Qunity

A Unified Language for Quantum and Classical Computing

FINN VOICHICK, University of Maryland, USA
LIYI LI, University of Maryland, USA
ROBERT RAND, University of Chicago, USA
MICHAEL HICKS*, University of Maryland and Amazon, USA

We introduce Qunity, a new quantum programming language designed to treat quantum computing as a natural generalization of classical computing. Qunity presents a unified syntax where familiar programming constructs can have both quantum and classical effects. For example, one can use sum types to implement the direct sum of linear operators, exception-handling syntax to implement projective measurements, and aliasing to induce entanglement. Further, Qunity takes advantage of the overlooked BQP subroutine theorem, allowing one to construct reversible subroutines from irreversible quantum algorithms through the uncomputation of "garbage" outputs. Unlike existing languages that enable quantum aspects with separate add-ons (like a classical language with quantum gates bolted on), Qunity provides a unified syntax and a novel denotational semantics that guarantees that programs are quantum mechanically valid. We present Qunity's syntax, type system, and denotational semantics, showing how it can cleanly express several quantum algorithms. We also detail how Qunity can be compiled into a low-level qubit circuit language like OpenQASM, proving the realizability of our design.

CCS Concepts: • **Software and its engineering** → **Multiparadigm languages**; *Syntax*; *Control structures*; Data types and structures; Compilers; Functional languages; Patterns; Semantics; • **Theory of computation** → *Denotational semantics*; *Quantum information theory*; Control primitives; Categorical semantics; Quantum query complexity; Assertions.

Additional Key Words and Phrases: algebraic data types, Kraus operators, quantum subroutines, reversible computing

ACM Reference Format:
Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 32 (January 2023), 31 pages. https://doi.org/10.1145/3571225

1 INTRODUCTION

Quantum computing *generalizes* classical computing. That is, any efficiently-implementable classical algorithm can also be efficiently implemented on a quantum computer. However, quantum programming languages today do not fully leverage this connection. Rather, to varying degrees, they impose a separation between their classical and quantum programming constructs. Such a separation owes in part to the QRAM computing model [Knill 1996], and is reflected in slogans

*Work completed before starting at Amazon.

Authors' addresses: Finn Voichick, University of Maryland, College Park, USA, finn@umd.edu; Liyi Li, University of Maryland, College Park, USA, liyili2@umd.edu; Robert Rand, University of Chicago, Chicago, USA, rand@uchicago.edu; Michael Hicks, University of Maryland and Amazon, College Park, USA, mwh@cs.umd.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/1-ART32
https://doi.org/10.1145/3571225

such as "quantum data, classical control" [Selinger 2004]. While keeping the quantum and classical separated in the language makes some sense in the near term (the "NISQ" era [Preskill 2018]), it artificially limits the long-term potential of quantum algorithm designs. It also fails to take advantage of a classical programmer's intuition and limits the reuse of classical code and ideas in quantum algorithms.

In this paper, we present *Qunity* ("KYOO-nih-tee"), a new programming language whose programming constructs will be familiar to classical programmers but are generalized to include quantum behavior. Thus, Qunity aims to *unify* quantum and classical concepts in a single language. Qunity draws inspiration from prior languages which contain some unified constructs [Altenkirch and Grattage 2005; Bichsel et al. 2020], but Qunity broadens and deepens that unification.

1.1 Motivating Example: Deutsch's Algorithm

To give a sense of Qunity's design, we present Deutsch's algorithm [Deutsch 1985]. Given black-box access to a function $f : \{0, 1\} \to \{0, 1\}$, this algorithm computes whether or not $f(0) \overset{?}{=} f(1)$ using only a single query to the function.

$$\begin{aligned}
&\texttt{deutsch}(f) \coloneqq \\
&\texttt{let } x =_{\texttt{Bit}} (\mathsf{had}\ \texttt{0}) \texttt{ in} \\
&\left(\texttt{ctrl }(f\ x)_{\texttt{Bit}} \begin{Bmatrix} \texttt{0} \mapsto x \\ \texttt{1} \mapsto x \triangleright \mathsf{gphase}_{\texttt{Bit}}(\pi) \end{Bmatrix}_{\texttt{Bit}}\right) \triangleright \mathsf{had}
\end{aligned}$$

The algorithm has three steps:

(1) Apply a Hadamard operator had to a qubit in the zero state, yielding a qubit in state $|+\rangle$; the `let` expression binds this qubit to x.
(2) Query an oracle to conditionally flip the phase of the qubit, coherently performing the linear map:
$$|x\rangle \mapsto (-1)^{f(x)} |x\rangle$$
This step is implemented by the `ctrl` expression: if $f x$ is `0` then $|x\rangle$ is unchanged, but if $f x$ is 1 then $x \triangleright \mathsf{gphase}_{\texttt{Bit}}(\pi)$ applies a phase of $e^{i\pi}$ (which is -1) to $|x\rangle$.
(3) Finally, apply a Hadamard operator to the qubit output from step 2.

If $f(0) \neq f(1)$, the output will be $|1\rangle$ up to global phase; otherwise, it will be $|0\rangle$.

Qunity's version of Deutsch's algorithm reads like a typical functional program and is more general than typically presented versions. Some presentations [Nielsen and Chuang 2010] require constructing a two-qubit unitary oracle U_f from the classical function f such that $U_f |x, y\rangle = |x, y \oplus f(x)\rangle$ for all $x, y \in \{0, 1\}$. In Qunity, no separate oracle is needed; the "oracle" is f itself. Existing programming languages [Amy et al. 2017; Li et al. 2022; Rand et al. 2019] support automatic construction of oracles U_f from explicitly-written programs f, but these require that the implementation of f is strictly classical. For example, Silq's [Bichsel et al. 2020] uncomputation construct requires that the program of interest be "qfree," with strictly classical behavior, and Quipper [Green et al. 2013] supports automatic oracle construction only from classical Haskell programs. In Qunity, f can be an *arbitrary* quantum algorithm that takes a qubit as input and produces a qubit as output, even if it uses measurement or interspersed classical operations.

Qunity takes advantage of the BQP subroutine theorem [Bennett et al. 1997; Watrous 2009], which allows for the construction of reversible subroutines from arbitrary (not necessarily reversible) quantum algorithms. If the quantum algorithm has probabilistic behavior, the reversible subroutine may have a degree of error, but the BQP subroutine theorem places reasonable bounds on this error.

These bounded-error subroutines are commonly used in the design of quantum algorithms [Kothari 2014], but existing programming languages provide no convenient way to construct and compose them, a gap that Qunity fills.

According to Qunity's type system (Section 3), the following typing judgment is valid:

$$\frac{\vdash f : \texttt{Bit} \Rrightarrow \texttt{Bit}}{\varnothing \parallel \varnothing \vdash \texttt{deutsch}(f) : \texttt{Bit}}$$

The rule says: "given an arbitrary quantum algorithm for computing a function $f : \{0, 1\} \to \{0, 1\}$, our `deutsch`$(f)$ program outputs a bit." Per Qunity's formal semantics (Section 4), `deutsch`(f) corresponds to a single-qubit pure state whenever f corresponds to a single-qubit quantum channel. This is possible because of the unique way that Qunity's semantics interweaves the usage of pure and mixed quantum states.

1.2 Design Principles

Qunity's design is motivated by four key principles: generalization of classical constructs, expressiveness, compositionality, and realizability.

Generalization of classical constructs. To make quantum computing easier for programmers, Qunity allows them to draw on intuition from classical computing. Many elements of Qunity's syntax are simply quantum generalizations of classical constructs: for example, tensor products generalize pairs, projective measurements generalize `try-catch`, and quantum control generalizes pattern matching. Rather than use linear types (as in Qwire [Paykin et al. 2017] and the Proto-Quipper languages [Fu et al. 2020; Ross 2015]), Qunity allows variables to be freely duplicated and discarded as in classical languages, but its semantics treats variable duplication as an entangling operation, and variable discarding as a partial trace.

Expressiveness. Qunity allows for writing algorithms at a higher level of abstraction than existing languages. One way that it does this is through algebraic data types: rather than manipulate fixed-length arrays of qubits directly, programmers can work with more complicated types. For example, in our quantum walk algorithm given in Section 5.3, we deal with superpositions of variable-length lists. Qunity's semantics also allows one to "implement" mathematical objects that are frequently used in algorithm analysis but seldom used in algorithm implementation. For example, the semantics of a Qunity program can be a superoperator [Kaye et al. 2007, p. 57], an isometry [Roman 2008, p. 210], or a projector [Nielsen and Chuang 2010, p. 70], and these can be composed in useful ways.

In detail, the operators that make up Qunity's semantics are drawn from a broad class called *Kraus operators* [Kaye et al. 2007, p. 60], which include norm-decreasing operators such as projectors. Projectors are used in quantum algorithms, but more often for *analyzing* quantum algorithms, and few quantum languages allow projectors to be directly implemented. Motivated by the fact that operators produced by the BQP subroutine theorem can be viewed as norm-*decreasing* rather than norm-preserving, we give Qunity programs norm non-increasing semantics. Expanding the language to include projectors turns out to be quite useful: We can treat the null space of a projector as a sort of "exception space," allowing us to reason about quantum projectors by a syntactic analogy with classical programming strategies, namely exception handling. This allows us to implement a few more useful program transformations, like a "reflection" in the style of Grover's diffusion operator. Given the ability to implement a projector P, it is straightforward to implement the unitary reflection $(2P - I)$, a common feature in quantum algorithms.

Compositionality. Qunity programs can be composed in useful ways. For example, our `ctrl` construct uses the BQP subroutine theorem [Bennett et al. 1997; Watrous 2009] to reversibly use an irreversible quantum algorithm as a condition for executing another—this happens in step 2 of our version of Deutsch's algorithm. In general, Qunity's denotational semantics allows composing programs that are mathematically described in different ways, like defining an expression's "pure" operator semantics in terms of a subexpression's "mixed" superoperator semantics. By contrast, prior languages have not managed such a compositional semantics [Grattage 2011, 2006; Ying 2016], as discussed below.

Realizability. We have designed Qunity so that it can be compiled into qubit-based unitary circuits written in a lower-level language such as OpenQasm. We have designed such a compilation procedure and proven that the circuits it produces correctly implement Qunity's denotational semantics. In Section 6, we give an overview of our compilation strategy, and we include the full details and proofs in the supplemental report [Voichick et al. 2022a]. To limit our analysis to finite-dimensional Hilbert spaces, we limit Qunity programs to working with finite types only. Some quantum languages [Ying 2016, Chapter 7] work with infinite-dimensional Hilbert spaces such as Fock spaces, making it possible to coherently manipulate superpositions of *unbounded* lists, for example. However, since qubit-based circuits work only with finite-dimensional Hilbert spaces, allowing Qunity to work with infinite data types would mean that compilation would have to approximate infinite-dimensional Hilbert spaces with finite ones, which is challenging to do satisfyingly. For this reason, Qunity has no notion of recursive types, though we use classical metaprogramming to define parameterized types.

1.3 Related Work

Table 1. Comparison with existing languages

	Language					
Feature	Qunity	Silq	QML	SPM	Quipper	QuGCL
Decoherence	✓	✓	✓	✗	✓	✓
Denotational Semantics	✓	✗	✓*	✗	✗	✓*
Quantum Sum Types	✓	✗	✓	✓	✗	✗
Isometries	✓	✓	✓*	✗	✓*	✗
Projectors	✓	✗	✗	✗	✗	✓
Classical Uncomputation	✓	✓	✗	✗	✓	✗
Quantum Uncomputation	✓	✗	✗	✗	✗	✗

Qunity's design was inspired by several existing languages, summarized in Table 1: Silq [Bichsel et al. 2020], QML [Altenkirch and Grattage 2005], the symmetric pattern matching language [Sabry et al. 2018] (hereafter SPM), Quipper [Green et al. 2013], and QuGCL [Ying 2016, Chapter 6]. This table shows some of Qunity's most interesting features and whether these features are present in existing languages. The features listed are:

- Decoherence. All of these languages support some form of measurement or discarding, except for SPM, which is restricted to unitary operators.
- Compositional denotational semantics. Most of these languages define an operational semantics, not a denotational one. QuGCL's denotational semantics is explicitly *non*-compositional,

meaning that its equivalence relation does not allow "equivalent" programs to be substituted as subexpressions. Later work on Qml [Grattage 2011, 2006] found that the denotational semantics derived from Qml's operational semantics was non-compositional because of entangled "garbage" outputs related to sum types, and Qml's denotational semantics is only partially defined even with sum types removed.

- Quantum sum types. Like Qml and Spm, Qunity can coherently manipulate tagged unions, and the "tag" qubit can be in superposition.
- Isometries. Silq is the only other language here with a semantics defined in terms of non-unitary isometries. Quipper and Qml allow isometries to be implemented through qubit initialization, but their semantics are defined only in terms of unitaries, meaning there is no notion of equivalence between different unitaries implementing the same isometry.
- Projectors. Like with isometries, Quipper allows projectors to be implemented through assertative termination, but there is no notion of equivalence between different unitaries implementing the same projector.
- Classical uncomputation. Silq and Quipper both have convenient facilities for converting (irreversible) classical programs into reversible quantum subroutines.
- Quantum uncomputation. Silq can only uncompute "qfree" programs that are strictly classical. Quipper does have some facilities for uncomputing values produced by quantum programs, such as the `with_computed` function. However, this function's correctness depends on conditions that are nontrivial to verify. In the words of Quipper's documentation: "This is a very general but relatively unsafe operation. It is the user's responsibility to ensure that the computation can indeed be undone." In Qunity, safety is assured.

1.4 Contributions and Roadmap

Our core contribution is a new quantum programming language, Qunity, designed to unify classical and quantum computing through an expressive generalization of classical programming constructs. Qunity's powerful semantics brings constructions commonly used in algorithm *analysis*—such as bounded-error quantum subroutines, projectors, and direct sums—into the realm of algorithm *implementation*. We describe Qunity's formal syntax (Section 2), an efficiently checkable typing relation (Section 3), a compositional denotational semantics (Section 4), and a strategy for compiling to lower-level unitary quantum circuits (Section 6). We prove that well-typed Qunity programs have a well-defined semantic denotation, and we show that denotation is realizable by proving that our compilation strategy does indeed produce correct circuits. We also show how Qunity can be used to program several interesting examples, including Grover's algorithm, the quantum Fourier transform, and a quantum walk (Section 5).

2 SYNTAX

Qunity's formal syntax is defined in Figures 1–3. Qunity's types are shown in Figure 1. The algebraic *data* types T have essentially the same interpretation as in a typical classical programming language, with the caveat that values can be in superposition. The symbols $\oplus$ and $\otimes$ are used because of how these data types will correspond to direct sums and tensor products. The two *program* types F differ in whether they decohere quantum states: programs of type $T \rightsquigarrow T'$ will have a semantics defined by an (often unitary) linear operator, while programs of type $T \Rrightarrow T'$ will have a semantics defined by a superoperator, a completely positive trace-non-increasing map that may involve measurement or discarding.

Qunity's term language is defined in Figure 3: *expressions* e are assigned data types T and *programs* f are assigned program types F. Figure 4 shows additional derived forms. In the figure, x ranges over some infinite set $\mathbb{X}$ of variables (for example Ascii strings), and r ranges over some representation

$$
\begin{array}{rlr}
T ::= & & \textit{(data type)} \\
 & \mathtt{Void} & \textit{(bottom)} \\
\mid & \mathtt{()} & \textit{(unit)} \\
\mid & T \oplus T & \textit{(sum)} \\
\mid & T \otimes T & \textit{(product)} \\
F ::= & & \textit{(program type)} \\
 & T \rightsquigarrow T & \textit{(coherent map)} \\
\mid & T \Rrightarrow T & \textit{(quantum channel)}
\end{array}
$$

Fig. 1. Qunity types

$$
\begin{array}{rlr}
\Gamma ::= & & \textit{(context)} \\
 & \varnothing & \textit{(empty)} \\
 & \mid \Gamma, x : T & \textit{(binding)} \\
\mathrm{dom}(\varnothing) \coloneqq & \varnothing & \textit{(dom-none)} \\
\mathrm{dom}(\Gamma, x : T) \coloneqq & \mathrm{dom}(\Gamma) \cup \{x\} & \textit{(dom-bind)}
\end{array}
$$

Fig. 2. Typing contexts

$$
\begin{array}{llr}
e ::= & & \textit{(expression)} \\
 & \mathtt{()} & \textit{(unit)} \\
 & \mid x & \textit{(variable)} \\
 & \mid \mathtt{(}e\mathtt{,}e\mathtt{)} & \textit{(pair)} \\
 & \mid \mathtt{ctrl}\ e\ {}_{T}\!\left\{\begin{array}{c} e \mapsto e \\ \cdots \\ e \mapsto e \end{array}\right\}_{T} & \textit{(coherent control)} \\
 & \mid \mathtt{try}\ e\ \mathtt{catch}\ e & \textit{(error recovery)} \\
 & \mid f\ e & \textit{(application)} \\
f ::= & & \textit{(program)} \\
 & \mathtt{u_3(}r\mathtt{,}r\mathtt{,}r\mathtt{)} & \textit{(qubit gate)} \\
 & \mid \mathtt{left}_{T \oplus T} & \textit{(left tag)} \\
 & \mid \mathtt{right}_{T \oplus T} & \textit{(right tag)} \\
 & \mid \lambda e \overset{T}{\mapsto} e & \textit{(abstraction)} \\
 & \mid \mathtt{rphase}_{T} \left\{\begin{array}{r} e \mapsto r \\ \mathtt{else} \mapsto r \end{array}\right\} & \textit{(relative phase)}
\end{array}
$$

Fig. 3. Base Qunity syntax

of real numbers (one example of which is defined in the supplemental report [Voichick et al. 2022a]). Some syntax elements use type annotations T, which are grayed out to reduce visual clutter in the more complex examples. A type inference algorithm could allow these annotations could to be removed, and throughout this paper, we occasionally omit type annotations for brevity.

Qunity makes use of standard classical programming language features generalized to the quantum setting. As examples, pairs generalize to creating tensor products of quantum states, sums generalize to allowing their data to be in superposition, and `ctrl` generalizes classical pattern-matching (where in branch $e \mapsto e'$ the e is a pattern which may bind variables in e') to *quantum control flow* using superposition. The type system requires that the left-hand-side patterns are non-overlapping and thus correspond to orthogonal subspaces, and that the right-hand-side expressions appropriately use all of the variables from the condition expression, ensuring that this data is reversibly uncomputed rather than discarded. In general, we allow the left-hand-side patterns to be non-exhaustive, in which case the semantics is a norm-*decreasing* operator rather than norm-preserving. Operationally, a decrease in norm corresponds to the probability of being in a special "exceptional state."

Existing languages like Proq [Li et al. 2020] have similarly used norm-decreasing operators such as projectors to describe assertions. Their system is effective for testing and debugging, but Qunity takes these projective assertions a step further by allowing them to be used in the control flow itself. Failed assertions are treated as *exceptions*, which can be dynamically caught and handled using the `try`-`catch` construct, generalizing another familiar classical programming construct to the quantum setting. The quantum behavior here is well-defined using the language of *projective measurements*.

$$
\begin{aligned}
\mathtt{Bit} &\coloneqq () \oplus () \\
\mathtt{0} &\coloneqq \mathtt{left}_{\mathtt{Bit}}() \\
\mathtt{1} &\coloneqq \mathtt{right}_{\mathtt{Bit}}() \\
f^{\dagger T} &\coloneqq (\lambda (f\ x) \overset{T}{\mapsto} x) \\
\mathtt{Maybe}\ (T) &\coloneqq () \oplus T \\
\mathtt{nothing}_{T} &\coloneqq \mathtt{left}_{\mathtt{Maybe}\ (T)}() \\
\mathtt{just}_{T} &\coloneqq \mathtt{right}_{\mathtt{Maybe}\ (T)} \\
T^{\otimes 0} &\coloneqq () \\
T^{\otimes (n+1)} &\coloneqq T \otimes T^{\otimes n} \\
e^{\otimes 0} &\coloneqq () \\
e^{\otimes (n+1)} &\coloneqq (e, e^{\otimes n}) \\
\mathtt{gphase}_{T}(r) &\coloneqq \mathtt{rphase}_{T} \begin{Bmatrix} x \mapsto r \\ \mathtt{else} \mapsto r \end{Bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
(e \triangleright f) &\coloneqq (f\ e) \\
(\mathtt{let}\ e_1 =_{T} e_2\ \mathtt{in}\ e_3) &\coloneqq (e_2 \triangleright \lambda e_1 \overset{T}{\mapsto} e_3) \\
(f \circ_{T} f') &\coloneqq (\lambda x \overset{T}{\mapsto} f(f'\ x)) \\
\mathtt{fst}_{T_0 \otimes T_1} &\coloneqq \lambda (x_0, x_1) \xmapsto{T_0 \otimes T_1} x_0 \\
\mathtt{snd}_{T_0 \otimes T_1} &\coloneqq \lambda (x_0, x_1) \xmapsto{T_0 \otimes T_1} x_1 \\
\mathtt{had} &\coloneqq \mathtt{u_3}(\pi/2, 0, \pi) \\
\mathtt{plus} &\coloneqq \mathtt{had}\ \mathtt{0} \\
\mathtt{minus} &\coloneqq \mathtt{had}\ \mathtt{1} \\
\mathtt{equals}_{T}(e) &\coloneqq \begin{pmatrix} \lambda x \overset{T}{\mapsto} \\ \mathtt{try}(x \triangleright \lambda e \overset{T}{\mapsto} \mathtt{1})\ \mathtt{catch}\ \mathtt{0} \end{pmatrix} \\
\mathtt{reflect}_{T}(e) &\coloneqq \mathtt{rphase}_{T} \begin{Bmatrix} e \mapsto 0 \\ \mathtt{else} \mapsto \pi \end{Bmatrix}
\end{aligned}
$$

Fig. 4. Syntactic sugar

Assuming projector P is implemented by Qunity program f_P, and state $|\psi\rangle$ is produced by Qunity program e_ψ, the Qunity expression $\mathtt{try}\ \mathtt{just}_T(f_P\ e_\psi)\ \mathtt{catch}\ \mathtt{nothing}_T$ (using Maybe syntax from Figure 4) produces the mixed state defined by the density operator:

$$P|\psi\rangle\langle\psi|P \oplus \langle\psi|\,(I-P)\,|\psi\rangle\,.$$

This state has $P\,|\psi\rangle$ in the "just" subspace and the norm of $(I-P)\,|\psi\rangle$ in the "nothing" subspace. Though we use the language of exception handling, this construct is also useful for non-exceptional conditions, like in the definition of the equals program in Figure 4. This function can be used to measure whether two states are equal; for example, a simple "coin flip" can be implemented by $(\mathtt{had}\,\mathtt{0} \triangleright \mathtt{equals}_{\mathtt{Bit}}(\mathtt{1}))$, which applies a Hadamard gate to a qubit in the $|\mathtt{0}\rangle$ state and then measures whether the result is $|\mathtt{1}\rangle$.

Notice that the innermost lambda in the definition of equals uses a non-exhaustive pattern on the left side; the program "$\lambda \mathtt{0} \xmapsto{\mathtt{Bit}} \mathtt{1}$" implements the projector $|\mathtt{1}\rangle\langle\mathtt{0}|$. While a classical operational semantics typically interprets lambdas in terms of *substitution* of arguments for parameters, Qunity's semantics is best interpreted as a more general linear mapping. This interpretation means that Qunity's type system can allow for a much wider range of expressions on the left "parameter" side of the lambda. As an extreme case, consider the definition of "$f^{\dagger T}$" syntax in Figure 4, which applies a function on the left side. Semantically, this lambda corresponds to the adjoint of f, and can be interpreted like this: "given $f(x)$ as input, output x." (However, this interpretation is imprecise when f is norm-decreasing and its adjoint is not its inverse.)

Two of Qunity's constructs have no classical analog. The $\mathtt{u_3}$ construct is a parameterized gate that allows us to implement any single-qubit gate [Cross et al. 2017]; e.g., it is used to implement had, per Figure 4. The rphase construct induces a relative phase of e^{ir}, where the value r comes from the first branch for states in the subspace spanned by the first branch's parameter e and from the second branch for states in the orthogonal subspace. It is used to implement gphase in Figure 4, used to implement conditional phase flip from Deutsch's algorithm (Section 1.1). More generally, it can be used to implement reflections, as used in Grover's search algorithm, the iterator for which is shown below. We see that it includes the same conditional phase flip as Deutsch's, composed with $\texttt{reflect}_{\texttt{Bit}^{\otimes n}}(\texttt{plus}^{\otimes n})$ to perform inversion about the mean. If the state $|\psi\rangle$ is implemented by the expression e_ψ, then $\texttt{reflect}_T(e)$ implements the reflection $(2|\psi\rangle\langle\psi| - I)$ by coherently applying a phase of $e^{i\pi} = -1$ to any state orthogonal to $|\psi\rangle$.

$$\texttt{grover}_n(f) \coloneqq \lambda x \xmapsto{\texttt{Bit}^{\otimes n}} \texttt{ctrl}\ f\ x\ {}_{\texttt{Bit}}\!\left\{ \begin{array}{l} \texttt{0} \mapsto x \\ \texttt{1} \mapsto x \triangleright \texttt{gphase}_{\texttt{Bit}^{\otimes n}}(\pi) \end{array} \right\}_{\texttt{Bit}^{\otimes n}} \triangleright \texttt{reflect}_{\texttt{Bit}^{\otimes n}}(\texttt{plus}^{\otimes n})$$

Several lambda calculi [Arrighi and Dowek 2017; Selinger and Valiron 2009; van Tonder 2004] have explored the use of higher-order functions in a quantum setting, but Qunity does not. We aim for Qunity's denotational semantics to closely correspond to existing notations and conventions used in quantum algorithms, and higher-order functions and "superpositions of programs" are uncommon and inconsistently defined. In our experience, quantum mechanical notation (and to some degree quantum computing in general) is ill-suited for higher-order programs. In particular, we interpret programs f as quantum operations acting on their argument, and expressions e as quantum operations acting on their free variables, but allowing for higher-order functions means that programs can also have free variables, and one must take some sort of tensor product of a program's two inputs. To avoid this extra complication, we have first-order functions only, and our typing relation prevents them from containing free variables.

3 TYPING

This section describes Qunity's type system. We prove that well-typed programs have a well-defined semantics (given in Section 4), and we have implemented a type checker for Qunity programs [Voichick et al. 2022b].

Qunity's type system takes the form of three different, interdependent typing judgments: *pure expression typing*, *mixed expression typing*, and *program typing*. The distinction between pure and mixed typing comes from the fact that there are two ways to mathematically represent quantum states depending on whether any classical probability is involved: pure states are described by state vectors and do not involve classical probability, while mixed states are described by density matrices, usually interpreted as a classical probability distribution over pure states. Program types are similarly divided into pure and mixed versions, where pure programs correspond to linear operators from pure states to pure states, and mixed programs correspond to *superoperators* from mixed states to mixed states. In the supplemental report [Voichick et al. 2022a], we include further discussion on the need for both of these in Qunity.

Judgments are parameterized by *typing contexts*, which are ordered lists of variable-type pairs, as defined in Fig. 2. We say that a typing context $(x_1 : T_1, \ldots, x_n : T_n)$ is *well-formed* if all variables are distinct; that is, if $x_j \neq x_k$ whenever $j \neq k$. Concatenating two typing contexts Γ_1 and Γ_2 is written Γ_1, Γ_2. Within the inference rules, there is an implicit assumption that contexts are well-formed (so concatenation requires $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$).

3.1 Pure Expression Typing

The pure expression typing judgment is written $\Gamma \parallel \Delta \vdash e : T$,[1] indicating that expression e has pure type T with respect to classical context Γ and quantum context Δ. Variables in the classical context are in a classical basis state, and are automatically uncomputed, while quantum context

[1]Whether a context is classical or quantum is based on its *position* in the typing judgment—left of $\parallel$ is classical and right of it is quantum. We write Γ (Δ, resp.) for classical (quantum, resp.) variables as a common but not universal convention.

variables are consumed and may be in superposition. Operationally, classical data is still compiled into qubits, but these qubits are only used as control wires for controlled operations, and they are uncomputed when they go out of scope. Whenever $\varnothing \parallel \varnothing \vdash e : T$ holds, the semantics corresponds to a pure state in $\mathcal{H}(T)$, the Hilbert space assigned to T.

$$\frac{}{\Gamma \parallel \varnothing \vdash () : ()}\ \textsc{T-Unit} \qquad \frac{}{\Gamma, x : T, \Gamma' \parallel \varnothing \vdash x : T}\ \textsc{T-CVar} \qquad \frac{x \notin \mathrm{dom}(\Gamma)}{\Gamma \parallel x : T \vdash x : T}\ \textsc{T-QVar}$$

$$\frac{\Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0 \qquad \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1}{\Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1}\ \textsc{T-PurePair}$$

$$\frac{\begin{array}{ccc}\Gamma, \Delta \Vdash e : T & \mathrm{ortho}_T(e_1, \dots, e_n) & \varnothing \parallel \Gamma_j \vdash e_j : T \text{ for all } j \\ \multicolumn{2}{c}{\mathrm{erases}_{T'}(x; e'_1, \dots, e'_n) \text{ for all } x \in \mathrm{dom}(\Delta)} & \Gamma, \Gamma', \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T' \text{ for all } j\end{array}}{\Gamma, \Gamma' \parallel \Delta, \Delta' \vdash \mathtt{ctrl}\ e\ {}_T\!\left\{\begin{array}{c} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{array}\right\}_{T'} : T'}\ \textsc{T-Ctrl}$$

$$\frac{\vdash f : T \rightsquigarrow T' \qquad \Gamma \parallel \Delta \vdash e : T}{\Gamma \parallel \Delta \vdash f\ e : T'}\ \textsc{T-PureApp} \qquad \frac{\Gamma \parallel \Delta \vdash e : T}{\pi_{\mathrm{G}}(\Gamma) \parallel \pi_{\mathrm{D}}(\Delta) \vdash e : T}\ \textsc{T-PurePerm}$$

Fig. 5. Pure expression typing rules

The pure expression typing rules are given in Figure 5. As is typical in quantum computing languages, these rules are *substructural* [Walker 2004]. In particular, as in QML [Altenkirch and Grattage 2005], quantum variables are *relevant*, meaning they must be used *at least once*. This invariant is evident from the T-QVar rule, which requires the quantum context to contain only the variable x of interest, and the T-CVar and T-Unit rules, which require the quantum context to be empty. Indeed, we can prove that $\Gamma \parallel \Delta \vdash e : T$ implies $\mathrm{dom}(\Delta) \subseteq \mathrm{FV}(e)$. On the other hand, the rules do not enforce $\mathrm{dom}(\Gamma) \subseteq \mathrm{FV}(e)$.

By enforcing variable relevance, Qunity's type system can control what parts of a program are allowed to discard information. Semantically, the non-use of a variable corresponds to a *partial trace*, a quantum operation described by decoherence. Quantum control flow is ill-defined in the presence of decoherence [Bădescu and Panangaden 2015], and decoherence is inherently irreversible, so we use relevant types wherever computation must be reversible or subject to quantum control.

A quantum lambda calculus [Selinger and Valiron 2009] uses *affine* types for quantum data, ensuring quantum variables are used at most once, while Proto-Quipper [Ross 2015] and QWIRE [Paykin et al. 2017] use *linear* types, requiring them to be used exactly once. This is done as static enforcement of the no-cloning theorem [Wootters and Zurek 1982], which makes sense in the QRAM computational model [Knill 1996]. However, Qunity's aim (like QML's [Altenkirch and Grattage 2005]) is to treat quantum computing as a generalization of classical computing. Qunity fixes a particular standard computational basis and treats expression (x, x) as *entangling* variable x when it is in superposition, essentially as a linear isometry $\alpha\,|0\rangle + \beta\,|1\rangle \mapsto \alpha\,|00\rangle + \beta\,|11\rangle$. The expression is deemed well typed by T-PurePair—notice that Δ appears when typing both e_0 and e_1, allowing duplication ("sharing") of quantum resources across the pair. In contrast, QRAM-based languages reject (x, x) to avoid confusion with the non-physical cloning function $\alpha\,|0\rangle + \beta\,|1\rangle \mapsto (\alpha\,|0\rangle + \beta\,|1\rangle) \otimes (\alpha\,|0\rangle + \beta\,|1\rangle)$.

The difference between "sharing" and "cloning" affects only non-basis states. *Sharing* a qubit $\alpha\,|0\rangle + \beta\,|1\rangle$ to a second qubit register produces the entangled state $\alpha\,|00\rangle + \beta\,|11\rangle$, while *cloning* would produce the unentangled state $(\alpha\,|0\rangle + \beta\,|1\rangle)(\alpha\,|0\rangle + \beta\,|1\rangle) = \alpha^2\,|00\rangle + \alpha\beta\,|01\rangle + \alpha\beta\,|10\rangle + \beta^2\,|11\rangle$. The former is basis-dependent and implementable in Qunity as "(x,x)," while the latter is basis-independent and physically prohibited by the no-cloning theorem.

The T-PureApp rule applies a linear operator f to its argument e. The T-PurePerm rule exists to allow for the usual structural rule of exchange, which is typically implicit. Here and throughout this work, the functions π are list permutation functions, arbitrarily permuting the bindings within a context. Making the exchange rule explicit allows compilation to be a direct function of the typing judgment, with swap gates introduced at uses of the explicit exchange rule (Section 6.2).

Rule T-Ctrl types pattern matching and quantum control. We defer discussing it to Section 3.4, after we have considered the other judgments.

3.2 Mixed Expression Typing

We write $\Delta \Vdash e : T$ to indicate that expression e has mixed type T under quantum context Δ;[2] the rules are in Figure 6. $\varnothing \Vdash e : T$ implies e's semantics corresponds to a *mixed state* in $\mathcal{H}(T)$.

$$\frac{\varnothing \parallel \Delta \vdash e : T}{\Delta \Vdash e : T}\ \textsc{T-Mix} \qquad \frac{\Delta \Vdash e : T}{\pi(\Delta) \Vdash e : T}\ \textsc{T-MixedPerm}$$

$$\frac{\Delta, \Delta_0 \Vdash e_0 : T_0 \qquad \Delta, \Delta_1 \Vdash e_1 : T_1}{\Delta, \Delta_0, \Delta_1 \Vdash (e_0, e_1) : T_0 \otimes T_1}\ \textsc{T-MixedPair}$$

$$\frac{\Delta_0 \Vdash e_0 : T \qquad \Delta_1 \Vdash e_1 : T}{\Delta_0, \Delta_1 \Vdash \texttt{try}\ e_0\ \texttt{catch}\ e_1 : T}\ \textsc{T-Try} \qquad \frac{\vdash f : T \Rrightarrow T' \qquad \Delta \Vdash e : T}{\Delta \Vdash f\ e : T'}\ \textsc{T-MixedApp}$$

Fig. 6. Mixed expression typing rules

Rule T-Mix allows pure expressions to be typed as mixed. (One could equivalently treat pure types as a *subtype* of mixed types.) Rules T-MixedPerm and T-MixedPair are analogous to the pure-expression versions. Rule T-MixedApp allows applying a quantum channel f on an expression e—such an f may perform measurements, as discussed below. Rule T-Try allows exceptions occurring in expression e_0 to be caught and replaced by expression e_1. Operationally, this is effectively "measuring whether an error occurred" and thus `try-catch` expressions have no *pure* type; it also cannot be done without perturbing the input data Δ_0 and thus expression e_1 is typed in a separate context.

3.3 Program Typing

We write $\vdash f : F$ to indicate that program f has type F; the rules are in Figure 7. Whenever $\vdash f : T \rightsquigarrow T'$ holds, the semantics corresponds to a linear operator mapping $\mathcal{H}(T)$ to $\mathcal{H}(T')$. Whenever $\vdash f : T \Rrightarrow T'$ holds, the semantics corresponds to a superoperator mapping mixed states in $\mathcal{H}(T)$ to mixed states in $\mathcal{H}(T')$.

T-Gate types a single-qubit unitary gate, and T-Left and T-Right type sum introduction. These are linear operations. T-PureAbs types a linear abstraction. In abstraction $\lambda e \overset{T}{\mapsto} e'$, variables introduced in e are *relevant* in e'—they must be present in quantum context Δ used to type both e

[2]We use the double-lined "$\Vdash$" symbol because this typing judgment can be used for measurements, and quantum circuit diagrams conventionally use double-lined wires to carry measurement results.

$$\frac{}{\vdash \mathtt{u_3}(r_\theta, r_\phi, r_\lambda) : \mathtt{Bit} \rightsquigarrow \mathtt{Bit}}\ \textsc{T-Gate}$$

$$\frac{}{\vdash \mathtt{left}_{T_0 \oplus T_1} : T_0 \rightsquigarrow T_0 \oplus T_1}\ \textsc{T-Left} \qquad \frac{}{\vdash \mathtt{right}_{T_0 \oplus T_1} : T_1 \rightsquigarrow T_0 \oplus T_1}\ \textsc{T-Right}$$

$$\frac{\varnothing \parallel \Delta \vdash e : T \qquad \varnothing \parallel \Delta \vdash e' : T'}{\vdash \lambda e \overset{T}{\mapsto} e' : T \rightsquigarrow T'}\ \textsc{T-PureAbs} \qquad \frac{\varnothing \parallel \Delta \vdash e : T}{\vdash \mathtt{rphase}_T \begin{Bmatrix} e \mapsto r \\ \mathtt{else} \mapsto r' \end{Bmatrix} : T \rightsquigarrow T}\ \textsc{T-Rphase}$$

$$\frac{\vdash f : T \rightsquigarrow T'}{\vdash f : T \Rrightarrow T'}\ \textsc{T-Channel} \qquad \frac{\varnothing \parallel \Delta, \Delta_0 \vdash e : T \qquad \Delta \Vdash e' : T'}{\vdash \lambda e \overset{T}{\mapsto} e' : T \Rrightarrow T'}\ \textsc{T-MixedAbs}$$

Fig. 7. Program typing rules

and e'. (It is not hard to prove that Δ can always be uniquely determined.) T-MixedAbs is more relaxed: variables in e can be contained in a context Δ_0 which is *not* used to type e'; such variables are *discarded* in e', which implies measuring them. So the typing judgment $\vdash \lambda x \xmapsto{\mathtt{Bit}} \mathtt{0} : \mathtt{Bit} \rightsquigarrow \mathtt{Bit}$ is invalid but the typing judgment $\vdash \lambda x \xmapsto{\mathtt{Bit}} \mathtt{0} : \mathtt{Bit} \Rrightarrow \mathtt{Bit}$ is valid. Rphase uses the expression e as a pattern for coherently inducing a phase, either e^{ir} or $e^{ir'}$ depending on whether pattern e is matched. T-Channel permits a linear operator to be treated as a superoperator.

Qunity programs are typed without context to help avoid scenarios where "entangling through variable re-use" might be confusing. For example, the program $\lambda x \xmapsto{\mathtt{Bit}} (x \triangleright (\lambda y \xmapsto{\mathtt{Bit}} x))$ is ill-typed in Qunity because the subprogram $(\lambda y \xmapsto{\mathtt{Bit}} x)$ has a free variable x. This is not a major loss in expressiveness because the rewritten program $\lambda x \xmapsto{\mathtt{Bit}} \mathtt{let}\ (x_0, x_1) =_{\mathtt{Bit} \otimes \mathtt{Bit}} (x, x)\ \mathtt{in}\ x_0$ is valid, with type $\mathtt{Bit} \Rrightarrow \mathtt{Bit}$. This program measures a qubit by re-using the variable to share the qubit to a second register and then discarding the entangled qubit, and the valid program makes this sharing explicit. Qunity is designed so that pairing is the only way to perform this kind of entanglement, by using a variable on both sides of the pair.

3.4 Typing Quantum Control

Qunity allows for quantum control by generalizing pattern matching via `ctrl`, which is typed via the T-Ctrl rule (Figure 5). The e_js in the premises of this rule refer to the indexed expressions in the conclusion. A "prime" symbol should be viewed as part of the variable name. The following is the T-Ctrl rule for $n = 2$:

$$\frac{\begin{matrix} \Gamma, \Delta \Vdash e : T \qquad \mathrm{ortho}_T\,(e_1, e_2) \qquad \varnothing \parallel \Gamma_1 \vdash e_1 : T \qquad \varnothing \parallel \Gamma_2 \vdash e_2 : T \\ \mathrm{erases}_{T'}(x; e'_1, e'_2) \text{ for all } x \in \mathrm{dom}(\Delta) \qquad \Gamma, \Gamma', \Gamma_1 \parallel \Delta \vdash e'_1 : T' \qquad \Gamma, \Gamma', \Gamma_2 \parallel \Delta \vdash e'_2 : T' \end{matrix}}{\Gamma, \Gamma' \parallel \Delta \vdash \mathtt{ctrl}\ e \ {}_T\!\begin{Bmatrix} e_1 \mapsto e'_1 \\ e_2 \mapsto e'_2 \end{Bmatrix}_{T'} : T'}$$

To ensure realizable circuits, the type rule imposes several restrictions on patterns. First, both pattern and body expressions e_j and e'_j must be pure expressions; this means they cannot include invocations of superoperators, which might involve measurements. Some recent work has tried to generalize the definition of quantum control to a notion of "quantum alternation" that allows measurements to be controlled. QuGcl [Ying 2016, Chapter 6] attempts this, but the resulting

$$\frac{\text{erases}_T(x; e_1, \dots, e_n)}{\text{erases}_T(x; e_1, \dots, e_{j-1}, e_j \triangleright \text{gphase}_T(r), e_{j+1}, \dots, e_n)} \text{ E-GPHASE}$$

$$\frac{\text{erases}_T(x; e_1, \dots, e_{j-1}, e_{j,1}, \dots, e_{j,m}, e_{j+1}, \dots, e_n)}{\text{erases}_T\left(x; e_1, \dots, e_{j-1}, \texttt{ctrl}\ e \ {}_{T'}\!\left\{\begin{array}{l} e'_1 \mapsto e_{j,1} \\ \cdots \\ e'_m \mapsto e_{j,m} \end{array}\right\}_T, e_{j+1}, \dots, e_n\right)} \text{ E-CTRL}$$

$$\frac{}{\text{erases}_T(x; x, x, \dots, x)} \text{ E-VAR} \qquad \frac{\text{erases}_{T_0}(x; e_{0,1}, \dots, e_{0,n})}{\text{erases}_{T_0 \otimes T_1}(x; (e_{0,1}, e_{1,1}), \dots, (e_{0,n}, e_{1,n}))} \text{ E-PAIR0}$$

$$\frac{\text{erases}_{T_1}(x; e_{1,1}, \dots, e_{1,n})}{\text{erases}_{T_0 \otimes T_1}(x; (e_{0,1}, e_{1,1}), \dots, (e_{0,n}, e_{1,n}))} \text{ E-PAIR1}$$

Fig. 8. Erasure inference rules

denotational semantics is non-compositional. Bădescu and Panangaden [Bădescu and Panangaden 2015] shed more light on the problems with quantum alternation, proving that quantum alternation is not monotone with respect to the Löwner order [Ying 2016, p. 17] and concluding that “quantum alternation is a fantasy arising from programming language semantics rather than from physics.”

For `ctrl` semantics to be physically meaningful, T-CTRL depends on two additional judgments applied to the body expressions—“ortho” for the left-hand-side (defined further below), and “erases” for the right-hand-side (defined in Figure 8). The ortho judgment ensures that the left-hand-side patterns e_j are purely classical and non-overlapping. The erases judgment (defined using gphase syntax from Figure 4) ensures that all of the right-hand-side patterns e'_j use the variables from e in a consistent way, and its name comes from the way this judgment is used by the compiler to coherently “erase” these variables. The contexts Γ_j are the primary motivation that the typing relation includes classical contexts Γ at all. Semantically, the variables they represent are purely classical; operationally, the information on these registers is used without being “consumed,” under the assumption that it will be uncomputed. When typing the expressions e_j, these contexts Γ_j appear in the “quantum” part of the context so that the type system can enforce variable relevance, but the restrictions placed by the ortho judgment ensure that these variables are still practically classical.

The orthogonality judgment is defined in terms of a “spanning” judgment defined in the supplemental report [Voichick et al. 2022a]. This judgment, written $\text{spanning}_T\ (e_1, \dots, e_n)$ and largely inspired by SPM’s [Sabry et al. 2018] “orthogonal decomposition” judgment, denotes that the set of expressions $\{e_1, \dots, e_n\}$ is a spanning set for type T. This judgment enforces that the expressions form an exhaustive set of patterns for the type T, which in our quantum setting semantically corresponds to a set of states that span the Hilbert space corresponding to T.

We write $\text{ortho}_T\ (e_1, \dots, e_n)$ to denote that the set of expressions $\{e_1, \dots, e_n\}$ is orthogonal. An orthogonal set of expressions is simply a subset of some spanning set. That is, orthogonality judgments can be defined by the following inference rule alone:

$$\frac{\text{spanning}_T\ (e'_1, \dots, e'_m) \qquad [e_1, \dots, e_n] \text{ is a subsequence of } [e'_1, \dots, e'_m]}{\text{ortho}_T\ (e_1, \dots, e_n)}$$

It is not hard to prove that orthogonality holds for a set of expressions regardless of the order they appear in the judgment (e.g., $\mathrm{ortho}_T\ (e_1, e_2)$ *iff* $\mathrm{ortho}_T\ (e_2, e_1)$).

4 SEMANTICS

In this section, we define Qunity's denotational semantics in terms of linear operators (for pure expressions) and superoperators (for mixed expressions). These definitions are designed to naturally generalize from a classical semantics defined on a classical sublanguage of Qunity. After presenting the semantics, we present some metatheoretical results, most notably that well-typed Qunity programs are well-defined according to the semantics.

4.1 Classical Sublanguage Semantics

Qunity's semantics may be more intuitive if we first restrict ourselves to a classical sublanguage.

Definition 4.1 (classical sublanguage). Qunity's classical sublanguage is defined by removing the `u3` and `rphase` constructs from the language, producing the following:

$$e ::= () \mid x \mid (e,e) \mid \texttt{ctrl}\ e\ {}_{T}\begin{Bmatrix} e \mapsto e \\ \cdots \\ e \mapsto e \end{Bmatrix}_{T} \mid \texttt{try}\ e\ \texttt{catch}\ e \mid f\ e$$

$$f ::= \texttt{left}_{T \oplus T} \mid \texttt{right}_{T \oplus T} \mid \lambda e \overset{T}{\mapsto} e$$

We can define a classical denotational semantics for this sublanguage using partial functions over *values* and *valuations*.

Definition 4.2 (value and valuation). For any type T, we write $\mathbb{V}(T)$ to denote the set of expressions that are values of that type.

$$\begin{aligned}
\mathbb{V}(\texttt{Void}) &:= \varnothing \\
\mathbb{V}(()) &:= \{()\} \\
\mathbb{V}(T_0 \oplus T_1) &:= \{\texttt{left}_{T_0 \oplus T_1}\ v_0 \mid v_0 \in \mathbb{V}(T_0)\} \cup \{\texttt{right}_{T_0 \oplus T_1}\ v_1 \mid v_1 \in \mathbb{V}(T_1)\} \\
\mathbb{V}(T_0 \otimes T_1) &:= \{(v_0, v_1) \mid v_0 \in \mathbb{V}(T_0), v_1 \in \mathbb{V}(T_1)\}
\end{aligned}$$

A valuation, written σ or τ, is a list of variable-value pairs:

$$\sigma ::= \varnothing \mid \sigma, x \mapsto v$$

Like with Γ and Δ, we will generally use the letter σ for classical data and τ for quantum data, but the two are interchangeable. Each typing context has a corresponding set of valuations, defined as follows:

$$\mathbb{V}(x_1 : T_1, \ldots, x_n : T_n) := \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \mid v_1 \in \mathbb{V}(T_1), \ldots, v_n \in \mathbb{V}(T_n)\}$$

Like with typing contexts, we use a comma to denote the concatenation of valuations, sometimes mixing valuations with explicit variable-value pairs. For example, if $\tau_0 \in \mathbb{V}(\Delta_0)$ and $v \in \mathbb{V}(T)$ and $\tau_1 \in \mathbb{V}(\Delta_1)$, then $\tau_0, x \mapsto v, \tau_1 \in \mathbb{V}(\Delta_0, x : T, \Delta_1)$.

In the supplemental report [Voichick et al. 2022a], we define a classical denotational semantics for this sublanguage. The semantics of an expression is a partial function from valuations to values, while the semantics of a program is a partial function from values to values. Given a purely-typed expression or program, this partial function will be injective. Stripped of quantum constructs, Qunity thus becomes a (classical) *reversible* programming language comparable to other programming languages for reversible computing.

Theorem 4.3 (expressiveness of classical sublanguage). *Any combinator written in the reversible language* Π *[James and Sabry 2012] can be translated into a pure program in Qunity's classical sublanguage, and any arrow computation written in the arrow metalanguage* ML_Π *[James and Sabry 2012] can be translated into a mixed program in Qunity's classical sublanguage. These translations preserve types and semantics.*

We prove Theorem 4.3 in the supplemental report. We choose the language ML_Π because its typing and semantics are directly comparable with Qunity's, and because a translation from a more typical `let`-based language to ML_Π already exists.

Qunity's quantum semantics, defined in the next section, can be viewed as a generalization of the classical sublanguage semantics. The advantage of this approach is that one does not have to explicitly convert between separate quantum and classical languages, as any program written in the classical sublanguage can be applied to quantum data.

Where the classical semantics uses finite sets for input and output, the quantum semantics uses finite-dimensional vector spaces. As we state more precisely and prove in the supplemental report, the classical semantics is simply the quantum semantics restricted to the standard computational basis.

Theorem 4.4 (generalization of classical semantics). *The classical semantics of any classical Qunity program coincides with its quantum semantics applied to values treated like basis states.*

4.2 Full Semantics

Definition 4.5. We associate Hilbert spaces with types and contexts as follows:

$$\begin{aligned}
\mathcal{H}(\texttt{Void}) &:= \{0\} \\
\mathcal{H}(()) &:= \mathbb{C} \\
\mathcal{H}(T_0 \oplus T_1) &:= \mathcal{H}(T_0) \oplus \mathcal{H}(T_1) \\
\mathcal{H}(T_0 \otimes T_1) &:= \mathcal{H}(T_0) \otimes \mathcal{H}(T_1) \\
\mathcal{H}(x_1 : T_1, \ldots, x_n : T_n) &:= \mathcal{H}(T_1) \otimes \cdots \otimes \mathcal{H}(T_n)
\end{aligned}$$

On the right-hand side above, we use the symbols $\oplus$ and $\otimes$ to refer to the usual direct sum and tensor product of Hilbert spaces [Roman 2008]. For those unfamiliar with the direct sum, we give an overview of its important properties in the supplemental report. Each Hilbert space $\mathcal{H}(T)$ has a canonical orthonormal basis $\{|v\rangle : v \in \mathbb{V}(T)\}$, where the meaning of $|v\rangle \in \mathcal{H}(T)$ is defined as follows:

$$\begin{aligned}
|()\rangle &:= 1 \\
\left|\texttt{left}_{T_0 \oplus T_1}\, v_0\right\rangle &:= |v_0\rangle \oplus 0 \\
\left|\texttt{right}_{T_0 \oplus T_1}\, v_1\right\rangle &:= 0 \oplus |v_1\rangle \\
|(v_0, v_1)\rangle &:= |v_0\rangle \otimes |v_1\rangle
\end{aligned}$$

We can do the same for typing contexts Δ, constructing an orthonormal basis $\{|\tau\rangle : \tau \in \mathbb{V}(\Delta)\} \subset \mathcal{H}(\Delta)$ as follows:

$$|x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\rangle := |v_1\rangle \otimes \cdots \otimes |v_n\rangle$$

Using notation from linear algebra [Axler 2015], we write $\mathcal{L}(\mathcal{H}_0, \mathcal{H}_1)$ to denote the space of linear operators from $\mathcal{H}_0$ to $\mathcal{H}_1$, with $\mathcal{L}(\mathcal{H}') := \mathcal{L}(\mathcal{H}', \mathcal{H}')$.

Qunity's semantics is defined by four mutually recursive functions of a valid typing judgment:

$$
\begin{aligned}
[\![\sigma : \Gamma \parallel \varnothing \vdash \texttt{()} : \texttt{()}]\!]\,|\varnothing\rangle &\coloneqq |\texttt{()}\rangle \\
[\![\sigma : \Gamma \parallel \varnothing \vdash x : T]\!]\,|\varnothing\rangle &\coloneqq |\sigma(x)\rangle \\
[\![\sigma : \Gamma \parallel x : T \vdash x : T]\!]\,|x \mapsto v\rangle &\coloneqq |v\rangle \\
[\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash \texttt{(}e_0\texttt{,}e_1\texttt{)} : T_0 \otimes T_1]\!]\,|\tau, \tau_0, \tau_1\rangle &\coloneqq [\![\sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0]\!]\,|\tau, \tau_0\rangle \\
&\quad \otimes [\![\sigma : \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1]\!]\,|\tau, \tau_1\rangle \\
[\![\sigma, \sigma' : \Gamma, \Gamma' \parallel \Delta, \Delta' \vdash \texttt{ctrl}\ e \underset{T}{} \begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_{T'} : T']\!]\,|\tau, \tau'\rangle &\coloneqq \sum_{v \in \mathbb{V}(T)} \langle v|\,[\![\Gamma, \Delta \Vdash e : T]\!]\,(|\sigma, \tau\rangle\!\langle \sigma, \tau|)\,|v\rangle \\
&\quad \cdot \sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \langle \sigma_j|\,[\![\varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T]\!]^{\dagger}\,|v\rangle \\
&\quad \cdot [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T']\!]\,|\tau, \tau'\rangle \\
[\![\sigma : \Gamma \parallel \Delta \vdash f\ e : T]\!]\,|\tau\rangle &\coloneqq [\![\vdash f : T \rightsquigarrow T']\!][\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]\,|\tau\rangle \\
[\![\pi_{\mathrm{G}}(\sigma) : \pi_{\mathrm{G}}(\Gamma) \parallel \pi_{\mathrm{D}}(\Delta) \vdash e : T]\!]\,|\pi_{\mathrm{D}}(\tau)\rangle &\coloneqq [\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]\,|\tau\rangle
\end{aligned}
$$

Fig. 9. Pure expression semantics

$$
\begin{aligned}
[\![\Delta, \Delta_0 \Vdash e : T]\!]\,\big(|\tau, \tau_0\rangle\!\langle \tau, \tau'_0|\big) &\coloneqq [\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!]|\tau\rangle\!\langle \tau'|[\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!]^{\dagger} \\
[\![\Delta, \Delta_0 \Vdash e : T]\!]\,\big(|\tau, \tau_0\rangle\!\langle \tau', \tau'_0|\big) &\coloneqq 0 \text{ if } (\varnothing : \varnothing \parallel \Delta \vdash e : T) \text{ holds and } \tau \neq \tau' \\
[\![\Delta, \Delta_0, \Delta_1 \Vdash \texttt{(}e_0\texttt{,}e_1\texttt{)} : T_0 \otimes T_1]\!]\,\big(|\tau, \tau_0, \tau_1\rangle\!\langle \tau', \tau'_0, \tau'_1|\big) &\coloneqq [\![\Delta, \Delta_0 \Vdash e_0 : T_0]\!]\,\big(|\tau, \tau_0\rangle\!\langle \tau', \tau'_0|\big) \\
&\quad \otimes [\![\Delta, \Delta_1 \Vdash e_1 : T_1]\!]\,\big(|\tau, \tau_1\rangle\!\langle \tau', \tau'_1|\big) \\
[\![\Delta_0, \Delta_1 \Vdash \texttt{try}\ e_0\ \texttt{catch}\ e_1 : T]\!]\,\big(|\tau_0, \tau_1\rangle\!\langle \tau'_0, \tau'_1|\big) &\coloneqq [\![\Delta_0 \Vdash e_0 : T]\!]\,\big(|\tau_0\rangle\!\langle \tau'_0|\big) \\
&\quad + (1 - \mathrm{tr}([\![\Delta_0 \Vdash e_0 : T]\!]\,\big(|\tau_0\rangle\!\langle \tau'_0|\big))) \\
&\quad \cdot [\![\Delta_1 \Vdash e_1 : T]\!]\,\big(|\tau_1\rangle\!\langle \tau'_1|\big) \\
[\![\Delta \Vdash f\ e : T']\!](|\tau\rangle\!\langle \tau'|) &\coloneqq [\![\vdash f : T \Rrightarrow T']\!]\,\big([\![\Delta \Vdash e : T]\!](|\tau\rangle\!\langle \tau'|)\big) \\
[\![\pi(\Delta) \Vdash e : T]\!](|\pi(\tau)\rangle\!\langle \pi(\tau')|) &\coloneqq [\![\Delta \Vdash e : T]\!](|\pi(\tau)\rangle\!\langle \pi(\tau')|)
\end{aligned}
$$

Fig. 10. Mixed expression semantics

- If $\Gamma \parallel \Delta \vdash e : T$ and $\sigma \in \mathbb{V}(\Gamma)$, then $[\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!] \in \mathcal{L}(\mathcal{H}(\Delta), \mathcal{H}(T))$ defines the pure semantics of expression e. We give the denotation in Figure 9. The σ is a sort of "classical data," so the pure expression semantics may be viewed as a two-parameter function $\mathbb{V}(\Gamma) \times \mathcal{H}(\Delta) \to \mathcal{H}(T)$, linear in its second argument.
- If $\Delta \Vdash e : T$, then $[\![\Delta \Vdash e : T]\!] \in \mathcal{L}(\mathcal{L}(\mathcal{H}(\Delta)), \mathcal{L}(\mathcal{H}(T)))$ defines the mixed semantics of expression e. We give the denotation in Figure 10.
- If $\vdash f : T \rightsquigarrow T'$, then $[\![\vdash f : T \rightsquigarrow T']\!] \in \mathcal{L}(\mathcal{H}(T), \mathcal{H}(T'))$ defines the pure semantics of program f. We give the denotation in Figure 11.
- If $\vdash f : T \Rrightarrow T'$, then $[\![\vdash f : T \Rrightarrow T']\!] \in \mathcal{L}(\mathcal{L}(\mathcal{H}(T)), \mathcal{L}(\mathcal{H}(T')))$ defines the mixed semantics of program f. We give the denotation in Figure 12.

$$\llbracket \vdash \mathtt{u_3(}r_\theta\mathtt{,}r_\phi\mathtt{,}r_\lambda\mathtt{)} : \mathtt{Bit} \rightsquigarrow \mathtt{Bit} \rrbracket \left|\mathtt{0}\right\rangle := \cos(r_\theta/2) \left|\mathtt{0}\right\rangle + e^{ir_\phi} \sin(r_\theta/2) \left|\mathtt{1}\right\rangle$$

$$\llbracket \vdash \mathtt{u_3(}r_\theta\mathtt{,}r_\phi\mathtt{,}r_\lambda\mathtt{)} : \mathtt{Bit} \rightsquigarrow \mathtt{Bit} \rrbracket \left|\mathtt{1}\right\rangle := -e^{ir_\lambda} \sin(r_\theta/2) \left|\mathtt{0}\right\rangle + e^{i(r_\phi + r_\lambda)} \cos(r_\theta/2) \left|\mathtt{1}\right\rangle$$

$$\llbracket \vdash \mathtt{left}_{T_0 \oplus T_1} : T_0 \rightsquigarrow T_0 \oplus T_1 \rrbracket \left|v\right\rangle := \left|\mathtt{left}_{T_0 \oplus T_1}\ v\right\rangle$$

$$\llbracket \vdash \mathtt{right}_{T_0 \oplus T_1} : T_1 \rightsquigarrow T_0 \oplus T_1 \rrbracket \left|v\right\rangle := \left|\mathtt{right}_{T_0 \oplus T_1}\ v\right\rangle$$

$$\llbracket \vdash \lambda e \overset{T}{\mapsto} e' : T \rightsquigarrow T' \rrbracket \left|v\right\rangle := \llbracket \varnothing : \varnothing \parallel \Delta \vdash e' : T' \rrbracket \llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger \left|v\right\rangle$$

$$\llbracket \vdash \mathtt{rphase}_T \begin{Bmatrix} e \mapsto r \\ \mathtt{else} \mapsto r' \end{Bmatrix} : T \rightsquigarrow T \rrbracket \left|v\right\rangle := e^{ir} \llbracket \varnothing \parallel \Delta \vdash e : T \rrbracket \llbracket \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger \left|v\right\rangle + e^{ir'} \left(\mathbb{I} - \llbracket \varnothing \parallel \Delta \vdash e : T \rrbracket \llbracket \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger\right) \left|v\right\rangle$$

Fig. 11. Pure program semantics

$$\llbracket \vdash f : T \Rrightarrow T' \rrbracket (|v\rangle\langle v'|) := \llbracket \vdash f : T \rightsquigarrow T' \rrbracket |v\rangle\langle v'| \llbracket \vdash f : T \rightsquigarrow T' \rrbracket^\dagger$$

$$\llbracket \Vdash \lambda e \overset{T}{\mapsto} e' : T \Rrightarrow T' \rrbracket (|v\rangle\langle v'|) := \llbracket \Delta \Vdash e' : T' \rrbracket \left(\mathrm{tr}_{\Delta_0} \left(\llbracket \varnothing : \varnothing \parallel \Delta, \Delta_0 \vdash e : T \rrbracket^\dagger |v\rangle\langle v'| \llbracket \varnothing : \varnothing \parallel \Delta, \Delta_0 \vdash e : T \rrbracket\right)\right)$$

Fig. 12. Mixed program semantics

We define Qunity semantics on the standard computational basis, but these should be understood to be linear operators after extending by linearity. This semantics is compositional by construction. For example, the pure semantics of $\mathtt{(}e_0\mathtt{,}e_1\mathtt{)}$ can be computed in terms of the semantics of e_0 and e_1. The operator for each subexpression is applied to the relevant part of the input basis state, and a tensor product of the resulting states defines $\mathtt{(}e_0\mathtt{,}e_1\mathtt{)}$.

The most complicated definition is the one for `ctrl`, which uses a superoperator to construct an operator. The definition uses $\langle v| \llbracket \Gamma, \Delta \Vdash e : T \rrbracket (|\sigma, \tau\rangle\langle\sigma, \tau|) |v\rangle$, the probability that expression e outputs v given input (σ, τ), as seen by a classical observer. This *probability* from the mixed semantics is then used directly as an *amplitude* in the resulting pure semantics. Note that this introduces an unavoidable source of error whenever probabilities between 0 and 1 are involved because there is no square root involved—the resulting semantics will be norm-decreasing even if the original superoperator was trace-preserving.

The definition for `try-catch` uses $(1 - \mathrm{tr}(\llbracket \Delta_0 \Vdash e_0 : T \rrbracket (|\tau_0\rangle\langle\tau_0'|)))$, the probability that e_0 throws an exception. If e_0 succeeds, then its results are used, but otherwise, the results from e_1 are used.

4.3 Metatheory

We are defining semantics as recursive functions on typing judgments rather than on programs and expressions directly. Qunity's typing relation is not syntax-directed, which means that there are multiple ways to type—and give semantics to—the same expression. Thus, we must prove that the semantics is truly a function, i.e., that the different proofs of the same judgment lead to the same semantic denotation.

As an example, note the two deduction trees in Figure 13 prove the same typing judgment, so they should give rise to the same semantics. The semantics takes a tensor product of two state vectors for T-PurePair, but a tensor product of two density operators for T-MixedPair. This is not

$$\dfrac{\dfrac{\varnothing \parallel \Gamma_0 \vdash e_0 : T_0 \qquad \varnothing \parallel \Gamma_1 \vdash e_1 : T_1}{\varnothing \parallel \Gamma_0, \Gamma_1 \vdash (e_0, e_1) : T_0 \otimes T_1}\ \textsc{T-PurePair}}{\Gamma_0, \Gamma_1 \Vdash (e_0, e_1) : T_0 \otimes T_1}\ \textsc{T-Mix}$$

$$\dfrac{\dfrac{\varnothing \parallel \Gamma_0 \vdash e_0 : T_0}{\Gamma_0 \Vdash e_0 : T_0}\ \textsc{T-Mix} \qquad \dfrac{\varnothing \parallel \Gamma_1 \vdash e_1 : T_1}{\Gamma_1 \Vdash e_1 : T_1}\ \textsc{T-Mix}}{\Gamma_0, \Gamma_1 \Vdash (e_0, e_1) : T_0 \otimes T_1}\ \textsc{T-MixedPair}$$

Fig. 13. Two proofs of the same typing judgment

a problem because a tensor product can equivalently be taken before or after the conversion to density operators: $(|\psi\rangle \otimes |\phi\rangle)(\langle\psi| \otimes \langle\phi|) = |\psi\rangle\langle\psi| \otimes |\phi\rangle\langle\phi|$ for any states $|\psi\rangle, |\phi\rangle$. We prove that this kind of equivalence always holds.

Theorem 4.6 (well-defined semantics). *Qunity has a well-defined semantics:*

- *Whenever* $(\Gamma \parallel \Delta \vdash e : T)$ *is valid and* $\sigma \in \mathbb{V}(\Gamma)$, *the denotation* $[\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]$ *is uniquely defined.*
- *Whenever* $(\Delta \Vdash e : T)$ *is valid, the denotation* $[\![\Delta \Vdash e : T]\!]$ *is uniquely defined.*
- *Whenever* $(\vdash f : T \rightsquigarrow T')$ *is valid, the denotation* $[\![\vdash f : T \rightsquigarrow T']\!]$ *is uniquely defined.*
- *Whenever* $(\vdash f : T \Rrightarrow T')$ *is valid, the denotation* $[\![\vdash f : T \Rrightarrow T']\!]$ *is uniquely defined.*

Whenever a typing judgment has more than one proof of validity, the derived semantics is independent of the proof.

The proof of Theorem 4.6, given in the supplemental report, was largely inspired by the proof of Newman's lemma [Huet 1980; Newman 1942], a standard tool for proving global confluence from local confluence when using (operational) reduction semantics. Qunity's semantics is denotational rather than operational, but one can imagine the operational procedure of *evaluating* the denotational semantics by repeatedly rewriting denotations in terms of the denotations of subexpressions. In this view, we have a terminating sequence of rewrite rules, and global confluence is exactly what is needed to prove the semantics well-defined. We do not use Newman's lemma directly, but the induction strategy is essentially the same, and this allows us to focus on particular cases of equivalence like the one in Figure 13, which are easy to verify algebraically.

Not all pure Qunity programs have a norm-preserving (isometric) semantics. Rather, programs that may throw an exception are norm-*decreasing* instead. We can characterize our semantics in terms of *Kraus operators.*

Definition 4.7 (Kraus operator). A Kraus operator is a linear operator E such that $E^\dagger E \sqsubseteq I$, where "$\sqsubseteq$" denotes the Löwner order [Ying 2016, p. 17].

Note that Kraus operators are typically defined as *sets* of operators E whose sum satisfies the property above. For our purposes, a single operator will suffice.

Theorem 4.8. $[\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]$ *and* $[\![\vdash f : T \rightsquigarrow T']\!]$ *are Kraus operators whenever well-defined.* $[\![\Delta \Vdash e : T]\!]$ *and* $[\![\vdash f : T \Rrightarrow T']\!]$ *are completely positive, trace non-increasing superoperators whenever well-defined.*

We do not prove this theorem directly, but it is a direct consequence of the correctness of our compilation procedure discussed in Section 6.

We can also be more precise about the semantics of the orthogonality and spanning judgments. In the absence of variables, the orthogonality judgment describes a set of orthogonal basis states, and the spanning judgment describes a spanning set of orthogonal basis states. With variables, the picture is a bit more complicated, because we are dealing with projectors onto orthogonal *subspaces* rather than orthogonal states. An algebraic description of these semantics is given by Lemmas 4.10 and 4.9.

Lemma 4.9 (spanning semantics). *Suppose* $\varnothing \parallel \Delta_j \vdash e_j : T$ *for all* $j \in \{1, \ldots, n\}$ *and suppose* $\text{spanning}_T\left(e_1, \ldots, e_n\right)$ *is true. Then* $\sum_{j=1}^{n} [\![\varnothing \parallel \Delta_j \vdash e_j : T]\!][\![\varnothing \parallel \Delta_j \vdash e_j : T]\!]^\dagger = \mathbb{I}$.

Lemma 4.10 (orthogonality semantics). *Suppose* $\varnothing \parallel \Delta_j \vdash e_j : T$ *for all* $j \in \{1, \ldots, n\}$ *and suppose* $\text{ortho}_T\left(e_1, \ldots, e_n\right)$ *is true. Then* $\langle v| \, [\![\varnothing \parallel \Delta_j \vdash e_j : T]\!] \left|\tau_j\right\rangle \in \{0, 1\}$ *for all* $\tau_j \in \mathbb{V}(\Delta_j), v \in \mathbb{V}(T)$, *and* $\sum_{j=1}^{n} [\![\varnothing \parallel \Delta_j \vdash e_j : T]\!][\![\varnothing \parallel \Delta_j \vdash e_j : T]\!]^\dagger \sqsubseteq \mathbb{I}$.

Finally, Qunity's semantics uses norm-decreasing operators and trace-decreasing superoperators, which can be interpreted operationally as a sort of "exception." In the supplemental report, we present an additional judgment "iso" that can be used to statically determine whether a program belongs to a checkable class of exception-free programs.

5 EXAMPLES

We have already shown two examples of programs we can write in Qunity: Deutsch's algorithm (Section 1.1) and Grover's algorithm (end of Section 2). This section presents two more—the quantum Fourier transform and the quantum walk—aiming to further illustrate Qunity's expressiveness. The quantum walk depends on *specialized erasure*, a technique that reverses `ctrl` expressions to implement a general form of reversible pattern-matching. We give a third example—the Deutsch-Jozsa algorithm—in the supplemental report.

5.1 Quantum Fourier Transform

In Figure 14, we use a presentation of the quantum Fourier transform that has a symmetric circuit diagram [Griffiths and Niu 1996]. This version uses a two-qubit "coupling" gate that swaps two qubits and coherently induces a particular global phase conditional on both qubits being $|1\rangle$.

5.2 Specialized Erasure

Quantum control—that is, programming with conditionals while maintaining quantum coherence—is a common feature of quantum algorithms, and Qunity's `ctrl` construct can be a powerful tool for implementing this pattern. However, the "erases" requirement of the T-Ctrl typing rule can be a frustrating limitation on practical quantum control. It effectively requires that any part of the input used for quantum control (the Δ context in the T-Ctrl typing rule) must be present in the output as well. We show here how programmers can get around this limitation with a general approach for "erasing" controlled data in a more customizable way, without adding any new primitives to the Qunity language.

The basic approach is fairly simple, and it is already used in the implementation of existing quantum algorithms [Childs and van Dam 2010, p. 8]. Suppose one has two Hilbert spaces $\mathcal{H}$ and $\mathcal{H}'$ with orthonormal sets $\{|1\rangle, \ldots, |n\rangle\} \subset \mathcal{H}$ and $\{|1'\rangle, \ldots, |n'\rangle\} \subset \mathcal{H}'$, and one would like to implement the operator $E \coloneqq \sum_{j=1}^{n} |j'\rangle\langle j| \in \mathcal{L}(\mathcal{H}, \mathcal{H}')$. If one can implement operators $E_1 \coloneqq \sum_{j=1}^{n} |j, j'\rangle\langle j| \in \mathcal{L}(\mathcal{H}, \mathcal{H} \otimes \mathcal{H}')$ and $E_2 \coloneqq \sum_{j=1}^{n} |j, j'\rangle\langle j'| \in \mathcal{L}(\mathcal{H}', \mathcal{H} \otimes \mathcal{H}')$, then one can implement $E = E_2^\dagger E_1$. This pattern can be useful in Qunity, where the operators E_1 and E_2 can be easy to implement using a `ctrl` expression, but the erases judgment prevents E from being implemented more directly. The $E_2^\dagger$ operator serves as a sort of "specialized erasure," erasing the input state $|j\rangle$.

$$\mathsf{and} \coloneqq \lambda x \xmapsto{\texttt{Bit}\otimes\texttt{Bit}}$$
$$\texttt{ctrl}\ x \quad {}_{\texttt{Bit}\otimes\texttt{Bit}}\begin{Bmatrix} (\texttt{0},\texttt{0}) \mapsto (x,\texttt{0}) \\ (\texttt{0},\texttt{1}) \mapsto (x,\texttt{0}) \\ (\texttt{1},\texttt{0}) \mapsto (x,\texttt{0}) \\ (\texttt{1},\texttt{1}) \mapsto (x,\texttt{1}) \end{Bmatrix}_{(\texttt{Bit}\otimes\texttt{Bit})\otimes\texttt{Bit}} \qquad \triangleright\ \mathsf{snd}_{(\texttt{Bit}\otimes\texttt{Bit})\otimes\texttt{Bit}}$$

$$\texttt{couple}(k) \coloneqq \lambda(x_0,x_1) \xmapsto{\texttt{Bit}\otimes\texttt{Bit}}$$
$$\texttt{ctrl}\ \mathsf{and}(x_0,x_1) \quad {}_{\texttt{Bit}}\begin{Bmatrix} \texttt{0} \mapsto (x_1,x_0) \\ \texttt{1} \mapsto (x_1,x_0) \triangleright \mathsf{gphase}_{\texttt{Bit}\otimes\texttt{Bit}}(2\pi\ /\ 2^{\mathsf{k}}) \end{Bmatrix}_{\texttt{Bit}\otimes\texttt{Bit}}$$

$$\texttt{rotations}(0) \coloneqq \lambda() \xmapsto{\texttt{Bit}^{\otimes 0}} ()$$

$$\texttt{rotations}(1) \coloneqq \lambda(x,()) \xmapsto{\texttt{Bit}^{\otimes 1}} (\mathsf{had}\,x,())$$

$$\texttt{rotations}(n+2) \coloneqq \lambda(x_0,x) \xmapsto{\texttt{Bit}^{\otimes(n+2)}}$$
$$\begin{pmatrix} \texttt{let}\ (x_0,(y_0',y)) =_{\texttt{Bit}^{\otimes(n+2)}} (x_0, x \triangleright \texttt{rotations}(n+1))\ \texttt{in} \\ \texttt{let}\ ((y_0,y_1),y) =_{(\texttt{Bit}\otimes\texttt{Bit})\otimes\texttt{Bit}^{\otimes n}} ((x_0,y_0') \triangleright \texttt{couple}(n+2), y)\ \texttt{in} \\ (y_0,(y_1,y)) \end{pmatrix}$$

$$\texttt{qft}(0) \coloneqq \lambda() \xmapsto{\texttt{Bit}^{\otimes 0}} ()$$

$$\texttt{qft}(n+1) \coloneqq \lambda x \xmapsto{\texttt{Bit}^{\otimes(n+1)}} \texttt{let}\ (x_0,x') =_{\texttt{Bit}^{\otimes n}} x \triangleright \texttt{rotations}(n+1)\ \texttt{in}\ (x_0, x' \triangleright \texttt{qft}(n))$$

Fig. 14. The quantum Fourier transform implemented in Qunity

$$\texttt{match}\ {}_{T}\begin{Bmatrix} e_1 \mapsto e_1' \\ \cdots \\ e_n \mapsto e_n' \end{Bmatrix}_{T'} \coloneqq \lambda x \overset{T}{\mapsto} \texttt{ctrl}\ x \ {}_{T}\begin{Bmatrix} e_1 \mapsto (x,e_1') \\ \cdots \\ e_n \mapsto (x,e_n') \end{Bmatrix}_{T\otimes T'}$$
$$\triangleright\ \lambda \begin{pmatrix} \texttt{ctrl}\ x' \ {}_{T'}\begin{Bmatrix} e_1' \mapsto (e_1,x') \\ \cdots \\ e_n' \mapsto (e_1,x') \end{Bmatrix}_{T\otimes T'} \end{pmatrix} \xmapsto{T\otimes T'} x'$$

Fig. 15. Reversible pattern matching via specialized erasure

This pattern can be used to implement a "`match`" program, shown in Figure 15. The typing and semantics of this program are very similar to the symmetric pattern matching language [Sabry et al. 2018]. In particular, the typing rule and semantics in Figure 16 can be inferred.

As another example, Figure 17 shows how one can use this pattern to implement the direct sum of linear operators, defined so that $[\![\vdash f_0 \oplus f_1 : T_0 \oplus T_1 \rightsquigarrow T_0' \oplus T_1']\!] = [\![\vdash f_0 : T_0 \rightsquigarrow T_0']\!] \oplus [\![\vdash f_1 : T_1 \rightsquigarrow T_1']\!]$.

These examples are admittedly fairly lengthy, and would not lead to an efficient implementation if compiled without optimizations using the compilation procedure described in Section 6. Our

$$\frac{\begin{array}{cc}\text{ortho}_T\ (e_1,\dots,e_n) & \varnothing \parallel \Delta_j \vdash e_j : T \text{ for all } j\\ \text{ortho}_{T'}\ (e'_1,\dots,e'_n) & \varnothing \parallel \Delta_j \vdash e'_j : T' \text{ for all } j\end{array}}{\vdash \texttt{match}\ {}_T\!\left\{\begin{array}{c} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n\end{array}\right\}_{T'} : T \rightsquigarrow T'}$$

$$\left[\!\left[\vdash \texttt{match}\ {}_T\!\left\{\begin{array}{c} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n\end{array}\right\}_{T'} : T \rightsquigarrow T'\right]\!\right] = \sum_{j=1}^{n} [\![\varnothing \parallel \Delta_j \vdash e'_j : T']\!] [\![\varnothing \parallel \Delta_j \vdash e_j : T]\!]^\dagger$$

Fig. 16. Typing and semantics for the match construct

$$f_0 \oplus f_1 := \lambda x \xmapsto{T_0 \oplus T_1} \texttt{ctrl}\ x\ {}_{T_0 \oplus T_1}\!\left\{\begin{array}{l} \texttt{left}_{T_0 \oplus T_1}\ x_0 \mapsto (x, \texttt{left}_{T'_0 \oplus T'_1}\ (f_0\ x_0)) \\ \texttt{right}_{T_0 \oplus T_1}\ x_1 \mapsto (x, \texttt{right}_{T'_0 \oplus T'_1}\ (f_1\ x_1)) \end{array}\right\}_{(T_0 \oplus T_1) \otimes (T'_0 \oplus T'_1)}$$
$$\triangleright\ \lambda\left(\texttt{ctrl}\ x' \left\{\begin{array}{l} \texttt{left}_{T'_0 \oplus T'_1}\ x'_0 \mapsto (\texttt{left}_{T_0 \oplus T_1}\left(f_0^\dagger x'_0\right), x') \\ \texttt{right}_{T'_0 \oplus T'_1}\ x'_1 \mapsto (\texttt{right}_{T_0 \oplus T_1}\left(f_1^\dagger x'_1\right), x') \end{array}\right\}\right) \xmapsto{(T_0 \oplus T_1) \otimes (T'_0 \oplus T'_1)} x'$$

Fig. 17. A Qunity implementation of the direct sum of linear operators

purpose here is to demonstrate the expressiveness of Qunity despite its small number of language primitives, but additional language primitives may make efficient compilation easier.

5.3 Quantum Walk

We use Qunity to implement a quantum walk algorithm for boolean formula evaluation [Ambainis et al. 2010; Childs et al. 2007]. The simplest version of this algorithm treats a NAND formula as a balanced binary tree where each leaf corresponds to a variable in the formula and each vertex corresponds to a NAND application. Given black-box oracle access to a function f : `Variable` $\to$ $\{0, 1\}$, the task is to evaluate the formula.

This algorithm performs a quantum walk, repeatedly applying a *diffusion step* and a *walk step* to two quantum registers: a vertex index and a qutrit "three-sided coin." The diffusion step uses coherent control to apply a different operator to the coin depending on the vertex index. If the vertex index is a leaf v, then the oracle is used to induce a conditional phase flip of $(-1)^{f(v)}$. Otherwise, a reflection operator $(2|\mathsf{u}\rangle\langle\mathsf{u}| - I)$ is applied to the coin register, where $|\mathsf{u}\rangle$ is a coin state whose value depends on whether the vertex index is one of two special root nodes. The walk step then performs a coherent permutation on the two registers, "walking" the vertex index to the direction specified by the coin and setting the coin to be the direction traveled from.

This algorithm presents some interesting challenges for implementation, and Quipper's [Green et al. 2013] implementation of this algorithm is quite long.[3] One challenge is the representation of the vertex index. The traversed graph is a tree, so a convenient representation is the path taken to get to this vertex from the root of the tree, a list like [`vleft`, `vright`, ...] indicating the sequence of child directions. The problem is that this list has an unknown length, so one cannot simply

[3]See `Quipper.Algorithms.BF.BooleanFormula` for reference.

use an array of qubits where each qubit corresponds to a direction taken. Quipper has no sum types, and every (quantum) Quipper type is effectively a fixed-length array of qubits, so Quipper's implementation must directly manipulate an encoding of variable-length lists into bitstrings, which requires developers to work at a lower level of abstraction than they would prefer. Qunity's sum types make it convenient to coherently manipulate variable-length lists by managing the qubit encoding automatically.

First, we must define the types and values used in this program in terms of existing ones. We use a recursively-defined `Vertex` type to represent a vertex in the tree as a variable-length list. This type and others are defined in Figure 18 in terms of Qunity's base types. Here, the parameter n is the height of the tree, a bound on the depth. A $\mathtt{Vertex}_{n+1}$ is thus either empty or a $\mathtt{Vertex}_n$ with an additional `Child` appended and can store a superposition of lists of different lengths. The algorithm augments the tree with two special vertices at the root of the tree, $\mathtt{root'}_n$ and $\mathtt{root}_{n+1}$, both of type $\mathtt{Vertex}_{n+2}$, where the "+2" comes from the extra depth incurred by these vertices. The leaves of the traversed tree are those where the path from the root is maximal, so we can use the fixed-length `Leaf` type to describe leaves separately from arbitrary vertices.

$$
\begin{aligned}
\mathtt{Child} &\coloneqq \mathtt{Bit} \\
\mathtt{vleft} &\coloneqq \mathtt{0} \\
\mathtt{vright} &\coloneqq \mathtt{1} \\
\mathtt{Coin} &\coloneqq \mathtt{Maybe}\ (\mathtt{Child}) \\
\mathtt{cdown} &\coloneqq \mathtt{nothing}_{\mathtt{Child}} \\
\mathtt{cleft} &\coloneqq \mathtt{just}_{\mathtt{Child}}\mathtt{vleft} \\
\mathtt{cright} &\coloneqq \mathtt{just}_{\mathtt{Child}}\mathtt{vright}
\end{aligned}
\qquad
\begin{aligned}
\mathtt{Vertex}_0 &\coloneqq \mathtt{Void} \\
\mathtt{Vertex}_{n+1} &\coloneqq () \oplus (\mathtt{Vertex}_n \otimes \mathtt{Child}) \\
\mathtt{root}_n &\coloneqq \mathtt{left}_{\mathtt{Vertex}_{n+1}}() \\
(e \hookrightarrow_n e_0) &\coloneqq \mathtt{right}_{\mathtt{Vertex}_{n+1}}(e, e_0) \\
\mathtt{root'}_n &\coloneqq \mathtt{root}_n \hookrightarrow_{n+1} \mathtt{vleft} \\
\mathtt{Leaf}_n &\coloneqq \mathtt{Child}^{\otimes n}
\end{aligned}
$$

Fig. 18. Types and values used in the quantum walk algorithm (also see Fig. 4)

Figure 19 defines some additional programs used in this algorithm. The `asleaf` program is essentially a projector onto the subspace of vertices spanned by leaves; $\mathtt{asleaf}_n$ has type $(\mathtt{Vertex}_{n+2} \rightsquigarrow \mathtt{Leaf}_n)$. This program converts the variable-length `Vertex` type into the fixed-length `Leaf` type, terminating exceptionally if the length is insufficient. The $\mathtt{downcast}_n$ program has type $(\mathtt{Vertex}_{n+1} \rightsquigarrow \mathtt{Vertex}_n)$, and is used to convert a vertex to a type with a smaller maximum length, terminating exceptionally if the length is maximal. This program, defined using the specialized erasure technique described in the previous section, is used by the $\mathtt{leftchild}_n$ and $\mathtt{rightchild}_n$ programs, both of type $(\mathtt{Vertex}_n \rightsquigarrow \mathtt{Vertex}_n)$, which reversibly "step" a vertex to a child in the tree.

We implement the diffusion step of the quantum walk in Figure 20. This implementation resembles the original algorithm description [Childs et al. 2007, p. 4] much more closely than Quipper's implementation, which involves more complicated qubit encodings and explicit uncomputation. The diffusion program has the following type:

$$\frac{\vdash f : \mathtt{Leaf}_n \Rrightarrow \mathtt{Bit}}{\vdash \mathtt{diffusion}_n(f) : \mathtt{Coin} \otimes \mathtt{Vertex}_{n+2} \rightsquigarrow \mathtt{Coin} \otimes \mathtt{Vertex}_{n+2}}$$

We implement the walk step in Figure 21. This program must update both the vertex and the coin while maintaining quantum coherence. To make this work, we use a program $\mathtt{nextcoin}_n$ of type $(\mathtt{Coin} \otimes \mathtt{Vertex}_{n+1} \rightsquigarrow (\mathtt{Coin} \otimes \mathtt{Vertex}_{n+1}) \otimes \mathtt{Coin})$ to both compute the updated coin and

$$
\begin{aligned}
\texttt{asleaf}_0 &\coloneqq \lambda\, \texttt{root'}_0 \xmapsto{\texttt{Vertex}_2} \texttt{()} \\
\texttt{asleaf}_{n+1} &\coloneqq \lambda(x \hookrightarrow_{n+2} x_0) \xmapsto{\texttt{Vertex}_{n+3}} (x_0\texttt{,}x \triangleright \texttt{asleaf}_n) \\
\texttt{downcast}_0 &\coloneqq \lambda v \xmapsto{\texttt{Vertex}_1} \texttt{ctrl}\ v\ {}_{\texttt{Vertex}_1}\{\}_{\texttt{Vertex}_0} \\
\texttt{downcast}_{n+1} &\coloneqq \lambda v \xmapsto{\texttt{Vertex}_{n+2}} \\
&\qquad \underset{\texttt{Vertex}_{n+2}}{\texttt{ctrl}\ v} \left\{\begin{array}{l} \quad\texttt{root}_{n+1} \mapsto (v\texttt{,}\texttt{root}_n) \\ v' \hookrightarrow_{n+1} x \mapsto (v\texttt{,}\texttt{downcast}_n v' \hookrightarrow_n x) \end{array}\right\}_{\texttt{Vertex}_{n+2}\otimes\texttt{Vertex}_{n+1}} \\
&\qquad \triangleright \lambda \quad \underset{\texttt{Vertex}_{n+1}}{\texttt{ctrl}\ v} \left\{\begin{array}{l} \quad\texttt{root}_n \mapsto (\texttt{root}_{n+1}\texttt{,}v) \\ v' \hookrightarrow_n x \mapsto (\texttt{downcast}_n^{\dagger} v' \hookrightarrow_{n+1} x\texttt{,}v) \end{array}\right\}_{\texttt{Vertex}_{n+2}\otimes\texttt{Vertex}_{n+1}} \xmapsto{\texttt{Vertex}_{n+2}\otimes\texttt{Vertex}_{n+1}} v \\
\texttt{leftchild}_n &\coloneqq \lambda v \xmapsto{\texttt{Vertex}_n} \texttt{downcast}_n(v \hookrightarrow_n \texttt{vleft}) \\
\texttt{rightchild}_n &\coloneqq \lambda v \xmapsto{\texttt{Vertex}_n} \texttt{downcast}_n(v \hookrightarrow_n \texttt{vright})
\end{aligned}
$$

Fig. 19. Programs and expressions used in the quantum walk implementation

$$
\begin{array}{l}
\texttt{diffusion}_n(f) \coloneqq \\
\lambda(c\texttt{,}v) \xmapsto{\texttt{Coin}\otimes\texttt{Vertex}_{n+2}} \\
\texttt{ctrl}\ v \\
\left\{\begin{array}{l}
v' \hookrightarrow_n x \hookrightarrow_{n+1} x' \mapsto \texttt{ctrl} \begin{pmatrix} \text{try just}_{\text{Leaf}(n)}((v' \hookrightarrow_n x \hookrightarrow_{n+1} x') \quad \triangleright\texttt{asleaf}_n) \\ \text{catch nothing}_{\text{Leaf}(n)} \end{pmatrix} \\
\qquad\qquad\qquad \left\{\begin{array}{l} \text{nothing}_{\text{Leaf}(n)} \mapsto (c \triangleright \texttt{reflect}_{\texttt{Coin}}(\texttt{u})\texttt{,}v) \\ \quad \text{just}_{\text{Leaf}(n)}\ell \mapsto \texttt{ctrl}\ \ (f\ \ell) \left\{\begin{array}{l} \texttt{0} \mapsto (c\texttt{,}v) \\ \texttt{1} \mapsto (c\texttt{,}v) \triangleright \text{gphase}\texttt{(}\pi\texttt{)} \end{array}\right\} \end{array}\right\} \\
\qquad \texttt{root'}_n \mapsto (c \triangleright \texttt{reflect}_{\texttt{Coin}}(\texttt{u'}_n)\texttt{,}v) \\
\qquad \texttt{root}_{n+1} \mapsto (c\texttt{,}v)
\end{array}\right\}
\end{array}
$$

Fig. 20. The diffusion step of the boolean formula algorithm. For brevity, we omit some type annotations and the implementation of the expressions u and $\texttt{u'}_n$. See the Coq implementation [Voichick et al. 2022b] for details.

to uncompute the previous coin. The walk program $\texttt{walk}_n$ then has type ($\texttt{Coin} \otimes \texttt{Vertex}_{n+1} \rightsquigarrow \texttt{Coin} \otimes \texttt{Vertex}_{n+1}$).

6 COMPILATION

We have developed an algorithm for compiling Qunity to a low-level qubit circuit language such as OpenQasm [Cross et al. 2021]. This section provides an overview of the algorithm; the details are given in the supplemental report [Voichick et al. 2022a].

$$\texttt{nextcoin}_n \coloneqq \lambda x \mapsto \texttt{ctrl}\ x \left\{ \begin{array}{r} (\texttt{cdown}, v \hookrightarrow_n \texttt{vleft}) \mapsto (x, \texttt{cleft}) \\ (\texttt{cdown}, v \hookrightarrow_n \texttt{vright}) \mapsto (x, \texttt{cleft}) \\ (\texttt{cright}, v) \mapsto (x, \texttt{cdown}) \\ (\texttt{cleft}, v) \mapsto (x, \texttt{cdown}) \end{array} \right\}_{\texttt{Coin} \otimes \texttt{Vertex}_{n+1}}^{(\texttt{Coin} \otimes \texttt{Vertex}_{n+1}) \otimes \texttt{Coin}}$$

$$\begin{aligned}
\texttt{walk}_n \coloneqq \lambda x &\xmapsto{\texttt{Coin} \otimes \texttt{Vertex}_{n+1}} \\
&\texttt{nextcoin}_n x \\
&\triangleright \lambda((c, v), c') \xmapsto{(\texttt{Coin} \otimes \texttt{Vertex}_{n+1}) \otimes \texttt{Coin}} \\
&\texttt{ctrl}\ (c, c') \left\{ \begin{array}{l} (\texttt{cdown}, \texttt{cleft}) \mapsto ((c', \texttt{leftchild}_{n+1}{}^{\dagger} v), c) \\ (\texttt{cdown}, \texttt{cright}) \mapsto ((c', \texttt{rightchild}_{n+1}{}^{\dagger} v), c) \\ (\texttt{cleft}, \texttt{cdown}) \mapsto ((c', \texttt{leftchild}_{n+1} v), c) \\ (\texttt{cright}, \texttt{cdown}) \mapsto ((c', \texttt{rightchild}_{n+1} v), c) \end{array} \right\} \\
&\triangleright \texttt{nextcoin}_n{}^{\dagger}
\end{aligned}$$

Fig. 21. The walk step of the boolean formula algorithm

Our compilation algorithm serves two purposes. First, it makes clear that Qunity is realizable on a quantum computer. Realizability is not immediately clear from the definitions of Qunity's semantics; some quantum languages like Lineal [Arrighi and Dowek 2017] and the zx-calculus [van de Wetering 2020] allow for programs to be written with norm-*increasing* semantics and thus have no physical interpretation. Second, it gives an intuition on what is operationally happening step-by-step in the physical computer and answers questions about data representation, automatic uncomputation, and the interplay between Qunity's pure and mixed modes of computation.

There are some obvious challenges in compiling Qunity to low-level circuits. In particular, the `ctrl` construct can make irreversible programs reversible, and the `try-catch` construct can make trace-decreasing programs trace-preserving. We solve these problems by augmenting the circuit with ancillary "garbage" and "flag" qubits, and then compiling it into a unitary circuit. The garbage qubits are used to store discarded information, taking advantage of the deferred measurement principle [Nielsen and Chuang 2010, p. 186], and the `ctrl` expression performs a reverse computation to "uncompute" this garbage and produce a reversible circuit. The flag qubits are used as a kind of "assertion," where a nonzero flag qubit corresponds to an error that can be caught by the `try-catch` expression. Our main result, proven in the supplemental report, is summarized by the following theorem:

Theorem 6.1. *There is a recursive procedure that compiles any well-typed Qunity program to a qubit-based circuit consisting only of single-qubit gates and controlled operators. The unitary semantics of this low-level circuit will correspond closely to the (potentially norm-decreasing) semantics of the Qunity program, as made precise by Definitions 6.2 and 6.3 in Section 6.3 below.*

6.1 Overview

Compiling Qunity programs to qubit circuits happens in two stages: from Qunity programs to high-level circuits, and from high-level circuits to low-level qubit circuits. More precisely:

The **input** is a valid typing judgment, as defined in Section 3, for either an expression or program, pure or mixed. We assume that the compiler has access to the proof of validity for the typing judgment, and our compiler (like our semantics) is defined as a recursive function of this proof.

An **intermediate representation** uses quantum circuits of the sort found in most quantum computing textbooks, with two notable exceptions:

(1) The wires in the circuit correspond to the Hilbert spaces defined in Definition 4.5, rather than single qubits. In our circuit diagrams, we will label wires with the Hilbert space they represent and a "slash" pointing to the labeled wire. This is useful because it is easier to analyze circuit semantics if we can algebraically manipulate vectors in the direct sum of Hilbert spaces rather than the corresponding qubit encodings of these vectors. Using wires to represent the direct sum of Hilbert spaces is unconventional, but not unheard of in graphical quantum languages [Chardonnet et al. 2022].

(2) The boxes in the circuit correspond to norm-non-increasing operators and trace-non-increasing superoperators rather than unitary operators and trace-preserving superoperators (CPTP maps). Again, this is not unheard of: Fault-tolerant quantum circuits are often drawn and analyzed with norm-decreasing "r-filter" projectors [Gottesman 2010].

Some of our circuits will be "pure," involving no measurement and described by norm-non-increasing linear operators. Others will be impure, potentially involving measurement and described by trace-non-increasing superoperators. In both cases, a vertical stacking of boxes represents a tensor product, a horizontal stacking of boxes represents function composition, and a bare wire represents the identity. Whenever we use a pure component described by the operator E within an impure circuit, it has superoperator semantics $\rho \mapsto E\rho E^\dagger$.

The **output** is a low-level qubit-based unitary quantum circuit of the sort standard in quantum computing literature. We provide circuit diagrams, and it should be obvious how to implement these circuits in a runnable quantum assembly language such as OpenQASM [Cross et al. 2021]. Here, wire labels indicate the number of qubits in a particular register, and unlabelled wires can be assumed to represent a single qubit. All of these circuits will be "pure" (unitary and measurement-free), and we assume that the controlled gate $|0\rangle\langle 0| \otimes \mathbb{I} + |1\rangle\langle 1| \otimes U$ is implementable whenever the unitary U can be, as this sort of quantum control is a primitive "gate modifier" in OpenQASM.

In what follows, we will refer to our intermediate circuit representation as "high-level circuits" and the target qubit language as "low-level circuits." High-level circuits semantically described by superoperators will be referred to as "decoherence-based," while those that are measurement-free will be referred to as "pure." Section 6.2 outlines the compilation from Qunity programs to high-level circuits, and Section 6.3 outlines the compilation of these high-level circuits to low-level circuits.

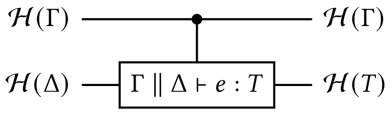
6.2 Qunity Programs to High-Level Circuits

Our compiler, like our denotational semantics, is a recursive function of a valid typing judgment. Sometimes we do not depend on the compiled subcircuit directly, but rather a transformed version of this circuit; we make it clear when we do this and show how to implement these circuit transformers in the supplemental report. For each kind of judgment, we describe below the correspondence between the Qunity semantics and the high-level circuit semantics and give an example of a compiled circuit.

Pure expressions. Given a judgment $(\Gamma \parallel \Delta \vdash e : T)$, the compiler produces a pure circuit with input space $\mathcal{H}(\Gamma) \otimes \mathcal{H}(\Delta)$ and output space $\mathcal{H}(\Gamma) \otimes \mathcal{H}(T)$. The semantics of this circuit C corresponds to Qunity expression semantics in the following way:

$$\langle \sigma, v|\, C\, |\sigma, \tau\rangle = \langle v|\, [\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]\, |\tau\rangle$$

Recall that the typing context $\Gamma \parallel \Delta$ is partitioned in two: left of the $\parallel$ is the "classical" context Γ and right of it is the "quantum" context Δ. The semantics makes the distinction clear – Γ determines the classical parameter to the denotation, while Δ determines input Hilbert space. In the compilation setting, the story is slightly different. Because even classical computation may need to be done reversibly on quantum data, our compiler uses $\mathcal{H}(\Gamma)$ as an auxiliary Hilbert space holding the variables used in a classical way. Practically, "used in a classical way" means that the wires in this ancilla register are used only as "control" wires, so any data on this register remains unchanged in the standard basis. For this reason, we depict these gates in circuit diagrams with a control on the classical Γ register:



Seven inference rules define the pure expression typing relation, and in the supplemental report we give a compiled circuit for each of these cases. As an example, Figure 22 shows the circuit for the T-PurePair typing rule, which includes as subcircuits the compiled subexpressions. It uses a "share" gate graphically represented by the "controlled cloud" component that copies quantum data in the standard basis, mapping $|\tau\rangle \mapsto |\tau, \tau\rangle$. We show how to implement this gate later, in Figure 26.

$$\frac{\Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0 \qquad \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1}{\Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1} \text{ T-PurePair}$$

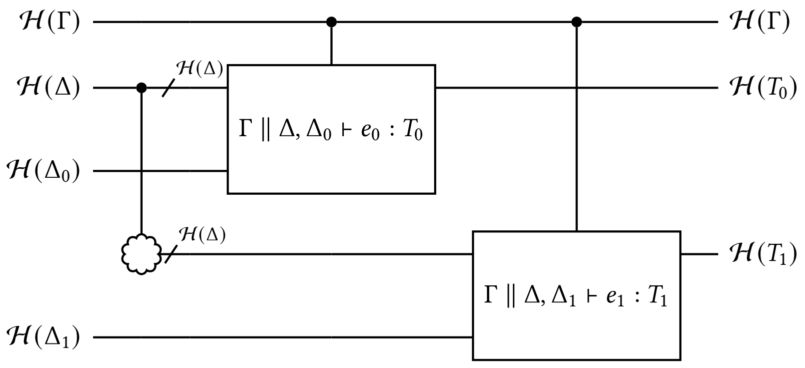


Fig. 22. T-PurePair compilation

Pure programs. Given a judgment $(\vdash f : T \rightsquigarrow T')$, the compiler produces a pure circuit with input space $\mathcal{H}(T)$ and output space $\mathcal{H}(T')$. This circuit's semantics is the same as $[\![\vdash f : T \rightsquigarrow T']\!]$. As an example, Figure 23 shows the circuit for the T-PureAbs typing rule. Note that this circuit does not use the compiled circuit for $(\varnothing \parallel \Delta \vdash e : T)$ directly; rather, we use a transformed version: its *adjoint*. As described in Section 6.3, norm-decreasing operators in our high-level circuits are implemented by unitary operators in our low-level circuits by treating some of the wires as "prep wires" (initialized to zero) and some of the wires as "flag wires" (asserted to be zero upon termination). Whenever a norm-decreasing operator can be implemented, its adjoint can also be implemented by taking the adjoint of the underlying unitary and swapping the prep and flag wires, reversing the circuit and swapping the processes of qubit initialization and qubit termination. We describe this circuit transformer (and others) in more detail in the supplemental report.

$$\frac{\varnothing \parallel \Delta \vdash e : T \qquad \varnothing \parallel \Delta \vdash e' : T'}{\vdash \lambda e \overset{T}{\mapsto} e' : T \rightsquigarrow T'} \text{ T-PureAbs}$$

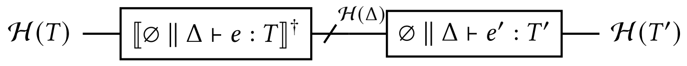

$\mathcal{H}(T)$
$[\![\varnothing \parallel \Delta \vdash e : T]\!]^\dagger$
$\mathcal{H}(\Delta)$
$\varnothing \parallel \Delta \vdash e' : T'$
$\mathcal{H}(T')$

Fig. 23. T-PureAbs compilation

Mixed expressions. Given a judgment ($\Delta \Vdash e : T$), the compiler produces a decoherence-based circuit with input space $\mathcal{H}(\Delta)$ and output space $\mathcal{H}(T)$. This circuit's semantics is the same as $[\![\Delta \Vdash e : T]\!]$. As an example, Figure 24 shows the circuit for the T-MixedPerm typing rule. The π gate here is a series of SWAP gates that permutes the data in Δ according to the permutation function π. This example demonstrates the benefit of the explicit exchange rules—exchange of variables corresponds to SWAP gates in the quantum circuit.

$$\frac{\Delta \Vdash e : T}{\pi(\Delta) \Vdash e : T} \text{ T-MixedPerm}$$

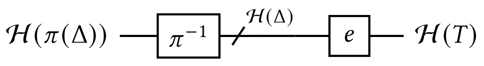


Fig. 24. T-MixedPerm compilation

Mixed programs. Given a judgment ($\vdash f : T \Rrightarrow T'$), the compiler produces a decoherence-based circuit with input space $\mathcal{H}(T)$ and output space $\mathcal{H}(T')$. This circuit's semantics is the same as $[\![\vdash f : T \Rrightarrow T']\!]$. As an example, Figure 25 shows the circuit for the T-MixedAbs typing rule. This is the same as the T-PureAbs circuit, except that there is an extra context Δ_0 for unused variables, which are discarded.

$$\frac{\varnothing \parallel \Delta, \Delta_0 \vdash e : T \qquad \Delta \Vdash e' : T'}{\vdash \lambda e \overset{T}{\mapsto} e' : T \Rrightarrow T'} \text{ T-MixedAbs}$$

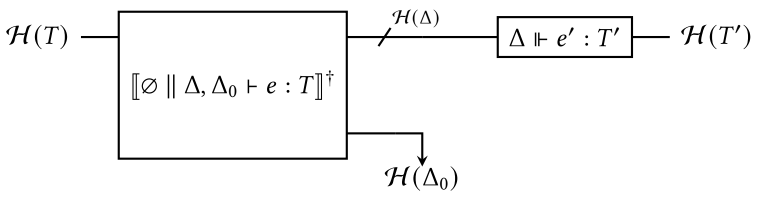


Fig. 25. T-MixedAbs compilation

6.3 High-Level Circuits to Low-Level Circuits

The high-level circuits constructed in the previous section are convenient for analysis, in particular for our proofs of correctness. For these circuits to be runnable on quantum hardware, however, an additional compilation stage is necessary, transforming these high-level circuits into low-level ones. In particular, our high-level circuits manipulate values in $\mathbb{V}(T)$ in Hilbert space $\mathcal{H}(T)$ with

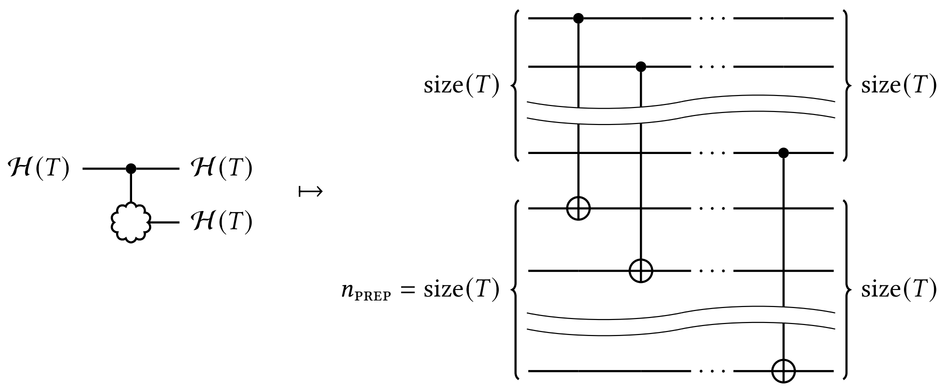


Fig. 26. A (high-level) "share" gate implemented by (low-level) CNOT gates

$\dim(\mathcal{H}(T)) = |\mathbb{V}(T)|$, but low-level circuits use only qubits, working in the Hilbert space $\mathbb{C}^{2^n}$ where n is the number of qubits on the quantum computer. Our compiler must then translate Qunity programs written for the Hilbert space $\mathcal{H}(T)$ into OpenQASM programs that use some $|\mathbb{V}(T)|$-dimensional subspace of $\mathbb{C}^{2^n}$, and values in $\mathbb{V}(T)$ will be encoded into bitstrings in $\{\mathtt{0},\mathtt{1}\}^n$. This number of qubits n is determined from T by a function size(), shown below. Values in $\mathbb{V}(T)$ will end up encoded into bitstrings whose length is size(T) using the function encode(), where "++" denotes bitstring concatenation.

$$\begin{aligned}
\text{size}(\mathtt{Void}) &\coloneqq 0 & \text{encode}(\mathtt{()}) &= \mathtt{""}\\
\text{size}(\mathtt{()}) &\coloneqq 0 & \text{encode}(\mathtt{left}_{T_0 \oplus T_1}\ v) &= \mathtt{"0"} \mathbin{+\!\!+} \text{encode}(v)\\
\text{size}(T_0 \oplus T_1) &\coloneqq 1 + \max\{\text{size}(T_0), \text{size}(T_1)\} & \text{encode}(\mathtt{right}_{T_0 \oplus T_1}\ v) &= \mathtt{"1"} \mathbin{+\!\!+} \text{encode}(v)\\
\text{size}(T_0 \otimes T_1) &\coloneqq \text{size}(T_0) + \text{size}(T_1) & \text{encode}(\mathtt{(}v_0\mathtt{,}v_1\mathtt{)}) &= \text{encode}(v_0) \mathbin{+\!\!+} \text{encode}(v_1)
\end{aligned}$$

We can now be precise about what it means to implement a norm-non-increasing operator with a low-level circuit.

Definition 6.2. We say that it is possible to implement a norm-non-increasing operator $E : \mathcal{H}(T) \to \mathcal{H}(T')$ if there is a low-level circuit implementing a unitary operator $U : \mathbb{C}^{2^{\text{size}(T)+n_{\text{PREP}}}} \to \mathbb{C}^{2^{\text{size}(T')+n_{\text{FLAG}}}}$ for some integers n_{PREP} and n_{FLAG} such that for all $v \in \mathbb{V}(T), v' \in \mathbb{V}(T')$:

$$\left\langle \text{encode}(v'), \mathtt{0}^{\otimes n_{\text{FLAG}}} \middle| U \middle| \text{encode}(v), \mathtt{0}^{\otimes n_{\text{PREP}}} \right\rangle = \langle v' | E | v \rangle$$

Here, n_{PREP} is the number of "prep" qubits initialized to $|\mathtt{0}\rangle$, and n_{FLAG} is the number of "flag" qubits asserted to be in the $|\mathtt{0}\rangle$ state upon termination. This definition requires that size$(T) + n_{\text{PREP}} =$ size$(T') + n_{\text{FLAG}}$.

For example, consider the "share" gate graphically represented by the "controlled cloud" in Figure 22. This gate can be defined for any type T, where it is mathematically represented by the isometry $\sum_{v \in \mathbb{V}(T)} |v, v\rangle\langle v|$, copying a value in the standard basis. By Definition 6.2, one can implement this operator with $n_{\text{PREP}} = \text{size}(T)$ prep qubits and $n_{\text{FLAG}} = 0$ flag qubits, using the series of CNOT gates shown in Figure 26.

Definition 6.2 can be adapted to the setting of decoherence and superoperators by including an additional "garbage" register. This is essentially the Stinespring dilation [Heinosaari and Ziman 2011, p. 186], using the principle of deferred measurement to purify our computation and organize all of our measurements onto one segment of our output.

Definition 6.3. We say that it is possible to implement a trace-non-increasing superoperator $\mathcal{E} : \mathcal{L}(\mathcal{H}(T)) \to \mathcal{L}(\mathcal{H}(T'))$ if there is a qubit circuit implementing a unitary operator $U : \mathbb{C}^{2^{\text{size}(T)+n_{\text{PREP}}}} \to \mathbb{C}^{2^{\text{size}(T')+n_{\text{FLAG}}+n_{\text{GARB}}}}$ for some integers n_{PREP}, n_{FLAG}, and n_{GARB} such that for all $\rho \in \mathcal{L}(\mathcal{H}(T)), v_1', v_2' \in \mathbb{V}(T')$:

$$\left\langle v_1' \middle| \mathcal{E}(\rho) \middle| v_2' \right\rangle = \sum_{b \in \{0,1\}^{n_{\text{GARB}}}} \left\langle \text{encode}(v_1'), 0^{\otimes n_{\text{FLAG}}}, b \middle| U \left(\rho \otimes |0\rangle\langle 0|^{\otimes n_{\text{PREP}}}\right) U^\dagger \middle| \text{encode}(v_2'), 0^{\otimes n_{\text{FLAG}}}, b \right\rangle$$

Like before, n_{PREP} is the number of "prep" qubits initialized to $|0\rangle$, and n_{FLAG} is the number of "flag" qubits asserted to be in the $|0\rangle$ state upon termination. The new parameter n_{GARB} is the number of "garbage" qubits discarded after use. This definition requires that $\text{size}(T) + n_{\text{PREP}} = \text{size}(T') + n_{\text{FLAG}} + n_{\text{GARB}}$.

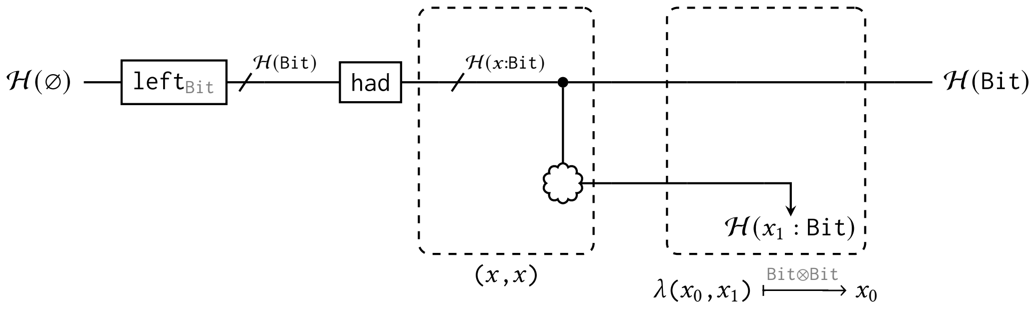
For example, under this definition the "discard" used for the $\mathcal{H}(\Delta_0)$ space in Figure 25 can be implemented by an empty (identity) circuit by setting $n_{\text{GARB}} = \text{size}(T)$. Any implementable pure circuit is also implementable as a decoherence-based circuit by setting $n_{\text{GARB}} = 0$.

6.4 Example

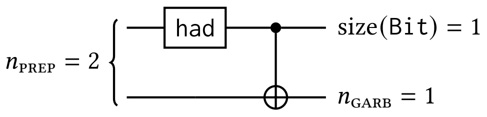
As a concrete example of compiling a simple program, consider a "coin flip" expression defined below. This program is defined in terms of a `meas` program that measures its argument in the standard basis, by copying in the standard basis and then discarding the copy.

$$\begin{aligned} \texttt{meas}_T &\coloneqq \lambda x \overset{T}{\longmapsto} (x, x) \triangleright \texttt{fst}_{\texttt{Bit}\otimes\texttt{Bit}} \\ \texttt{coin} &\coloneqq \texttt{meas}_{\texttt{Bit}}\ (\texttt{had}\ \texttt{0}) \\ &= ()\triangleright \texttt{left}_{\texttt{Bit}} \triangleright \texttt{had} \triangleright \lambda x \xmapsto{\texttt{Bit}} (x, x) \triangleright \lambda(x_0, x_1) \xmapsto{T\otimes T} x_0 \end{aligned}$$

The typing judgment ($\varnothing \Vdash \texttt{coin} : \texttt{Bit}$) is compiled into the following high-level circuit. Here you can see that the (x, x) expression implements the isometry $(|0,0\rangle\langle 0| + |1,1\rangle\langle 1|)$, and the `fst` program implements a partial trace.


$\mathcal{H}(\varnothing)$
$\texttt{left}_{\texttt{Bit}}$
$\mathcal{H}(\texttt{Bit})$
had
$\mathcal{H}(x{:}\texttt{Bit})$
$\mathcal{H}(\texttt{Bit})$
$\mathcal{H}(x_1 : \texttt{Bit})$
(x, x)
$\lambda(x_0, x_1) \xmapsto{\texttt{Bit}\otimes\texttt{Bit}} x_0$

This is then compiled into the following low-level circuit, with $n_{\text{PREP}} = 2$, $n_{\text{FLAG}} = 0$, $n_{\text{GARB}} = 1$, $\text{size}(\varnothing) = 0$ contextual input wires, and $\text{size}(\texttt{Bit}) = 1$ data output wire.



7 CONCLUSION

Qunity is designed to unify classical and quantum computing through an expressive generalization of classical programming constructs. Its syntax allows programmers to write quantum algorithms using familiar classical programming constructs, like exception handling and pattern matching. Our type system leverages algebraic data types and relevant (substructural) types, differentiating between unitary maps and quantum channels but allowing them to be usefully nested. Qunity's semantics brings constructions commonly used in algorithm *analysis*—such as bounded-error quantum subroutines, projectors, and direct sums—into the realm of algorithm *implementation*.

We have formally defined the Qunity programming language, proven that its semantics is well-defined, and shown how it can be used to implement some complicated quantum algorithms. We have demonstrated a strategy for compiling Qunity programs to low-level qubit-based unitary circuits and proven that our procedure preserves the semantics, demonstrating that this language does indeed have a physical interpretation and could be run on quantum hardware. While classical computers can be modeled by logic gates acting on classical bits, it is far more convenient to use higher-level programming constructs for most tasks. Qunity's design similarly abstracts away low-level qubit-based gates using techniques from quantum algorithms that are overlooked in existing languages. We hope that Qunity's features can ease the implementation and analysis of complicated quantum algorithms written at a high level of abstraction.

DATA AVAILABILITY STATEMENT

An extended version of this paper supplements this work with additional background, proofs, and diagrams [Voichick et al. 2022a]. We have implemented and verified a type checker for Qunity programs within the Coq proof assistant [Voichick et al. 2022b].

ACKNOWLEDGMENTS

We thank Andrew Childs for helping us to understand the quantum algorithm designer's perspective and for originally pointing us toward the BQP subroutine theorem, and we thank the anonymous referees for their helpful comments on a draft of this paper. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040, and the Air Force Office of Scientific Research under Grant No. FA95502110051. This work is funded in part by EPiQC, an NSF Expedition in Computing, under award CCF-1730449.

REFERENCES

T Altenkirch and J Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE, Chicago, IL, 249–258. https://doi.org/10.1109/LICS.2005.1 arXiv:quant-ph/0409065

Andris Ambainis, Andrew M Childs, Ben W Reichardt, Robert Špalek, and Shengyu Zhang. 2010. Any AND-OR formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a quantum computer. *SIAM J. Comput.* 39, 6 (2010), 2513–2530. https://doi.org/10.1137/080712167

Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–21. https://doi.org/10.1007/978-3-319-63390-9_1 arXiv:1603.01635 [quant-ph]

Pablo Arrighi and Gilles Dowek. 2017. Lineal: A linear-algebraic Lambda-calculus. *Logical Methods in Computer Science* 13, 1 (March 2017). https://doi.org/10.23638/LMCS-13(1:8)2017

Sheldon Axler. 2015. *Linear Algebra Done Right* (third ed.). Springer, Cham. xvii + 340 pages. https://doi.org/10.1007/978-3-319-11080-6

Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. 1997. Strengths and Weaknesses of Quantum Computing. *SIAM J. Comput.* 26, 5 (1997), 1510–1523. https://doi.org/10.1137/S0097539796300933 arXiv:quant-ph/9701001

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London) *(PLDI 2020)*. Association for Computing Machinery, New York, 286–300. https://doi.org/10.1145/3385412.3386007 https://www.sri.inf.ethz.ch/publications/bichsel2020silq

Costin Bădescu and Prakash Panangaden. 2015. Quantum Alternation: Prospects and Problems. In Proceedings of the 12th International Workshop on *Quantum Physics and Logic,* Oxford, U.K., July 15-17, 2015 *(Electronic Proceedings in Theoretical Computer Science, Vol. 195)*, Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.). Open Publishing Association, Oxford, UK, 33–42. https://doi.org/10.4204/EPTCS.195.3 arXiv:1511.01567 [cs.PL]

Kostia Chardonnet, Marc de Visme, Benoît Valiron, and Renaud Vilmart. 2022. The Many-Worlds Calculus: Representing Quantum Control. arXiv:2206.10234 [cs.LO]

Andrew M Childs, Ben W Reichardt, Robert Spalek, and Shengyu Zhang. 2007. Every NAND formula of size N can be evaluated in time $N^{\frac{1}{2}+o(1)}$ on a quantum computer. arXiv:quant-ph/0703015

Andrew M. Childs and Wim van Dam. 2010. Quantum algorithms for algebraic problems. *Rev. Mod. Phys.* 82 (Jan 2010), 1–52. Issue 1. https://doi.org/10.1103/RevModPhys.82.1 arXiv:0812.0380 [quant-ph]

Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429 [quant-ph] https://github.com/Qiskit/openqasm/tree/OpenQASM2.x

Andrew W Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, John Smolin, Jay M Gambetta, and Blake R Johnson. 2021. OpenQASM 3: A broader and deeper quantum assembly language. arXiv:2104.14722 [quant-ph] https://qiskit.github.io/openqasm/

David Deutsch. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A* 400, 1818 (1985), 97–117. https://doi.org/10.1098/rspa.1985.0070

Peng Fu, Kohei Kishida, and Peter Selinger. 2020. Linear Dependent Type Theory for Quantum Programming Languages. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. Association for Computing Machinery, New York, NY, USA, 440–453. https://doi.org/10.1145/3373718.3394765 arXiv:2004.13472

Daniel Gottesman. 2010. An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation. In *Quantum Information Science and Its Contributions to Mathematics* (Washington, DC) *(Proceedings of Symposia in Applied Mathematics, Vol. 68)*, Samuel J Lomonaco, Jr (Ed.). American Mathematical Society, Providence, RI. https://doi.org/10.1090/psapm/068 arXiv:0904.2557 [quant-ph]

Jonathan Grattage. 2011. An Overview of QML With a Concrete Implementation in Haskell, In Proceedings of the Joint 5th International Workshop on Quantum Physics and Logic and 4th Workshop on Developments in Computational Models (QPL/DCM 2008). *Electronic Notes in Theoretical Computer Science* 270, 1, 165–174. https://doi.org/10.1016/j.entcs.2011.01.015 arXiv:0806.2735 [quant-ph]

Jonathan James Grattage. 2006. *A functional quantum programming language.* Ph. D. Dissertation. University of Nottingham. http://eprints.nottingham.ac.uk/10250/

Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, WA). Association for Computing Machinery, New York, NY, 333–342. https://doi.org/10.1145/2491956.2462177 arXiv:1304.3390 [cs.PL]

Robert B. Griffiths and Chi-Sheng Niu. 1996. Semiclassical Fourier Transform for Quantum Computation. *Phys. Rev. Lett.* 76 (Apr 1996), 3228–3231. Issue 17. https://doi.org/10.1103/PhysRevLett.76.3228 arXiv:quant-ph/9511007

Teiko Heinosaari and Mário Ziman. 2011. *The Mathematical Language of Quantum Theory: From Uncertainty to Entanglement.* Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9781139031103

Gérard Huet. 1980. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM* 27, 4 (Oct. 1980), 797–821. https://doi.org/10.1145/322217.322230

Roshan P. James and Amr Sabry. 2012. Information Effects. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2103656.2103667 https://legacy.cs.indiana.edu/~sabry/papers/information-effects.pdf

Phillip Kaye, Raymond Laflamme, and Michele Mosca. 2007. *An Introduction to Quantum Computing.* Oxford University Press, Oxford. 274 pages.

E Knill. 1996. *Conventions for quantum pseudocode.* Technical Report LA-UR-96-2724. Los Alamos National Lab. https://doi.org/10.2172/366453

Robin Kothari. 2014. *Efficient algorithms in quantum query complexity.* Ph. D. Dissertation. University of Waterloo. http://hdl.handle.net/10012/8625

Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 150 (nov 2020), 29 pages. https://doi.org/10.1145/3428218 arXiv:1911.12855 [cs.PL]

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified Compilation of Quantum Oracles. In *Proceedings of the ACM on Programming Languages*, Vol. 6. Association for Computing Machinery, New York, NY, USA, Article 146, 27 pages. https://doi.org/10.1145/3563309 arXiv:2112.06700 [quant-ph]

M. H. A. Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. https://doi.org/10.2307/1968867

Michael A Nielsen and Isaac L Chuang. 2010. *Quantum Computation and Quantum Information* (10th anniversary ed.). Cambridge University Press, Cambridge, UK. xxxi + 676 pages. https://worldcat.org/en/title/700706156

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. Qwire: A Core Language for Quantum Circuits. In *Proceedings of the 44th Acm Sigplan Symposium on Principles of Programming Languages* (Paris, France). Association for Computing Machinery, New York, NY, 846–858. https://doi.org/10.1145/3009837.3009894 https://www.cis.upenn.edu/~stevez/papers/abstracts.html#PRZ17

John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018). https://doi.org/10.22331/q-2018-08-06-79 arXiv:1801.00862 [quant-ph]

Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: Reasoning about Reversible Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic (QPL), Halifax, Canada, June 3–7, 2018 (Electronic Proceedings in Theoretical Computer Science, Vol. 287)*, Peter Selinger and Giulio Chiribella (Eds.). Open Publishing Association, Waterloo, New South Wales, 299–312. https://doi.org/10.4204/EPTCS.287.17 arXiv:1901.10118 [cs.LO]

Steven Roman. 2008. *Advanced Linear Algebra* (third ed.). Graduate Texts in Mathematics, Vol. 135. Springer, New York. xviii + 525 pages. https://doi.org/10.1007/978-0-387-72831-5

Neil Julien Ross. 2015. *Algebraic and Logical Methods in Quantum Computation.* Ph. D. Dissertation. Dalhousie University. arXiv:1510.02198 [quant-ph] https://hdl.handle.net/10222/60819

Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, Switzerland, 348–364. https://doi.org/10.1007/978-3-319-89366-2_19 arXiv:1804.00952 [cs.LO]

Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586. https://doi.org/10.1017/S0960129504004256 https://www.mathstat.dal.ca/~selinger/papers.html#qpl

Peter Selinger and Benoît Valiron. 2009. Quantum Lambda Calculus. In *Semantic Techniques in Quantum Computation*, Simon Gay and Ian Mackie (Eds.). Cambridge University Press, Cambridge, UK, 135–172. https://doi.org/10.1017/CBO9781139193313.005 https://www.mathstat.dal.ca/~selinger/papers/papers/#qlambdabook

John van de Wetering. 2020. ZX-calculus for the working quantum computer scientist. arXiv:2012.13966 [quant-ph]

André van Tonder. 2004. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, 5 (2004), 1109–1135. https://doi.org/10.1137/S0097539703432165 arXiv:quant-ph/0307150

Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2022a. Qunity: A Unified Language for Quantum and Classical Computing (Extended Version). arXiv:2204.12384 [quant-ph]

Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2022b. Qunity: A Unified Language for Quantum and Classical Computing (Type Checker). https://doi.org/10.5281/zenodo.7150282 https://gitlab.umiacs.umd.edu/finn/qunity/

David Walker. 2004. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C Pierce (Ed.). The MIT Press, Cambridge, MA.

John Watrous. 2009. Quantum Computational Complexity. In *Encyclopedia of Complexity and Systems Science*, Robert A Meyers (Ed.). Springer, New York, NY, 7174–7201. https://doi.org/10.1007/978-0-387-30440-3_428 arXiv:0804.3401 [quant-ph]

W K Wootters and W H Zurek. 1982. A single quantum cannot be cloned. *Nature* 299 (1982), 802–803. https://doi.org/10.1038/299802a0

Mingsheng Ying. 2016. *Foundations of Quantum Programming.* Elsevier Science, Cambridge, MA. xii + 357 pages. https://worldcat.org/en/title/1027777388

Received 2022-07-07; accepted 2022-11-07