversion of typing, $\mathtt{measure}(\mathtt{ref}[\alpha]^\pi)$ also has Boolean type.

In case S-SplitPure, by inversion of $\Gamma \vdash_{\Delta_1,\Delta_2} \mathtt{split}_{\mathbf{P}}([q_1, q_2]^{\mathbf{P}}) : {\varrho_1}^{\mathbf{P}} \times {\varrho_2}^{\mathbf{P}}$ it is the case that $\vdash_{\Delta_1} q_1 : \varrho_1$ and $\vdash_{\Delta_2} q_2 : \varrho_2$, meaning $\Gamma \vdash_{\Delta_1,\Delta_2} ({q_1}^{\mathbf{P}}, {q_2}^{\mathbf{P}}) : {\varrho_1}^{\mathbf{P}} \times {\varrho_2}^{\mathbf{P}}$. The premise is the separability condition that implies compatibility.

In cases S-AppL, S-AppR, S-PairL, S-PairR, S-LetS, S-SplitS, S-EntangleS, S-CastS, S-U1S, S-U2S, S-IfS, S-MeasureS, and S-SplitS, the IH directly implies type preservation and state/expression compatibility. □

A.1.3 Purity Soundness (Theorem 7)

First, two properties of the language are stated that hold as a consequence of it being a variant of the linear simply-typed λ-calculus with type safety.

Lemma 4 (Strong Normalization)**.** *If $\cdot \vdash_\Delta e : \tau$ then for all $|\psi\rangle$ such that $\Delta \subseteq \mathrm{dom}\,|\psi\rangle$, there exists v such that $|\psi\rangle; e \mapsto^* |\psi'\rangle; v$ and v val.*

This lemma implies the existence of an evaluation $|\psi\rangle; e \mapsto^* |\psi'\rangle; v$ for some $|\psi'\rangle$ and v.

Lemma 5 (Call-By Equivalence)**.** *$|\psi\rangle; (\lambda x.e)(e') \mapsto^* |\psi'\rangle; v$ such that v val if and only if $|\psi\rangle; [e'/x]e \mapsto^* |\psi'\rangle; v$. Likewise, $|\psi\rangle; \mathtt{let}\ (x, y) = (e_1, e_2)\ \mathtt{in}\ e \mapsto^* |\psi'\rangle; v$ where v val if and only if $|\psi\rangle; [e_1, e_2/x, y]e \mapsto^* |\psi'\rangle; v$.*

In this statement, e', e_1 and e_2 need not be values, and thus this lemma asserts the equivalence of call-by-name and call-by-value evaluation strategies. Because in a linear λ-calculus every variable occurs once without discarding or duplication, any effects (measurement or unitary operator) of the eagerly-evaluated argument of a function application or `let`-binding still occur exactly once if they are instead substituted before evaluation. Furthermore, the order of effects of two components of a pair does not matter, because linearity requires them to refer to disjoint sets of qubits, and reversing the order of measurements or unitary operators on disjoint sets of qubits cannot change the computation outcome.

The purity soundness theorem is proved by logical relations, strengthening the induction hypothesis to describe types other than $\mathbf{Q}^{\mathbf{P}}$ and typing judgments that involve non-empty contexts. The relation is purity at a type τ, denoted $|\psi\rangle; e\ \mathsf{pure}_\tau$.

$$
\begin{aligned}
|\psi\rangle; e\ \mathsf{pure}_{\mathbf{Q}^{\mathbf{P}}} &= |\psi\rangle; e\ \mathsf{pure} \\
|\psi\rangle; e\ \mathsf{pure}_{\mathbf{Q}^{\mathbf{M}}} &= \top \\
|\psi\rangle; e\ \mathsf{pure}_{\mathtt{bool}} &= \top \\
|\psi\rangle; e\ \mathsf{pure}_{\tau_1 \to \tau_2} &= \forall e', |\psi\rangle; e'\ \mathsf{pure}_{\tau_1} \Rightarrow |\psi\rangle; e(e')\ \mathsf{pure}_{\tau_2} \\
|\psi\rangle; e\ \mathsf{pure}_{\tau_1 \times \tau_2} &= \forall e_1, e_2, e', (|\psi\rangle; e_1\ \mathsf{pure}_{\tau_1} \Rightarrow |\psi\rangle; e_2\ \mathsf{pure}_{\tau_2} \Rightarrow |\psi\rangle; [e_1, e_2/x, y]e'\ \mathsf{pure}_\tau) \\
&\quad \Rightarrow |\psi\rangle; \mathtt{let}\ (x, y) = e\ \mathtt{in}\ e'\ \mathsf{pure}_\tau
\end{aligned}
$$

Purity at type $\mathbf{Q}^{\mathbf{P}}$ simply invokes the existing definition. At mixed or Boolean type, purity contains no information. A function $\tau_1 \to \tau_2$ is pure when applying it to a pure argument at τ_1 yields an output pure at τ_2. Finally, a pure product $\tau_1 \times \tau_2$ contains two elements that are pure at τ_1 and τ_2 respectively. This is represented by stating the elimination form of a product, a `let`-expression, is pure at τ if its body e' is pure at τ after being substituted with two expressions pure at τ_1 and τ_2.

The following lemma states that reversal of deterministic steps preserves the purity relation.

Lemma 6. *If* $|\psi\rangle; e \mapsto_1 |\psi'\rangle; e'$ *deterministically and* $|\psi'\rangle; e'\ \mathsf{pure}$ *then* $|\psi\rangle; e\ \mathsf{pure}$.

The proof follows directly from the definition of purity. Next, the standard notion of a substitution γ is defined to map an open expression e to a closed expression $\gamma(e)$. Define the compatibility judgment $\gamma \vDash \Gamma, \Delta, |\psi\rangle$ to mean that γ maps every variable $x_i : \tau_i$ in Γ to a closed term e_i such that $\cdot \vdash_{\Delta_i} e_i : \tau_i$ where Δ and all Δ_i are disjoint, and $|\psi\rangle \vDash \Delta_i$ and $|\psi\rangle; e_i\ \mathsf{pure}_{\tau_i}$.

Now the strengthened purity soundness theorem is stated. When τ is $\mathbf{Q}^{\mathbf{P}}$ and Γ is empty, the strengthening implies the original theorem.

Theorem 9. *If* $\Gamma \vdash_\Delta e : \tau$ *and* $\gamma \vDash \Gamma, \Delta, |\psi\rangle$ *and* $|\psi\rangle \vDash \gamma(e)$*, then* $|\psi\rangle; \gamma(e)\ \mathsf{pure}_\tau$.

Proof. Proceed by induction on the derivation of $\Gamma \vdash_\Delta e : \tau$.

The case for T-Var holds directly from the hypothesis. In case T-Qval, no substitution can occur. If $a = \mathbf{M}$ then purity is trivial. Otherwise, because $|\psi\rangle \vDash e$ it is the case that $|\psi\rangle; e\ \mathsf{pure}$. In case T-Qinit, no substitution can occur, and e has only one deterministic transition and purity is trivial.

In case T-U1, by the IH, $\cdot \vdash_\Delta \gamma(e) : \mathtt{qubit}^{\mathbf{P}}$ and $|\psi\rangle \vDash \gamma(e)$ so $|\psi\rangle; \gamma(e)\ \mathsf{pure}$. Any evaluation $|\psi\rangle; U(\gamma(e)) \mapsto^* |\psi'\rangle; e'$ where $e'\ \mathsf{val}$ is of the form $|\psi\rangle; U(\gamma(e)) \mapsto \psi_1; U(e_1) \mapsto \ldots \mapsto |\psi'\rangle; U(\mathtt{ref}[\alpha]^{\mathbf{P}}) \mapsto_1 U_\alpha |\psi\rangle; \mathtt{ref}[\alpha]^{\mathbf{P}}$. Inverting each step, because $|\psi\rangle; \gamma(e)\ \mathsf{pure}$ it is the case that the execution up to $|\psi'\rangle$ is unique. By preservation, $U_\alpha |\psi'\rangle \vDash \mathtt{ref}[\alpha]^{\mathbf{P}}$ so $U_\alpha |\psi'\rangle; \mathtt{ref}[\alpha]^{\mathbf{P}}\ \mathsf{pure}$. By Lemma 6, it is the case that $|\psi'\rangle; U(\mathtt{ref}[\alpha]^{\mathbf{P}})\ \mathsf{pure}$. Stitching the two deterministic executions together, it is the case that $|\psi\rangle; \gamma(U(e))\ \mathsf{pure}$.

Case T-U2 follows by the same reasoning as for T-U1, with the only difference being that a two-qubit operator $U_{\alpha,\beta}$ is executed on a pair of qubits in the deterministic step.

Case T-ENTANGLE also follows by similar reasoning. If $a = \mathbf{M}$ then purity is trivial. Otherwise, every execution first evaluates $\gamma(e)$ to a compatible final state and value $|\psi\rangle \vDash [q_1, q_2]^{\mathbf{P}}$, implying $|\psi\rangle; [q_1, q_2]$ pure. Reverse this step and stitch together the deterministic executions of $\gamma(e_1)$ and $\gamma(e_2)$ to obtain $|\psi\rangle; \gamma(\mathtt{entangle}_{\pi}(e_1, e_2))$ pure.

In case T-ABS, it is the case that $x : \tau \vdash_{\Delta} \gamma(e) : \tau'$. By the IH it is the case that if $|\psi\rangle; e'$ pure_{τ} then $|\psi\rangle; [e'/x]\gamma(e)$ $\mathsf{pure}_{\tau'}$. From this it is the case that $|\psi\rangle; (\lambda x.\gamma(e))(e')$ $\mathsf{pure}_{\tau'}$ by Lemma 5. It also holds that $|\psi\rangle; \gamma(\lambda x.e)$ $\mathsf{pure}_{\tau \to \tau'}$.

In case T-PAIR, it is the case that $\cdot \vdash_{\Delta_1} \gamma(e_1) : \tau_1$ and $\cdot \vdash_{\Delta_2} \gamma(e_2) : \tau_2$. By the IH it is the case that $|\psi\rangle; \gamma(e_1)$ pure_{τ_1} and $|\psi\rangle; \gamma(e_2)$ pure_{τ_2}. Thus, assuming $|\psi\rangle; [\gamma(e_1), \gamma(e_2)/x, y]e$ pure_{τ}, it is the case that $|\psi\rangle;$ $\mathtt{let}$ $(x, y) = (\gamma(e_1), \gamma(e_2))$ $\mathtt{in}$ e pure_{τ} by Lemma 5, and so $|\psi\rangle; (\gamma(e_1), \gamma(e_2))$ $\mathsf{pure}_{\tau_1 \times \tau_2}$.

In case T-APP, it is the case that $\cdot \vdash_{\Delta_1} \gamma(e_1) : \tau_1 \to \tau_2$ and $\cdot \vdash_{\Delta_2} \gamma(e_2) : \tau_1$. By the IH it is the case that $|\psi\rangle; \gamma(e_1)$ $\mathsf{pure}_{\tau_1 \to \tau_2}$ and $|\psi\rangle; \gamma(e_2)$ pure_{τ_1}, thus $|\psi\rangle; \gamma(e_1(e_2))$ pure_{τ_2}.

In case T-LET, it is the case that $\cdot \vdash_{\Delta_1} \gamma(e) : \tau_1 \times \tau_2$ and $x : \tau_1, y : \tau_2 \vdash_{\Delta_2}$ $\gamma(e') : \tau$. By the IH it is the case that $|\psi\rangle; \gamma(e)$ $\mathsf{pure}_{\tau_1 \times \tau_2}$ and that if $|\psi\rangle; e_1$ pure_{τ_1} and $|\psi\rangle; e_2$ pure_{τ_2} then $|\psi\rangle; [e_1, e_2/x, y]\gamma(e')$ pure_{τ}. Thus, $|\psi\rangle; \gamma(\mathtt{let}$ $(x, y) = e$ $\mathtt{in}$ $e')$ pure_{τ}.

In case T-SPLITPURE, it is the case that $\cdot \vdash_{\Delta} \gamma(e) : (\mathsf{Q}_1 \,\&\, \mathsf{Q}_2)^{\mathbf{P}}$ and need to show that
$|\psi\rangle; \mathtt{split}_{\mathbf{P}}(\gamma(e))$ $\mathsf{pure}_{\mathsf{Q}_1{}^{\mathbf{P}} \times \mathsf{Q}_2{}^{\mathbf{P}}}$. By the IH it is the case that $|\psi\rangle; \gamma(e)$ pure.

Proceed by the same reasoning as case T-U1 to fully evaluate $\gamma(e)$ by rule S-SPLITS and deterministically obtain $[q_1, q_2]^{\mathbf{P}}$ under the unique state $|\psi'\rangle$.

First assume that the separability condition holds. The next step is S-SPLITPURE, and inverting it yields its premises $|\psi'\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_0\rangle$, $\mathrm{dom}\,|\psi_1\rangle = \mathsf{Refs}(q_1)$, $\mathrm{dom}\,|\psi_2\rangle = \mathsf{Refs}(q_2)$, implying that $|\psi'\rangle; q_1$ pure and $|\psi'\rangle; q_2$ pure. Thus, assuming $|\psi'\rangle; [q_1{}^{\mathbf{P}}, q_2{}^{\mathbf{P}}/x, y]e'$ pure_{τ}, after applying Lemma 5 it is also the case that $|\psi'\rangle;$ $\mathtt{let}$ $(x, y) = (q_1{}^{\mathbf{P}}, q_2{}^{\mathbf{P}})$ $\mathtt{in}$ e' pure_{τ}.

Now assume that the separability condition fails. Then, every evaluation of $\mathtt{split}_{\mathbf{P}}(\gamma(e))$ will next take step S-SPLITFAIL, meaning that every evaluation fails and purity holds vacuously.

The remaining cases yield outputs of mixed or Boolean type, for which purity holds trivially. □

A.2 Static Analysis for Purity

A.2.1 Purity Soundness (Theorem 8)

That all $\mathbf{P}$ annotations introduced by the semantics of Twist without $\mathtt{cast}_{\mathbf{P}}$ are on pure expressions has already been shown. It remains to show that if the static analysis assigns an expression e a pure quantum type, then e is pure. If so, rule S-CAST only introduces $\mathbf{P}$ annotations on pure expressions, implying the soundness theorem.

Proof. Proceed by induction on the derivation of $\Gamma \vdash_A e : \varrho^{\mathbf{P}}$. It is only necessary to examine new $\mathbf{P}$ annotations introduced by the $\vdash_A$ judgment.

In case A-CastPure, if e has pure type then by the IH it is pure, meaning $\texttt{cast}_{\mathbf{P}}(e)$ is also pure.

In case A-SplitMixed, $g = \mathsf{Split}(f, j)$ is never $\mathbf{P}$, so purity is trivially satisfied.

In case A-Entangle, if h is not $\mathbf{P}$ then purity is trivial. In addition, if f and g are both $\mathbf{P}$ then the same reasoning as in the original proof applies. If f and g are not $\mathbf{P}$ but h is pure, it is necessary to show that $\texttt{entangle}_{\pi}(e_1, e_2)$ is pure. Suppose it is not. The first possibility is that it evaluates to an entangled pair $[e_1, e_2]$ that is entangled with some qubit α not owned by e_1 or e_2. But this means one of e_1 or e_2 is entangled with α; without loss of generality let it be e_1. Then, some $\texttt{split}_{\mathbf{M}}$ occurred with α on one side and e_1 on the other, introducing a term for x_i in the history f. Because α is not owned by e_2, x_i is absent from the history g and must be present in $h = \mathsf{Combine}(f, g)$, contradicting the fact that h is $\mathbf{P}$. The second possibility is that the evaluation of $\texttt{entangle}_{\pi}(e_1, e_2)$ encounters an `if`-expression that evaluates to a mixed state that is not discarded. But then this `if`-expression appears in the evaluation of e_1 or e_2, meaning by preservation that either f or g is $\mathbf{M}$ and thus $h = \mathsf{Combine}(f, g)$ is $\mathbf{M}$, which is also a contradiction.

All other rules follow by the same reasoning as in the proof of Theorem 9. □

Appendix B

Full Benchmark Programs

This chapter presents the full source code for each benchmark. Several benchmarks use syntactic features of Twist not discussed in the main paper, including inference of purity assertions and polymorphic purity annotations, which are described in Chapter 9.

B.1 Teleport-Deferred

```
fun bell_pair () : (qubit & qubit)<P> =
  CNOT (H (qinit ()), qinit ())

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 = H (q1) in
  let (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
  let (q1 : qubit<M>, q3 : qubit<M>) = CZ (q1, q3) in
  let all : ((qubit & qubit) & qubit)<P> = ((q1, q2), q3) in
  let (_ : (qubit & qubit)<P>, q3 : qubit<P>) = all in q3

fun main () : qubit<P> = teleport (H (qinit ()))
```

B.2 Teleport-NoCZ

All other functions are identical to the *Teleport-Deferred* example.

```
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 (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
  let (q1 : qubit<M>, q3 : qubit<M>) = CNOT (q1, q3) in
  let all : ((qubit & qubit) & qubit)<P> = ((q1, q2), q3) in
  (* Dynamic separability check failure: argument entangled *)
  let (discard : (qubit & qubit)<P>, q3 : qubit<P>) = all in
  let discard = measure (discard) in q3
```

B.3 Teleport-Measure

All other functions are identical to the *Teleport-Deferred* example.

```
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) (* Static analysis failure: q1 and q2 not covered *)
```

B.4 AndOracle

```
fun and_oracle (p0 : qubit<P>, p1 : qubit<P>) : (qubit & qubit)<P> =
  let x = qinit () in
  let (p0 : qubit<M>, (p1 : qubit<M>, x : qubit<M>)) = TOF (p0, (p1, x)) in
  let (p0 : qubit<M>, (p1 : qubit<M>, x : qubit<M>)) = TOF (p0, (p1, x)) in
  let qs : (qubit & (qubit & qubit))<P> = (x, (p0, p1)) in
  let (x : qubit<P>, rest : (qubit & qubit)<P>) = qs in rest
fun main () : (qubit & qubit)<P> = and_oracle (H (qinit ()), X (qinit ()))
```

B.5 AndOracle-NotUncomputed

```
fun and_oracle (p0 : qubit<P>, p1 : qubit<P>) : (qubit & qubit)<P> =
  let x = qinit () in
  let (p0 : qubit<M>, (p1 : qubit<M>, x : qubit<M>)) = TOF (p0, (p1, x)) in
  let p0 = Z (p0) in
  let _ = measure x in
  entangle<P>(p0, p1) (* Type error: p0 and p1 are mixed *)
fun main () : (qubit & qubit)<P> = and_oracle (H (qinit ()), X (qinit ()))
```

B.6 Bell-GHZ

```
fun main () : (qubit & qubit)<P> =
  let q1 = H (qinit ()) in
  let (q1 : qubit<M>, q2 : qubit<M>) = CNOT (q1, qinit ()) in
  let (q1 : qubit<M>, q3 : qubit<M>) = CNOT (q1, qinit ()) in
  let _ = measure q3 in
  entangle<P>(q1, q2) (* Type error: q1 and q2 are mixed *)
```

B.7 Deutsch

```
fun deutsch (uf : (qubit & qubit)<P> -> (qubit & qubit)<P>) : bool =
  let input : (qubit & qubit)<P> = (H (qinit ()), H (X (qinit ()))) in
  let (x : qubit<P>, _ : qubit<P>) = uf (input) in
  measure (H (x))

fun cnot (xy : (qubit & qubit)<P>) : (qubit & qubit)<P> = CNOT (xy)

fun always_true (xy : (qubit & qubit)<P>) : (qubit & qubit)<P> =
  let (x : qubit<M>, y : qubit<M>) = xy in
  cast<P>(entangle<M>(x, X (y)))

fun always_false (xy : (qubit & qubit)<P>) : (qubit & qubit)<P> = xy

fun main () : ((bool * bool) * bool) =
  ((deutsch (always_false), deutsch (always_true)), deutsch (cnot))
```

B.8 Deutsch-BadResultBasis

All other functions are identical to the *Deutsch* example.

```
fun deutsch (uf : (qubit & qubit)<P> -> (qubit & qubit)<P>) : bool =
  (* Dynamic separability check failure: argument entangled *)
  let (x : qubit<P>, y : qubit<P>) =
    uf (entangle<P>(H (qinit ()), (X (qinit ())))) in
  let _ = measure (y) in
  measure (H (x))
```

B.9 DeutschJozsa

```
type oracle = ((qubit & qubit) & qubit)<P> -> ((qubit & qubit) & qubit)<P>
type domain = (qubit & qubit)<P>

(* An (entangled) domain-codomain pair *)
type graph_pt = ((qubit & qubit) & qubit)<P>

(* Prepare domain qubits in |0> + |1> *)
fun init_domain () : (qubit<P> * qubit<P>) = (H (qinit ()), H (qinit ()))

(* Prepare output qubit in |0> - |1> *)
fun init_output () : qubit<P> = H (X qinit ())

fun test_oracle (f : oracle) : graph_pt =
  let out : qubit<P> = init_output () in
  let dom : (qubit<P> * qubit<P>) = init_domain () in
  let all : graph_pt = (dom, out) in
  let inout : graph_pt = f (all) in
  let ((d0 : qubit<M>, d1 : qubit<M>), out: qubit<M>) = inout in
  (* Hadamard the domain qubits *)
  let (inout_post : ((qubit & qubit) & qubit)<M>) =
    (((H d0), (H d1)), out) in
```

```
  cast<P>(inout_post)

(* A balanced function {0, 1}^2 -> {0, 1} that selects states with second
    qubit |1> *)
fun is_odd (pt : graph_pt) : graph_pt =
  let ((d0 : qubit<M>, d1 : qubit<M>), out : qubit<M>) = pt in
  let (d1 : qubit<M>, out : qubit<M>) = (CNOT (d1, out)) in
  let (inout : ((qubit & qubit) & qubit)<M>) = ((d0, d1), out) in
  cast<P>(inout)

fun main () : graph_pt = test_oracle (is_odd)
```

B.10 DeutschJozsa-MixedInit

The other functions are identical to the *DeutschJozsa* example.

```
fun init_domain () : (qubit<M> * qubit<M>) =
  let (x : qubit<M>, y : qubit<M>) = CNOT (H (qinit ()), qinit ()) in
  let _ = measure (y) in
  (x, cast<M>(qinit ()))

fun test_oracle (f : oracle) : graph_pt =
  let out : qubit<P> = init_output () in
  let dom : (qubit<M> * qubit<M>) = init_domain () in
  let all : graph_pt = (dom, out) in
  let inout : graph_pt = f (all) in
  let ((d0 : qubit<M>, d1 : qubit<M>), out: qubit<M>) = inout in
  (* Hadamard the domain qubits *)
  let (inout_post : ((qubit & qubit) & qubit)<M>) =
    (((H d0), (H d1)), out) in
  (* Static analysis failure: mixed output of init_domain () not covered *)
  cast<P>(inout_post)
```

B.11 Grover

```
type addr = (qubit & qubit)<P>
type oracle = (qubit & qubit)<P> -> (qubit & qubit)<P>
fun init_addr () : addr = entangle<P>(H qinit(), H qinit ())
fun diffuse (p : addr) : addr =
  let (p0 : qubit<M>, p1 : qubit<M>) = p in
  let (p0 : qubit<M>, p1 : qubit<M>) = (H p0, H p1) in
  let (p0 : qubit<M>, p1 : qubit<M>) = (Z p0, Z p1) in
  let (p0 : qubit<M>, p1 : qubit<M>) = CZ (p0, p1) in
  let (p0 : qubit<M>, p1 : qubit<M>) = (H p0, H p1) in
  let p = entangle<M>(p0, p1) in
  cast<P>(p)

fun grover (f : oracle) : addr =
  let addr = init_addr () in
  let addr = f (addr) in
```

```
  let addr = diffuse (addr) in
  addr

fun final_addr (p : addr) : addr = (CZ (p))
fun main () : addr = grover (final_addr)
```

B.12 Grover-BadOracle

The other functions are identical to the *Grover* example.

```
fun final_addr (p : addr) : addr =
  let (p0 : qubit<M>, p1 : qubit<M>) = p in
  let x = qinit () in
  let (p0 : qubit<M>, (p1 : qubit<M>, x : qubit<M>)) = TOF (p0, (p1, x)) in
  let (x : qubit<M>, p0 : qubit<M>) = CZ (x, p0) in
  let _ = measure x in
  cast<P>(entangle<M>(p0, p1)) (* Static analysis failure: x not covered *)
```

B.13 QFT

```
fun qft_sub_1 (q : qubit<'p>) : qubit<'p> = H q

fun qft_sub_2 (qs : (qubit & qubit)<'p>) : (qubit & qubit)<'p> =
  let (q0 : qubit<M>, q1 : qubit<M>) = cast<M>(qs) in
  let q0 = qft_sub_1 (q0) in
  (* Controlled phase of 2 pi / 2 ** 2 *)
  let (qs : (qubit & qubit)<M>) = CPHASE 0.250 (q1, q0) in
  let (q1 : qubit<M>, q0 : qubit<M>) = qs in
  let (qs : (qubit & qubit)<M>) = (q0, q1) in
  cast<'p>(qs)

fun qft_sub_3 (qs : ((qubit & qubit) & qubit)<'p>) :
  ((qubit & qubit) & qubit)<'p> =
  let (qs : (qubit & qubit)<M>, q2 : qubit<M>) = qs in
  let qs = qft_sub_2 (qs) in
  let (q0 : qubit<M> , q1 : qubit<M>) = qs in
  (* Controlled phase of 2 pi / 2 ** 3 *)
  let (q2 : qubit<M>, q0 : qubit<M>) = CPHASE 0.125 (q2, q0) in
  let (qs : ((qubit & qubit) & qubit)<M>) = ((q0, q1), q2) in
  cast<'p>(qs)

fun qft_1 (q : qubit<'p>) : qubit<'p> = qft_sub_1 (q)

fun qft_2 (qs : (qubit & qubit)<'p>) : (qubit & qubit)<'p> =
  let qs = qft_sub_2 (qs) in
  let (q0 : qubit<M>, q1 : qubit<M>) = qs in
  let q1 = qft_1 (q1) in
  let (qs: (qubit & qubit)<M>) = (q0, q1) in
  cast<'p>(qs)

```

```
fun qft_3 (qs : ((qubit & qubit) & qubit)<'p>) :
  ((qubit & qubit) & qubit)<'p> =
  let qs = qft_sub_3 (qs) in
  let ((q0 : qubit<M>, q1 : qubit<M>), q2 : qubit<M>) = qs in
  let tail : (qubit & qubit)<M> = (q1, q2) in
  let (q1 : qubit<M>, q2 : qubit<M>) = qft_2 (tail) in
  let (qs: ((qubit & qubit) & qubit)<M>) = ((q0, q1), q2) in
  cast<'p>(qs)

fun main () : ((qubit & qubit) & qubit)<P> =
  let qs : ((qubit & qubit) & qubit)<pure> =
    ((qinit(), X qinit()), qinit()) in
  qft_3 (qs)
```

B.14 ShorCode

```
type triple_p = (qubit & (qubit & qubit))<P>
type triple_m = (qubit & (qubit & qubit))<M>
type nonuple_p = ((qubit & (qubit & qubit)) &
                  ((qubit & (qubit & qubit)) &
                   (qubit & (qubit & qubit))))<P>
type nonuple_m = ((qubit & (qubit & qubit)) &
                  ((qubit & (qubit & qubit)) &
                   (qubit & (qubit & qubit))))<M>

(* Encode a qubit with the three-qubit bit-flip code *)
fun enc_bit (q : qubit<'p>) : (qubit & (qubit & qubit))<'p> =
  let (a1 : qubit<M>) = qinit () in
  let (a2 : qubit<M>) = qinit () in
  let (q : qubit<M>, a2 : qubit<M>) = (CNOT (q, a2)) in
  let (q : qubit<M>, a1 : qubit<M>) = (CNOT (q, a1)) in
  let (out : (qubit & (qubit & qubit))<M>) = (q, (a1, a2)) in
  cast<'p>(out)

(* Encode a qubit with the three-qubit phase-flip code *)
fun enc_phase (q : qubit<'p>) : (qubit & (qubit & qubit))<'p> =
  let (x : qubit<M>, (y : qubit<M>, z : qubit<M>)) = enc_bit (q) in
  let (out : (qubit & (qubit & qubit))<M>) = (H x, (H y, H z)) in
  cast<'p>(out)

(* Encode a qubit with the nine-qubit Shor code by concatenating the bit- and
 * phase-flip codes *)
fun enc_shor (q : qubit<P>) : nonuple_p =
  let (x : qubit<M>, (y : qubit<M>, z : qubit<M>)) = enc_phase (cast<M>(q)) in
  let (out : nonuple_m) = (enc_bit (x), (enc_bit (y), enc_bit (z))) in
  cast<P>(out)

(* Decode the three-qubit bit-flip code *)
fun dec_bit (enc : (qubit & (qubit & qubit))<'p>) :
  (qubit & (qubit & qubit))<'p> =
  let (q0 : qubit<M>, tail : (qubit & qubit)<M>) = enc in
  let (q1 : qubit<M>, q2 : qubit<M>) = tail in
```

```
  let (q0 : qubit<M>, q1 : qubit<M>) = (CNOT (q0, q1)) in
  let (q0 : qubit<M>, q2 : qubit<M>) = (CNOT (q0, q2)) in
  let (q2 : qubit<M>, (q1 : qubit<M>, q0 : qubit<M>)) = TOF (q2, (q1, q0)) in
  let env : (qubit & qubit)<M> = entangle<M>(q1, q2) in
  let out : (qubit & (qubit & qubit))<M> = entangle<M>(q0, env) in
  cast<'p>(out)

(* Decode the three-qubit phase-flip code *)
fun dec_phase (enc : (qubit & (qubit & qubit))<'p>) :
  (qubit & (qubit & qubit))<'p> =
  let (x : qubit<M>, (y : qubit<M>, z : qubit<M>)) = enc in
  let qs : (qubit & (qubit & qubit))<M> = (H x, (H y, H z)) in
  let out : (qubit & (qubit & qubit))<'p> = dec_bit (cast<'p>(qs)) in
  out

(* Decode the Shor code, without discarding the extra bits *)
fun dec_shor (enc : nonuple_p) : nonuple_p =
  let (x : triple_m, (y : triple_m, z : triple_m)) = enc in
  let (x : triple_m, (y : triple_m, z : triple_m)) =
    (dec_bit (x), (dec_bit (y), dec_bit(z))) in

  let (x0 : qubit<M>, (x1 : qubit<M>, x2 : qubit<M>)) = x in
  let (y0 : qubit<M>, (y1 : qubit<M>, y2 : qubit<M>)) = y in
  let (z0 : qubit<M>, (z1 : qubit<M>, z2 : qubit<M>)) = z in
  let heads : triple_m = (x0, (y0, z0)) in
  let (x0 : qubit<M>, (y0 : qubit<M>, z0 : qubit<M>)) = dec_phase (heads) in

  let x = (x0, (x1, x2)) in
  let y = (y0, (y1, y2)) in
  let z = (z0, (z1, z2)) in
  let dec : nonuple_m = (x, (y, z)) in
  cast<P>(dec)

fun test_bitflip (q : qubit<P>) : qubit<P> =
  let (q0 : qubit<M>, tail : (qubit & qubit)<M>) = enc_bit (q) in
  let (q1 : qubit<M>, q2 : qubit<M>) = tail in
  let qs : (qubit & (qubit & qubit))<M> = (q0, (q1, q2)) in
  let (q : qubit<P>, env : (qubit & qubit)<P>) =
    dec_bit (cast<P>(qs)) in
  q

(* Accept a qubit and a noise operation.
 * Encode the qubit, apply the given noise
 * operation on the physical qubits,
 * then decode and discard the extra bits. *)
fun shor_ecc (qop : (qubit<P> * (nonuple_p -> nonuple_p))) : qubit<P> =
  (* Encode the qubit *)
  let (q : qubit<P>, op : nonuple_p -> nonuple_p) = qop in
  let enc = enc_shor (q) in
  (* Disturb the encoded state with the noise operation *)
  let enc = op (enc) in
  (* Decode and discard the parity bits *)
  let dec = dec_shor (enc) in
  let (x : triple_p, (y : triple_p, z : triple_p)) = dec in
```

```
  let (dec : qubit<P>, others : (qubit & qubit)<P>) = x in
  dec

(* A unitary channel that produces a single phase flip *)
fun phaseflip_channel (enc : nonuple_p) : nonuple_p =
  let (x : triple_m, (y : triple_m, z : triple_m)) = enc in
  let (x0 : qubit<M>, x_tail : (qubit & qubit)<M>) = x in
  let x0 = Z x0 in
  let x : triple_m = (x0, x_tail) in
  let enc : nonuple_p = (x, (y, z)) in
  cast<P>(enc)

fun main () : qubit<P> = shor_ecc ((H (qinit ()), phaseflip_channel))
```

B.15 ShorCode-Drop

All other functions are identical to the *ShorCode* example.

```
fun enc_bit (q : qubit<'p>) : ((qubit & qubit) & qubit)<'p> =
  let (a1 : qubit<M>) = qinit () in
  let (a2 : qubit<M>) = qinit () in
  let (q : qubit<M>) = qinit () in (* Type error: drops shadowed q *)
  let (q : qubit<M>, a2 : qubit<M>) = (CNOT (q, a2)) in
  let (q : qubit<M>, a1 : qubit<M>) = (CNOT (q, a1)) in
  let (out : ((qubit & qubit) & qubit)<M>) = ((q, a1), a2) in
  cast<'p>(out)
```

B.16 ModMul(4)

Next is the modular multiplication benchmark for $n = 4$. All other benchmarks in this family are very similar.

```
type five = (qubit & (((qubit & qubit) & qubit) & qubit))<P>
type four_m = (((qubit & qubit) & qubit) & qubit)<M>
type four_p = (((qubit & qubit) & qubit) & qubit)<P>

(* controlled multiply by 7 mod 15,
 * using negation followed by three controlled swaps *)
fun mult7 (cqs : five) : five =
  let (c : qubit<M>, qs : four_m) = cqs in
  let (((q1 : qubit<M>, q2 : qubit<M>), q3 : qubit<M>), q4 : qubit<M>) = qs in
  let (c : qubit<M>, q1 : qubit<M>) = CNOT (c, q1) in
  let (c : qubit<M>, q2 : qubit<M>) = CNOT (c, q2) in
  let (c : qubit<M>, q3 : qubit<M>) = CNOT (c, q3) in
  let (c : qubit<M>, q4 : qubit<M>) = CNOT (c, q4) in
  let (c : qubit<M>, (q2 : qubit<M>, q3 : qubit<M>)) = FRED (c, (q2, q3)) in
  let (c : qubit<M>, (q1 : qubit<M>, q2 : qubit<M>)) = FRED (c, (q1, q2)) in
  let (c : qubit<M>, (q1 : qubit<M>, q4 : qubit<M>)) = FRED (c, (q1, q4)) in
  let res : five = (c, (((q1, q2), q3), q4)) in
  res

```

```
(* controlled multiply by 13 mod 15 *)
fun mult13 (cqs : five) : five =
  let (c : qubit<M>, qs : four_m) = cqs in
  let (((q1 : qubit<M>, q2 : qubit<M>), q3 : qubit<M>), q4 : qubit<M>) = qs in
  let (c : qubit<M>, q1 : qubit<M>) = CNOT (c, q1) in
  let (c : qubit<M>, q2 : qubit<M>) = CNOT (c, q2) in
  let (c : qubit<M>, q3 : qubit<M>) = CNOT (c, q3) in
  let (c : qubit<M>, q4 : qubit<M>) = CNOT (c, q4) in
  let (c : qubit<M>, (q1 : qubit<M>, q4 : qubit<M>)) = FRED (c, (q1, q4)) in
  let (c : qubit<M>, (q1 : qubit<M>, q2 : qubit<M>)) = FRED (c, (q1, q2)) in
  let (c : qubit<M>, (q2 : qubit<M>, q3 : qubit<M>)) = FRED (c, (q2, q3)) in
  let res : five = (c, (((q1, q2), q3), q4)) in
  res

fun z () : qubit<P> = qinit ()
fun o () : qubit<P> = H (qinit ())
fun main () : (qubit<P> * four_p) =
  let c = o () in
  (* 0b1001 = 9 *)
  let num : four_p = ((((o ()), z ()), z ()), o ()) in
  let (c : qubit<P>, rest : four_p) = mult13 (mult7 (entangle<P>(c, num))) in
  (* restored to 0b1001 *)
  (c, rest)
```

B.17 ModMul(4)-NotInverse

All other functions are identical to the *ModMul(4)* example.

```
fun mult13 (cqs : (qubit & (((qubit & qubit) & qubit) & qubit))<P>) :
  (qubit & (((qubit & qubit) & qubit) & qubit))<P> =
  let (c : qubit<M>, qs : (((qubit & qubit) & qubit) & qubit)<M>) = cqs in
  let (((q1 : qubit<M>, q2 : qubit<M>), q3 : qubit<M>), q4 : qubit<M>) = qs in
  let (c : qubit<M>, q1 : qubit<M>) = CNOT (c, q1) in
  let (c : qubit<M>, q2 : qubit<M>) = CNOT (c, q2) in
  let (c : qubit<M>, q3 : qubit<M>) = CNOT (c, q3) in
  let (c : qubit<M>, q4 : qubit<M>) = CNOT (c, q4) in
  let (c : qubit<M>, (q1 : qubit<M>, q4 : qubit<M>)) = FRED (c, (q1, q4)) in
  (* WRONG *)
  let (c : qubit<M>, (q1 : qubit<M>, q3 : qubit<M>)) = FRED (c, (q1, q3)) in
  let (c : qubit<M>, (q2 : qubit<M>, q3 : qubit<M>)) = FRED (c, (q2, q3)) in
  let res : (qubit & (((qubit & qubit) & qubit) & qubit))<P> =
    (c, (((q1, q2), q3), q4)) in
  res

fun main () : (qubit<P> * four_p) =
  let c = o () in
  let num : four_p = ((((o ()), z ()), z ()), o ()) in
  (* Dynamic separability check failure: argument entangled *)
  let (c : qubit<P>, rest : four_p) = mult13 (mult7 (entangle<P>(c, num))) in
  (c, rest)
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

H. Abraham et al. Qiskit: An open-source framework for quantum computing, 2019.

T. Altenkirch and J. Grattage. A functional quantum programming language. In *IEEE Symposium on Logic in Computer Science*, 2005. doi: 10.1109/LICS.2005.1.

M. Amy, M. Roetteler, and K. M. Svore. Verified compilation of space-efficient reversible circuits. In *Computer Aided Verification*, 2017.

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

G. Barthe, J. Hsu, M. Ying, N. Yu, and L. Zhou. Relational proofs for quantum programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019. doi: 10.1145/3371089.

G. Barthe, J. Hsu, and K. Liao. A probabilistic separation logic. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2020. doi: 10.1145/3371123.

J. S. Bell. On the einstein podolsky rosen paradox. *Physics*, 1, Nov 1964. doi: 10.1103/PhysicsPhysiqueFizika.1.195.

C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560, 2014. doi: 10.1016/j.tcs.2014.05.025.

C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70, Mar 1993. doi: 10.1103/PhysRevLett.70.1895.

B. Bichsel, M. Baader, T. Gehr, and M. Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020. doi: 10.1145/3385412.3386007.

R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Symposium on Principles of Programming Languages*, 2005. doi: 10.1145/1047659.1040327.

J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, 2003. doi: 10.1007/3-540-44898-5_4.

H. Buhrman, R. Cleve, J. Watrous, and R. de Wolf. Quantum fingerprinting. *Phys. Rev. Lett.*, 87, Sep 2001. doi: 10.1103/PhysRevLett.87.167902.

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

A. M. Childs, D. Maslov, Y. Nam, N. J. Ross, and Y. Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences*, 115(38), 2018. doi: 10.1073/pnas.1801723115.

P. Clairambault and M. de Visme. Full abstraction for the quantum lambda-calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019. doi: 10.1145/3371131.

D. Coppersmith. An approximate fourier transform useful in quantum factoring, 1994.

D. Deutsch. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London: Mathematical and Physical Sciences*, 439(1907), 1992. doi: 10.1098/rspa.1992.0167.

T. G. Draper. Addition on a quantum computer, 2000.

S. J. Gay, R. Nagarajan, and N. Papanikolaou. Qmc: A model checker for quantum systems. In *International Conference on Computer Aided Verification*, 2008. doi: 10.1007/978-3-540-70545-1_51.

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

O. Giraud. Distribution of bipartite entanglement for random pure states. *Journal of Physics A: Mathematical and Theoretical*, 40(11), Feb. 2007. doi: 10.1088/1751-8113/40/11/014.

G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numer. Math.*, 14(5), Apr. 1970. doi: 10.1007/BF02163027.

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

A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013. doi: 10.1145/2491956.2462177.

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

L. K. Grover. A fast quantum mechanical algorithm for database search. In *ACM Symposium on Theory of Computing*, 1996. doi: 10.1145/237814.237866.

L. Gurvits. Classical deterministic complexity of edmonds' problem and quantum entanglement. In *ACM Symposium on Theory of Computing*, 2003. doi: 10.1145/780542.780545.

A. Haar. Der massbegriff in der theorie der kontinuierlichen gruppen. *Annals of Mathematics*, 34, 1933. doi: 10.2307/1968346.

T. Häner, T. Hoefler, and M. Troyer. Assertion-based optimization of quantum programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2020. doi: 10.1145/3428201.

A. W. Harrow and A. Montanaro. Testing product states, quantum merlin-arthur games and tensor optimization. *J. ACM*, 60(1), Feb. 2013. doi: 10.1145/2432622.2432625.

P. Hayden, K. Milner, and M. M. Wilde. Two-message quantum interactive proofs and the quantum separability problem. *Quantum Info. Comput.*, 14(5-6), Apr. 2014. doi: 10.5555/2638661.2638663.

S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. In *Workshop on Formal Techniques for Java-Like Programs*, 2011. doi: 10.1145/2076674.2076675.

K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks. A verified optimizer for quantum circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2021. doi: 10.1145/3434318.

K. Honda. Analysis of quantum entanglement in quantum programs using stabilizer formalism. In *QPL*, 2015. doi: 10.4204/EPTCS.195.19.

M. Horodecki, P. Horodecki, and R. Horodecki. Separability of mixed states: necessary and sufficient conditions. *Physics Letters A*, 223(1-2), Nov 1996. doi: 10.1016/S0375-9601(96)00706-2.

Y. Huang and M. Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *International Symposium on Computer Architecture*, 2019. doi: 10.1145/3307650.3322213.

S.-H. Hung, K. Hietala, S. Zhu, M. Ying, M. Hicks, and X. Wu. Quantitative robustness analysis of quantum programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019. doi: 10.1145/3290344.

A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi. Scaffcc: A framework for compilation and analysis of quantum computing programs. In *ACM Conference on Computing Frontiers*, 2014. doi: 10.1145/2597917.2597939.

R. Jozsa and N. Linden. 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. doi: 10.1098/rspa.2002.1097.

I. Kassal, J. D. Whitfield, A. Perdomo-Ortiz, M.-H. Yung, and A. Aspuru-Guzik. Simulating chemistry using quantum computers. *Annual Review of Physical Chemistry*, 62(1), 2011. doi: 10.1146/annurev-physchem-032210-103512.

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

E. Knill. Conventions for quantum pseudocode, 1996.

A. Kumar, S. Haddadi, M. Pourkarimi, B. Behera, and P. Panigrahi. Experimental realization of controlled quantum teleportation of arbitrary qubit states via cluster states. *Scientific Reports*, 10, 08 2020. doi: 10.1038/s41598-020-70446-8.

G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie. Projection-based runtime assertions for testing and debugging quantum programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2020. doi: 10.1145/3428218.

J. Liu and H. Zhou. Systematic approaches for precise and approximate quantum state runtime assertion. In *IEEE International Symposium on High-Performance Computer Architecture*, 2021. doi: 10.1109/HPCA51647.2021.00025.

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

I. L. Markov and M. Saeedi. Constant-optimized quantum circuits for modular multiplication and exponentiation. *Quantum Info. Comput.*, 12(5-6), May 2012.

W. A. Miller, G. Kreymerman, C. Tison, P. Alsing, and J. McDonald. Quantum computing in a piece of glass. In *SPIE - The International Society for Optical Engineering*, 2011. doi: 10.1117/12.883332.

M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

A. Paradis, B. Bichsel, S. Steffen, and M. Vechev. Unqomp: Synthesizing uncomputation in quantum circuits. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021. doi: 10.1145/3453483.3454040.

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

J. Paykin, R. Rand, and S. Zdancewic. Qwire: A core language for quantum circuits. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017. doi: 10.1145/3009837.3009894.

S. Perdrix. Quantum patterns and types for entanglement and separability. In *International Workshop on Quantum Programming Languages*, 2005. doi: 10.1016/j.entcs.2006.12.015.

S. Perdrix. Quantum entanglement analysis based on abstract interpretation. In *International Symposium on Static Analysis*, 2008. doi: 10.1007/978-3-540-69166-2_18.

F. Prost and C. Zerrari. Reasoning about entanglement and separability in quantum higher-order functions. *Unconventional Computation*, 2009. doi: 10.1007/978-3-642-03745-0_25.

R. Rand, J. Paykin, D.-H. Lee, and S. Zdancewic. Reqwire: Reasoning about reversible quantum circuits. *Electronic Proceedings in Theoretical Computer Science*, 287, Jan 2019. doi: 10.4204/EPTCS.287.17.

R. Rand, A. Sundaram, K. Singhal, and B. Lackey. Extending gottesman types beyond the clifford group. In *Workshop on Programming Languages for Quantum Computing*, 2021a.

R. Rand, A. Sundaram, K. Singhal, and B. Lackey. Static analysis of quantum programs via gottesman types, 2021b.

M. Rennela and S. Staton. Classical control, quantum circuits and linear logic in enriched category theory. In *Conference on Mathematical Foundations of Programming Semantics*, 2017. doi: 10.23638/LMCS-16(1:30)2020.

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

E. Schmidt. Zur theorie der linearen und nichtlinearen integralgleichungen. *Mathematische Annalen*, 63, 1907. doi: 10.1007/BF01449770.

P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14, 08 2004. doi: 10.1017/S0960129504004256.

P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Typed Lambda Calculi and Applications*, 2005. doi: 10.1017/S0960129506005238.

Y. Shi, R. Tao, X. Li, A. Javadi-Abhari, A. W. Cross, F. T. Chong, and R. Gu. Certiq: A mostly-automated verification of a realistic quantum compiler, 2020.

P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5), Oct 1997. doi: 10.1137/S0097539795293172.

K. Singhal. Quantum hoare type theory, 2020.

K. Suenaga and N. Kobayashi. Fractional ownerships for safe memory deallocation. In *Asian Symposium on Programming Languages and Systems*, 2009. doi: 10.1007/978-3-642-10672-9_11.

K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Real World Domain Specific Languages Workshop*, 2018. doi: 10.1145/3183895.3183901.

R. Tao, Y. Shi, J. Yao, J. Hui, F. T. Chong, and R. Gu. Gleipnir: Toward practical error analysis for quantum programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021. doi: 10.1145/3453483.3454029.

D. Unruh. Quantum hoare logic with ghost variables. In *IEEE Symposium on Logic in Computer Science*, 2019. doi: 10.1109/LICS.2019.8785779.

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

S. P. Walborn, P. H. Souto Ribeiro, L. Davidovich, F. Mintert, and A. Buchleitner. Experimental determination of entanglement with a single measurement. *Nature*, 440(7087), Apr. 2006. doi: 10.1038/nature04627.

D. Wecker, K. M. Svore, and K. M. Svore. Liqui|>: A software design architecture and domain-specific language for quantum computing. February 2014.

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

M. Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016. ISBN 0128023066.

N. Yu and J. Palsberg. Quantum abstract interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021. doi: 10.1145/3453483.3454061.

L. Zhou, G. Barthe, J. Hsu, M. Ying, and N. Yu. A quantum interpretation of bunched logic amp; quantum separation logic. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2021. doi: 10.1109/LICS52264.2021.9470673.