```
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 ())
```

Fig. 16. Definitions of the four Bell states. The listing uses Twist syntax extensions described in Appendix C.

A ENTANGLEMENT FOR PURE AND MIXED STATES

In this work, we focus 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.

In this section, we briefly illustrate 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.

A.1 Definitions of Entanglement: Concurring Case

We present example programs that we analyze using the frameworks of pure- and mixed-state entanglement. First, we show a case where the pure- and mixed-state definitions align.

Figure 16 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 sub-state of the program state. For example, consider the program in Figure 17, in which we produce a qubit x in state $|1\rangle$ and two qubits y and z in the Bell state $|\Phi^+\rangle$. When we measure z, the whole program enters a mixed state: $|1\rangle_{\mathsf{x}} \otimes |0\rangle_{\mathsf{y}}$ with probability 1/2 and $|1\rangle_{\mathsf{x}} \otimes |1\rangle_{\mathsf{y}}$ 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: $|1\rangle_{\mathsf{x}}$.

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

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

The fact that the density matrix ρ is separable into density matrices for x and y, where the factor $|1\rangle\langle 1|_{\mathsf{x}}$ 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.

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

Fig. 17. A quantum program that produces a mixed state with a sub-state for the qubit x that is pure.

```
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 ()
```

Fig. 18. A quantum program that produces one of the four Bell states at random.

A.2 Definitions of Entanglement: Contrasting Case

We now examine a case where the mixed-state definition of entanglement yields a different conclusion from the pure-state definition. Figure 18 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. 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 we, for example, extract one qubit (and measure the other), 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, we can confirm:

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

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.

A.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 our reasoning about the purity of expressions. After all, we are interested in 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, we can recover the information of simple separability through pure-state reasoning. After all, assuming without loss of generality that $\rho_1 = |\phi\rangle\langle\phi|$ and $\rho_2 = \sum_j p_j |\psi_j\rangle\langle\psi_j|$, we have

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

which is equivalent to the statement 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.

B SEPARABILITY TESTING IN HARDWARE

In this section, we briefly discuss a potential implementation of separability testing on a hardware quantum computer. We utilize concepts from the density matrix formalism of quantum mechanics, including reduced density matrices and the quantity of purity. We refer the reader to Nielsen and Chuang [2010] for the technical background to this section.

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 $|\psi\rangle$, we divide it into subsystems A and B of dimensions d_A and d_B with reduced density matrices ρ_A and ρ_B.

First, we produce two copies $|\psi\rangle \otimes |\psi\rangle$ of the state under test. We swap the A subsystems of the two states, conditioned on an ancilla qubit, and apply the same procedure 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, we perform a measurement 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, we can express the test 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.

C ADDITIONAL SYNTACTIC FEATURES OF TWIST

In this section, we discuss syntactic features of Twist that enable writing more concise programs.

C.1 Affine Pure Types

As discussed, Twist allows pure expressions to be discarded and implicitly measured. Formally, we perform a syntactic translation to insert measurements of unused variables of type ϱ^{P} and discard the result of the measurement. Specifically, we translate functions and `let`-bindings:

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

when x has type ϱ^{P} and does not appear in e, and likewise for y if necessary. It is irrelevant when the implicit measurement takes place, and we do so before evaluating e. To discard a pair containing only pure and classical types, we recursively measure and discard its contents.

C.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:

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

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

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

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 $\tau = (\varrho_1 \mathbin{\&} \varrho_2)^{\pi}$ 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 $\tau' = (\varrho_1 \mathbin{\&} \varrho_2)^{\pi}$, 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 us to express 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
```

C.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 Section 8 to be more precise. We extend the syntax of purities to allow *variables*:

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

We require that every purity variable be introduced exactly once in the argument of the function in which it is used. We do not permit $\texttt{split}_{\pi}(e)$ for generic π because its operation fundamentally differs for **P** or **M**,[22] but we permit generics in `entangle` and `cast`, which are statically checked.

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 in the system. We also augment the type system to instantiate 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 following 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. A more sophisticated interprocedural static analysis might realize this fact, but a superior solution is to allow `f` to be polymorphic in the purity of `q`. We can annotate the argument to `f` 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)
```

The example now passes the static analysis and more clearly expresses the effect of `f` on purity.

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

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

Fig. 19. Definition of well-formed programs in μQ and Twist.

$$\text{V-FUN}\ \frac{}{f \text{ val}} \qquad \text{V-BOOL}\ \frac{}{b \text{ val}} \qquad \text{V-QREF}\ \frac{}{\mathtt{ref}[\alpha] \text{ val}} \qquad \text{V-PAIR}\ \frac{e_1 \text{ val} \qquad e_2 \text{ val}}{(e_1, e_2) \text{ val}}$$

Fig. 20. Definition of value judgment in μQ.

$$\text{S-APP}\ \frac{\phi(f) = \lambda x.e \qquad e' \text{ val}}{|\psi\rangle; f(e') \xmapsto{\emptyset}_1 |\psi\rangle; [e'/x]e} \qquad \text{S-APPL}\ \frac{|\psi\rangle; e_1 \xmapsto{O}_p |\psi'\rangle; e_1'}{|\psi\rangle; e_1(e_2) \xmapsto{O}_p |\psi'\rangle; e_1'(e_2)} \qquad \text{S-APPR}\ \frac{e_1 \text{ val} \qquad |\psi\rangle; e_2 \xmapsto{O}_p |\psi'\rangle; e_2'}{|\psi\rangle; e_1(e_2) \xmapsto{O}_p |\psi'\rangle; e_1(e_2')}$$

$$\text{S-PAIRL}\ \frac{|\psi\rangle; e_1 \xmapsto{O}_p |\psi'\rangle; e_1'}{|\psi\rangle; (e_1, e_2) \xmapsto{O}_p |\psi'\rangle; (e_1', e_2)} \qquad \text{S-PAIRR}\ \frac{e_1 \text{ val} \qquad |\psi\rangle; e_2 \xmapsto{O}_p |\psi'\rangle; e_2'}{|\psi\rangle; (e_1, e_2) \xmapsto{O}_p |\psi'\rangle; (e_1, e_2')}$$

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

$$\text{S-LETS}\ \frac{|\psi\rangle; e_1 \xmapsto{O}_p |\psi'\rangle; e_1'}{|\psi\rangle; \mathtt{let}\ (x, y) = e_1\ \mathtt{in}\ e_2 \xmapsto{O}_p |\psi'\rangle; \mathtt{let}\ (x, y) = e_1'\ \mathtt{in}\ e_2} \qquad \text{E-VAL}\ \frac{v \text{ val}}{|\psi\rangle; v \xmapsto{\emptyset}{}^*_1 |\psi\rangle; v}$$

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

Fig. 21. Operational semantics for classical constructs in μQ and Twist.

D FULL LANGUAGE SEMANTICS

D.1 μQ Language

Figure 19 defines the typing judgment for programs in μQ and Twist. Figure 20, Figure 21, and Figure 22 contain the full operational semantics of μQ.

D.2 Twist Language

Figure 23 contains the full type system of Twist. Figure 24, Figure 21, and Figure 25 contain the full operational semantics of Twist.

E FULL STATIC ANALYSIS TYPE SYSTEM

Figure 26 contains the full type system. There is no rule for q^π which is not written by the user.

F PROOFS OF SEMANTIC PROPERTIES

In this section, we provide proofs of progress, preservation, and purity soundness theorems for Twist.

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

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

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

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

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

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

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

S-MEASURES
$$\frac{|\psi\rangle; e \xmapsto{O}_p |\psi'\rangle; e'}{|\psi\rangle; \texttt{measure}(e) \xmapsto{O}_p |\psi'\rangle; \texttt{measure}(e')}$$

S-IFT
$$\frac{}{|\psi\rangle; \texttt{if T then } e_1 \texttt{ else } e_2 \xmapsto{\emptyset}_1 |\psi\rangle; e_1}$$

S-IFF
$$\frac{}{|\psi\rangle; \texttt{if F then } e_1 \texttt{ else } e_2 \xmapsto{\emptyset}_1 |\psi\rangle; e_2}$$

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

Fig. 22. Full operational semantics of quantum constructs in μQ.

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

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

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

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
$$\frac{\Gamma \vdash_\Delta e : \varrho^{\pi'}}{\Gamma \vdash_\Delta \texttt{cast}_\pi(e) : \varrho^\pi}$$

Fig. 23. Full type system of Twist.

V-FUN
$$\frac{}{f \ \texttt{val}}$$

V-BOOL
$$\frac{}{b \ \texttt{val}}$$

V-QVAL
$$\frac{}{q^\pi \ \texttt{val}}$$

V-PAIR
$$\frac{e_1 \ \texttt{val} \qquad e_2 \ \texttt{val}}{(e_1, e_2) \ \texttt{val}}$$

Fig. 24. Definition of value judgment in Twist.

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

S-U1
$$\frac{}{|\psi\rangle;U(\texttt{ref}[\alpha]^\pi) \xmapsto{\emptyset}_1 U_\alpha\,|\psi\rangle;\texttt{ref}[\alpha]^\pi}$$

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

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

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

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

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

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

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

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

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

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

S-EntangleS
$$\frac{|\psi\rangle;e \xmapsto{O}_p |\psi'\rangle;e'}{|\psi\rangle;\texttt{entangle}_\pi(e) \xmapsto{O}_p |\psi'\rangle;\texttt{entangle}_\pi(e')}$$

S-SplitS
$$\frac{|\psi\rangle;e \xmapsto{O}_p |\psi'\rangle;e'}{|\psi\rangle;\texttt{split}_\pi(e) \xmapsto{O}_p |\psi'\rangle;\texttt{split}_\pi(e')}$$

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

S-SplitPure
$$\frac{\begin{array}{c}|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_0\rangle \\ \mathrm{dom}\,|\psi_1\rangle = \mathsf{Refs}(q_1) \qquad \mathrm{dom}\,|\psi_2\rangle = \mathsf{Refs}(q_2)\end{array}}{|\psi\rangle;\texttt{split}_{\mathbf{P}}([q_1,q_2]^{\mathbf{P}}) \xmapsto{\emptyset}_1 |\psi\rangle;({q_1}^{\mathbf{P}},{q_2}^{\mathbf{P}})}$$

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

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

Fig. 25. Operational semantics of quantum constructs in Twist.

F.1 Twist Language

F.1.1 Progress (Theorem 7.1). First, we state canonical forms lemmas for values:

Lemma F.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 F.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 $\vdash_\Delta q : \varrho$. □

$$\textsc{A-Var}\ \frac{}{x : \tau \vdash_A x : \tau} \qquad \textsc{A-Fun}\ \frac{\Gamma, x : \tau_1 \vdash_A e : \tau_2}{\Gamma \vdash_A \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{A-App}\ \frac{\Gamma_1 \vdash_A e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash_A e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_A e_1(e_2) : \tau_2} \qquad \textsc{A-Pair}\ \frac{\Gamma_1 \vdash_A e_1 : \tau_1 \qquad \Gamma_2 \vdash_A e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_A (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\textsc{A-Let}\ \frac{\Gamma_2 \vdash_A e : \tau_1 \times \tau_2 \qquad \Gamma_1, x : \tau_1, y : \tau_2 \vdash_A e' : \tau}{\Gamma_1, \Gamma_2 \vdash_A \texttt{let}\ (x, y) = e\ \texttt{in}\ e' : \tau} \qquad \textsc{A-If}\ \frac{\Gamma_1 \vdash_A e : \texttt{bool} \quad \Gamma_2 \vdash_A e_1 : \varrho^f \quad \Gamma_2 \vdash_A e_2 : \varrho^g}{\Gamma_1, \Gamma_2 \vdash_A \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \varrho^{\mathrm{M}}} \qquad \textsc{A-Bool}\ \frac{}{\cdot \vdash_A b : \texttt{bool}}$$

$$\textsc{A-Qinit}\ \frac{}{\cdot \vdash_A \texttt{qinit}\ () : \texttt{qubit}^{\mathrm{P}}} \qquad \textsc{A-U1}\ \frac{\Gamma \vdash_A e : \texttt{qubit}^f}{\Gamma \vdash_A U(e) : \texttt{qubit}^f} \qquad \textsc{A-U2}\ \frac{\Gamma \vdash_A e : (\texttt{qubit} \,\&\, \texttt{qubit})^f}{\Gamma \vdash_A U_2(e) : (\texttt{qubit} \,\&\, \texttt{qubit})^f} \qquad \textsc{A-Measure}\ \frac{\Gamma \vdash_A e : \texttt{qubit}^f}{\Gamma \vdash_A \texttt{measure}(e) : \texttt{bool}}$$

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

$$\textsc{A-SplitPure}\ \frac{\Gamma \vdash_A e : (\varrho_1 \,\&\, \varrho_2)^{\mathrm{P}}}{\Gamma \vdash_A \texttt{split}_{\mathbf{P}}(e) : {\varrho_1}^{\mathrm{P}} \times {\varrho_2}^{\mathrm{P}}} \qquad \textsc{A-CastMixed}\ \frac{\Gamma \vdash_A e : \varrho^f}{\Gamma \vdash_A \texttt{cast}_{\mathbf{M}}(e) : \varrho^f} \qquad \textsc{A-CastPure}\ \frac{\Gamma \vdash_A e : \varrho^{\mathrm{P}}}{\Gamma \vdash_A \texttt{cast}_{\mathbf{P}}(e) : \varrho^{\mathrm{P}}}$$

Fig. 26. Full static analysis type system.

Next, we prove the main theorem:

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 we have 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 we have $|\psi\rangle; e \mapsto \cdot$ 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 we have $|\psi\rangle; e \mapsto \cdot$ or e val. In the former case, apply S-CastS. In the latter case, apply S-Cast. □

F.1.2 Preservation (Theorem 7.4). In the following, we use $|\psi'\rangle; q$ pure as shorthand for $|\psi'\rangle; q^{\mathrm{P}}$ pure. We first state a helpful lemma. 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 F.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, we prove the main theorem:

Proof. Proceed by induction on the derivation of $|\psi\rangle; e \mapsto |\psi'\rangle; e'$. In each case, to show that $|\psi'\rangle \vDash e'$, we show 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 $\Gamma_1, \Gamma_2 \vdash_{\Delta_1,\Delta_2} (\lambda x.e)(e') : \tau$ we have that $\Gamma_1, x : \tau' \vdash_{\Delta_1} e : \tau$ and $\Gamma_2 \vdash_{\Delta_2} e' : \tau'$. Thus we have $\Gamma_1, \Gamma_2 \vdash_{\Delta_1,\Delta_2} [e'/x]e : \tau$.

In case S-Let, by inversion of typing we have 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 we have $\Gamma, \Gamma_1, \Gamma_2 \vdash_{\Delta,\Delta_1,\Delta_2} [e_1, e_2/x, y]e : \tau$.

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 $\Gamma \vdash_{\Delta_1,\Delta_2} \texttt{split}_{\mathbf{M}}([q_1, q_2]^{\mathbf{M}}) : (\varrho_1 \,\&\, \varrho_2)^{\mathbf{M}}$, we have 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 $\Gamma \vdash_{\Delta_1,\Delta_2} \texttt{entangle}_\pi({q_1}^\pi, {q_2}^\pi) : (\varrho_1 \,\&\, \varrho_2)^\pi$ we have $\vdash_{\Delta_1} q_1 : \varrho_1$ 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 we know that $|\psi\rangle; q_1$ pure and $|\psi\rangle; q_2$ pure, meaning $|\psi\rangle; [q_1, q_2]$ pure and we have $|\psi\rangle \vDash [q_1, q_2]^{\mathbf{P}}$.

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

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

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

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

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

In case S-SplitPure, by inversion of $\Gamma \vdash_{\Delta_1,\Delta_2} \texttt{split}_{\mathbf{P}}([q_1, q_2]^{\mathbf{P}}) : {\varrho_1}^{\mathbf{P}} \times {\varrho_2}^{\mathbf{P}}$ we have 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. □

F.1.3 Purity Soundness (Theorem 7.5). First, we state two properties of the language that hold as a consequence of it being a variant of the linear simply-typed λ-calculus with type safety.

Lemma F.4 (Strong Normalization). *If $\cdot \vdash_\Delta e : \tau$ then for all $|\psi\rangle$ such that $\Delta \subseteq \operatorname{dom} |\psi\rangle$, 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 F.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, $|\psi\rangle; \texttt{let } (x, y) = (e_1, e_2) \texttt{ in } e \mapsto^* |\psi'\rangle; v$ 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 we operate in a linear λ-calculus where 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.

We now give a proof for the purity soundness theorem by logical relations, strengthening the induction hypothesis to describe types other than $\varrho^{\mathbf{P}}$ and typing judgments that involve non-empty contexts. For our relation, we define the notion of purity at a type τ, denoted $|\psi\rangle; e\ \mathsf{pure}_\tau$.

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

Purity at type $\varrho^{\mathbf{P}}$ 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. We represent this 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 F.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, we define the standard notion of a substitution γ mapping 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 we state the strengthened purity soundness theorem. When τ is $\varrho^{\mathbf{P}}$ and Γ is empty, the strengthening implies the original theorem.

Theorem F.7. *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 $|\psi\rangle \vDash e$ we have $|\psi\rangle; e\ \mathsf{pure}$. 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' 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 $|\psi\rangle; \gamma(e)$ pure we have that the execution up to $|\psi'\rangle$ is unique. By preservation, $U_\alpha |\psi'\rangle \vDash \mathtt{ref}[\alpha]^{\mathbf{P}}$ so $U_\alpha |\psi'\rangle; \mathtt{ref}[\alpha]^{\mathbf{P}}$ pure. By Lemma F.6, we have $|\psi'\rangle; U(\mathtt{ref}[\alpha]^{\mathbf{P}})$ pure. Stitching the two deterministic executions together, we conclude that $|\psi\rangle; \gamma(U(e))$ pure.

Case T-U2 follows by the same reasoning as for T-U1, with the only difference being that we execute a two-qubit operator $U_{\alpha,\beta}$ 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 $|\psi\rangle; \gamma(\texttt{entangle}_\pi(e_1, e_2))$ pure.

In case T-Abs, we have that $x : \tau \vdash_\Delta \gamma(e) : \tau'$. By the IH we have that if $|\psi\rangle; e'$ pure_τ then $|\psi\rangle; [e'/x]\gamma(e)$ $\text{pure}_{\tau'}$. From this we have $|\psi\rangle; (\lambda x.\gamma(e))(e')$ $\text{pure}_{\tau'}$ by Lemma F.5. We also have that $|\psi\rangle; \gamma(\lambda x.e)$ $\text{pure}_{\tau \to \tau'}$.

In case T-Pair, we have that $\cdot \vdash_{\Delta_1} \gamma(e_1) : \tau_1$ and $\cdot \vdash_{\Delta_2} \gamma(e_2) : \tau_2$. By the IH we have that $|\psi\rangle; \gamma(e_1)$ pure_{τ_1} and $|\psi\rangle; \gamma(e_2)$ pure_{τ_2}. Thus, if we assume $|\psi\rangle; [\gamma(e_1), \gamma(e_2)/x, y]e$ pure_τ, then we have $|\psi\rangle; \texttt{let}\ (x, y) = (\gamma(e_1), \gamma(e_2))\ \texttt{in}\ e$ pure_τ by Lemma F.5, and so $|\psi\rangle; (\gamma(e_1), \gamma(e_2))$ $\text{pure}_{\tau_1 \times \tau_2}$.

In case T-App, we have 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 we have $|\psi\rangle; \gamma(e_1)$ $\text{pure}_{\tau_1 \to \tau_2}$ and $|\psi\rangle; \gamma(e_2)$ pure_{τ_1}, thus $|\psi\rangle; \gamma(e_1(e_2))$ pure_{τ_2}.

In case T-Let, we have 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 we have $|\psi\rangle; \gamma(e)$ $\text{pure}_{\tau_1 \times \tau_2}$ 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, we have $|\psi\rangle; \gamma(\texttt{let}\ (x, y) = e\ \texttt{in}\ e')$ pure_τ.

In case T-SplitPure, we have $\cdot \vdash_\Delta \gamma(e) : (\varrho_1 \,\&\, \varrho_2)^{\mathbf{P}}$ and need to show that $|\psi\rangle; \texttt{split}_{\mathbf{P}}(\gamma(e))$ $\text{pure}_{\varrho_1{}^{\mathbf{P}} \times \varrho_2{}^{\mathbf{P}}}$. By the IH we have 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 $|\psi'\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_0\rangle$, $\text{dom}\,|\psi_1\rangle = \text{Refs}(q_1)$, $\text{dom}\,|\psi_2\rangle = \text{Refs}(q_2)$, implying that $|\psi'\rangle; q_1$ pure and $|\psi'\rangle; q_2$ pure. Thus, if we assume $|\psi'\rangle; [q_1{}^{\mathbf{P}}, q_2{}^{\mathbf{P}}/x, y]e'$ pure_τ, from Lemma F.5 we have that $|\psi'\rangle; \texttt{let}\ (x, y) = (q_1{}^{\mathbf{P}}, q_2{}^{\mathbf{P}})\ \texttt{in}\ e'$ pure_τ.

Now assume that the separability condition fails. Then, every evaluation of $\texttt{split}_{\mathbf{P}}(\gamma(e))$ 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. □

F.2 Static Analysis for Purity

F.2.1 Purity Soundness (Theorem 8.2). We have already shown that all **P** annotations introduced by the semantics of Twist without $\texttt{cast}_{\mathbf{P}}$ are on pure expressions. We next 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 **P** annotations on pure expressions, implying the soundness theorem.

Proof. Proceed by induction on the derivation of $\Gamma \vdash_A e : \varrho^{\mathbf{P}}$. We only need to examine new **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, $g = \text{Split}(f, j)$ is never **P**, so purity is trivially satisfied.

In case A-Entangle, if h is not **P** then purity is trivial. In addition, if f and g are both **P** then the same reasoning as in the original proof applies. If f and g are not **P** but h is pure, we must 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 $h = \text{Combine}(f, g)$, contradicting the fact that h is **P**. The second possibility is that the evaluation of $\texttt{entangle}_\pi(e_1, e_2)$ encounters an $\texttt{if}$-expression that evaluates to a mixed state that is not discarded. But then this $\texttt{if}$-expression appears in the evaluation of e_1 or e_2, meaning by preservation that either f or g is **M** and thus $h = \text{Combine}(f, g)$ is **M**, which is also a contradiction.

All other rules follow by the same reasoning as in the proof of Theorem F.7. □

G FULL BENCHMARK DESCRIPTIONS

Teleport-Deferred. This program implements the deferred-measurement teleportation from Section 3. We also implemented 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. We also implement 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. We also implemented 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). We also implemented 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 $|11\rangle$. We also implemented 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. The purity specification of the QFT 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. We also implemented 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). We followed the scheme of Markov and Saeedi [2012] for quantum circuits for modular multiplication, which generalizes a predictable structural pattern for arbitrary n, with the number of gates linear in n. For each n, we implemented conditional multiplication mod $2^n - 1$ by some k and also by k^{-1}. 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. We also implemented an erroneous variant *ModMul(n)-NotInverse* where multiplication by k^{-1} is defective, resulting in a register and condition that are still entangled.

G.1 Analysis Result Descriptions

Teleport-Deferred. We determine that the teleportation circuit satisfies its purity specification, taking a pure qubit to another pure qubit. We also determine 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. We determine that the oracle correctly uncomputes ancillas and yields a pure output. We also determine that the variant that does not uncompute an ancilla may yield a mixed output, violating its specification.

Bell-GHZ. We determine 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. We determine that the algorithm satisfies its specification, including the fact that an ancilla is separable from the output and can be safely dropped. We also determine that the erroneous variant results in an entangled ancilla and rejects the attempt to drop it at runtime.

DeutschJozsa. We determine that the algorithm satisfies its specification. For the erroneous variant with a defective initial state, we determine 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. We determine 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. We determine 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. We determine 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.

ModMul(n). We determine 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.

H FULL BENCHMARK PROGRAMS

In this section, we present 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 Appendix C.

H.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 ()))
```

H.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
```

H.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 *)
```

H.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 ()))
```

H.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 ()))
```

H.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 *)
```

H.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))
```

H.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))
```

H.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)
```

H.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)
```

H.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)
```

H.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 *)
```

H.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)
```

H.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))
```

H.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)
```

H.16 ModMul(4)

We next show 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)
```

H.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
  let (c : qubit<M>, (q1 : qubit<M>, q3 : qubit<M>)) = FRED (c, (q1, q3)) in (* WRONG *)
  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)
```