Compositional Quantum Control Flow with Efficient Compilation in Qunity

MIKHAIL MINTS, California Institute of Technology, USA
FINN VOICHICK, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA
ROBERT RAND, University of Chicago, USA

Most existing quantum programming languages are based on the quantum circuit model of computation, as higher-level abstractions are particularly challenging to implement—especially ones relating to quantum control flow. The Qunity language, proposed by Voichick et al., offered such an abstraction in the form of a quantum control construct, with great care taken to ensure that the resulting language is still realizable. However, Qunity lacked a working implementation, and the originally proposed compilation procedure was very inefficient, with even simple quantum algorithms compiling to unreasonably large circuits.

In this work, we focus on the efficient compilation of high-level quantum control flow constructs, using Qunity as our starting point. We introduce a wider range of abstractions on top of Qunity's core language that offer compelling trade-offs compared to its existing control construct. We create a complete implementation of a Qunity compiler, which converts high-level Qunity code into the quantum assembly language OpenQASM 3. We develop optimization techniques for multiple stages of the Qunity compilation procedure, including both low-level circuit optimizations as well as methods that consider the high-level structure of a Qunity program, greatly reducing the number of qubits and gates used by the compiler.

CCS Concepts: • **Software and its engineering** → Syntax; *Semantics*; **Compilers**; Interpreters; Preprocessors; Functional languages; Data types and structures; **Control structures**; *Procedures, functions and subroutines*; *Patterns*; Domain specific languages; • **Theory of computation** → *Quantum computation theory*; **Control primitives**; *Denotational semantics.*

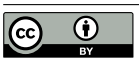
Additional Key Words and Phrases: quantum programming languages, high-level programming languages, quantum control flow, quantum subroutines, compiler optimizations

1 Introduction

In recent years, many *high-level quantum programming languages* have been proposed, aiming to allow algorithm implementers to work at a higher level of abstraction compared to the quantum circuit model of computation. Languages such as QML [1], Silq [3], Tower [37], and Qunity [31] walk a fine line between providing constructs familiar from classical computing and being realizable in quantum hardware. High-level quantum control flow constructs are particularly challenging since quantum programs must be compilable to fixed-length quantum circuits generated by classical computation [38]. As a result, proposed high-level quantum languages offer branching statements with significant restrictions on the expressive powers of their individual branches.

Among these languages, Qunity stands out for its unique handling of control flow that emphasizes compositionality. Qunity naturally extends classical programming constructs into the domain of quantum computation and admits a *compositional denotational semantics* defined in terms of quantum operators (acting on state vectors) and superoperators (acting on density matrices), allowing for more flexible design than what is possible with unitary gate-based quantum circuits.

Authors' Contact Information: Mikhail Mints, California Institute of Technology, Pasadena, USA, mmints@caltech.edu; Finn Voichick, University of Maryland, College Park, USA, finn@umd.edu; Leonidas Lampropoulos, University of Maryland, College Park, USA, leonidas@umd.edu; Robert Rand, University of Chicago, Chicago, USA, rand@uchicago.edu.



Qunity's `ctrl` construct offers a generalization of pattern matching that can coherently control on the output of an arbitrary (possibly irreversible) quantum or classical computation, relying on the BQP subroutine theorem [33]. The construct matches the scrutinee against a set of reversible classical patterns and outputs a superposition of the corresponding outcomes, provided that certain conditions (like the orthogonality of the patterns and the consistent erasure of quantum variables) hold. Consider, for example, Deutsch's algorithm [8] as written in abstract Qunity syntax:

$$\begin{array}{l}\texttt{deutsch}(f) \coloneqq \\ \quad \texttt{let } x = _{\text{Bit}} (\texttt{had } \texttt{0}) \texttt{ in} \\ \quad \left(\texttt{ctrl } (f\ x) \ {}_{\text{Bit}}\left\{\begin{array}{l}\texttt{0} \mapsto x \\ \texttt{1} \mapsto x \triangleright \texttt{gphase}_{\text{Bit}}(\pi)\end{array}\right\}_{\text{Bit}}\right) \triangleright \texttt{had}\end{array}$$

Given an *arbitrary* quantum oracle f that takes in and outputs a single qubit, `deutsch` will test whether f is constant or not using only a single evaluation (rather than two as in classical computing). Unlike in other quantum programming languages, Qunity's quantum control flow allows the programmer to write subroutines in a style similar to classical programming, without having to explicitly translate irreversible programs into unitary circuit forms. For instance, to input the constant-one oracle into the above example of Deutsch's algorithm, the programmer can just use $\lambda x \overset{\text{Bit}}{\longmapsto} 1$ instead of constructing a reversible map of the form $U_f \left|x, y\right\rangle = \left|x, y \oplus f(x)\right\rangle = \left|x, y \oplus 1\right\rangle$ as is typical in circuit-style programming. However, Qunity's type system places several constraints on the expressions in the `ctrl` block to ensure its realizability: specifically, the right-hand-side patterns must *erase* the quantum variables in the scrutinee expression in a consistent way so that the compiler can perform automatic uncomputation.

To address these limitations, we introduce two new pattern matching constructs to Qunity, which trade off the subroutine capabilities of `ctrl` for greater flexibility in designing pure and mixed quantum computations. The new `match` construct corresponds to classical or mixed pattern matching and has minimal restrictions, allowing the programmer to easily write classical logic (which may then become a quantum subroutine). By contrast, `pmatch` (or pure match) allows for symmetric pattern matching between orthogonal sets of pure expressions, similarly to Sabry et al. [25]. This allows us to easily express reversible quantum programs without the erasure restrictions of `ctrl` and simultaneously aids in compiling Qunity code to efficient circuits.

Compiling Qunity is a key contribution of this work. Qunity's high degree of abstraction from the quantum circuit model of computation makes compilation particularly challenging. In addition to control flow and pattern matching, Qunity also generalizes the notion of error handling through `try/catch` statements (which can be viewed as projective measurements) and provides support for quantum sum types (whose semantics corresponds to direct sums of Hilbert spaces). To represent these abstractions, the compiler must encode Qunity types and contexts into quantum registers and allocate ancillary qubits to express Qunity's non-unitary semantics in terms of unitary gates, a process which can lead to significant inefficiencies. While Voichick et al. describe an algorithm for converting Qunity into low-level qubit circuits, this procedure is more of a proof-of-realizability than a practical compiler: even simple quantum algorithms tend to be compiled into prohibitively large circuits. In this work, we develop a practical Qunity compiler, incorporating optimizations at various stages of the compilation process that significantly reduce the number of qubits and gates used in the final compiled circuits.

We begin by providing technical background on Qunity (Section 2), followed by a series of examples to introduce the intricacies of the language. Then, we make the following contributions:

- We develop the first implementation of the Qunity language. We add a surface syntax to the core Qunity language, allowing the user to define parameterized types and subroutines that may involve recursion and higher-order operations (Section 3).
- We introduce new pattern-matching constructs into the core Qunity language, to avoid the restrictions imposed by the quantum control construct and improve the language's expressiveness (Section 4).
- We implement the first working Qunity compiler that converts high-level Qunity code into low-level quantum circuits in OpenQASM 3, as well as a Qunity interpreter (Section 5).
- We design optimizations for the compilation procedure to reduce the number of qubits and gates used in the final compiled circuit, acting at several stages of the compilation process (Section 6 and Section 7).

2 Qunity's Goals, Syntax, and Typing

In this section, we discuss the central ideas and motivations behind the construction of Qunity's type system and semantics, and we will introduce our new surface syntax.

2.1 Unified Programming in Qunity

Qunity's primary goal is a *unified* treatment of quantum and classical computation. Traditional models such as Knill's quantum random access machine (QRAM) [17] and dynamic circuits [6] have a classical computer constructing and running quantum circuits, using the measurement results in classical control flow to determine what circuits to run next. This reflects the paradigm of "quantum data, classical control" [26], creating a clear separation between classical and quantum components of an algorithm.

On the other hand, Qunity aims to blur the line between "classical" and "quantum" as much as possible, introducing quantum language constructs that generalize classical ones while emphasizing compositionality. While Qunity still allows, in principle, to compile Qunity programs to dynamic circuits, it is not tailored for quantum algorithms that rely heavily on classical computation (e.g. ones that require floating point manipulation such as variational hybrid quantum-classical algorithms [21] and quantum machine learning [32]). Instead, Qunity is designed to support the implementation of complex quantum algorithms that operate on a level of abstraction above the circuit model, require manipulating complex data structures in quantum superposition, or use irreversible programs as subroutines in a reversible quantum computation.

2.2 Surface Syntax

Qunity's core language does not support higher-order functions and recursion, due to theoretical limitations on ways in which these ideas can be generalized to the quantum setting [38]. Furthermore, Qunity's core syntax does not allow subroutines to be named and called in a program multiple times. For these reasons, we augment Qunity with a *surface syntax* which provides a layer of *metaprogramming* on top of the core Qunity language. This system allows the user to create parameterized types, expressions, programs, and real number expressions that get evaluated at compile time during the *preprocessing* stage. User-defined types can be written as *variants* with named *constructors*, which are evaluated into the core language's left and right injections during the preprocessing stage. The full grammar of the surface syntax, as used in Qunity source files, can be found in Appendix B.

For concreteness, let us revisit the Deutsch example from the introduction and present it in terms of the new surface syntax:

```
def $deutsch{@f : Bit → Bit} : Bit :=
  let x = $plus in
  ctrl @f(x) [
    $0 → x;
    $1 → x ▷ gphase{pi}
  ]
  ▷ @had
end
```

In this syntax, the symbols `$`, `@`, and `#` are used as sigils to write names for Qunity expressions, programs, and numbers, respectively. Qunity types have names starting with a capital letter, and quantum variables (which are part of the core syntax and remain after the preprocessing stage) start with a lowercase letter or underscore. The syntax `▷` is a shorthand for function application: `x ▷ @f` is equivalent to `@f(x)`. Additionally, `let x = y in z` is syntactic sugar for `(lambda x → z)(y)`. Global phase (`gphase`) can be defined as syntactic sugar over the primitive `rphase` (relative phase), with `gphase{r}` equivalent to `rphase{_, r, r}`.

The type `Bit` is our first example of a user-defined datatype, which is provided by the Qunity standard library:

```
type Bit := $0 | $1 end
```

Here, `Bit` is defined as a variant type with two constructors that do not take any arguments. When the preprocessor evaluates this into the base Qunity syntax, it converts the variant types into sum types and the constructors into left and right injections. So, `$0` becomes $\texttt{left}_{\texttt{Unit}\oplus\texttt{Unit}}\texttt{()}$, and `$1` becomes $\texttt{right}_{\texttt{Unit}\oplus\texttt{Unit}}\texttt{()}$. We can also define `@had` and `$plus` to represent the Hadamard gate and the $|+\rangle$ state:

```
def @had : Bit → Bit := u3{pi/2, 0, pi} end
def $plus : Bit := @had($0) end
```

where `u3` is a Qunity primitive for general single-qubit gates with the type $\texttt{Unit} \oplus \texttt{Unit} \rightsquigarrow \texttt{Unit} \oplus \texttt{Unit}$ in the core Qunity language.

2.3 Typing and Semantics

Qunity has two distinct typing judgments for expressions: *pure* expression typing and *mixed* expression typing. These correspond to two distinct but interrelated semantics: a pure semantics that is defined in terms of norm non-increasing operators acting on state vectors and a mixed semantics that is defined in terms of trace non-increasing superoperators acting on density matrices.

We write $\Gamma \parallel \Delta \vdash e : T$ to indicate that expression e has pure type T with respect to classical context Γ and quantum context Δ. Similarly, we write $\Gamma \parallel \Delta \Vdash e : T$ (using $\Vdash$ instead of $\vdash$) to indicate that e has *mixed type* T. The classical contexts here are not to be viewed as literally containing classical variables—a variable being in the classical context indicates that it can only be accessed by copying it in the classical basis, and, unlike quantum variables, its relevance is not enforced by the type system. Appendix A.2 contains a more detailed discussion of the role of these contexts.

We write $\vdash f : T \rightsquigarrow T'$ to indicate that the program f is typed as a *coherent map* from T to T', and we write $\vdash f : T \Rrightarrow T'$ to indicate that f is typed as a *quantum channel*. Coherent maps have operator semantics, acting on pure states, while quantum channels act on density matrices.

The semantics of a pure expression corresponds to a linear map $[\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!] \in \mathcal{L}(\mathcal{H}(\Delta), \mathcal{H}(T))$, sending quantum states in the Hilbert space $\mathcal{H}(\Delta)$ associated with the quantum context Δ to states in the space $\mathcal{H}(T)$ associated with the type T (see Appendix A.3 for definitions

of these Hilbert spaces). Similarly, $[\![\vdash f : T \rightsquigarrow T']\!] \in \mathcal{L}(\mathcal{H}(T), \mathcal{H}(T'))$ sends states from the space $\mathcal{H}(T)$ to $\mathcal{H}(T')$. Pure expressions and coherent maps are used to represent *reversible* quantum computation, which does not discard quantum information and has a well-defined *adjoint*. Note that "reversible" does not necessarily mean "invertible" and the adjoint of an operator representing the semantics of a pure expression or coherent map does not need to be its inverse. Thus, unlike in the quantum circuit model and languages such as SPM [25], these linear maps do not necessarily need to be unitary. Instead, they are restricted to the much broader class of *contractions* (norm non-increasing operators), which includes *projectors* and non-unitary *isometries*. Consider, for example, the expression `lambda $0 → $0`, which has the operator semantics of a projector:

$$|0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

Using `$0` as a pattern in the `lambda` effectively creates an assertion that the input is the $|0\rangle$ state. If `$1` is given as input, the result is 0 (the zero vector), signifying an "error state" or an "exception" being thrown and if `$plus` (whose semantics is $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$) is given as input, the result is $\frac{1}{\sqrt{2}}|0\rangle$, which can be viewed as a quantum superposition of the assertion succeeding and failing.

Despite the freedom gained by not restricting pure semantics to unitaries, Qunity's type system ensures that pure expressions and programs never discard quantum information, placing relevance constraints on variables in quantum contexts. However, sometimes maintaining reversibility is unnecessary and requires a large amount of tedious bookkeeping. For instance, if a programmer wishes to implement an AND gate reversibly, they would need to implement a 3-bit operation such as $(a, b, c) \mapsto (a, b, (a \wedge b) \oplus c)$, corresponding to the Toffoli gate. Keeping track of this extra data can be inconvenient and unnecessary. Mixed typing and semantics allow the Qunity programmer to create decoherence by irreversibly discarding quantum information when necessary. The semantics of mixed expressions and quantum channels is described by *trace non-increasing superoperators* acting on the space of *density matrices*. In quantum mechanics, density matrices describe the state of an open quantum system that has interacted with its environment. Quantum algorithm designers often work with both the state vector and density matrix formalisms, and the design of the Qunity language allows both to be implemented in a convenient way. Consider, for example, the Qunity program `lambda x → (x, x) ▷ lambda (x, y) → x`. Here, `lambda x → (x, x)` implements the isometry $|0,0\rangle\langle 0| + |1,1\rangle\langle 1|$,[1] while `lambda (x, y) → x` cannot be typed as a coherent map since it introduces decoherence by discarding the variable y. Taken as a whole, the program shares `x` along the classical basis and discards the new copy of it, which effectively performs a *measurement*. If the pure state $|+\rangle$ is input into this program, it transforms into the maximally mixed state $\frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|)$, which is a probabilistic mixture (rather than a quantum superposition) of $|0\rangle$ and $|1\rangle$.

Finally, Qunity's `try/catch` construct can be viewed as performing a measurement to determine whether an exception has occurred and performing another computation if it has. This can convert a norm-decreasing operator into a trace-preserving superoperator. For example, the Qunity expression `try $plus ▷ lambda $0 → $0 catch $plus` corresponds to the density matrix

$$\frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|+\rangle\langle +| = \begin{bmatrix} 3/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}.$$

A density matrix can always be viewed as a partial trace of a pure state on a larger system, and a quantum channel can always be represented as the application of an isometry followed by a partial trace using the Stinespring dilation [14].

[1]Note that Qunity allows variables to be reused: This does not violate the no-cloning theorem [34], since this corresponds to *sharing* a state along the classical basis (an operation that creates entanglement) rather than *cloning* it.

The design of Qunity's type system and semantics thus allows for a rich interplay between pure and mixed programming, allowing concepts prominent in quantum algorithm *analysis* to be easily extended into the realm of quantum algorithm *implementation*.

3 Qunity by Example

In this section, we demonstrate several examples of quantum algorithms implemented in Qunity. These examples showcase the power of Qunity's compositional quantum control flow constructs: `ctrl`, as well as the new `pmatch` and `match` constructs that we introduce to overcome some of the limitations of `ctrl`. Section 4 will discuss these in more technical detail.

3.1 Order Finding

The following example is a simplified demonstration of the order finding algorithm, which is the quantum part of Shor's algorithm for integer factorization [27]. Given relatively prime integers N and $a < N$, the goal is to find an integer r such that $a^r \equiv 1 \pmod{N}$. For the purposes of this demonstration, we will assume that N is a power of two – the resulting program is not useful for Shor's algorithm (as it assumes N is *odd* and orders modulo 2^n are easy to calculate classically [16]), but it is simpler to implement and illustrative of Qunity's features.

In our surface language, we can define a datatype of arrays of a given length, and then represent numbers `Num{#n}` in little-endian form using this datatype:

```
type Array{#n, 'a} := if #n <= 0 then Unit
                      else 'a * Array{#n - 1, 'a} endif end
type Num{#n} := Array{#n, Bit} end
```

Using this representation, we can write the following code to add a fixed number `#a` to a quantum `#n`-bit number (overflowing so that the addition is modulo $2^{\#n}$) in a reversible way:

```
def @add_const{#n, #a} : Num{#n} → Num{#n} :=
  if #n <= 0 then
    @id{Unit}
  else
    pmatch [
      ($0, x) → (if #a % 2 = 0 then $0 else $1 endif,
                 x ▷ @add_const{#n - 1, (#a-#a%2)/2});
      ($1, x) → (if #a % 2 = 0 then $1 else $0 endif,
                 x ▷ @add_const{#n - 1, (#a-#a%2)/2 + #a%2})
    ]
  endif
end
```

This showcases the use of the newly added `pmatch` construct. This form of pattern matching, which has *pure program* semantics, allows us to reversibly transform between two orthogonal sets of Qunity expressions. The type system imposes orthogonality requirements (Appendix F) on *both* sides of the pattern-matching block in `pmatch`, while `ctrl` only imposes them on the left-hand side patterns. However, since `pmatch` does not need to perform automatic uncomputation, it avoids the *erasure judgment* (Appendix H) required by `ctrl`, so the use of variables on the right-hand side of the pattern matching block is much less restricted in this respect. It would be difficult to define this operation using only the `ctrl` construct.

For this example, `@add_const` is defined recursively, using versions of itself with `#n - 1` and different values of `#a` depending on whether there is a carry bit. It is clear that `@add_const{0, #a}` has unitary semantics since it is the identity map. Now, for any integer `#n`, assuming that `@add_const{#n - 1, #a}` is unitary, the typechecker will correctly type the `pmatch` statement and conclude that `@add_const{#n, #a}` is also unitary because the two sets of orthogonal expressions *span* their respective spaces. This ensures that the orthogonality requirements are satisfied.

We can now use constant addition to define modular multiplication by an odd constant `#a` modulo $2^{\#n}$, using the approach in Gidney [10]:

```
def @mod_mult{#n, #a} : Num{#n} → Num{#n} :=
  if #n = 1 then
    @id{Num{#n}}
  else
    lambda (x0, x1) →
    let (x0, x1) = (x0, x1 ▷ @mod_mult{#n - 1, #a}) in
    ctrl x0 [
      $0 → (x0, x1);
      $1 → (x0, @add_const{#n - 1, (#a - 1) / 2}(x1))
    ]
  endif
end
```

This procedure works because if a is odd, then

$$(1+2x)a \equiv 1 + 2xa + a - 1 \equiv 1 + 2\left(xa + \frac{a-1}{2}\right)$$

where all addition is modulo 2^n. Now, we can define a modular exponentiation program that implements the operation $|x, y\rangle \mapsto |x, y \cdot a^x\rangle$, where we use `ctrl` to coherently condition on the bits of x and apply a modular multiplication by the current power of `#a` (note that now we cannot use `pmatch` since the RHS is not necessarily orthogonal):

```
def @mod_exp{#m, #n, #a} : Num{#m}*Num{#n} → Num{#m}*Num{#n} :=
  if #m = 0 then @id{Num{#m} * Num{#n}}
  else
    lambda ((x0, x1), y) → let ((x0, x1), y) = ctrl x0 [
      $0 → ((x0, x1), y);
      $1 → ((x0, x1), @mod_mult{#n, #a}(y)) ]
    in let (x0, (x1, y)) =
      (x0, @mod_exp{#m - 1, #n, #a * #a}(x1, y))
    in ((x0, x1), y)
  endif
end
```

With all the above machinery in place, we can finally define the main order finding procedure:

```
def $order_finding{#n, #a} : Num{#n} :=
  ($repeated{#n, Bit, $plus}, $num_to_state{#n, 1})
  ▷ @mod_exp{#n, #n, #a}
  ▷ @fst{Num{#n}, Num{#n}}
  ▷ @adjoint{Num{#n}, Num{#n}, @qft{#n}}
  ▷ @reverse{#n, Bit}
end
```

The order finding procedure starts with a uniform superposition of the values of x and $y = 1$, applies the modular exponentiation, then discards the second register (`@fst` is defined as just `lambda (x, y) → x`) and applies the inverse quantum Fourier transform (see Appendix D.2 for the implementation), reversing the result to display it in big-endian. Measuring the output will result in a random number of the form Nj/r for some integer j. For instance, running `$order_finding{5, 13}` would output random multiples of 4, since the order of 13 modulo 2^5 is 8 and $2^5/8 = 4$. The procedure relies on the reversibility of the operations and the pure typing judgment to maintain quantum coherence before applying the inverse quantum Fourier transform.

3.2 Grover's Algorithm

An interesting example of the use of `ctrl` is Grover's search algorithm [12]: Given any quantum oracle that takes in a value of some type and outputs a bit, we can start from a uniform superposition of the type and amplify the amplitude of states for which the oracle outputs 1. The generalized code for Grover's algorithm in Qunity is as follows:

```
def @grover_iter{'a, $equal_superpos : 'a,
                 @f : 'a → Bit} : 'a → 'a :=
  lambda x → ctrl @f(x) [
    $0 → x;
    $1 → x ▷ gphase{pi}
  ] ▷ @reflect{'a, $equal_superpos}
end

def $grover{'a, $equal_superpos : 'a,
            @f : 'a → Bit, #n_iter} : 'a :=
  if #n_iter = 0 then
    $equal_superpos
  else
    $grover{'a, $equal_superpos, @f, #n_iter - 1} ▷
    @grover_iter{'a, $equal_superpos, @f}
  endif
end
```

Here `@grover_iter` takes in a type parameter `'a`, a user-provided expression `$equal_superpos` expected to create an equal superposition of all values of type `'a` (as there is no way to do this generically), and an oracle `@f : 'a → Bit` that is the function input to Grover's algorithm. The result is a program that transforms inputs of type `'a`, bringing them closer to the input that satisfies the oracle. Then, `$grover` is a parameterized expression rather than a program: it becomes a Qunity

expression after the parameters are substituted in the preprocessing stage. It repeatedly applies `@grover_iter` to a state that begins as an equal superposition, for a given number of iterations. Observe the use of the `ctrl` construct in this example: this code works with any function `@f`, including an irreversible one. This allows us to define quantum oracles for Grover's algorithm in a completely classical way, without ever needing to worry about the reversibility of the computations.

The construction of such subroutines is greatly facilitated by the newly added `match` construct - a form of mixed or irreversible pattern matching that has minimal restrictions, functioning like the matching constructs in classical programming languages. Consider, for instance, the following example of a function testing whether a list of bits has an odd number of ones:

```
def @is_odd_sum{#n} : List{#n, Bit} → Bit :=
  if #n = 0 then
    lambda l → $0
  else
    lambda l → match l [
       $ListEmpty {#n, Bit} → $0;
       @ListCons {#n, Bit}($0, l') → @is_odd_sum{#n - 1}(l');
       @ListCons {#n, Bit}($1, l') → @not(@is_odd_sum{#n - 1}(l'))
    ]
  endif
end
```

Here, the `List` datatype is a list of variable but bounded length, defined as:

```
type List{#n, 'a} :=
  |  $ListEmpty
  |  @ListCons  of (if #n <= 0 then Void
                    else 'a * List{#n - 1, 'a} endif)
end
```

The definition of `@is_odd_sum` is typed as a mixed program, with the `match` block typed as a mixed expression. It is defined recursively, using `@is_odd_sum{#n - 1}` in the RHS of the pattern-matching block. This means that it would be difficult to write this in terms of just the `ctrl` construct, which types its RHS expressions as pure. Since `match` has mixed semantics, it is not directly responsible for performing any uncomputation and is able to discard quantum data: hence, it can avoid the erasure requirements present in `ctrl`.

Now, we can use this as an oracle in Grover's algorithm simply by writing

```
def #n := 2 end
$grover{List{#n, Bit}, $equal_superpos_list{#n},
        @is_odd_sum{#n}, 3}
```

where `$equal_superpos_list` is an expression generating an equal superposition of all possible lists of bits with lengths bounded by `#n` (implementation in Appendix D.7). This code uses Grover's algorithm to amplify the probability of measuring the lists of bits (of length at most 2) the sum of whose elements is odd.

4 Generalizing Qunity's Control Flow

Qunity's `ctrl` construct is very powerful, as it allows an arbitrary quantum computation, potentially involving decoherence, to be used as a subroutine in a pure, reversible computation. However, this

feature is accompanied by some notable restrictions that can make this construct more difficult to use than necessary.

The first significant restriction is that the left-hand-side (LHS) expressions of the `ctrl`-block must be classical—that is, they cannot include any invocations of `u3` or `rphase`, and so their semantics correspond to classical basis states. The second restriction arises from the requirement that the right-hand-side (RHS) expressions must satisfy the *erasure judgment*, which stipulates that all variables in the quantum context Δ of the scrutinee expression must be present "in the same way" in each of the RHS expressions. If, for instance, Δ involves the variable x, this essentially forces the programmer to make all the RHS expressions have the form $(x, \dots)$ if they want to output something other than just x itself. See Appendix H for the definition of the erasure inference rules.

The erasure requirement is essential to ensure that it is possible to implement `ctrl`: without it, the semantics of `ctrl` will not be reversible. At the circuit level, the context Δ needs to be shared to produce the RHS expressions, but applying the adjoint of the purified scrutinee expression inevitably creates another copy of $\mathcal{H}(\Delta)$ alongside the output $\mathcal{H}(T')$, and the erasure judgment is needed to coherently delete this. The classical requirement in the orthogonality judgment does not seem as obvious; indeed, applying an isometric transformation to an orthogonal set of states maintains their orthogonality. However, dropping the classical requirement from `ctrl` typing would allow for non-physical norm-*increasing* semantics. For instance, using the definition of `ctrl`'s semantics from Appendix A.3 on the expression

```
$0 ▷ lambda x → ctrl x [$plus → x; $minus → x]
```

would result in the physically impossible state $\sqrt{2}\,|0\rangle$ (with norm $\sqrt{2} > 1$). The correctness of the circuit construction in Appendix L.3 relies on this classical-basis assumption.

It would be useful to have a construct that defines a reversible transformation between two orthogonal sets of expressions, which do not necessarily need to be in the classical basis, while avoiding the erasure requirement. Also, if the programmer is implementing classical logic (which may be part of a larger quantum algorithm), it may be useful to have a construct that defines *irreversible* control flow in a manner most similar to a `match` statement in classical programming languages, again without the restrictions of the erasure judgment. These ideas give rise to two new Qunity primitives: `pmatch` and `match`.

Table 1. Comparison of Qunity's three pattern-matching constructs (extended version in Appendix C).

Feature	`ctrl`	`match`	`pmatch`
Purely typed	✓	✗	✓
Mixed scrutinee allowed	✓	✓	✗
Mixed expressions allowed on RHS	✗	✓	✗
Non-orthogonal expressions allowed on RHS	✓	✓	✗
Non-classical expressions allowed on LHS	✗	✗	✓
Avoids erasure requirements	✗	✓	✓
Can be used in the RHS of a `ctrl`	✓	✗	✓
Can be used in the RHS of a `match`	✓	✓	✓
Can be used in the LHS or RHS of a `pmatch`	✓*	✗	✓*

*if the isometry judgment holds

4.1 The pmatch Construct

The pmatch construct (the "p" stands for "pure") has a design that is very similar to that of the symmetric pattern-matching language (SPM) described by Sabry et al. [25]. We can write a typing judgment for pmatch as follows:

$$\frac{\begin{array}{ll}\mathrm{ortho}_T\ (e_1, \ldots, e_n) & \varnothing \parallel \Delta_j \vdash e_j : T\ \forall j \\ \mathrm{ortho}_{T'}\ (e'_1, \ldots, e'_n) & \varnothing \parallel \Delta_j \vdash e'_j : T'\ \forall j\end{array}}{\vdash \mathtt{pmatch}\ {}_T\left\{\begin{array}{c} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{array}\right\}_{T'} : T \rightsquigarrow T'}\ \textsc{T-Pmatch}$$

Each pattern and branch must be appropriately typed using the same quantum context Δ_j, and both the LHS and RHS must satisfy the orthogonality judgment (Appendix F). If the classical requirement as in ctrl was still in place, then pmatch[2] could be described as syntactic sugar over the ctrl construct, using a technique known as *specialized erasure* [31]. Instead, we make pmatch more general by allowing expressions in an arbitrary basis on both sides. This is done at the expense of losing some of the capabilities of ctrl: since the semantics of pmatch is that of a pure program rather than a pure expression, it does not include a scrutinee directly and thus does not have the power to uncompute arbitrary mixed expressions via purification. It also has a requirement dictating that the same contexts must be used symmetrically in the LHS and RHS, making it possible to simply flip the two sides to invert the program.

We can describe the semantics of pmatch as:

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

Each term of the above sum is formed by applying the adjoint of the semantics of an LHS expression, $[\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T]\!]^\dagger : \mathcal{H}(T) \to \mathcal{H}(\Delta_j)$, followed by an application of the corresponding RHS expression $[\![\varnothing : \varnothing \parallel \Delta_j \vdash e'_j : T']\!] : \mathcal{H}(\Delta_j) \to \mathcal{H}(T')$. The images of the operators $[\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T]\!]$ correspond to a set of orthogonal subspaces of $\mathcal{H}(T)$. In the special case where all the LHS expressions are classical and $|v\rangle$ is a classical basis state, at most one of the terms in the sum will be nonzero. The variables in the pattern will be extracted from the input, and the result will be the RHS expression applied to their values. In general, all the branches of the pmatch block can be taken in superposition. If the LHS patterns are *spanning* (Appendix F), then pmatch will act as an *isometry*, as every input can be described as a linear combination of the patterns. If the RHS expressions are also spanning, then the semantics of pmatch will correspond to a *unitary* operator.

For an example of the benefit provided by pmatch, consider the following Qunity code which prepares an equal superposition of Maybe{Bit}, which is isomorphic to a qutrit:

```
$0
▷ u3{2 * arccos(sqrt(1 / 3)), 0, 0}
▷ pmatch [
  $0 → $Nothing {Bit};
  $1 → @Just {Bit}($plus)
]
```

[2]Called "match" by Voichick et al., but we change the terminology as match will be a different construct.

Here, we are easily able to transform between two orthogonal bases of expressions, one in the type `Bit` and the other in the type `Maybe{Bit}`. The semantics of this can be written as:

$$
\begin{aligned}
&[\![\texttt{pmatch}\{\dots\}]\!][\![\texttt{u3}(\arccos(\sqrt{1/3}, 0, 0))]\!]\,|\texttt{\$0}\rangle = \\
&= \left([\![\texttt{Nothing\{Bit\}}]\!][\![\texttt{\$0}]\!]^\dagger + [\![\texttt{@Just\{Bit\}}]\!][\![\texttt{\$plus}]\!][\![\texttt{\$1}]\!]^\dagger\right)\left(\frac{1}{\sqrt{3}}\,|\texttt{\$0}\rangle + \frac{\sqrt{2}}{\sqrt{3}}\,|\texttt{\$1}\rangle\right) = \\
&= \left(|\texttt{Nothing\{Bit\}}\rangle\langle\texttt{\$0}| + \frac{1}{\sqrt{2}}\,|\texttt{@Just\{Bit\}(\$0)}\rangle\langle\texttt{\$1}| + \frac{1}{\sqrt{2}}\,|\texttt{@Just\{Bit\}(\$1)}\rangle\langle\texttt{\$1}|\right) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\left(\frac{1}{\sqrt{3}}\,|\texttt{\$0}\rangle + \frac{\sqrt{2}}{\sqrt{3}}\,|\texttt{\$1}\rangle\right) = \\
&= \frac{1}{\sqrt{3}}\left(|\texttt{Nothing\{Bit\}}\rangle + |\texttt{@Just\{Bit\}(\$0)}\rangle + |\texttt{@Just\{Bit\}(\$1)}\rangle\right).
\end{aligned}
$$

If we were to implement this only using `ctrl`, we would need to write something like the following:

```
$0
▷ u3 {2 * arccos(sqrt(1 / 3)), 0, 0}
▷ lambda x → ctrl x [
  $0 → (x, $Nothing {Bit});
  $1 → (x, @Just{Bit}($plus))
]
▷ lambda (
  ctrl x' [
    $Nothing{Bit} → ($0, x');
    @Just{Bit}(_) → ($1, x')
  ]
) → x'
```

This is an instance of the specialized erasure pattern: in order to respect the erasure judgment, we must explicitly uncompute the variable `x` through an inverted `ctrl` expression.

Moreover, isometries can be combined with `pmatch` in more complicated ways, such as the following piece of code which allows us to view the rest of the tuple in the standard or Fourier basis depending on the value of the first element:

```
$repeated{4, Bit, $minus} ▷ pmatch [
  ($0, (x, y)) → (@qft{2}($plus, (x, ())), @qft{2}(y));
  ($1, @qft{3}(x, y)) →
      (@qft{2}($minus, (x, ())), @add_const{2, 1}(y));
]
```

Note that we can use the Quantum Fourier Transform (`@qft`, given in Appendix D.2) and `@add_const` (from Section 3.1) on both sides of the pattern-matching block since the type system can recognize that both of them have isometric (and in fact, unitary) semantics.

4.2 The `match` Construct

In a sense, `match` is a step in the opposite direction from `pmatch`: while `pmatch` extends the "pure" capabilities of `ctrl`, `match` extends its "mixed" capabilities. The goal of the `match` construct is to

have as few restrictions as possible. We can create a typing judgment for `match`, ensuring that the types of the scrutinee, patterns, and branches align in the current context, while also requiring orthogonality:

$$\frac{\Gamma \parallel \Delta, \Delta_0 \Vdash e : T \qquad \mathrm{ortho}_T\,(e_1, \ldots, e_n) \qquad \varnothing \parallel \Gamma_j \vdash e_j : T\ \forall j \qquad \mathrm{classical}(e_j)\ \forall j \qquad \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e'_j : T'\ \forall j}{\Gamma \parallel \Delta, \Delta_0, \Delta_1 \Vdash \texttt{match}\ e\ {}_T\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_{T'} : T'}\ \textsc{T-Match}$$

and we can define its semantics as:

$$\begin{aligned}\left[\!\left[\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \Vdash \texttt{match}\ e\ {}_T\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_{T'} : T'\right]\!\right]\left(|\tau, \tau_0, \tau_1\rangle \left\langle \tau', \tau'_0, \tau'_1\right|\right) \\ = \sum_{v \in \mathbb{V}(T)} \langle v| \left(\left[\!\left[\sigma : \Gamma \parallel \Delta, \Delta_0 \Vdash e : T\right]\!\right]\left(|\tau, \tau_0\rangle \left\langle \tau', \tau'_0\right|\right)\right) |v\rangle \cdot \\ \cdot \sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j\right| \left[\!\left[\varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T\right]\!\right]^\dagger |v\rangle \cdot \\ \cdot \left[\!\left[\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e'_j : T'\right]\!\right]\left(|\tau, \tau_1\rangle \left\langle \tau', \tau'_1\right|\right)\end{aligned}$$

This is very similar to the semantics of `ctrl`, described in Appendix A.3. Here,

$$\langle v| \left(\left[\!\left[\sigma : \Gamma \parallel \Delta, \Delta_0 \Vdash e : T\right]\!\right]\left(|\tau, \tau_0\rangle \left\langle \tau', \tau'_0\right|\right)\right) |v\rangle$$

gives the *probability* of the expression e being measured as $|v\rangle$ in the classical basis, with the given classical and quantum context variables. The expression e is thus treated as a "classical black box": thus, there is no way to distinguish the pure state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ from the mixed state $\frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|)$, since their probability distributions when measured in the classical basis are identical. This behavior is exactly the same as in `ctrl`. However, unlike `ctrl`, which has pure semantics, `match` has mixed semantics: it uses the superoperator $[\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e'_j : T']\!]$ formed from the RHS expression e'_j.

The `match` construct is designed to be used in circumstances where we do not care if our computation is reversible or not. For instance, it is convenient to implement a logical-AND operation using `match` as follows:

```
def @and : Bit * Bit → Bit :=
  lambda x → match x [
    ($1, $1) → $1;
    else → $0;
  ]
end
```

Note that the `else` keyword here is a syntactic construct that gets converted to the remaining expressions spanning `Bit * Bit` (that is, `($1, $0)` and `($0, _)`) during the preprocessing stage. The `match` construct no longer has the erasure requirement from `ctrl`, simply because it does not need to be reversible and can always discard any extra data it has, without the need to uncompute it. Like for `pmatch`, there exists a version of `match` that can be built as syntactic sugar over `ctrl`. This version can be constructed by simply combining all the variables in Δ into a tuple and pairing

it with each of the RHS expressions, and then feeding the result into `@snd`. However, the additional feature of `match` when it is implemented as a primitive is the fact that the RHS expressions, unlike in `ctrl`, do not need to be pure. See the example in Section 3.2 for an instance where this becomes useful for quantum algorithm implementation. The `match` construct is designed to be maximally similar to pattern matching in classical languages, avoiding the restrictions placed on `ctrl` other than the orthogonality requirement for the LHS. Using the `match` construct, Qunity programmers may write long and complex classical algorithms without worrying about reversibility, and then use them as subroutines in larger quantum algorithms.

5 Compiler Design

Figure 1 shows an outline of the main components of the Qunity interpreter and compiler, which are implemented in OCaml.

The preprocessor first evaluates the surface syntax definitions and produces a single Qunity expression, with the same abstract syntax as in Voichick et al. The typechecker then uses Qunity's extended typing judgments to check if the expression is well-typed and outputs a data structure representing a proof of typing, effectively elaborating the Qunity expression with helpful information that can later be used by the interpreter and the compiler, such as the contexts associated with its subexpressions. We can then either interpret or compile this elaborated expression: On the left side of Figure 1, we depict a simulator explicitly calculating the matrices corresponding to its semantics. This is an exponentially slow process, however, as it is equivalent to simulating a quantum computer on a classical computer. The main focus of this work is the compilation process shown on the right side, transforming a high-level elaborated Qunity expression into OpenQASM 3. This process can be performed efficiently on a classical computer, and the final output is a quantum circuit in a form that can be run on quantum hardware.

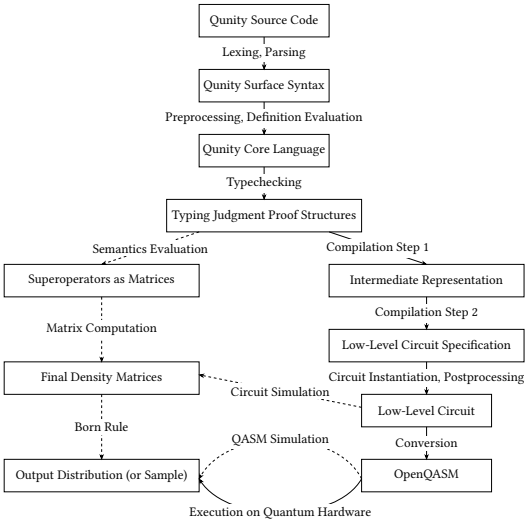


Fig. 1. Diagram showing the procedures used in the Qunity interpreter (left path), and the Qunity compiler (right path). Solid and dashed arrows indicate polynomial and exponential time processes respectively.

5.1 Formalism

Before making improvements to Qunity's original proof-of-concept compilation scheme, we found it necessary to revise some of the mathematical formalism that governs how the low-level circuits should be constructed in order to establish stronger correctness invariants throughout the process. In particular, many Qunity values are stored in a quantum register, represented as bitstrings using a type-based *encoding*. However, the space of all possible encodings does not necessarily span the space of all possible register states—for instance, a qutrit (a 3-level quantum system) can be stored in two bits, but only 3 out of 4 possible bit strings will be valid encodings. The compilation procedure in Voichick et al. includes some low-level circuits (namely, the direct sum) that can potentially output invalid encodings due to the way flag and output registers in the circuit's components are combined. The compilation lemmas do not address this possibility, which makes some parts of the compilation procedure incorrect. Allowing redundancy in the encodings would be problematic

because the extra qubits would need to be treated as flag or garbage qubits when inverting the direct sum injections: a "non-standard" encoding would either cause an error to be raised or introduce unwanted decoherence.

We considered two methods of resolving this problem. The first possibility was to create an error-checking circuit component, which would ensure that any state in the invalid encoding subspace is sent to the error subspace (with nonzero values for the flag qubits). However, the issue with this approach is that such an error-checking component would need to be inserted after every circuit component in the compilation procedure, making it much more complex and inefficient. The alternative approach was to ensure that no states enter the invalid encoding subspace under any conditions. We can create a mathematical definition of this as follows:

Definition 5.1. For a type T, define the space of valid encodings as

$$\mathbb{W}(T) = \operatorname{span}\{|\operatorname{enc}(v)\rangle \mid v \in \mathbb{V}(T)\},$$

where enc is the encoding function that maps values to bitstrings:

$$\begin{aligned}
\operatorname{enc}(()) &= \texttt{""} \\
\operatorname{enc}(\texttt{left}_{T_0 \oplus T_1}\ v) &= \texttt{"0"} +\!\!+ \operatorname{enc}(v) +\!\!+ \texttt{"0"}^{\max\{\operatorname{size}(T_0),\operatorname{size}(T_1)\}-\operatorname{size}(T_0)} \\
\operatorname{enc}(\texttt{right}_{T_0 \oplus T_1}\ v) &= \texttt{"1"} +\!\!+ \operatorname{enc}(v) +\!\!+ \texttt{"0"}^{\max\{\operatorname{size}(T_0),\operatorname{size}(T_1)\}-\operatorname{size}(T_1)} \\
\operatorname{enc}((v_0, v_1)) &= \operatorname{enc}(v_0) +\!\!+ \operatorname{enc}(v_1),
\end{aligned}$$

and the size of a type is defined as

$$\begin{aligned}
\operatorname{size}(\texttt{Void}) &= 0 \\
\operatorname{size}(\texttt{Unit}) &= 0 \\
\operatorname{size}(T_0 \oplus T_1) &= 1 + \max\{\operatorname{size}(T_0), \operatorname{size}(T_1)\} \\
\operatorname{size}(T_0 \otimes T_1) &= \operatorname{size}(T_0) + \operatorname{size}(T_1).
\end{aligned}$$

Definition 5.2. 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 $U : \mathbb{C}^{2^{\operatorname{size}(T)+p}} \to \mathbb{C}^{2^{\operatorname{size}(T')+f}}$ for some $p, f \in \mathbb{N}$ such that the following conditions hold:

- $\left\langle \operatorname{enc}(v'), 0^{\otimes f} \right| U \left| \operatorname{enc}(v), 0^{\otimes p} \right\rangle = \langle v' | E | v \rangle$ for all $v \in \mathbb{V}(T)$, $v' \in \mathbb{V}(T')$,
- $U \left| \operatorname{enc}(v), 0^{\otimes p} \right\rangle \in \mathbb{W}(T') \otimes \mathbb{C}^{2^f}$ for all $v \in \mathbb{V}(T)$, and
- $U^\dagger \left| \operatorname{enc}(v'), 0^{\otimes f} \right\rangle \in \mathbb{W}(T) \otimes \mathbb{C}^{2^p}$ for all $v' \in \mathbb{V}(T')$.

Definition 5.3. We say that it is possible to implement a trace non-increasing operator $\mathcal{E} : \mathcal{L}(\mathcal{H}(T)) \to \mathcal{L}(\mathcal{H}(T'))$ if there is a low-level circuit implementing a unitary operator $U : \mathbb{C}^{2^{\operatorname{size}(T)+p}} \to \mathbb{C}^{2^{\operatorname{size}(T')+f+g}}$ for some $p, f, g \in \mathbb{N}$ such that for all $v_1, v_2 \in \mathbb{V}(T)$, $v_1', v_2' \in \mathbb{V}(T')$,

$$\left\langle v_1' \right| \mathcal{E}\left(|v_1\rangle\langle v_2|\right) \left| v_2' \right\rangle = \sum_{b \in \{0,1\}^g} \left\langle \operatorname{enc}(v_1'), 0^{\otimes f}, b \right| U |\operatorname{enc}(v_1), 0^{\otimes p}\rangle\langle \operatorname{enc}(v_2), 0^{\otimes p}| U^\dagger \left| \operatorname{enc}(v_2'), 0^{\otimes f}, b \right\rangle,$$

and, for all $v \in \mathbb{V}(T)$,

$$U \left| \operatorname{enc}(v), 0^{\otimes p} \right\rangle \in \mathbb{W}(T') \otimes \mathbb{C}^{2^{f+g}}.$$

These definitions are formed by augmenting Definitions 6.2 and 6.3 from Voichick et al. with conditions that guarantee encoding validity is preserved. Now, any optimizations to the low-level circuit components need to respect the encoding-validity-preserving property.

5.2 Gates, Circuits, and Circuit Specifications

The final output of the compiler takes the form of a low-level qubit circuit, consisting of a series of gates, which can be:

- The identity
- A single-qubit $U_3(\theta, \phi, \lambda)$ gate
- A global phase gate
- A reset gate, which measures and resets a qubit to the $|0\rangle$ state
- An error-measurement gate, which measures a qubit and signals an error if it is in the $|1\rangle$ state
- A swap gate
- A controlled gate, with a list of control qubits
- Special zero-state labels and potential deletion labels, whose uses are detailed in Section 6.3.

These gates can very straightforwardly be converted into OpenQASM 3 code. However, each gate must already contain information that specifies the indices of the qubits it is applied to, which makes this data structure lack modularity. When compiling the intermediate representation into low-level circuits, we often want to separately construct two circuit components representing different operators, and then feed the output of one into the input of the other, or maybe reuse the same input and prep qubits for controlled versions of two different operators (see Section 6.1), which is difficult to do with just the gates. The Qunity compiler uses a system of *circuits* and *circuit specifications* to allow for the construction of low-level circuits out of smaller components. This system is constructed in a way that facilitates the tracking of qubits in various registers to ensure that the components fit together properly.

A circuit consists of a series of gates combined with additional data that specifies the roles of the qubits used by it - specifically, which qubit indices constitute the input registers, the output registers, the prep register (additional inputs prepared in the $|0\rangle$ state), the flag register (output qubits expected to be in the $|0\rangle$ state in the absence of error), and the garbage register (output qubits that are measured and discarded, corresponding to a partial trace operation).

A circuit specification is a wrapper around a function, `circ_fun`, that can construct a circuit given a list of input registers, a set of used wires (qubits that are in use elsewhere and cannot be used as new prep qubits), and additional instantiation settings that can dictate certain aspects of how the circuit is built. In addition to the circuit, this function also outputs the updated set of used wires. The circuit specification also includes data on the required input and output register dimensions. Circuit specifications are more modular than gates or circuits, because they are not tied to specific qubit indices - these are not realized until their `circ_fun` is called. Thus, it is possible to easily construct circuit specifications from other circuit specifications through various circuit construction functions. For instance, one such function could take in two circuit specifications, and define its `circ_fun` to instantiate the first specification into a circuit, and then use its output registers and used wires set to instantiate the second specification into a circuit, and then output a combined circuit whose gate is a sequence of the two original circuits' gates. The possible ways in which circuit specifications can be combined correspond closely to the low-level circuits defined in Appendix K.

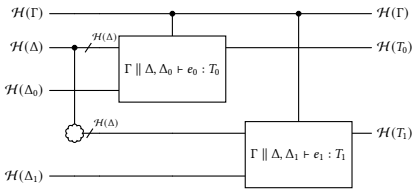
5.3 The Intermediate Representation

The intermediate representation used in the Qunity compiler describes high-level circuit diagrams in which the wires correspond to Hilbert spaces associated with Qunity types and contexts. This representation is built from *operators* that take in some list of input registers and output some registers. This level abstracts away the "flag" and "garbage" utility registers and can thus describe operators with non-unitary semantics. Operators in the intermediate representation can be either

built from smaller components using primitive constructors (corresponding to specific low-level circuit components), or they can be constructed from a series of commands in a fashion similar to a simple imperative language. These custom operators can associate registers with variable names and apply other operators to them, enforcing the condition that every variable must be used exactly once. This design makes it convenient to describe high-level circuits corresponding to each Qunity typing rule.

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

(a)



(b)

```
| TPurePair { t0; t1; d; d0; d1; e0; e1; _ } → begin
    let op0 = compile_pure_expr_to_inter_op e0 in
    let op1 = compile_pure_expr_to_inter_op e1 in
      inter_func_marked "TPurePair" iso un
        [("g", gsize); ("dd0d1", dsize)]
        [
          inter_comment "Starting TPurePair";
          inter_letapp ["d"; "d0d1"]
            (IContextPartition (d_whole, map_dom d))
            ["dd0d1"];
          inter_letapp ["d0"; "d1"]
            (IContextPartition (map_merge_noopt false d0 d1,
      map_dom d0))
            ["d0d1"];
          inter_letapp ["d"; "d*"] (IContextShare d) ["d"];
          inter_letapp ["dd0"] (IContextMerge (d, d0)) ["d"; "d0"
      ];
          inter_letapp ["d*d1"] (IContextMerge (d, d1)) ["d*"; "
      d1"];
          inter_letapp ["g"; "t0"] op0 ["g"; "dd0"];
          inter_letapp ["g"; "t1"] op1 ["g"; "d*d1"];
          inter_letapp ["res"] (IPair (t0, t1)) ["t0"; "t1"];
          inter_comment "Finished TPurePair";
        ]
        [("g", gsize); ("res", tsize)]
end
```

(c)

Fig. 2. (a) The typing judgment T-PurePair for typing pairs of Qunity expressions with a product type.
(b) The compilation circuit for T-PurePair as presented in Appendix L.3.
(c) Excerpt from the `compile_pure_expr_to_inter_op` function in the Qunity compiler, written in OCaml. This shows the compilation of the T-PurePair typing judgment into the intermediate representation, corresponding to the circuit in (b).

In the first step of the compilation procedure, the typing judgment proofs obtained from Qunity expressions by the typechecker are converted into custom operators in the intermediate representation. This can be done easily due to the close correspondence between the intermediate representation and the circuit diagrams described in Appendix L. Figure 2c shows the code for compiling T-PurePair into the intermediate representation. We can see that this describes an operator that is a map of the form $\mathcal{H}(\Gamma) \otimes \mathcal{H}(\Delta, \Delta_0, \Delta_1) \to \mathcal{H}(\Gamma) \otimes \mathcal{H}(T)$. First, the typing judgment proofs of e_0 and e_1 are compiled into the intermediate representation. Then, they are used in the `inter_func_marked`, which is a wrapper around the `IFunc` (user-defined operator) constructor, adding an extra operator around it using isometry and unitary judgment information. The `inter_comment` is just used to create annotations for debugging purposes. Next, the input register corresponding to the space $\mathcal{H}(\Delta, \Delta_0, \Delta_1)$ is partitioned into separate registers corresponding to $\mathcal{H}(\Delta)$, $\mathcal{H}(\Delta_0)$, and $\mathcal{H}(\Delta_1)$. Note that the compiler maintains an invariant that in a register corresponding to a context, the values of all variables must be stored in lexicographic order. Thus, the partitioning and merging circuits are not necessarily the identity and may involve some rearrangement of wires, which is done at the low level through a special circuit construction function. Then, the share gate (whose low-level implementation involves a new prep register of the same size as the input and CNOT gates) is applied, followed by the previously compiled operator circuits, and the result is combined into a single register. While it can be somewhat verbose, this code is essentially equivalent to the circuit diagram shown in Figure 2b.

6 Low-Level Optimizations

This section describes optimizations that modify the second step of the Qunity compilation procedure (where the intermediate representation is instantiated into low-level circuits that keep track of qubit indices) and additional post-processing steps that are applied to the resulting circuits. Each time a circuit component is instantiated, the compiler needs to recursively instantiate its constituent parts, and decide how to connect the inputs and outputs of the parts to each other. This process may also require allocation of additional prep qubits, which increases the complexity of the generated circuit. The goal of the low-level optimizations is to improve the process of constructing these low-level circuits, to avoid unnecessarily using many qubits at once.

6.1 Improving the Direct Sum Circuits

An important component of the second step of the compilation procedure is being able to implement the direct sum of two operators. Given circuits U_0 implementing $E_0 : \mathcal{H}(T_0) \to \mathcal{H}(T_0')$ (with input size s_0, output size s_0', prep size p_0, and flag size f_0), and U_1 implementing $E_1 : \mathcal{H}(T_1) \to \mathcal{H}(T_1')$ (with corresponding sizes s_1, s_1', p_1, f_1), the goal is to implement the direct sum circuit $E_0 \oplus E_1 : \mathcal{H}(T_0 \oplus T_1) \to \mathcal{H}(T_0' \oplus T_1')$. The implementation must be in accordance with Definition 5.2.

An initial naïve implementation that still respects the encoding-validity-preserving property specified in Definition 5.2 uses $\min\{s_0, s_1\} + p_0 + p_1$ prep wires. Because the number of prep wires scales with the size of the input register, this can cause the number of qubits used by the circuit to grow rapidly when an operator is summed with itself multiple times, which is a common pattern in the compilation of quantum control.

We have created an improved version of the direct sum circuit, which still ensures encoding validity but reuses input and prep qubits as much as possible. Table 2 shows the number of prep qubits needed for this improved circuit in different cases. The full circuits and a proof of correctness for them are shown in Appendix I.

Table 2. Prep qubits needed for the improved direct sum circuit in 4 different cases. The 4 other cases can be obtained from these ones by applying a commutativity isomorphism (which is just an X gate).

Case	Number of required prep qubits
$s_0 \geq s_1, s_0' \geq s_1', p_0 \geq p_1$	$p_0 + \max\{0, f_1 - f_0\}$
$s_0 \geq s_1, s_0' \geq s_1', p_0 \leq p_1$	$p_1 + \max\{0, f_1 - f_0 + p_0 - p_1\}$
$s_0 \geq s_1, s_0' \leq s_1', p_0 \geq p_1$	$p_0 + s_1' - s_0'$
$s_0 \geq s_1, s_0' \leq s_1', p_0 \leq p_1$	$p_1 + \max\{0, s_1' - s_0' + p_0 - p_1\}$

6.2 Recycling of Garbage and Flag Qubits

An important optimization to the Qunity compiler comes from the possibility of measuring and resetting garbage and flag qubits in the middle of a circuit and reusing them as new prep qubits for subsequent circuit components - that is, *recycling* the garbage and flag qubits. This has to be done with some care, because there are situations in which we want to avoid resetting these qubits. Specifically, we may be taking the purification of a circuit, which feeds its garbage qubits into the output register, or we may be using an error-handling circuit, which uses the flag register to coherently toggle a new prep qubit, outputting values in the space $\mathcal{H} \oplus \mathbb{C}$, where $\mathcal{H}$ is the output space of the original circuit. This means that a circuit specification has to "know," at the time of its instantiation into a circuit, whether or not it is allowed to recycle its garbage or flag register. This is done by using the instantiation settings, which are passed to a circuit specification at the

time of instantiation, and are used to decide whether the qubits can be recycled. Purification is implemented as a function that takes in a circuit specification and outputs another one, whose instantiation function passes in a value of `false` for the instantiation settings' `reset_garb` field when calling the wrapped circuit specification's instantiation function. The same thing happens with the error-handling circuits and the `reset_flag` field. This optimization reduces the qubit count of the circuits output by the compiler, since now, if there are many circuit components in series that each can throw an error and are not part of a larger error-handling component, the compiler does not have to allocate new qubits every time.

While more advanced recycling techniques exist, such as the one found in Jiang [15], the simple recycling technique we introduced here is closely tied to the specifics of Qunity's compilation procedure, and thus is not directly comparable. A potential future step would be to incorporate some of the more advanced qubit recycling methods from the literature as part of the *postprocessing* stage, where they can be directly applied to a quantum circuit without the need to integrate with any Qunity-specific data structures.

6.3 Post-Processing Optimizations

We have introduced a post-processing step that optimizes the quantum circuits generated by the Qunity compiler. The overall procedure for low-level optimization converts the input circuit into a list of gates and then repeatedly applies *optimization passes* to this list, until it reaches a fixed point. In a single optimization pass, the gate list is processed from left to right, detecting possible patterns that can be optimized. This process is guaranteed to eventually terminate, since each change made to the circuit has to decrease the number of qubits or gates. The currently implemented pass includes the following:

- Canceling a gate that is immediately followed by its adjoint.
- Combining adjacent controlled-U and anti-controlled-U gates into just U.
- Removing physical swap gates without controls and relabeling the subsequent wires.
- Performing a commutation pass: taking a gate and commuting it to the right until either it cannot commute past something (in which case it is reverted to its original position) or it cancels with something.
- Classical propagation - moving through the circuit and keeping track of qubits we know to be in a classical state, modifying any gates that are controlled on them by removing controls or deleting the gates.
- Detecting regions where a CNOT gate is applied, sharing one qubit to another one initially in the $|0\rangle$ state, then any other gates are the Pauli X gate or only use either qubit as a control, and then an identical CNOT is applied. In this case, it is possible to transfer the controls on the second qubit to the first one, which can make it possible to delete the second qubit with the procedure described below.
- Deleting qubits (or gates on a qubit between measurements) if certain conditions are met.

We will now discuss this last point in more detail. This mechanism relies on high-level information from the *isometry judgment* in the Qunity typechecker (see Appendix G) to identify candidate qubits for potential deletion. These candidates are either garbage qubits or flag qubits that we are certain will always be in the $|0\rangle$ state when measured.

The candidate qubits are identified during the procedure of instantiating a circuit specification into a concrete circuit. The instantiation settings, which include Boolean fields `reset_garb` and `reset_flag` for whether or not the garbage and flag registers should be reset, as described in Section 6.2, also contain a field `iso` for whether or not this circuit can be interpreted as an isometry and thus whether we can expect the flag qubits to be in the $|0\rangle$ state. This variable is set to `true` by

a special wrapper circuit specification that acts very similarly to the purification and error-handling circuits, and is added to the circuit based on the results of the isometry judgment in the first step of the compilation process. Now, during instantiation, if `reset_garb` is `true`, then a special gate called a *potential deletion label* is added to the qubits in the garbage register. Similarly, if both `reset_flag` and `iso` are `true`, the potential deletion labels are also added to the flag register. Additionally, instead of measuring flag qubits known to be in the $|0\rangle$ state, the compiler adds a special *zero-state label* onto them.

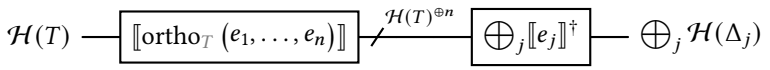
However, not all candidate qubits are safe to delete because they may be entangled with the output qubits at some point during the computation, possibly being used as ancillas. In the post-processing stage, all the deletion labels are first shifted to the left as far as possible until they hit a measurement gate or a zero-state label to ensure that they are selecting an entire region of interest. Then, during an optimization pass, when seeing a potential deletion label, the segment between it and the next measurement is evaluated to determine whether it is safe to delete. The procedure determines that all gates in the region involving the selected qubit are safe to delete if they are all Pauli X gates, controlled X gates with the qubit as a target, or uncontrolled global phase gates. This is because if the qubit starts in the $|0\rangle$ state, then applying only these gates will not have any effect on the reduced density matrix of the system obtained by removing the qubit. At the end of the optimization procedure, if a qubit has no more gates remaining on it, it is completely removed from the circuit.

7 High-Level Optimizations

This section describes optimizations that modify the first step of the Qunity compilation procedure, where the typing judgment proof structures are converted to the intermediate representation.

7.1 Simplifying the Orthogonality Circuit

A significant part of the circuit for compiling the `ctrl`, `match`, and `pmatch` constructs is the following circuit component:



This circuit is quite inefficient: it uses $O(n)$ extra qubits to store the register corresponding to $\mathcal{H}(T)^{\oplus n}$, and it involves many expensive associativity isomorphisms to transform the direct sum structure of the given space to something of the form $\mathcal{H}(T) \oplus (\mathcal{H}(T) \oplus (\mathcal{H}(T) \oplus \ldots))$. In this work, we introduce two major changes to this: allowing direct sums of operators to be taken *along a binary tree*, and replacing the two above circuit components with a single one that goes directly from $\mathcal{H}(T)$ to $\bigoplus_j \mathcal{H}(\Delta_j)$, without the intermediate step of $\mathcal{H}(T)^{\oplus n}$. Refer to Appendix J for the definitions and notations used to describe sums of operators along a binary tree and leveling a tree to a given height.

We can derive a binary tree from the structure of the orthogonality judgment (Appendix F), as follows:

Definition 7.1 (Tree derived from the orthogonality judgment).

- The trees derived from O-Void, O-Unit, and O-Var are all Leaf.
- The tree derived from O-IsoApp is the same as that derived from $\text{ortho}_T\ (e_1, \ldots, e_n)$.
- The O-Sum rule states that if $e_1, \ldots, e_m$ are orthogonal in T and $e'_1, \ldots, e'_m$ are orthogonal in T', then the left injections of the e_j and the right injections of the e'_j together are orthogonal in $T \oplus T'$. The tree derived from O-Sum is a root whose left subtree is the tree derived from the `left` expressions and whose right subtree is the tree derived from the `right` expressions.

- The O-Pair rule states that if $e_1, \ldots, e_m$ are orthogonal in T, and for each $1 \leq j \leq m$, the expressions $e'_{j,1}, \ldots, e'_{j,n_j}$ are orthogonal in T', then the list of pairs $(e_j, e'_{j,k})$ are orthogonal in $T \otimes T'$. The tree derived from S-Pair is constructed by taking the tree $\mathcal{R}_0$ derived from $\text{ortho}_T\ (e_1, \ldots, e_m)$, and replacing each jth leaf with the tree $\mathcal{R}_1^j$ derived from $\text{ortho}_{T'}\ (e'_{j,1}, \ldots, e'_{j,n_j})$. This essentially decomposes a tensor product into a direct sum structure.
- The tree derived from from O-Sub is constructed by removing all subtrees whose leaves only contain discarded expressions.

Now, we want to define $[\![\text{ortho}_T\ (e_1, \ldots, e_n)]\!]$ in such a way that when $\text{ortho}_T\ (e_1, \ldots, e_n)$ holds with tree structure $\mathcal{R}$, then for all j, for all $\tau_j \in \mathbb{V}(\Delta_j)$, we have that

$$[\![\text{ortho}_T\ (e_1, \ldots, e_n)]\!][\![e_j]\!] \left|\tau_j\right\rangle = \text{inj}_j^{\mathcal{R}} \left|\tau_j\right\rangle .$$

That is, we are constructing an operator

$$[\![\text{ortho}_T\ (e_1, \ldots, e_n)]\!] : \mathcal{H}(T) \to \bigoplus_{j:\mathcal{R}} \mathcal{H}(\Delta_j).$$

This construction is given in Appendix L.2.

With this change, we can now modify the circuit for the compilation of T-Ctrl to use a single circuit component that converts the input $\mathcal{H}(T)$ directly into a direct sum of the Δ_j over the tree derived from the orthogonality judgment. Then, for the rest of the circuit, all direct sums can be taken along this tree instead of a standardized associativity structure.

7.2 Compilation of `pmatch`

The same construction as above can be used to construct a high-level circuit to compile T-Pmatch into the intermediate representation. This can be done as follows:

where $\mathcal{R}_0$ and $\mathcal{R}_1$ are binary trees derived from the respective orthogonality judgments, except their leaves are also labeled with the indices of the corresponding expressions. Then, TreeRearrange is an algorithm for transforming one binary tree into another using only operations from a certain set, encoded as a quantum circuit.

The possible operations are:

- A right tree rotation, which can be implemented as the associativity isomorphism $(T_0 \oplus T_1) \oplus T_2 \to T_0 \oplus (T_1 \oplus T_2)$ (low-level circuit implementation in Appendix K).
- A left tree rotation, implementable as the adjoint of the associativity isomorphism.
- Switching the left and right subtree, which can be implemented with a single Pauli X gate.
- Other transformations applied to the subtrees of a given tree, which can be implemented using the direct sum circuit.
- A sequence of two transformations, which can be implemented as a sequence of two circuits.
- A special *conditional commutation* transformation, shown in Figure 3. It can be implemented with a multi-controlled X gate, where the target is the qubit corresponding to the root node, and the controls have states corresponding to the path taken.

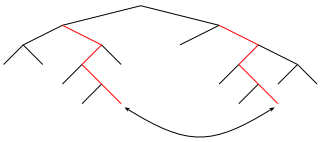


Fig. 3. An example of the conditional commutation tree transformation, which swaps the two vertices shown in this diagram. Their paths must be identical except for the very first step from the root.

The algorithm for transforming trees, expressed in terms of these allowed operations, is:

(1) Transform the input tree so it has the same shape as the goal, ignoring the values at the leaves. This can be done by transferring nodes from one subtree to another making sure they have the correct numbers, and then making recursive calls on each subtree. The transferring of nodes can be accomplished by a sequence of tree rotations.
(2) For each leaf value that needs to be transferred to a different location, do the following to switch the leaves at the two locations:
    (a) Bring the leaf at the larger depth (the one farther from the root) to the same depth as the other one. This can be accomplished by a sequence of tree rotations and commutations.
    (b) Make the trees have identical paths from the root except for the first place in which their paths differ. This can be done by a sequence of commutation operations applied to the appropriate subtrees.
    (c) Apply the special conditional commutation operation to switch the two leaves.
    (d) Apply the operations in the first two steps in reverse order so that the only effect of the entire transformation is swapping the two leaves of interest.

In the OCaml implementation of the compiler, running this algorithm creates a record of all the tree transformations that need to be applied to perform the operation. Then, the resulting tree transformations are converted into the intermediate representation in the manner described above, thus completing the compilation of `pmatch`. Beside the fact that this allows `pmatch` to work in an arbitrary basis, it is also much more efficient than implementing a classical-basis version of it as syntactic sugar over `ctrl`: the tree rearrangement algorithm attempts to avoid significantly increasing the height of the tree at any intermediate step, which means that it does not unnecessarily use additional qubits.

8 Evaluation

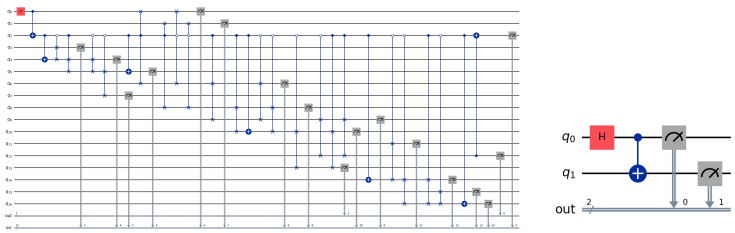
In this section we detail our evaluation of the Qunity compiler. We primarily focus on the effectiveness of the compiler in reducing compiled circuit size, but we also report on our differential unit testing that we used to gain confidence in the correctness of our implementation.

Compilation Efficiency. We evaluate the Qunity compiler on several benchmark programs, shown in Table 3. We compare the most recent version of the compiler with the initially implemented version, which followed directly the compilation procedure of Voichick et al. (with the direct sum circuit corrected to preserve encoding validity but not optimized). This baseline did not support the new pattern-matching constructs, so where possible, the benchmark code was rewritten to equivalent expressions using only `ctrl`. We also compare the performance of our compiler to low-level circuits constructed directly in Qiskit [23] version 1.1.1, using circuits from `qiskit.circuit.library` when relevant. The gate counts reported in this table are after the circuit has been transpiled by Qiskit with `basis_gates=["u3", "cx"], optimization_level=3` (to only use single-qubit gates and CNOT).

The results show a very significant improvement of the optimized Qunity compiler compared to the unoptimized one. In many instances, the optimized Qunity compiler reaches a gate count that is comparable to the handcrafted Qiskit implementation, whereas the baseline Qunity compiler resulted in orders of magnitude larger circuits. Still there is room for improvement: to uncover the limits of our compiler implementation, we included a benchmark that implements Grover's algorithm using the list sum oracle, which compiles into 13,470 low-level gates, while it is possible to manually construct a small circuit that accomplishes the same task. The inefficiencies arise due to isomorphisms and transformations between types that are introduced during the compilation procedure: while the optimizations introduced in Section 7 help to simplify them to a large extent, there are still many inefficiencies present that are nontrivial to eliminate.

Table 3. Benchmark tests comparing the performance of the optimized and unoptimized Qunity compilers with an implementation in Qiskit. The code used for these benchmarks is shown in Appendix D. The missing entries in the unoptimized Qunity compiler result from the unoptimized compiler not supporting the new pattern-matching constructs. We also do not have a suitable reference for the order-finding circuit. The entry marked with ?? has such a large circuit that it was infeasible to transpile it to obtain a gate count.

Benchmark	Qunity (unoptimized)		Qunity (optimized)		Qiskit	
	Qubits	Gates	Qubits	Gates	Qubits	Gates
Phase conditioned on AND of 5 qubits	769	147,871	9	121	5	101
Quantum Fourier Transform (5 qubits)	15	282	6	143	5	54
Phase estimation example (5 qubits)	75	1465	6	275	5	60
Order finding (`#n=5, #a=13`)	–	–	10	460	–	–
Reversible CDKM Adder [7] (5 bits)	1031	160,321	11	152	11	179
Grover (5-bit match oracle, 1 iteration)	1131	??	7	760	6	389
Grover (List sum oracle, `#n=2`, 1 iteration)	–	–	11	13,470	5	116



(a) (b)

Fig. 4. (a) The circuit produced by compiling a Qunity program for preparing the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ without compiler optimizations. (b) The circuit produced using the the optimized compiler for the same program.[3]

Still, for most programs, including smaller ones that we used for testing, our Qunity compiler results in practical circuits. For one example, Figure 4 demonstrates the improvement in the efficiency of the compiler resulting from the optimizations implemented in this work when preparing a Bell state using the quantum control construct. Note that the simplest way to prepare this state in Qunity is using entanglement through variable reuse: `$plus ▷ lambda x → (x, x)`, which compiles to the circuit in Figure 4b even with the unoptimized compiler. However, for this example,

[3]These circuits have measurements because the output register is always measured at the end when running the Qunity compiler.

the entanglement is achieved by defining a CNOT gate in terms of the `ctrl` construct, which has a much more complicated compilation procedure. (This can be used to write circuit-style programs in Qunity: see Appendix E). The circuit produced by the unoptimized compiler, in Figure 4a, needs 17 qubits and 27 CNOT and controlled-SWAP gates just to represent an operation equivalent to a single CNOT gate. This is due to a combination of several factors, including the unoptimized direct sum circuits, the unnecessary associativity isomorphisms, and the redundancy in the original orthogonality circuit. It is difficult to simplify this circuit by post-processing alone: the higher-level optimizations described in this work are necessary. Using the combination of these optimization techniques, the compiler is able to reduce this circuit to a form that can be more easily handled by the post-processing optimizations and converted to the circuit in Figure 4b.

Differential Unit Testing. Finally, we also created a test suite to verify the correctness of the compilation procedure, checking it against the output of the interpreter. The test proceeds as follows: given a Qunity expression e of type T, we evaluate its semantics as a density matrix $\rho = [\![\varnothing : \varnothing \parallel \varnothing \Vdash e : T]\!]$ using the Qunity interpreter. We run the compiler to generate a low-level qubit circuit, and simulate this circuit. Taking a projection onto the subspace where the flag register is in the $|0\rangle$ state and taking a partial trace over the garbage registers, we obtain a state $\tilde{\rho}$. Then, for each $v \in \mathbb{V}(T)$, we check that $\langle v| \rho |v\rangle = \langle \mathrm{enc}(v)| \tilde{\rho} |\mathrm{enc}(v)\rangle$.

Most programs in this suite are small, and many are just simple demonstrations of various patterns (such as pattern matching constructs nested in different ways, operations on datatypes, and error handling). We exclude programs from Table 3 that are too expensive to simulate.

9 Related Work

Many existing quantum programming languages are based on the paradigm of "quantum data, classical control," which was popularized by Selinger [26]. In this model, quantum circuits can be constructed by a classical computer, using classical data which may be obtained by performing measurements on qubits. Most of the widely-used quantum languages rely on this paradigm: from low-level languages like OpenQASM [6], Cirq [9], and Qiskit [23] to higher-level languages like Quipper [11], Q# [29] and Qualtran [13].

However, we often want to express algorithms that use quantum control flow and manipulate complex data structures in quantum superposition: for instance, Grover's algorithm [12] over arbitrary oracles or quantum walk-based algorithms such as Childs et al. [4]. This runs into problems, as fully general quantum control flow is physically infeasible, as argued by Bădescu and Panangaden [2] and later Yuan, Charles and Villanyi, Agnes and Carbin, Michael [38]. A number of languages have attempted to address this issue through restricting control flow, including QML [1], Symmetric Pattern Matching (SPM) [25], Silq [3], and Tower [37]. Of these, Silq and Tower compile to circuits.

Venev's [30] compiler for Silq programs is focused on the problem of *uncomputation*, specifically in compiling and optimizing away the results of the common compute-copy-uncompute pattern used in the generation of quantum oracles. Silq's uncomputation, however, is restricted to applications of `qfree` (strictly classical) functions - Qunity's `ctrl` construct, on the other hand, enables uncomputation of an arbitrary mixed program (although doing this on a program that introduces superposition will lead to norm-decreasing semantics). While none of our examples make use of this feature, it may be useful in more advanced quantum algorithms such as the triangle finding algorithm [19], which uses a quantum subroutine that is itself a probabilistic quantum algorithm.

Tower's compiler, Spire [36] is closer in its aims to this work. Like Qunity, Tower is a high-level language with sophisticated (QRAM-based) control flow, and Spire aims to optimize it at a high level. Spire uses two key optimizations: conditional flattening (which consists of combining nested

if statements, reducing the controls on internal unitaries) and conditional narrowing, which pulls statements out of quantum ifs (again, reducing the number of controls). These two optimizations prove surprisingly valuable in practice, enabling a significant reduction in the circuits' T-complexity, and showing that high-level optimizations of quantum programs could be useful in practice.

Qunity is set apart from the previous work listed above by its focus on compositionality and its denotational semantics. It takes the unique and unusual approach of defining both a pure and mixed typing judgment, allowing for a richer interplay of reversible and irreversible programming. Qunity is similar to classical functional languages, while languages like Silq and Tower follow an imperative paradigm. The metaprogramming layer allows for higher-order functions and recursion without running into the fundamental limitations of implementing them as quantum constructs [2]. Qunity's pattern-matching constructs offer a novel and expressive way of expressing quantum control flow.

10 Conclusion and Future Work

In this work, we show how to efficiently compile Qunity, a high-level quantum programming language with compositional semantics, by combining low-level optimizations that simplify the construction of qubit circuit components with higher-level optimizations that simplify manipulations of Hilbert spaces.

There are more optimizations that we would like to implement in the future. For example, in instances where no purification is performed, the `try`/`catch` statement could be implemented by measuring the flag register, and applying some gates depending on the measurement outcome. Similarly, the `match` construct could be implemented by measuring the scrutinee expression and using the result to evaluate one of the RHS expressions. Including this capability can simplify many Qunity programs with classical computation outside of quantum subroutines.

We could also change the target language of the Qunity compiler from OpenQASM to a quantum intermediate representation such as QIR [18], MLIR [20], or HUGR [28]. Note that these representations are not directly comparable to Qunity's IR, which is defined using operators closely associated to Qunity's control flow constructs. Compiling from Qunity's IR to one of these representations would allow Qunity to more easily take advantage of existing optimizers at the post-processing stage, and make the compilation to hybrid quantum-classical programs more feasible.

With improvements like these, we believe Qunity can truly become a practical quantum programming language, offering a promising alternative to circuit-based quantum languages and advancing the development of tools for high-level quantum programming and algorithm design.

Data Availability Statement

The implementation of the Qunity interpreter and compiler is publicly available [22].

Acknowledgments

We thank Michael Vanier for his advice on this work. Mikhail Mints was supported by the Samuel P. and Frances Krown SURF fellowship from Caltech. This material is based upon work supported by the Air Force Office of Scientific Research under Grant No. FA95502310406.

References

[1] 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. eprint: quant-ph/0409065. DOI: 10.1109/LICS.2005.1.

[2] Costin Bădescu and Prakash Panangaden. 2015. Quantum Alternation: Prospects and Problems. *Electronic Proceedings in Theoretical Computer Science*, 195, (Nov. 2015), 33–42. DOI: 10.4204/eptcs.195.3.

[3] 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* (PLDI 2020). Association for Computing Machinery, London, UK, 286–300. ISBN: 9781450376136. DOI: 10.1145/3385412.3386007.

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

[5] Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. 2022. Symmetries in Reversible Programming: From Symmetric Rig Groupoids to Reversible Programming Languages. *Proc. ACM Program. Lang.*, 6, POPL, (Jan. 2022). DOI: 10.1145/3498667.

[6] Andrew Cross et al. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing*, 3, 3, (Sept. 2022), 1–50. DOI: 10.1145/3505636.

[7] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. 2004. A new quantum ripple-carry addition circuit. (2004). https://arxiv.org/abs/quant-ph/0410184 arXiv: quant-ph/0410184 [quant-ph].

[8] David Deutsch and Roger Penrose. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400, 1818, 97–117. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1985.0070. DOI: 10.1098/rspa.1985.0070.

[9] Cirq Developers. 2025. Cirq. (Aug. 2025). DOI: 10.5281/ZENODO.4062499.

[10] Craig Gidney. 2017. Simple Algorithm for Multiplicative Inverses mod 2^n. (2017). https://algassert.com/post/1709.

[11] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. *SIGPLAN Not.*, 48, 6, (June 2013), 333–342. DOI: 10.1145/2499370.2462177.

[12] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (STOC '96). Association for Computing Machinery, Philadelphia, Pennsylvania, USA, 212–219. ISBN: 0897917855. DOI: 10.1145/237814.237866.

[13] Matthew P. Harrigan, Tanuj Khattar, Charles Yuan, Anurudh Peduri, Noureldin Yosri, Fionn D. Malone, Ryan Babbush, and Nicholas C. Rubin. 2024. Expressing and Analyzing Quantum Algorithms with Qualtran. (2024). eprint: 2409.04643. DOI: 10.48550/arXiv.2409.04643.

[14] Teiko Heinosaari and Mário Ziman. 2011. *The Mathematical Language of Quantum Theory: From Uncertainty to Entanglement*. Cambridge University Press, Cambridge. DOI: 10.1017/CBO9781139031103.

[15] Hanru Jiang. 2024. Qubit Recycling Revisited. *Proc. ACM Program. Lang.*, 8, PLDI, (June 2024). DOI: 10.1145/3656428.

[16] Soon-Mo Jung, Doyun Nam, and Michael Th. Rassias. 2019. On the Order of Odd Integers Modulo 2^n. *Applicable Analysis and Discrete Mathematics*, 13, 2, 619–631. Retrieved July 26, 2025 from https://www.jstor.org/stable/26799953.

[17] E Knill. 1996. Conventions for quantum pseudocode. Tech. rep. Los Alamos National Lab., NM (United States), (June 1996). DOI: 10.2172/366453.

[18] Junjie Luo and Jianjun Zhao. 2025. Formalization of Quantum Intermediate Representations for code safety. *Journal of Systems and Software*, 219, 112236. DOI: https://doi.org/10.1016/j.jss.2024.112236.

[19] Frédéric Magniez, Miklos Santha, and Mario Szegedy. 2007. Quantum Algorithms for the Triangle Problem. *SIAM Journal on Computing*, 37, 2, 413–424. eprint: quant-ph/0310134. DOI: 10.1137/050643684.

[20] Alexander McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. (2021). https://arxiv.org/abs/2101.11365 arXiv: 2101.11365 [quant-ph].

[21] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. 2016. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18, 2, (Feb. 2016), 023023. DOI: 10.1088/1367-2630/18/2/023023.

[22] [SW] Mikhail Mints, Finn Voichick, Leonidas Lampropoulos, and Robert Rand, Artifact for OOPSLA 2025: Compositional Quantum Control Flow with Efficient Compilation in Qunity 2025. DOI: 10.5281/zenodo.16567634, URL: https://github.com/mikhailmints/qunity.

[23] Qiskit contributors. 2023. Qiskit: an open-source framework for quantum computing. (2023). DOI: 10.5281/zenodo.2573505.

[24] Steven Roman. 2008. *Advanced Linear Algebra*. (Third ed.). *Graduate Texts in Mathematics*. Vol. 135. Springer, New York, xviii + 525. DOI: 10.1007/978-0-387-72831-5.

[25] 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, 348–364. ISBN: 978-3-319-89366-2. DOI: 10.1007/978-3-319-89366-2_19.

[26] Peter Selinger. 2004. Towards a Quantum Programming Language. *Mathematical Structures in Computer Science*, 14, 4, (Aug. 2004), 527–586. DOI: 10.1017/S0960129504004256.

[27] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26, 5, (Oct. 1997), 1484–1509. DOI: 10.1137/s0097539795293172.

[28] Seyon Sivarajah et al. 2025. HUGR: A Quantum-Classical Intermediate Representation. (2025). https://popl25.sigplan.org/details/planqc-2025-papers/6/HUGR-A-Quantum-Classical-Intermediate-Representation.

[29] Krysta Svore et al. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018* (RWDSL2018). Association for Computing Machinery, Vienna, Austria. ISBN: 9781450363556. DOI: 10.1145/3183895.3183901.

[30] Hristo Venev, Timon Gehr, Dimitar Dimitrov, and Martin Vechev. 2024. Modular Synthesis of Efficient Quantum Uncomputation. *Proceedings of the ACM on Programming Languages*, 8, OOPSLA2, 2097–2124. arXiv: 2406.14227 [cs.PL]. DOI: 10.1145/3689785.

[31] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. *Proceedings of the ACM on Programming Languages*, 7, POPL, (Jan. 2023), 921–951. arXiv: 2204.12384 [cs.PL]. DOI: 10.1145/3571225.

[32] Yunfei Wang and Junyu Liu. 2024. A comprehensive review of quantum machine learning: from NISQ to fault tolerance. *Reports on Progress in Physics*, 87, 11, (Oct. 2024), 116402. DOI: 10.1088/1361-6633/ad7f69.

[33] 2009. *Quantum Computational Complexity. Encyclopedia of Complexity and Systems Science*. Springer New York, New York, NY, 7174–7201. ISBN: 978-0-387-30440-3. DOI: 10.1007/978-0-387-30440-3_428.

[34] W K Wootters and W H Zurek. 1982. A single quantum cannot be cloned. *Nature*, 299, 802–803. DOI: 10.1038/299802a0.

[35] Donald Yau. 2021. Symmetric Bimonoidal Categories and Monoidal Bicategories. (2021). https://nilesjohnson.net/En-monoidal.html.

[36] Charles Yuan and Michael Carbin. 2024. The T-Complexity Costs of Error Correction for Control Flow in Quantum Computation. *Proceedings of the ACM on Programming Languages*, 8, PLDI, 492–517. arXiv: 2311.12772 [cs.PL]. DOI: 10.1145/3656397.

[37] Charles Yuan and Michael Carbin. 2022. Tower: Data Structures in Quantum Superposition. *Proceedings of the ACM on Programming Languages*, 6, OOPSLA2, 259–288. arXiv: 2205.10255 [cs.PL]. DOI: 10.1145/3563297.

[38] Yuan, Charles and Villanyi, Agnes and Carbin, Michael. 2024. Quantum control machine: the limits of control flow in quantum programming. *Proc. ACM Program. Lang.*, 8, OOPSLA1, (Apr. 2024). DOI: 10.1145/3649811.

Appendices

A Formal Definitions of Syntax, Typing Judgments, and Semantics

A.1 Syntax

Figure 7 shows the base abstract syntax of the Qunity language. Expressions e are assigned types T and programs f are assigned program types F (Figure 5). Here, x ranges over some set of variable names and r ranges over a representation of real numbers. This work adds two new primitive syntactic constructs to the original Qunity language: `match` and `pmatch`, discussed in Section 4.

A.2 Type System

Consider an expression e typed as $\Gamma \parallel \Delta \vdash e : T$, that is, e has pure type T with classical context Γ and quantum context Δ. The semantics of such an expression corresponds to a *pure state* in the Hilbert space $\mathcal{H}(T)$ associated with the type T. Figure 8 shows the typing rules for the pure expression typing judgment. While the separation into "classical" and "quantum" contexts may seem unexpected for a language like Qunity that aims to unify classical and quantum computation, the classical contexts do in fact play an important role in the type system. When a variable x is in a classical context Γ, it does not mean that it is literally in a classical register - it means that from the perspective of the current expression, we view it as being in a classical basis state and interact with it by sharing it relative to the classical basis. This also means that the type system does not place the same relevance restrictions on variables in classical contexts as it does on quantum variables: we may assume that the compiler automatically uncomputes the variable when it goes out of scope. The primary use case for these classical contexts is the `ctrl` construct: as seen in the T-Ctrl rule, the quantum contexts of the left-hand-side (LHS) pattern expressions become classical contexts when typing the right-hand-side (RHS) expressions. This allows the programmer to safely ignore these variables in the RHS expressions, knowing that when compiling T-Ctrl, their uncomputation will be handled automatically.

The semantics of an expression typed with the mixed typing judgment as $\Gamma \parallel \Delta \Vdash e : T$ corresponds to a *mixed state* in the Hilbert space $\mathcal{H}(T)$ associated with the type T. Observe that the rule T-Mix allows pure expressions to be typed as mixed.

Note that this presentation differs from the original mixed expression typing rules in Voichick et al., which did not include classical contexts since the mixed expression typing judgment has no relevance restrictions on the quantum variables. The primary reason for this change is consistency between `ctrl` and the newly introduced `match`: for instance, it allows expressions of the following

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

Fig. 5. Qunity types.

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

Fig. 6. Typing contexts.

$$
\begin{array}{rll}
e ::= & & \textit{(expression)} \\
& () & \textit{(unit)} \\
\mid & x & \textit{(variable)} \\
\mid & (e, e) & \textit{(pair)} \\
\mid & \texttt{ctrl}\ e\ {}_{T}\left\{\begin{array}{c} e \mapsto e \\ \cdots \\ e \mapsto e \end{array}\right\}_{T} & \textit{(coherent control)} \\
\mid & \texttt{match}\ e\ {}_{T}\left\{\begin{array}{c} e \mapsto e \\ \cdots \\ e \mapsto e \end{array}\right\}_{T} & \textit{(decoherent match)} \\
\mid & \texttt{try}\ e\ \texttt{catch}\ e & \textit{(error recovery)} \\
\mid & f\ e & \textit{(application)} \\
f ::= & & \textit{(program)} \\
& \mathsf{u}_3(r, r, r) & \textit{(qubit gate)} \\
\mid & \texttt{left}_{T \oplus T} & \textit{(left tag)} \\
\mid & \texttt{right}_{T \oplus T} & \textit{(right tag)} \\
\mid & \lambda e \overset{T}{\mapsto} e & \textit{(abstraction)} \\
\mid & \texttt{rphase}_{T}\left\{\begin{array}{c} e \mapsto r \\ \texttt{else} \mapsto r \end{array}\right\} & \textit{(relative phase)} \\
\mid & \texttt{pmatch}\ {}_{T}\left\{\begin{array}{c} e \mapsto e \\ \cdots \\ e \mapsto e \end{array}\right\}_{T'} & \textit{(symmetric matching)}
\end{array}
$$

Fig. 7. Base Qunity syntax.

form to be accepted by the typechecker:

$$
\texttt{match}\ x\ {}_{T}\left\{\begin{array}{c} y \mapsto \texttt{ctrl}\ y\ {}_{T}\{(\text{expressions that do not erase } y)\}_{T'} \\ \cdots \end{array}\right\}_{T'}
$$

Here, y is passed into the *classical* context of the `ctrl`, which is initially typed as mixed by the T-Match rule, and then as pure through T-Mix rule. Because y is in the classical context of the `ctrl` expression, it is not required to satisfy the erasure judgment in `ctrl` since it only applies to quantum context variables.

The T-Discard rule is also a new addition, as Voichick et al. used the T-MixedAbs program typing rule to make variable discarding possible. This new change is made to more easily deal with discarded variables in the `match` construct.

These rules rely on several additional judgments, namely `ortho`, and `erases`. These are listed in Appendix H and Appendix F.

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

$$\frac{\Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0 \qquad \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1}{\Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1}\ \textsc{T-PurePair} \qquad \frac{\vdash f : T \rightsquigarrow T' \qquad \Gamma \parallel \Delta \vdash e : T}{\Gamma \parallel \Delta \vdash f\ e : T'}\ \textsc{T-PureApp}$$

$$\frac{\begin{array}{c}\Gamma \parallel \Delta \Vdash e : T \qquad \mathrm{ortho}_T\,(e_1, \ldots, e_n) \qquad \varnothing \parallel \Gamma_j \vdash e_j : T\ \forall j \qquad \mathrm{classical}(e_j)\ \forall j \\ \mathrm{erases}_{T'}(x; e'_1, \ldots, e'_n)\ \forall x \in \mathrm{dom}(\Delta) \qquad \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T'\ \forall j\end{array}}{\Gamma \parallel \Delta, \Delta' \vdash \texttt{ctrl}\ e\ {}_T\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_{T'} : T'}\ \textsc{T-Ctrl}$$

Fig. 8. Pure expression typing rules. Here, $\texttt{classical}(e)$ holds if e does not include any use of `u3` or `rphase`.

$$\frac{\Gamma \parallel \Delta \vdash e : T}{\Gamma \parallel \Delta \Vdash e : T}\ \textsc{T-Mix} \qquad \frac{\Gamma \parallel \Delta \Vdash e : T}{\Gamma \parallel \Delta, \Delta_0 \Vdash e : T}\ \textsc{T-Discard}$$

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

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

$$\frac{\begin{array}{c}\Gamma \parallel \Delta, \Delta_0 \Vdash e : T \qquad \mathrm{ortho}_T\,(e_1, \ldots, e_n) \qquad \varnothing \parallel \Gamma_j \vdash e_j : T\ \forall j \\ \mathrm{classical}(e_j)\ \forall j \qquad \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e'_j : T'\ \forall j\end{array}}{\Gamma \parallel \Delta, \Delta_0, \Delta_1 \Vdash \texttt{match}\ e\ {}_T\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_{T'} : T'}\ \textsc{T-Match}$$

Fig. 9. Mixed expression typing rules.

A.3 Semantics

Qunity's semantics is defined in terms of operators and superoperators mapping between Hilbert spaces associated with Qunity types and contexts. These Hilbert spaces are defined as follows:

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

$$\frac{}{\vdash \mathtt{u}_3(r_\theta, r_\phi, r_\lambda) : \mathrm{Bit} \rightsquigarrow \mathrm{Bit}}\ \textsc{T-Gate}$$

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

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

$$\frac{\begin{array}{ll} \mathrm{ortho}_T\ (e_1, \ldots, e_n) & \varnothing \parallel \Delta_j \vdash e_j : T\ \forall j \\ \mathrm{ortho}_{T'}\ (e'_1, \ldots, e'_n) & \varnothing \parallel \Delta_j \vdash e'_j : T'\ \forall j \end{array}}{\vdash \mathtt{pmatch}\ {}_T\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_{T'} : T \rightsquigarrow T'}\ \textsc{T-Pmatch}$$

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

Fig. 10. Program typing rules.

For a type T, we define the set of classical expressions of that type as follows:

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

For a context Δ, we have a set of *valuations*, defined as

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

The space $\mathcal{H}(T)$ is spanned by an orthonormal basis $\{|v\rangle : v \in \mathbb{V}(T)\}$, where we define

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

The basis states for the space $\mathcal{H}(\Delta)$ are defined by $|\tau\rangle$ for valuations $\tau \in \mathbb{V}(\Delta)$.

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

Now, we can define the denotational semantics of Qunity. For pure expression semantics, we say that 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 e. Here σ is a valuation of Γ, representing "classical data". For mixed expression semantics, we have that if $\Gamma \parallel \Delta \Vdash e : T$ and $\sigma \in \mathbb{V}(\Gamma)$, then $[\![\sigma : \Gamma \parallel \Delta \Vdash e : T]\!] \in \mathcal{L}(\mathcal{L}(\mathcal{H}(\Delta)), \mathcal{L}(\mathcal{H}(T)))$, defining a superoperator acting on the space of density matrices. For program semantics, we have that $[\![\vdash f : T \rightsquigarrow T']\!] \in \mathcal{L}(\mathcal{H}(T), \mathcal{H}(T'))$, and $[\![\vdash f : T \Rrightarrow T']\!] \in \mathcal{L}(\mathcal{L}(\mathcal{H}(T)), \mathcal{L}(\mathcal{H}(T')))$.

$$
\begin{aligned}
\llbracket \sigma : \Gamma \parallel \varnothing \vdash () : \mathtt{Unit} \rrbracket \, |\varnothing\rangle &\coloneqq |()\rangle \\
\llbracket \sigma : \Gamma \parallel \varnothing \vdash x : T \rrbracket \, |\varnothing\rangle &\coloneqq |\sigma(x)\rangle \\
\llbracket \sigma : \Gamma \parallel x : T \vdash x : T \rrbracket \, |x \mapsto v\rangle &\coloneqq |v\rangle \\
\llbracket \sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1 \rrbracket \, |\tau, \tau_0, \tau_1\rangle &\coloneqq \llbracket \sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0 \rrbracket \, |\tau, \tau_0\rangle \\
&\quad \otimes \llbracket \sigma : \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1 \rrbracket \, |\tau, \tau_1\rangle \\
\llbracket \sigma : \Gamma \parallel \Delta, \Delta' \vdash \mathtt{ctrl}\ e \underset{T}{\begin{Bmatrix} e_1 \mapsto e_1' \\ \cdots \\ e_n \mapsto e_n' \end{Bmatrix}}_{T'} : T' \rrbracket \, |\tau, \tau'\rangle &\coloneqq \sum_{v \in \mathbb{V}(T)} \langle v| \llbracket \Gamma \parallel \Delta \Vdash e : T \rrbracket \, (|\sigma, \tau\rangle\langle\sigma, \tau|) \, |v\rangle \\
&\quad \cdot \sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| \llbracket \varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T \rrbracket^\dagger \, |v\rangle \\
&\quad \cdot \llbracket \sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e_j' : T' \rrbracket \, |\tau, \tau'\rangle \\
\llbracket \sigma : \Gamma \parallel \Delta \vdash f \ e : T' \rrbracket \, |\tau\rangle &\coloneqq \llbracket \vdash f : T \rightsquigarrow T' \rrbracket \llbracket \sigma : \Gamma \parallel \Delta \vdash e : T \rrbracket \, |\tau\rangle
\end{aligned}
$$

Fig. 11. Pure expression semantics.

$$
\begin{aligned}
\llbracket \vdash \mathtt{u_3}(r_\theta, r_\phi, r_\lambda) : \mathrm{Bit} \rightsquigarrow \mathrm{Bit} \rrbracket \, |0\rangle &\coloneqq \cos(r_\theta/2) \, |0\rangle + e^{ir_\phi} \sin(r_\theta/2) \, |1\rangle \\
\llbracket \vdash \mathtt{u_3}(r_\theta, r_\phi, r_\lambda) : \mathrm{Bit} \rightsquigarrow \mathrm{Bit} \rrbracket \, |1\rangle &\coloneqq -e^{ir_\lambda} \sin(r_\theta/2) \, |0\rangle + e^{i(r_\phi + r_\lambda)} \cos(r_\theta/2) \, |1\rangle \\
\llbracket \vdash \mathtt{left}_{T_0 \oplus T_1} : T_0 \rightsquigarrow T_0 \oplus T_1 \rrbracket \, |v\rangle &\coloneqq \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 \, |v\rangle &\coloneqq \left| \mathtt{right}_{T_0 \oplus T_1} \, v \right\rangle \\
\llbracket \vdash \lambda e \overset{T}{\mapsto} e' : T \rightsquigarrow T' \rrbracket \, |v\rangle &\coloneqq \llbracket \varnothing : \varnothing \parallel \Delta \vdash e' : T' \rrbracket \llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger \, |v\rangle \\
\llbracket \vdash \mathtt{rphase}_T \begin{Bmatrix} e \mapsto r \\ \mathtt{else} \mapsto r' \end{Bmatrix} : T \rightsquigarrow T \rrbracket \, |v\rangle &\coloneqq e^{ir} \llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket \llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger \, |v\rangle \\
&\quad + e^{ir'} \left(\mathbb{I} - \llbracket \varnothing \parallel \Delta \vdash e : T \rrbracket \llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger \right) |v\rangle \\
\llbracket \vdash \mathtt{pmatch} \underset{T}{\begin{Bmatrix} e_1 \mapsto e_1' \\ \cdots \\ e_n \mapsto e_n' \end{Bmatrix}}_{T'} : T \rightsquigarrow T' \rrbracket \, |v\rangle &\coloneqq \sum_{j=1}^{n} \llbracket \varnothing : \varnothing \parallel \Delta_j \vdash e_j' : T' \rrbracket \llbracket \varnothing : \varnothing \parallel \Delta_j \vdash e_j : T \rrbracket^\dagger \, |v\rangle
\end{aligned}
$$

Fig. 12. Pure program semantics.

$$
\begin{aligned}
[\![\sigma : \Gamma \parallel \Delta \Vdash e : T]\!]\left(|\tau\rangle\langle\tau'|\right) &\coloneqq [\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]|\tau\rangle\langle\tau'|[\![\sigma : \Gamma \parallel \Delta \vdash e : T]\!]^\dagger \\
[\![\sigma : \Gamma \parallel \Delta, \Delta_0 \Vdash e : T]\!]\left(\rho \otimes \rho_0\right) &\coloneqq \mathrm{tr}(\rho_0)[\![\sigma : \Gamma \parallel \Delta \Vdash e : T]\!](\rho) \\
[\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \Vdash (e_0, e_1) : T_0 \otimes T_1]\!]\left(|\tau, \tau_0, \tau_1\rangle\langle\tau', \tau_0', \tau_1'|\right) &\coloneqq [\![\sigma : \Gamma \parallel \Delta, \Delta_0 \Vdash e_0 : T_0]\!]\left(|\tau, \tau_0\rangle\langle\tau', \tau_0'|\right) \\
&\quad \otimes [\![\sigma : \Gamma \parallel \Delta, \Delta_1 \Vdash e_1 : T_1]\!]\left(|\tau, \tau_1\rangle\langle\tau', \tau_1'|\right) \\
[\![\sigma : \Gamma \parallel \Delta_0, \Delta_1 \Vdash \texttt{try}\ e_0\ \texttt{catch}\ e_1 : T]\!]\left(\rho_0 \otimes \rho_1\right) &\coloneqq \mathrm{tr}(\rho_1)[\![\sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T]\!](\rho_0) \\
&\quad + (\mathrm{tr}(\rho_0) - \mathrm{tr}([\![\sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T]\!](\rho_0))) \\
&\quad \cdot [\![\sigma : \Gamma \parallel \Delta_1 \Vdash e_1 : T]\!](\rho_1)
\end{aligned}
$$

$$
\begin{aligned}
[\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \Vdash \texttt{match}\ e\ {}_{T}\begin{Bmatrix} e_1 \mapsto e_1' \\ \cdots \\ e_n \mapsto e_n' \end{Bmatrix}_{T'} : T']\!]\left(|\tau, \tau_0, \tau_1\rangle \left\langle \tau', \tau_0', \tau_1' \right|\right) \coloneqq \\
= \sum_{v \in \mathbb{V}(T)} \langle v| \left([\![\sigma : \Gamma \parallel \Delta, \Delta_0 \Vdash e : T]\!]\left(|\tau, \tau_0\rangle \left\langle \tau', \tau_0' \right|\right)\right) |v\rangle \cdot \\
\cdot \sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![\varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T]\!]^\dagger |v\rangle \cdot \\
\cdot [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e_j' : T']\!]\left(|\tau, \tau_1\rangle \left\langle \tau', \tau_1' \right|\right)
\end{aligned}
$$

Fig. 13. Mixed expression semantics.

$$
\begin{aligned}
&[\![\vdash f : T \Rrightarrow T']\!](|v\rangle\langle v'|) \coloneqq [\![\vdash f : T \rightsquigarrow T']\!]|v\rangle\langle v'|[\![\vdash f : T \rightsquigarrow T']\!]^\dagger \\
&[\![\vdash \lambda e \overset{T}{\mapsto} e' : T \Rrightarrow T']\!](|v\rangle\langle v'|) \coloneqq \\
&\quad [\![\varnothing : \varnothing \parallel \Delta \Vdash e' : T']\!]\left([\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!]^\dagger |v\rangle\langle v'| [\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!]\right)
\end{aligned}
$$

Fig. 14. Mixed program semantics.

B Grammar of the Surface Syntax

A type name ⟨*tname*⟩ consists of a capital letter followed by zero or more alphanumeric characters, underscores, or apostrophes. An expression name ⟨*ename*⟩ consists of a dollar sign **$** followed by one or more alphanumeric characters, underscores, or apostrophes. Program names ⟨*fname*⟩, real names ⟨*rname*⟩, and type variables ⟨*tvar*⟩ are similar, using **@**, **#**, and **'** respectively. Quantum variables ⟨*qvar*⟩ must start with a lowercase letter or underscore. Real constants ⟨*const*⟩ are integers (one or more digits possibly preceded by a minus sign).

⟨*qfile*⟩ ::= ⟨*def*⟩* ⟨*e*⟩

⟨*def*⟩ ::= **type** ⟨*tname*⟩ ⟨*sig*⟩ **:=** ⟨*t*⟩ **end**
| **type** ⟨*tname*⟩ ⟨*sig*⟩ **:=** (| ⟨*tname*⟩ | ⟨*fname*⟩ **of** ⟨*t*⟩)+ **end**
| **def** ⟨*ename*⟩ ⟨*sig*⟩ **:** ⟨*t*⟩ **:=** ⟨*e*⟩ **end**
| **def** ⟨*fname*⟩ ⟨*sig*⟩ **:** ⟨*t*⟩ **->** ⟨*t*⟩ **:=** ⟨*f*⟩ **end**
| **def** ⟨*rname*⟩ ⟨*sig*⟩ **:=** ⟨*r*⟩ **end**

⟨*sig*⟩ ::= **{** ⟨*param*⟩ (**,** ⟨*param*⟩)* **}**

⟨*param*⟩ ::= ⟨*tvar*⟩
| ⟨*tname*⟩ **:** ⟨*t*⟩
| ⟨*fname*⟩ **:** ⟨*t*⟩ **->** ⟨*t*⟩
| ⟨*rname*⟩

⟨*ge*⟩ ::= ⟨*t*⟩ | ⟨*e*⟩ | ⟨*f*⟩ | ⟨*r*⟩

⟨*t*⟩ ::= **Void**
| **Unit**
| ⟨*t*⟩ ***** ⟨*t*⟩
| ⟨*tvar*⟩
| ⟨*tname*⟩ (**{** ⟨*ge*⟩, (**,** ⟨*ge*⟩)* **}**)?
| **(** ⟨*t*⟩ **)**
| **if** ⟨*be*⟩ **then** ⟨*t*⟩ **else** ⟨*t*⟩ **endif**

⟨*e*⟩ ::= **()**
| ⟨*qvar*⟩
| **(** ⟨*e*⟩ **,** ⟨*e*⟩ **)**
| **ctrl** ⟨*e*⟩ **[** (⟨*e*⟩ **->** ⟨*e*⟩ **;**)* (**else** -> ⟨*e*⟩)? **]**
| **match** ⟨*e*⟩ **[** (⟨*e*⟩ **->** ⟨*e*⟩ **;**)* (**else** -> ⟨*e*⟩)? **]**
| **try** ⟨*e*⟩ **catch** ⟨*e*⟩
| ⟨*f*⟩ **(** ⟨*e*⟩ **)**
| ⟨*e*⟩ **|>** ⟨*f*⟩
| **let** ⟨*e*⟩ **=** ⟨*e*⟩ **in** ⟨*e*⟩
| ⟨*ename*⟩ (**{** ⟨*ge*⟩, (**,** ⟨*ge*⟩)* **}**)?
| **(** ⟨*e*⟩ **)**
| **if** ⟨*be*⟩ **then** ⟨*e*⟩ **else** ⟨*e*⟩ **endif**

⟨*f*⟩ ::= **u3 {** ⟨*r*⟩ **,** ⟨*r*⟩ **,** ⟨*r*⟩ **}**
| **lambda** ⟨*e*⟩ **->** ⟨*e*⟩
| **gphase {** ⟨*r*⟩ **}**
| **rphase {** ⟨*e*⟩ **,** ⟨*r*⟩ **,** ⟨*r*⟩ **}**
| **pmatch [** (⟨*e*⟩ **->** ⟨*e*⟩ **;**)* **]**

```
    | ⟨fname⟩ ({ ⟨ge⟩, (, ⟨ge⟩)* })?
    | ( ⟨f⟩ )
    | if ⟨be⟩ then ⟨f⟩ else ⟨f⟩ endif

⟨r⟩ ::= pi
    | euler
    | ⟨const⟩
    | ⟨r⟩ + ⟨r⟩
    | ⟨r⟩ - ⟨r⟩
    | ⟨r⟩ * ⟨r⟩
    | ⟨r⟩ / ⟨r⟩
    | ⟨r⟩ ^ ⟨r⟩
    | ⟨r⟩ % ⟨r⟩
    | sin ( ⟨r⟩ )
    | cos ( ⟨r⟩ )
    | tan ( ⟨r⟩ )
    | arcsin ( ⟨r⟩ )
    | arccos ( ⟨r⟩ )
    | arctan ( ⟨r⟩ )
    | exp ( ⟨r⟩ )
    | ln ( ⟨r⟩ )
    | log2 ( ⟨r⟩ )
    | sqrt ( ⟨r⟩ )
    | ceil ( ⟨r⟩ )
    | floor ( ⟨r⟩ )
    | ⟨rname⟩ ({ ⟨ge⟩, (, ⟨ge⟩)* })?
    | ( ⟨r⟩ )
    | if ⟨be⟩ then ⟨r⟩ else ⟨r⟩ endif

⟨be⟩ ::= ! ⟨be⟩
    | ⟨be⟩ && ⟨be⟩
    | ⟨be⟩ || ⟨be⟩
    | ⟨r⟩ ⟨cmp⟩ ⟨r⟩
    | ( ⟨be⟩ )

⟨cmp⟩ ::= = | != | <= | < | >= | >
```

C Comparison of Control Flow Constructs

	`ctrl`	`match`	`pmatch`
What kind of semantics?	Pure expression	Mixed expression	Pure program
Restrictions	• LHS must satisfy orthogonality • LHS must be pure • RHS must be pure • RHS must satisfy erasure judgment for scrutinee quantum context • LHS must be classical	• LHS must satisfy orthogonality • LHS must be pure • LHS must be classical	• LHS must satisfy orthogonality • RHS must satisfy orthogonality • LHS must be pure • RHS must be pure • Each pair of matching expressions must share the same (quantum) context • Since it is a program, it is typed without context and does not contain a scrutinee directly
Benefits	• Scrutinee can be mixed while the entire expression is still pure • Can use classical context variables originating from an outer `ctrl` or `match` in an unrestricted way • Can be used to apply relative phases (conditional global phases)	• RHS can be mixed • Erasure judgment is not required • Can use classical context variables originating from an outer `ctrl` or `match` in an unrestricted way • No "unavoidable source of error" when scrutinee is not classical	• LHS and RHS can be in an arbitrary basis • Erasure judgment is not required • Can be used to apply relative phases (conditional global phases)
Can be used in a pattern / has an adjoint	Yes	No	Yes
Can be used in the LHS of a `ctrl` or `match`	No	No	No
Can be used in the RHS of a `ctrl`	Yes	No	Yes

Can be used the RHS of a `match`	Yes	Yes	Yes
Can be used in the LHS or RHS of a `pmatch`	Yes, if the isometry judgment holds	No	Yes, if the isometry judgment holds
Can contain a mixed expression in the scrutinee	Yes	Yes	N/A
Can contain a mixed expression on the LHS	No	No	No
Can contain a mixed expression on the RHS	No	Yes	No
Applying a global phase to a RHS expression makes a semantic difference	Yes	No	Yes

When to use?	• You want to control something on a subroutine which may involve measurement/decoherence, but you want your entire expression to be reversible • You want to conditionally apply a global phase depending on the result of a mixed expression • You are able to keep the scrutinee variables on the RHS in satisfaction of the erasure judgment	• You do not care whether your expression is reversible • You want to conditionally apply some expressions that involve measurement/decoherence • You want something that most closely corresponds to a `match` in classical programming languages • What you want to do can be accomplished by measuring the outcome of the input expression and then evaluating the appropriate RHS expression • You want to avoid the erasure requirement	• You want to reversibly map between two different bases • The LHS patterns are not in the classical basis. • You want to avoid the erasure requirement • You do not need any outside variables

D Benchmark Test Programs

For these examples, the quantum registers in all circuits were initialized to a superposition state to prevent classical propagation optimizations from taking place. Not all of the examples are useful quantum algorithms, but they serve to evaluate the performance of the Qunity compiler.

D.1 Phase Conditioned on AND of 5 Qubits

Qunity implementation:

```
def @multi_and{#n} : Array{#n, Bit} → Bit :=
  if #n = 0 then
    lambda () → $1
  else
    lambda (x0, x1) → @and(x0, @multi_and{#n - 1}(x1))
  endif
end

$repeated{5, Bit, $plus} ▷ lambda x → ctrl @multi_and{5}(x) [
  $0 → x;
  $1 → x ▷ gphase{pi}
]
```

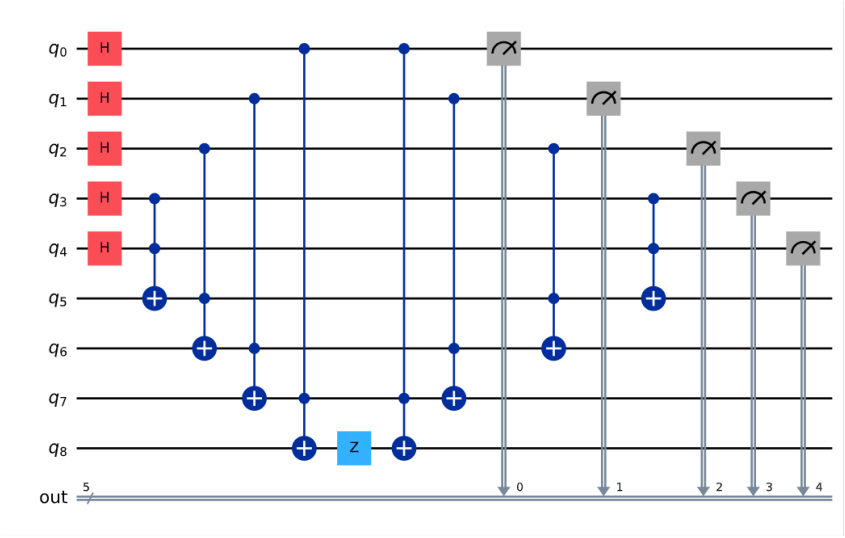


Fig. 15. Compiled circuit for phase conditioned on AND of 5 qubits.

Qiskit implementation:

```
circuit = QuantumCircuit(5, 5)
circuit.h(range(5))
circuit.mcp(math.pi, [0, 1, 2, 3], [4])
circuit.measure(range(5), range(5))
```

D.2 Quantum Fourier Transform (5 Qubits)

Qunity implementation:

```
def @couple{#k} : Bit * Bit → Bit * Bit :=
  lambda (x0, x1) → (x1, x0) ▷ rphase{($1, $1), 2 * pi / (2 ^
    #k), 0}
end

def @rotations{#n} : Array{#n, Bit} → Array{#n, Bit} :=
  if #n <= 0 then
    @id{Unit}
  else if #n = 1 then
    lambda (x, ()) → (@had(x), ())
  else
    lambda (x0, x) →
      let (x0, (y0', y)) = (x0, @rotations{#n - 1}(x)) in
      let ((y0, y1), y) = (@couple{#n}(x0, y0'), y) in
      (y0, (y1, y))
  endif
  endif
end

def @qft{#n} : Array{#n, Bit} → Array{#n, Bit} :=
  if #n <= 0 then
    @id{Unit}
  else
    lambda x →
      let (x0, x') = @rotations{#n}(x) in
      (x0, @qft{#n - 1}(x'))
  endif
end

@qft{5}($0, ($plus, ($plus, ($0, ($0, ())))))) ▷ @reverse{5, Bit}
```

Qiskit implementation:

```
from qiskit.circuit.library import QFT
circuit = QuantumCircuit(5, 5)
circuit.h([1, 2])
circuit.append(QFT(5), [0, 1, 2, 3, 4])
circuit.measure([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
```

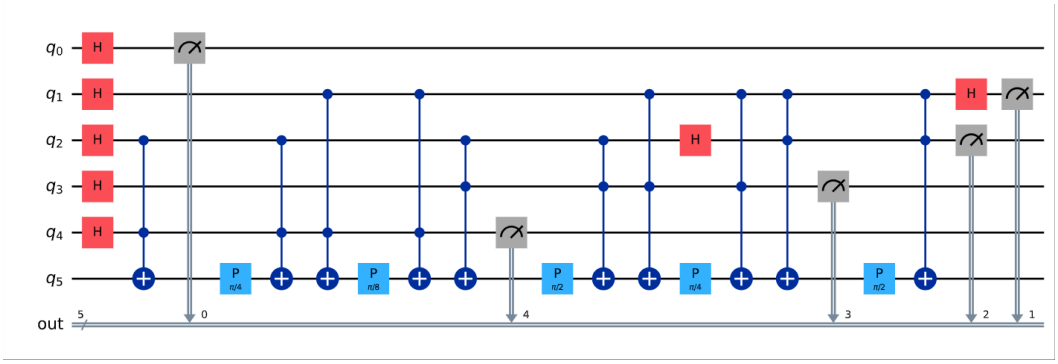


Fig. 16. Compiled circuit for the Quantum Fourier Transform.

D.3 Phase Estimation Example (5 Qubits)

Qunity implementation:

```
def @apply_phase{#n, #p} : Num{#n} → Num{#n} :=
  if #n = 0 then
    @id{Unit}
  else
    lambda (x0, x') →
    (ctrl x0 [
      $0 → x0;
      $1 → x0 ▷ gphase{2 * pi * #p}
    ], @apply_phase{#n - 1, 2 * #p}(x'))
  endif
end

def $phase_estimation{#n, #p} : Num{#n} :=
  $repeated{#n, Bit, $plus}
  ▷ @apply_phase{#n, #p}
  ▷ @adjoint{Num{#n}, Num{#n}, @qft{#n}}
  ▷ @reverse{#n, Bit}
end

$phase_estimation{5, 1/3}
```

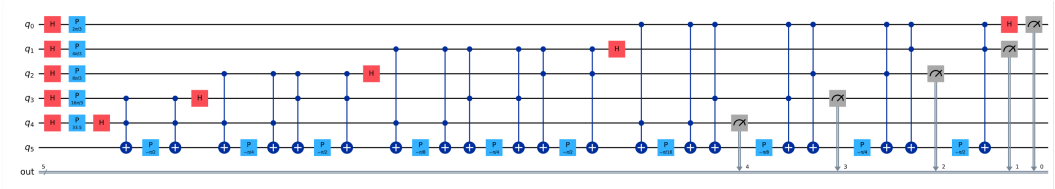


Fig. 17. Compiled circuit for phase estimation.

Qiskit implementation:

```
from qiskit.circuit.library import PhaseEstimation,
    GlobalPhaseGate

circuit = QuantumCircuit(5, 5)
circuit.append(PhaseEstimation(5, GlobalPhaseGate(2 * math.pi /
    3)), range(5))
circuit.measure(range(5), range(5))
```

D.4 Order Finding

For the Qunity implementation, see Section 3.1.

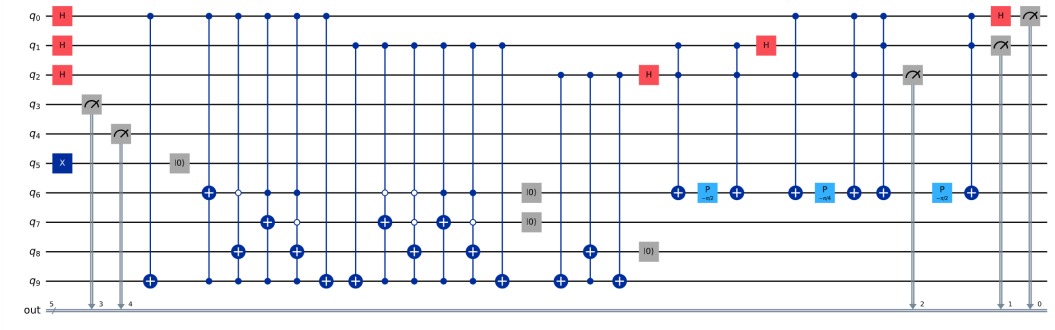


Fig. 18. Compiled circuit for order finding.

D.5 Reversible CDKM Adder (5 Bits)

Qunity implementation:

```
def @cdkm_maj : Bit * Bit * Bit → Bit * Bit * Bit :=
  lambda ((a, b), c) → ctrl c [
    $0 → (c, (a, b));
    $1 → (c, (@not(a), @not(b)))
  ] ▷ lambda (c, (a, b)) → ctrl (a, b) [
    ($1, $1) → ((a, b), @not(c));
    else → ((a, b), c);
  ]
end

def @cdkm_uma : Bit * Bit * Bit → Bit * Bit * Bit :=
  lambda ((a, b), c) → ctrl (a, b) [
    ($1, $1) → ((a, b), @not(c));
    else → ((a, b), c);
  ] ▷ lambda ((a, b), c) →
  (a, (b, (c, ()))) ▷ @cnot{3, 2, 0} ▷ @cnot{3, 0, 1} ▷
```

```
  lambda (a, (b, (c, ()))) → ((a, b), c)
end

def @rev_adder_helper{#n} : Num{#n} * Num{#n} * Bit → Num{#n} *
    Num{#n} * Bit :=
  if #n <= 0 then
    @id{Num{#n} * Num{#n} * Bit}
  else
    lambda (((a0, a1), (b0, b1)), c) →
    let (((ca, ba), c'), (a1, b1)) =
      (@cdkm_maj((c, b0), a0), (a1, b1)) in
    let ((ca, ba), ((a1, s1), c'')) =
      ((ca, ba), @rev_adder_helper{#n - 1}((a1, b1), c')) in
    let ((a1, s1), ((c, s0), a0)) =
      ((a1, s1), @cdkm_uma((ca, ba), c'')) in
    (((a0, a1), (s0, s1)), c)
  endif
end

def @rev_adder{#n} : Num{#n} * Num{#n} → Num{#n} * Num{#n} :=
  lambda (a, b) → ((a, b), $0) ▷ @rev_adder_helper{#n} ▷
   lambda (x, $0) → x
end

@rev_adder{5}($repeated{5, Bit, $plus}, $repeated{5, Bit, $plus})
```

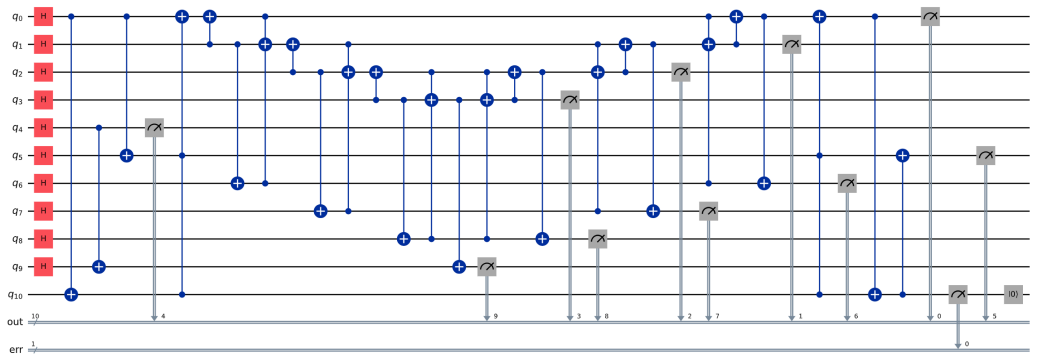


Fig. 19. Compiled circuit for the reversible CDKM adder.

Qiskit implementation:

```
from qiskit.circuit.library import CDKMRippleCarryAdder

circuit = QuantumCircuit(11, 10)
```

```
circuit.h(range(10))
circuit.append(CDKMRippleCarryAdder(5, kind="fixed"), range(11))
circuit.measure(range(10), range(10))
```

D.6 Grover (5-Bit Match Oracle, 1 Iteration)

Qunity implementation (for the definition of `$grover`, see Section 3.2):

```
def #n := 5 end
def $answer : Num{#n} := ($0, ($1, ($1, ($0, ($0, ())))))) end

def @f : Num{#n} → Bit :=
  lambda x → ctrl x [
    @answer → (x, $1);
    else → (x, $0)
  ] ▷ @snd{Num{#n}, Bit}
end

$grover{Num{#n}, $repeated{#n, Bit, $plus}, @f, 1}
```

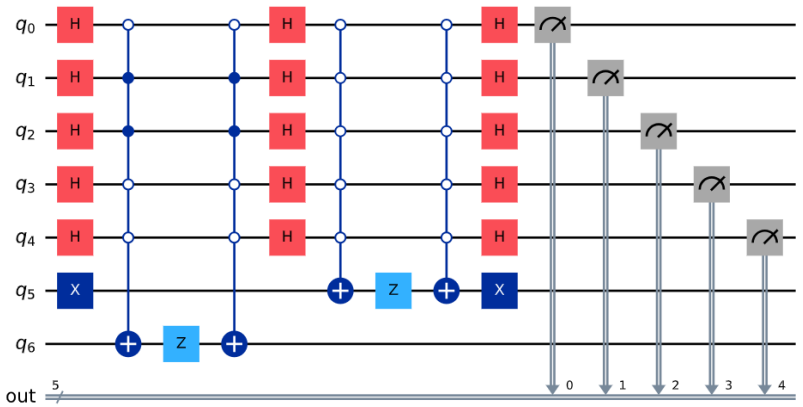


Fig. 20. Compiled circuit for Grover's algorithm with a simple `match` oracle.

Qiskit implementation:

```
circuit = QuantumCircuit(6, 5)
circuit.x(5)
circuit.h(range(6))
circuit.mcx([0, 1, 2, 3, 4], 5, ctrl_state="01100")
circuit.h(range(5))
circuit.mcx([0, 1, 2, 3, 4], 5, ctrl_state="00000")
circuit.h(range(5))
circuit.measure([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
```

D.7 Grover (List Sum Oracle)

Qunity implementation (for the definition of `$grover`, see Section 3.2):

```
def $equal_superpos_list{#n} : List{#n, Bit} :=
  if #n = 0 then
     $ListEmpty {0, Bit}
  else
    $0
    ▷ u3{2 * arccos(sqrt(1 / (2 ^ (#n + 1) - 1))), 0, 0}
    ▷ pmatch [
      $0 → $ListEmpty {#n, Bit};
      $1 → @ListCons {#n, Bit}($plus,
                               $equal_superpos_list{#n - 1})
    ]
  endif
end

def @is_odd_sum{#n} : List{#n, Bit} → Bit :=
  if #n = 0 then
    lambda l → $0
  else
    lambda l → match l [
       $ListEmpty {#n, Bit} → $0;
       @ListCons {#n, Bit}($0, l') → @is_odd_sum{#n - 1}(l');
       @ListCons {#n, Bit}($1, l') → @not(@is_odd_sum{#n - 1}(l'))
    ]
  endif
end

def #n := 2 end

$grover{List{#n, Bit}, $equal_superpos_list{#n},
                       @is_odd_sum{#n}, 1}
```

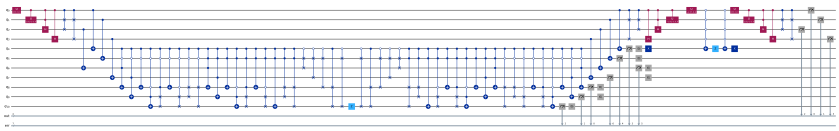


Fig. 21. Compiled circuit for Grover's algorithm with a "list sum" oracle.

Qiskit implementation:

```
circuit = QuantumCircuit(5, 4)
circuit.x(4)
circuit.h(4)
circuit.ry(2 * math.acos(math.sqrt(1 / (2**3 - 1))), 0)
circuit.ch(0, 1)
circuit.cry(2 * math.acos(math.sqrt(1 / (2**2 - 1))), 0, 2)
circuit.ch(2, 3)
circuit.cx(1, 4)
circuit.cx(3, 4)
circuit.ch(2, 3)
circuit.cry(-2 * math.acos(math.sqrt(1 / (2**2 - 1))), 0, 2)
circuit.ch(0, 1)
circuit.ry(-2 * math.acos(math.sqrt(1 / (2**3 - 1))), 0)
circuit.mcx([0, 1, 2, 3], 4, ctrl_state="0000")
circuit.ry(2 * math.acos(math.sqrt(1 / (2**3 - 1))), 0)
circuit.ch(0, 1)
circuit.cry(2 * math.acos(math.sqrt(1 / (2**2 - 1))), 0, 2)
circuit.ch(2, 3)
circuit.measure([0, 1, 2, 3], [0, 1, 2, 3])
```

E Circuit-Style Programming

While Qunity allows for programming on a much higher level of abstraction than the quantum circuit model, it is possible to work with Qunity as if it was a low-level circuit language, by defining the following constructs:

```
def @gate_1q{#n, #i, @f : Bit → Bit} : Array{#n, Bit} → Array{
    #n, Bit} :=
  if #i <= 0 then
    lambda (x, y) → (@f(x), y)
  else
    lambda (x, y) → (x, @gate_1q{#n - 1, #i - 1, @f}(y))
  endif
end

def @controlled_1q{#n, #i, #j, @f : Bit → Bit} : Array{#n, Bit}
    → Array{#n, Bit} :=
  if #i > #j then
    lambda x → x
    ▷ @reverse{#n, Bit}
    ▷ @controlled_1q{#n, #n - 1 - #i, #n - 1 - #j, @f}
    ▷ @reverse{#n, Bit}
  else
    if #i <= 0 then
```

```
      lambda (x, y) → ctrl x [
        $0 → (x, y);
        $1 → (x, @gate_1q{#n - 1, #j - 1, @f}(y))
      ]
    else
      lambda (x, y) → (x, @cnot{#n - 1, #i - 1, #j - 1}(y))
    endif
  endif
end

def @cnot{#n, #i, #j} : Array{#n, Bit} → Array{#n, Bit} :=
  @controlled_1q{#n, #i, #j, @not}
end
```

This allows us to use a Qunity array as a quantum register and apply gates to it, accessing qubits by index. Since we have defined single-qubit gates and CNOT in this way, this can in principle be used to represent any quantum computation. If performance is critical, a Qunity programmer may switch to writing in circuit style for some subroutines, since these constructs compile efficiently to the corresponding low-level gates.

F The Orthogonality and Spanning Judgments

We generalize the orthogonality judgment (used for typing Qunity's control flow constructs), as well as the spanning judgment (now used for the isometry judgment in Appendix G). The orthogonality judgment in Figure 22 allows the Qunity typechecker to statically check whether a given set of expressions corresponds to a set of orthogonal subspaces in the Hilbert space corresponding to their type. The spanning judgment in Figure 23 corresponds to checking that the expressions are orthogonal *and* that the direct sum of their corresponding subspaces is the whole Hilbert space. Note that both of these judgments correspond to sufficient but not necessary conditions: anything that they recognize as orthogonal (spanning) must be orthogonal (spanning), but the converse is not true.

The main modifications are the O-IsoApp rule allowing the application of arbitrary *isometric* programs to all expressions in the set (which maintains orthogonality), and the S-UnApp rule allowing the application of arbitrary *unitary* programs to all expressions in the set (which maintains the spanning property). This depends on the *unitary judgment*, which is itself defined in terms of the isometry judgment (Appendix G) as follows:

$$\frac{\mathrm{iso}(f) \quad \dim(T) = \dim(T')}{\mathrm{un}(\vdash f : T \rightsquigarrow T')}\ \text{U-Prog}$$

That is, if a Qunity program has isometric semantics and maps between two types whose associated Hilbert spaces have the same dimension, it must have unitary semantics.

$$\frac{}{\mathrm{ortho}_{\mathtt{Void}}\ ()}\ \textsc{O-Void} \qquad \frac{}{\mathrm{ortho}_{\mathtt{Unit}}\ (())}\ \textsc{O-Unit} \qquad \frac{}{\mathrm{ortho}_{T}\ (x)}\ \textsc{O-Var}$$

$$\frac{\mathrm{ortho}_{T}\ (e_1, \ldots, e_n) \qquad \mathrm{iso}(\vdash f : T \rightsquigarrow T')}{\mathrm{ortho}_{T'}\ (f\ e_1, \ldots, f\ e_n)}\ \textsc{O-IsoApp}$$

$$\frac{\mathrm{ortho}_{T_0}\ (e_1, \ldots, e_n) \qquad \mathrm{ortho}_{T_1}\ (e'_1, \ldots, e'_{n'})}{\mathrm{ortho}_{T_0 \oplus T_1}\begin{pmatrix} \mathtt{left}_{T_0 \oplus T_1}\, e_1, \ldots, \mathtt{left}_{T_0 \oplus T_1}\, e_n, \\ \mathtt{right}_{T_0 \oplus T_1}\, e'_1, \ldots, \mathtt{right}_{T_0 \oplus T_1}\, e'_{n'} \end{pmatrix}}\ \textsc{O-Sum}$$

$$\frac{\begin{array}{c}\mathrm{ortho}_{T_0}\ (e_1, \ldots, e_m) \qquad \mathrm{ortho}_{T_1}\ (e'_{j,1}, \ldots, e'_{j,n_j})\ \ \forall j \\ \mathrm{FV}(e_j) \cap \bigcup_{k=1}^{n_m} \mathrm{FV}(e'_{j,k}) = \varnothing\ \forall j\end{array}}{\mathrm{ortho}_{T_0 \otimes T_1}\begin{pmatrix} (e_1, e'_{1,1}), \ldots, (e_1, e'_{1,n_1}), \\ \ldots, \\ (e_m, e'_{m,1}), \ldots, (e_m, e'_{m,n_m}) \end{pmatrix}}\ \textsc{O-Pair}$$

$$\frac{\mathrm{ortho}_{T}\ (e'_1, \ldots, e'_m) \qquad [e_1, \ldots, e_n] \text{ is a subsequence of } [e'_1, \ldots, e'_m]}{\mathrm{ortho}_{T}\ (e_1, \ldots, e_n)}\ \textsc{O-Sub}$$

Fig. 22. Orthogonality inference rules, modified from the original spanning rules to allow for arbitrary isometry application. Here, $\mathrm{FV}(e)$ means the set of free variables in e (this does not include variables in patterns).

$$\frac{}{\mathrm{spanning}_{\mathtt{Void}}\ ()}\ \textsc{S-Void} \qquad \frac{}{\mathrm{spanning}_{\mathtt{Unit}}\ (\mathtt{Unit})}\ \textsc{S-Unit} \qquad \frac{}{\mathrm{spanning}_{T}\ (x)}\ \textsc{S-Var}$$

$$\frac{\mathrm{spanning}_{T}\ (e_1, \ldots, e_n) \qquad \mathrm{un}(\vdash f : T \rightsquigarrow T')}{\mathrm{spanning}_{T'}\ (f\ e_1, \ldots, f\ e_n)}\ \textsc{S-UnApp}$$

$$\frac{\mathrm{spanning}_{T_0}\ (e_1, \ldots, e_n) \qquad \mathrm{spanning}_{T_1}\ (e'_1, \ldots, e'_{n'})}{\mathrm{spanning}_{T_0 \oplus T_1}\begin{pmatrix} \mathtt{left}_{T_0 \oplus T_1}\, e_1, \ldots, \mathtt{left}_{T_0 \oplus T_1}\, e_n, \\ \mathtt{right}_{T_0 \oplus T_1}\, e'_1, \ldots, \mathtt{right}_{T_0 \oplus T_1}\, e'_{n'} \end{pmatrix}}\ \textsc{S-Sum}$$

$$\frac{\begin{array}{c}\mathrm{spanning}_{T_0}\ (e_1, \ldots, e_m) \qquad \mathrm{spanning}_{T_1}\ (e'_{j,1}, \ldots, e'_{j,n_j})\ \ \forall j \\ \mathrm{FV}(e_j) \cap \bigcup_{k=1}^{n_m} \mathrm{FV}(e'_{j,k}) = \varnothing\ \forall j\end{array}}{\mathrm{spanning}_{T_0 \otimes T_1}\begin{pmatrix} (e_1, e'_{1,1}), \ldots, (e_1, e'_{1,n_1}), \\ \ldots, \\ (e_m, e'_{m,1}), \ldots, (e_m, e'_{m,n_m}) \end{pmatrix}}\ \textsc{S-Pair}$$

Fig. 23. Spanning inference rules, extended to allow the application of arbitrary unitary programs.

Lemma F.1. *Suppose that* $\text{ortho}_T\left(e_1, \ldots, e_n\right)$ *holds. Take any* e_i, e_j *with* $i \neq j$*. Let*

$$[\![e_i]\!] = [\![\varnothing : \varnothing \parallel \Delta_i \vdash e_i : T]\!] : \mathcal{H}(\Delta_i) \to \mathcal{H}(T)$$

and similarly for $[\![e_j]\!]$*. Then, the images of the operators* $[\![e_i]\!]$ *and* $[\![e_j]\!]$ *as subspaces of* $\mathcal{H}(T)$ *are orthogonal.*

Proof. We prove this by induction on the rule used to prove the orthogonality judgment.

O-Void, O-Unit, O-Var: The statement is vacuously true in these cases, since the sets contain less than two expressions.

O-IsoApp: In this case, we use Lemma G.1 to claim that the semantics $[\![\vdash f : T \rightsquigarrow T']\!]$ is isometric. Since an isometric operator applied to orthogonal vectors preserves their orthogonality, it must be that $[\![f e_i]\!]$ and $[\![f e_j]\!]$ are orthogonal subspaces of T'.

O-Sum: If the two expressions are both "left" or both "right", then the claim follows by induction, since we assume that both subsequences form orthogonal sets and the left and right injections are isometric. If one is "left" and the other is "right," the image of $\mathcal{H}(T_0)$ under left and the image of $\mathcal{H}(T_1)$ under right are orthogonal subspaces in $\mathcal{H}(T_0 \oplus T_1) = \mathcal{H}(T_0) \oplus \mathcal{H}(T_1)$.

O-Pair:

The O-Pair rule assumes that for all j, we have

$$\text{FV}(e_j) \cap \bigcup_{k=1}^{n_m} \text{FV}(e'_{j,k}) = \varnothing,$$

that is, e_j and $e'_{j,k}$ do not share any free variables. Thus, in the typing rule T-PurePair for the expression $(e_j, e'_{j,k})$ we must have $\Delta = \varnothing$, and so substituting quantum contexts Δ_j and $\Delta'_{j,k}$ for Δ_0, Δ_1, the semantics (omitting empty classical contexts for brevity) is given by

$$[\![\Delta_j, \Delta'_{j,k} \vdash (e_j, e'_{j,k}) : T_0 \otimes T_1]\!] \left|\tau_0, \tau_1\right\rangle = [\![\Delta_j, \vdash e_j : T_0]\!] \left|\tau_0\right\rangle \otimes [\![\Delta'_{j,k}, \vdash e'_j : T_1]\!] \left|\tau_1\right\rangle.$$

Consider two expressions $(e_i, e'_{i,k})$ and $(e_j, e'_{j,l})$. We then have that the images are

$$[\![(e_i, e'_{i,k})]\!](\mathcal{H}(\Delta_i, \Delta'_{i,k})) = [\![e_i]\!](\mathcal{H}(\Delta_i)) \otimes [\![e'_{i,k}]\!](\mathcal{H}(\Delta'_{i,k}))$$

$$[\![(e_j, e'_{j,l})]\!](\mathcal{H}(\Delta_j, \Delta'_{j,l})) = [\![e_j]\!](\mathcal{H}(\Delta_j)) \otimes [\![e'_{j,l}]\!](\mathcal{H}(\Delta'_{j,l})).$$

In the case where $i \neq j$, we have that $[\![e_i]\!](\mathcal{H}(\Delta_i))$ and $[\![e_j]\!](\mathcal{H}(\Delta_j))$ are orthogonal subspaces by the assumption that $\text{ortho}_{T_0}\left(e_1, \ldots, e_m\right)$. If $i = j$, then $[\![e'_{i,k}]\!](\mathcal{H}(\Delta'_{i,k}))$ and $[\![e'_{j,l}]\!](\mathcal{H}(\Delta'_{j,l}))$ are orthogonal subspaces by the assumption that $\text{ortho}_{T_1}\left(e'_{j,1}, \ldots, e'_{j,n_j}\right)$. So, in all cases, the claim holds.

O-Sub: The claim holds trivially in this case, since any subsequence of a sequence of orthogonal expressions is orthogonal. This completes the proof.

□

Lemma F.2. *Suppose that* $\text{spanning}_T\left(e_1, \ldots, e_n\right)$ *holds. Then,* $\text{ortho}_T\left(e_1, \ldots, e_n\right)$ *holds and*

$$\bigoplus_{j=1}^{n} [\![e_j]\!](\mathcal{H}(\Delta_j)) \cong \mathcal{H}(T).$$

Proof. The first claim is clear, since each typing rule for the spanning judgment is stronger than the corresponding rule for the orthogonality judgment. Now, we prove the spanning property by induction:

S-Void: The claim holds trivially since $\mathcal{H}(\texttt{Void}) = \{0\}$, which we consider to be the direct sum of an empty set of spaces.

S-Unit: The image of $[\![\texttt{()}]\!]$ is $\text{span}\{|\texttt{()}\rangle\} = \mathbb{C} = \mathcal{H}(\texttt{Unit})$.

S-Var: Since $[\![x : T]\!]$ maps every $|x \mapsto v\rangle \in \mathcal{H}(\Delta)$ for $v \in \mathbb{V}(T)$ to $|v\rangle \in \mathcal{H}(T)$, it is clear by definition of the Hilbert spaces that it implements an isomorphism $\mathcal{H}(\Delta) \to \mathcal{H}(T)$, and thus its image is all of $\mathcal{H}(T)$.

S-UnApp: By the unitary rule, we must have that T and T' have the same dimension and $[\![f]\!]$ is an isometry (and thus a unitary). Thus, it sends orthogonal vectors in $\mathcal{H}(T)$ to orthogonal vectors in $\mathcal{H}(T')$. Since the dimensions are equal and the images of $[\![e_1]\!], \ldots, [\![e_n]\!]$ contain a basis of $\mathcal{H}(T)$, we must have that the images of $[\![fe_1]\!], \ldots, [\![fe_n]\!]$ contain a basis of $\mathcal{H}(T')$ and so their direct sum is all of $\mathcal{H}(T')$. Note that this depends on the correctness of the isometry judgment (G), which in turn depends on the spanning judgment, but this is not an issue since we are really performing simultaneous induction.

S-Sum: By assumption, we have $\text{spanning}_{T_0}\left(e_1, \ldots, e_n\right)$ and $\text{spanning}_{T_1}\left(e'_1, \ldots, e'_{n'}\right)$, so the direct sum of the images contains all of

$$\left\{\left|\texttt{left}_{T_0 \oplus T_1}\, v_0\right\rangle \mid v_0 \in \mathbb{V}(T_0)\right\} \cup \left\{\left|\texttt{right}_{T_0 \oplus T_1}\, v_1\right\rangle \mid v_1 \in \mathbb{V}(T_1)\right\},$$

which forms a basis for $\mathcal{H}(T_0 \oplus T_1)$.

S-Pair: By assumption, we have $\text{spanning}_{T_0}\left(e_1, \ldots, e_m\right)$, which means that

$$\bigoplus_{j=1}^{m} [\![e_j]\!](\mathcal{H}(\Delta_j)) = \mathcal{H}(T_0).$$

And, since for each j, we have $\text{spanning}_{T_0}\left(e'_{j,1}, \ldots, e'_{j,n_j}\right)$, we also have

$$\bigoplus_{k=1}^{n_j} [\![e'_{j,k}]\!](\mathcal{H}(\Delta'_{j,k})) = \mathcal{H}(T_1),$$

so it must be the case that

$$\begin{aligned}
&\bigoplus_{j=1}^{m} \bigoplus_{k=1}^{n_j} [\![(e_j, e'_{j,k})]\!](\mathcal{H}(\Delta_j, \Delta'_{j,k})) \\
&= \bigoplus_{j=1}^{m} \bigoplus_{k=1}^{n_j} \text{span}\{[\![(e_j, e'_{j,k})]\!]\,|\tau_0, \tau_1\rangle \mid \tau_0 \in \mathbb{V}(\Delta_j), \tau_1 \in \mathbb{V}(\Delta'_{j,k})\} = \\
&= \bigoplus_{j=1}^{m} \bigoplus_{k=1}^{n_j} \text{span}\{[\![e_j]\!]\,|\tau_0\rangle \otimes [\![e'_{j,k}]\!]\,|\tau_1\rangle \mid \tau_0 \in \mathbb{V}(\Delta_j), \tau_1 \in \mathbb{V}(\Delta'_{j,k})\} = \\
&= \bigoplus_{j=1}^{m} [\![e_j]\!](\mathcal{H}(\Delta_j)) \otimes \bigoplus_{k=1}^{n_j} [\![e'_{j,k}]\!](\mathcal{H}(\Delta'_{j,k})) = \mathcal{H}(T_0) \otimes \mathcal{H}(T_1) = \mathcal{H}(T_0 \otimes T_1).
\end{aligned}$$

Note that in the first equality above we used the assumption that the sets of free variables are disjoint to decompose the context spaces as tensor products. This completes the proof. □

G The Isometry Judgment

The isometry judgment in Figure 24 allows the Qunity typechecker to determine if the given expression or program is guaranteed to have isometric (norm-preserving) semantics, and thus never raise an error. This information can be used by the compiler to perform certain optimizations, such as deleting certain flag qubits that are known to always be in the $|0\rangle$ state due to the fact that the corresponding Qunity expression is isometric.

$$\frac{}{\mathrm{iso}(())}\ \textsc{I-Unit} \qquad \frac{}{\mathrm{iso}(x)}\ \textsc{I-Var} \qquad \frac{\mathrm{iso}(e_0) \qquad \mathrm{iso}(e_1)}{\mathrm{iso}((e_0, e_1))}\ \textsc{I-Pair}$$

$$\frac{\mathrm{classical}(e) \qquad \mathrm{iso}(e) \qquad \mathrm{spanning}_T\left(e_1, \ldots, e_n\right) \qquad \mathrm{iso}(e'_j)\ \forall j}{\mathrm{iso}\left(\texttt{ctrl}\ e\ \underset{T}{\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}}_{T'}\right)}\ \textsc{I-Ctrl}$$

$$\frac{\mathrm{classical}(e) \qquad \mathrm{iso}(e) \qquad \mathrm{spanning}_T\left(e_1, \ldots, e_n\right) \qquad \mathrm{iso}(e'_j)\ \forall j}{\mathrm{iso}\left(\texttt{match}\ e\ \underset{T}{\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}}_{T'}\right)}\ \textsc{I-Match}$$

$$\frac{\mathrm{iso}(e_0)}{\mathrm{iso}(\texttt{try}\ e_0\ \texttt{catch}\ e_1)}\ \textsc{I-Try} \qquad \frac{\mathrm{iso}(e_0)}{\mathrm{iso}(\texttt{try}\ e_0\ \texttt{catch}\ e_1)}\ \textsc{I-Catch} \qquad \frac{\mathrm{iso}(f) \qquad \mathrm{iso}(e)}{\mathrm{iso}(f\ e)}\ \textsc{I-App}$$

$$\frac{}{\mathrm{iso}(\mathsf{u}_3(r_\theta, r_\phi, r_\lambda))}\ \textsc{I-Gate} \qquad \frac{}{\mathrm{iso}(\texttt{left}_{T_0 \oplus T_1})}\ \textsc{I-Left} \qquad \frac{}{\mathrm{iso}(\texttt{right}_{T_0 \oplus T_1})}\ \textsc{I-Right}$$

$$\frac{\mathrm{spanning}_T\left(e\right) \qquad \mathrm{iso}(e')}{\mathrm{iso}(\lambda e \overset{T}{\mapsto} e')}\ \textsc{I-Abs} \qquad \frac{\mathrm{iso}(e)}{\mathrm{iso}\left(\texttt{rphase}_T \begin{Bmatrix} e \mapsto r \\ \texttt{else} \mapsto r' \end{Bmatrix}\right)}\ \textsc{I-Rphase}$$

$$\frac{\mathrm{spanning}_T\left(e_1, \ldots, e_n\right) \qquad \mathrm{iso}(e'_1) \quad \cdots \quad \mathrm{iso}(e'_n)}{\mathrm{iso}\left(\texttt{pmatch}\ \underset{T}{\begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}}_{T'}\right)}\ \textsc{I-Pmatch}$$

Fig. 24. Isometry inference rules.

Lemma G.1.

- *If* $\Gamma \parallel \Delta \vdash e : T$ *holds and* $\mathrm{iso}(e)$*, then for all* $|\psi\rangle \in \mathcal{H}(\Delta)$*, we have*
$$\left\| [\![\Gamma \parallel \Delta \vdash e : T]\!]\, |\psi\rangle \right\| = \||\psi\rangle\| .$$
- *If* $\Gamma \parallel \Delta \Vdash e : T$ *holds and* $\mathrm{iso}(e)$*, then for all* $\rho \in \mathcal{L}(\mathcal{H}(\Delta))$*, we have*
$$\mathrm{tr}\, [\![\Gamma \parallel \Delta \Vdash e : T]\!](\rho) = \mathrm{tr}(\rho).$$
- *If* $\vdash e : T \rightsquigarrow T'$ *holds and* $\mathrm{iso}(e)$*, then for all* $|\psi\rangle \in \mathcal{H}(\Delta)$*, we have*
$$\left\| [\![\vdash e : T \rightsquigarrow T']\!]\, |\psi\rangle \right\| = \||\psi\rangle\| .$$
- *If* $\vdash e : T \Rrightarrow T'$ *holds and* $\mathrm{iso}(e)$*, then for all* $\rho \in \mathcal{L}(\mathcal{H}(T))$*, we have*
$$\mathrm{tr}\, [\![\vdash e : T \Rrightarrow T']\!](|\tau\rangle\langle\tau|) = \mathrm{tr}(\rho).$$

Proof. I-Unit:

$$\left\|[\![\sigma : \Gamma \parallel \Delta \vdash () : \mathtt{Unit}]\!]\,|\psi\rangle\right\| = |\langle\varnothing|\psi\rangle|\,\||()\rangle\| = \||\psi\rangle\|\,.$$

I-Var:

$$\begin{aligned}&\left\|[\![\sigma : \Gamma \parallel \Delta \vdash x : T]\!]\,|\psi\rangle\right\|^2 = \\ &= \left\|\sum_{v\in\mathbb{V}(T)} \langle v|\psi\rangle\,[\![\sigma : \Gamma \parallel \Delta \vdash x : T]\!]\,|x \mapsto v\rangle\right\|^2 = \left\|\sum_{v\in\mathbb{V}(T)} \langle v|\psi\rangle\,|v\rangle\right\|^2 = \sum_{v\in\mathbb{V}(T)} |\langle v|\psi\rangle|^2 = \||\psi\rangle\|^2\,.\end{aligned}$$

I-Pair:

$$\begin{aligned}&\left\|[\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1]\!]\,|\psi\rangle\right\|^2 = \\ &= \langle\psi|\,[\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1]\!]^\dagger [\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1]\!]\,|\psi\rangle = \\ &= \langle\psi|\,[\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash (e_0, e_1) : T_0 \otimes T_1]\!]^\dagger \sum_{(\tau,\tau_0,\tau_1)\in\mathbb{V}(\Delta,\Delta_0,\Delta_1)} \langle\tau,\tau_0,\tau_1|\psi\rangle\,([\![\sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0]\!]\,|\tau,\tau_0\rangle \otimes \\ &\qquad\qquad \otimes [\![\sigma : \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1]\!]\,|\tau,\tau_1\rangle) \\ &= \langle\psi| \sum_{(\tau',\tau_0',\tau_1')\in\mathbb{V}(\Delta,\Delta_0,\Delta_1)} \sum_{(\tau,\tau_0,\tau_1)\in\mathbb{V}(\Delta,\Delta_0,\Delta_1)} \langle\tau,\tau_0,\tau_1|\psi\rangle \left|\tau',\tau_0',\tau_1'\right\rangle \cdot \\ &\qquad\qquad \cdot \left\langle\tau',\tau_0'\right| [\![\sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0]\!]^\dagger [\![\sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0]\!]\,|\tau,\tau_0\rangle \cdot \\ &\qquad\qquad \cdot \left\langle\tau',\tau_1'\right| [\![\sigma : \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1]\!]^\dagger [\![\sigma : \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1]\!]\,|\tau,\tau_1\rangle = \\ &= \langle\psi| \sum_{(\tau',\tau_0',\tau_1')\in\mathbb{V}(\Delta,\Delta_0,\Delta_1)} \sum_{(\tau,\tau_0,\tau_1)\in\mathbb{V}(\Delta,\Delta_0,\Delta_1)} \langle\tau,\tau_0,\tau_1|\psi\rangle \left\langle\tau',\tau_0'|\tau,\tau_0\right\rangle \left\langle\tau',\tau_1'|\tau,\tau_1\right\rangle \left|\tau',\tau_0',\tau_1'\right\rangle = \\ &= \langle\psi| \sum_{(\tau,\tau_0,\tau_1)\in\mathbb{V}(\Delta,\Delta_0,\Delta_1)} \langle\tau,\tau_0,\tau_1|\psi\rangle = \||\psi\rangle\|^2\,.\end{aligned}$$

I-Ctrl:

$$\begin{aligned}&\left\|[\![\sigma : \Gamma \parallel \Delta, \Delta' \vdash \mathtt{ctrl}\ e \underset{T}{\begin{Bmatrix} e_1 \mapsto e_1' \\ \cdots \\ e_n \mapsto e_n' \end{Bmatrix}}_{T'} : T']\!]\,|\psi\rangle\right\| = \\ &= \left\|\sum_{(\tau,\tau')\in\mathbb{V}(\Delta,\Delta')} \langle\tau,\tau'|\psi\rangle \sum_{v\in\mathbb{V}(T)} \langle v|\,[\![\Gamma \parallel \Delta \Vdash e : T]\!]\,(|\sigma,\tau\rangle\langle\sigma,\tau|)\,|v\rangle \right. \\ &\qquad \left. \cdot \sum_{j=1}^{n} \sum_{\sigma_j\in\mathbb{V}(\Gamma_j)} \left\langle\sigma_j\right| [\![\varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T]\!]^\dagger\,|v\rangle \cdot [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e_j' : T']\!]\,|\tau,\tau'\rangle \right\|\end{aligned}$$

Since e is classical and $\mathrm{tr}\left([\![\Gamma \parallel \Delta \Vdash e : T]\!]\,(|\sigma,\tau\rangle\langle\sigma,\tau|)\right) = 1$ by the isometry assumption for the scrutinee, we must have that

$$[\![\Gamma \parallel \Delta \Vdash e : T]\!]\,(|\sigma,\tau\rangle\langle\sigma,\tau|) = |v\rangle\,\langle v|$$

for some particular $v \in \mathbb{V}(T)$. Then, the entire expression becomes

$$\left\|\sum_{(\tau,\tau')\in\mathbb{V}(\Delta,\Delta')} \langle\tau,\tau'|\psi\rangle \sum_{j=1}^{n} \sum_{\sigma_j\in\mathbb{V}(\Gamma_j)} \left\langle\sigma_j\right| [\![\varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T]\!]^\dagger\,|v\rangle\,[\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e_j' : T']\!]\,|\tau,\tau'\rangle\right\|.$$

Now, since the e_j satisfy the spanning judgment, the direct sum of their images in $\mathcal{H}(T)$ must contain all of $|v'\rangle \in \mathbb{V}(T)$. Furthermore, since we assume the e_j are all classical (from T-Ctrl), each must map each σ_j to some $v' \in \mathbb{V}(T)$, and there must be a one-to-one correspondence between v' and (j, σ_j). Thus, exactly one value of j makes the value of summand nonzero. So, the expression becomes

$$\left\| \sum_{(\tau,\tau') \in \mathbb{V}(\Delta,\Delta')} \langle \tau, \tau' | \psi \rangle \, [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T']\!] \, |\tau, \tau'\rangle \right\| =$$
$$= \left\| [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T']\!] \, |\psi\rangle \right\| = \| |\psi\rangle \| ,$$

which follows from the isometry assumption on the RHS.

I-Match:

$$\operatorname{tr}\left([\![\sigma : \Gamma \parallel \Delta, \Delta_0, \Delta_1 \vdash \texttt{match}\ e \begin{Bmatrix} e_1 \mapsto e'_1 \\ \cdots \\ e_n \mapsto e'_n \end{Bmatrix}_T : T']\!](\rho) \right) =$$
$$= \sum_{(\tau,\tau_0,\tau_1),(\tau',\tau'_0,\tau'_1) \in \mathbb{V}(\Delta,\Delta_0,\Delta_1)} \langle \tau, \tau_0, \tau_1 | \, \rho \, | \tau', \tau'_0, \tau'_1 \rangle \sum_{v \in \mathbb{V}(T)} \langle v | \left([\![\sigma : \Gamma \parallel \Delta, \Delta_0 \Vdash e : T]\!] \left(|\tau, \tau_0\rangle \left\langle \tau', \tau'_0 \right| \right) \right) |v\rangle \cdot$$
$$\cdot \sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![\varnothing : \varnothing \parallel \Gamma_j \vdash e_j : T]\!]^\dagger |v\rangle \cdot \operatorname{tr}\left([\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e'_j : T']\!] \left(|\tau, \tau_1\rangle \left\langle \tau', \tau'_1 \right| \right) \right)$$

Then, by the isometry assumption for the scrutinee, we have that $\operatorname{tr}\left([\![\Gamma \parallel \Delta, \Delta_0 \Vdash e : T]\!] \left(|\tau, \tau_0\rangle\langle\tau', \tau'_0| \right) \right) = \operatorname{tr} |\tau, \tau_0\rangle\langle\tau', \tau'_0| = \delta_{\tau,\tau'} \delta_{\tau_0,\tau'_0}$. So then, by the same argument as for T-Ctrl, we can eliminate the inner sums and obtaining

$$\sum_{(\tau,\tau_0,\tau_1) \in \mathbb{V}(\Delta,\Delta_0,\Delta_1), \tau'_1 \in \mathbb{V}(\Delta_1)} \langle \tau, \tau_0, \tau_1 | \, \rho \, | \tau, \tau_0, \tau'_1 \rangle \operatorname{tr}\left([\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta_1 \Vdash e'_j : T']\!] \left(|\tau, \tau_1\rangle \left\langle \tau, \tau'_1 \right| \right) \right) =$$
$$= \sum_{(\tau,\tau_0,\tau_1) \in \mathbb{V}(\Delta,\Delta_0,\Delta_1), \tau'_1 \in \mathbb{V}(\Delta_1)} \langle \tau, \tau_0, \tau_1 | \, \rho \, | \tau, \tau_0, \tau'_1 \rangle \operatorname{tr}\left(|\tau, \tau_1\rangle \left\langle \tau, \tau'_1 \right| \right) =$$
$$= \sum_{(\tau,\tau_0,\tau_1) \in \mathbb{V}(\Delta,\Delta_0,\Delta_1), \tau'_1 \in \mathbb{V}(\Delta_1)} \langle \tau, \tau_0, \tau_1 | \, \rho \, | \tau, \tau_0, \tau'_1 \rangle \operatorname{tr}\left(|\tau, \tau_1\rangle \left\langle \tau, \tau'_1 \right| \right) =$$
$$= \sum_{(\tau,\tau_0,\tau_1) \in \mathbb{V}(\Delta,\Delta_0,\Delta_1)} \langle \tau, \tau_0, \tau_1 | \, \rho \, | \tau, \tau_0, \tau_1 \rangle = \operatorname{tr}(\rho).$$

I-Try: Assuming $\operatorname{iso}(e_0)$,

$$\operatorname{tr}\left([\![\sigma : \Gamma \parallel \Delta_0, \Delta_1 \Vdash \texttt{try}\ e_0\ \texttt{catch}\ e_1 : T]\!](\rho) \right) =$$
$$= \sum_{(\tau_0,\tau_1),(\tau'_0,\tau'_1) \in \mathbb{V}(\Delta_0,\Delta_1)} \langle \tau_0, \tau_1 | \, \rho \, | \tau'_0, \tau'_1 \rangle \left[\delta_{\tau_1,\tau'_1} \operatorname{tr}\left([\![\sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T]\!] \left(|\tau_0\rangle\langle\tau'_0| \right) \right) + \right.$$
$$\left. + \delta_{\tau_0,\tau'_0} (1 - \operatorname{tr}([\![\sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T]\!] \left(|\tau_0\rangle\langle\tau'_0| \right))) \cdot \operatorname{tr}\left([\![\sigma : \Gamma \parallel \Delta_1 \Vdash e_1 : T]\!] \left(|\tau_1\rangle\langle\tau'_1| \right) \right) \right] =$$
$$= \sum_{\tau_0 \in \mathbb{V}(\Delta_0), \tau_1 \in \mathbb{V}(\Delta_1)} \langle \tau_0, \tau_1 | \, \rho \, | \tau_0, \tau_1 \rangle = \operatorname{tr}(\rho).$$

I-Catch: Assuming $\mathrm{iso}(e_1)$,

$$\begin{aligned}
&\mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta_0, \Delta_1 \Vdash \texttt{try}\ e_0\ \texttt{catch}\ e_1 : T \rrbracket(\rho)\right) = \\
&= \sum_{(\tau_0,\tau_1),(\tau_0',\tau_1')\in \mathbb{V}(\Delta_0,\Delta_1)} \langle \tau_0,\tau_1|\,\rho\,|\tau_0',\tau_1'\rangle \left[\delta_{\tau_1,\tau_1'}\,\mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0'|\right)\right) + \right. \\
&\quad \left. +\delta_{\tau_0,\tau_0'}(1-\mathrm{tr}(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0'|\right)))\cdot \mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta_1 \Vdash e_1 : T \rrbracket\left(|\tau_1\rangle\langle\tau_1'|\right)\right)\right] = \\
&= \sum_{(\tau_0,\tau_1),(\tau_0',\tau_1')\in \mathbb{V}(\Delta_0,\Delta_1)} \langle \tau_0,\tau_1|\,\rho\,|\tau_0',\tau_1'\rangle \left[\delta_{\tau_1,\tau_1'}\,\mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0'|\right)\right) + \right. \\
&\quad \left. +\delta_{\tau_0,\tau_0'}(1-\mathrm{tr}(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0|\right)))\delta_{\tau_1,\tau_1'}\right] = \\
&= \sum_{\tau_0\in\mathbb{V}(\Delta_0),\tau_1\in\mathbb{V}(\Delta_1)} \langle \tau_0,\tau_1|\,\rho\,|\tau_0,\tau_1\rangle \left[\mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0|\right)\right) + \right. \\
&\quad \left. +(1-\mathrm{tr}(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0|\right)))\right] = \\
&= \sum_{\tau_0\in\mathbb{V}(\Delta_0),\tau_1\in\mathbb{V}(\Delta_1)} \langle \tau_0,\tau_1|\,\rho\,|\tau_0,\tau_1\rangle = \mathrm{tr}(\rho).
\end{aligned}$$

Note that in the above, we used the trace non-increasing property to say that

$$\mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T \rrbracket\left(|\tau_0\rangle\langle\tau_0'|\right)\right) \le \mathrm{tr}\left(|\tau_0\rangle\langle\tau_0'|\right) = \delta_{\tau_0,\tau_0'}$$

so it is zero when $\tau_0 \neq \tau_0'$: this can be seen as a consequence of the correctness of the compilation procedure (Appendix L.3).

I-App: Here, we must consider both pure and mixed typing. If f can be typed as a pure program, we have that

$$\begin{aligned}
&\left\|\llbracket \sigma : \Gamma \parallel \Delta \vdash f e : T' \rrbracket\,|\psi\rangle\right\| = \left\|\llbracket \vdash f : T \rightsquigarrow T' \rrbracket\llbracket \sigma : \Gamma \parallel \Delta \vdash e : T \rrbracket\,|\psi\rangle\right\| = \\
&= \left\|\llbracket \sigma : \Gamma \parallel \Delta \vdash e : T \rrbracket\,|\psi\rangle\right\| = \||\psi\rangle\|\,.
\end{aligned}$$

Similarly, for mixed typing, we have that

$$\begin{aligned}
&\mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta \Vdash f e : T' \rrbracket(\rho)\right) = \\
&= \mathrm{tr}\left(\llbracket \vdash f : T \Rrightarrow T' \rrbracket\left(\llbracket \sigma : \Gamma \parallel \Delta \vdash e : T \rrbracket(\rho)\right)\right) = \\
&= \mathrm{tr}\left(\llbracket \sigma : \Gamma \parallel \Delta \vdash e : T \rrbracket(\rho)\right) = \mathrm{tr}(\rho).
\end{aligned}$$

I-Gate: This is clear since single-qubit unitary gates are isometries.

I-Left, I-Right: The direct sum injections map between $\mathcal{H}(T_0)$ (resp. $\mathcal{H}(T_1)$) and the corresponding isomorphic subspace in $\mathcal{H}(T_0 \oplus T_1)$, and hence they are clearly isometries.

I-Abs: Here, we again consider both pure and mixed typing for e'. If e' can be typed as a pure expression, then

$$\begin{aligned}
&\left\|\llbracket \vdash \lambda e \overset{T}{\mapsto} e' : T \rightsquigarrow T' \rrbracket\,|\psi\rangle\right\| = \\
&= \left\|\llbracket \varnothing : \varnothing \parallel \Delta \vdash e' : T' \rrbracket\llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger\,|\psi\rangle\right\| = \\
&= \left\|\llbracket \varnothing : \varnothing \parallel \Delta \vdash e : T \rrbracket^\dagger\,|\psi\rangle\right\|
\end{aligned}$$

Now, consider the circumstances under which a single expression e can satisfy the spanning judgment. It is only possible for S-Sum to have been applied if one of the two types summed is `Void`. So, in all cases, the dimension of $\mathcal{H}(T)$ must be equal to the dimension of $\mathcal{H}(\Delta)$ (for the case of S-Pair this is enforced by the "no shared free variables" restriction). Since e is formed from just variables, pairs, and applications of isometric programs, it has isometric, and thus unitary, semantics. This means that the norm of the expression above must be equal to $\||\psi\rangle\|$.

Now, for mixed typing:

$$\begin{aligned}
&\operatorname{tr}\left(\left[\!\left[\vdash \lambda e \overset{T}{\mapsto} e' : T \Rrightarrow T'\right]\!\right](\rho)\right) = \\
&= \sum_{v,v' \in \mathbb{V}(T)} \langle v| \rho |v'\rangle \operatorname{tr}\left(\left[\!\left[\varnothing : \varnothing \parallel \Delta \Vdash e' : T'\right]\!\right] \left(\left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]^\dagger |v\rangle\langle v'| \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]\right)\right) = \\
&= \sum_{v,v' \in \mathbb{V}(T)} \langle v| \rho |v'\rangle \operatorname{tr}\left(\left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]^\dagger |v\rangle\langle v'| \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]\right) = \\
&= \sum_{v,v' \in \mathbb{V}(T)} \langle v| \rho |v'\rangle \operatorname{tr}\left(\langle v'| \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right] \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]^\dagger |v\rangle\right).
\end{aligned}$$

Now, by the same argument as above, $[\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!]$ must be unitary, so the expression simplifies to $\operatorname{tr}(\rho)$.

I-Rphase:

$$\begin{aligned}
&\left[\!\left[\vdash \texttt{rphase}_T \begin{Bmatrix} e \mapsto r \\ \texttt{else} \mapsto r' \end{Bmatrix} : T \rightsquigarrow T\right]\!\right] |\psi\rangle = \\
&= e^{ir} \left[\!\left[\varnothing \parallel \Delta \vdash e : T\right]\!\right] \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]^\dagger |\psi\rangle + e^{ir'} \left(\mathbb{I} - \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right] \left[\!\left[\varnothing : \varnothing \parallel \Delta \vdash e : T\right]\!\right]^\dagger\right) |\psi\rangle
\end{aligned}$$

Letting $P = [\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!] [\![\varnothing : \varnothing \parallel \Delta \vdash e : T]\!]^\dagger$, and observe that by the isometry assumption, $P^2 = P$ and $P^\dagger = P$. So, $P(\mathbb{I} - P) = P - P^2 = 0$ and $(\mathbb{I} - P)^2 = \mathbb{I} - 2P + P^2 = \mathbb{I} - P$. Then,

$$\begin{aligned}
&\left\| \left[\!\left[\vdash \texttt{rphase}_T \begin{Bmatrix} e \mapsto r \\ \texttt{else} \mapsto r' \end{Bmatrix} : T \rightsquigarrow T\right]\!\right] |\psi\rangle \right\|^2 = \\
&= \left\| e^{ir} P |\psi\rangle + e^{ir'} (\mathbb{I} - P) |\psi\rangle \right\| = \\
&= \left(e^{-ir} \langle\psi| P + e^{-ir'} \langle\psi| (\mathbb{I} - P)\right) \left(e^{ir} P |\psi\rangle + e^{ir'} (\mathbb{I} - P) |\psi\rangle\right) = \\
&= \langle\psi| P |\psi\rangle + \langle\psi| (\mathbb{I} - P) |\psi\rangle = \||\psi\rangle\|^2 .
\end{aligned}$$

I-Pmatch:

$$\begin{aligned}
&\left\| \left[\!\left[\vdash \texttt{pmatch} \quad \underset{T}{\begin{Bmatrix} e_1 \mapsto e_1' \\ \cdots \\ e_n \mapsto e_n' \end{Bmatrix}}_{T'} : T \rightsquigarrow T'\right]\!\right] |\psi\rangle \right\|^2 = \\
&= \left\| \sum_{j=1}^n \left[\!\left[\varnothing : \varnothing \parallel \Delta_j \vdash e_j' : T'\right]\!\right] \left[\!\left[\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T\right]\!\right]^\dagger |\psi\rangle \right\|^2 = \\
&= \sum_{j=1}^n \sum_{k=1}^n \langle\psi| \left[\!\left[\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T\right]\!\right] \left[\!\left[\varnothing : \varnothing \parallel \Delta_j \vdash e_j' : T'\right]\!\right]^\dagger \left[\!\left[\varnothing : \varnothing \parallel \Delta_k \vdash e_k' : T'\right]\!\right] \left[\!\left[\varnothing : \varnothing \parallel \Delta_k \vdash e_k : T\right]\!\right]^\dagger |\psi\rangle
\end{aligned}$$

By the orthogonality assumption for the RHS in T-Pmatch, we must have $[\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j' : T']\!]^\dagger [\![\varnothing : \varnothing \parallel \Delta_j \vdash e_k' : T']\!] = 0$ unless $j = k$, in which case it must be the identity by the isometry

assumption for the RHS, so the expression becomes

$$\sum_{j=1}^{n} \langle\psi| [\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T]\!][\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T]\!]^\dagger |\psi\rangle =$$

$$= \sum_{v in \mathbb{V}(T)} |\langle v|\psi\rangle| \sum_{j=1}^{n} \langle v| [\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T]\!][\![\varnothing : \varnothing \parallel \Delta_j \vdash e_j : T]\!]^\dagger |v\rangle = \||\psi\rangle\|^2,$$

which follows by the spanning assumption on the LHS expressions. This completes the analysis of all the cases for the isometry judgment.

□

H The Erasure Judgment

The erasure judgment (Figure 25) is used for typing Qunity's `ctrl` construct, ensuring that it is possible to correctly perform the necessary uncomputation. This requires that all quantum variables in the scrutinee must be present "in the same way" in all the RHS expressions. If the purpose of the control is to apply a controlled phase, this is easily satisfied. If it needs to output some additional data, it needs to be paired with the original variables, possibly inside a nested `ctrl`. This condition is necessary to ensure that `ctrl` does not discard any quantum information and can be typed as a pure expression.

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

$$\frac{\mathrm{erases}_T(x; e_1, \ldots, e_{j-1}, e_{j,1}, \ldots, e_{j,m}, e_{j+1}, \ldots, e_n)}{\mathrm{erases}_T\left(x; e_1, \ldots, e_{j-1}, \mathtt{ctrl}\ e\ \underset{T'}{\left\{\begin{array}{c} e'_1 \mapsto e_{j,1} \\ \cdots \\ e'_m \mapsto e_{j,m} \end{array}\right\}_T}, e_{j+1}, \ldots, e_n\right)}\ \text{E-Ctrl}$$

$$\frac{}{\mathrm{erases}_T(x; x, x, \ldots, x)}\ \text{E-Var} \qquad \frac{\mathrm{erases}_{T_0}(x; e_{0,1}, \ldots, e_{0,n})}{\mathrm{erases}_{T_0 \otimes T_1}(x; (e_{0,1}, e_{1,1}), \ldots, (e_{0,n}, e_{1,n}))}\ \text{E-Pair0}$$

$$\frac{\mathrm{erases}_{T_1}(x; e_{1,1}, \ldots, e_{1,n})}{\mathrm{erases}_{T_0 \otimes T_1}(x; (e_{0,1}, e_{1,1}), \ldots, (e_{0,n}, e_{1,n}))}\ \text{E-Pair1}$$

Fig. 25. Erasure inference rules.

I Direct Sum Circuit Correctness Proof

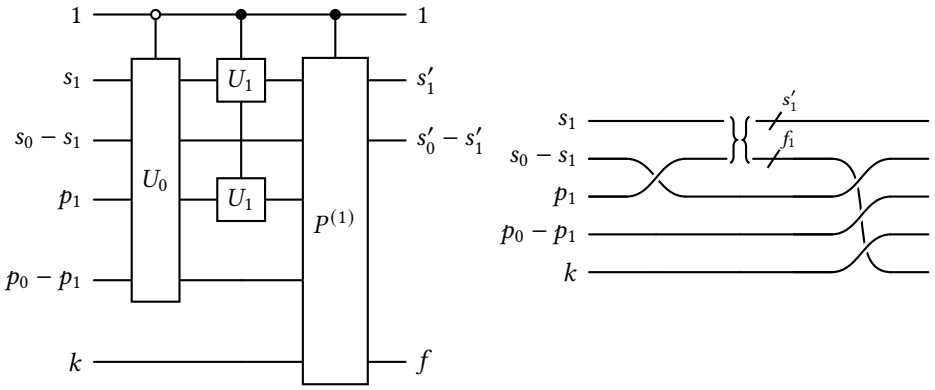


Fig. 26. Improved direct sum circuit for Case 1: $s_0 \geq s_1, s'_0 \geq s'_1, p_0 \geq p_1$. For this circuit, let $k = \max\{0, f_1 - f_0\}$, and let $f = \max\{f_0, f_1\} = f_0 + k$. The circuit on the left implements the direct sum, while the one on the right is the implementation of the component $P^{(1)}$. Note that where U_1 is drawn as two separate boxes, these are not two separate gates, but rather a single gate that only acts on the s_1 and p_1 registers and not on the one in between them. The curly braces in the circuit on the right indicate a repartitioning of a set of registers taken together into registers of different sizes, preserving the order of qubits.

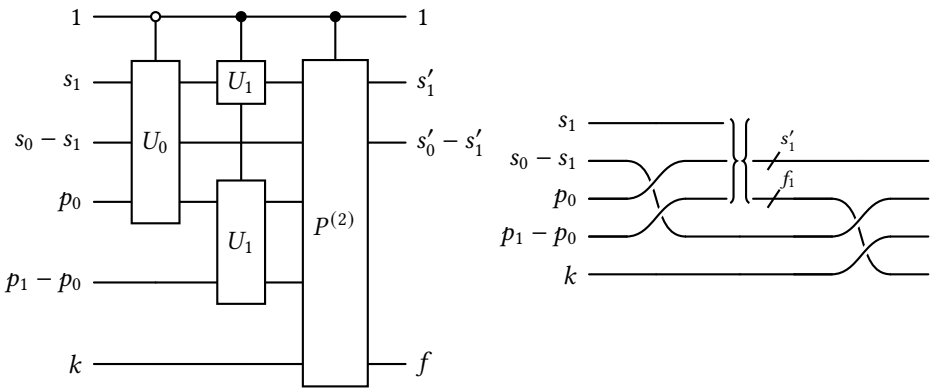


Fig. 27. Improved direct sum circuit for Case 2: $s_0 \geq s_1, s'_0 \geq s'_1, p_0 \leq p_1$. For this circuit, let $k = \max\{0, f_1 - f_0 + p_0 - p_1\}$, and let $f = \max\{f_0 + p_1 - p_0, f_1\}$. The circuit on the left implements the direct sum, while the one on the right is the implementation of the component $P^{(2)}$.

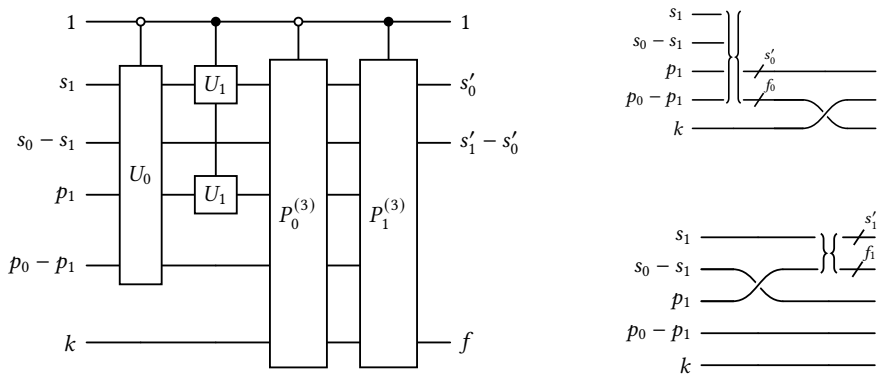


Fig. 28. Improved direct sum circuit for Case 3: $s_0 \geq s_1, s'_0 \leq s'_1, p_0 \geq p_1$. For this circuit, let $k = s'_1 - s'_0$, and let $f = f_0$. The circuit on the left implements the direct sum, while the ones on the right are the implementations of the components $P_0^{(3)}$ and $P_1^{(3)}$.

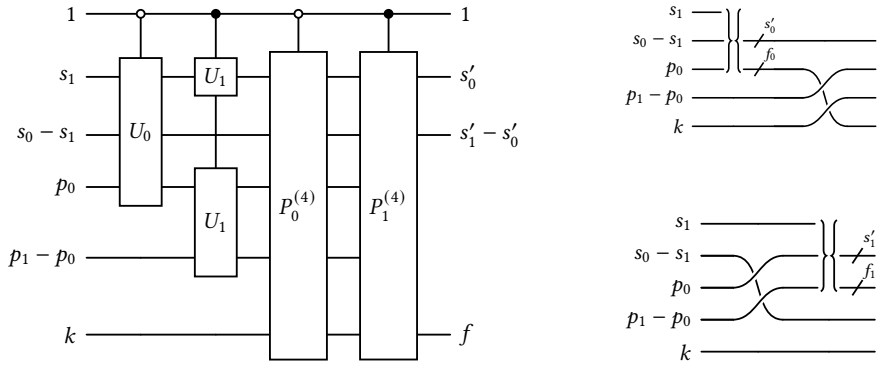


Fig. 29. Improved direct sum circuit for Case 4: $s_0 \geq s_1, s'_0 \leq s'_1, p_0 \leq p_1$. For this circuit, let $k = \max\{0, s'_1 - s'_0 + p_0 - p_1\}$, and let $f = s_0 + p_1 - s'_1 + k$. The circuit on the left implements the direct sum, while the ones on the right are the implementations of the components $P_0^{(4)}$ and $P_1^{(4)}$.

Lemma I.1. *Suppose it is possible to implement (as specified in Definition 5.2) the operators $E_0 : \mathcal{H}(T_0) \to \mathcal{H}(T_0')$ and $E_1 : \mathcal{H}(T_1) \to \mathcal{H}(T_1')$. Then it is possible to implement the operator $E_0 \oplus E_1 : \mathcal{H}(T_0 \oplus T_1) \to \mathcal{H}(T_0' \oplus T_1')$, using the circuits shown in Figures 26, 27, 28, and 29.*

Proof. Let $s_0 = \text{size}(T_0)$, $s_0' = \text{size}(T_0')$, $s_1 = \text{size}(T_1)$, $s_1' = \text{size}(T_1')$. Suppose that E_0 is implemented by a unitary U_0 using p_0 prep qubits and p_1 flag qubits, and E_1 is implemented by a unitary U_1 using p_1 prep qubits and f_1 flag qubits. We have that

$$s_0 + p_0 = s_0' + f_0$$
$$s_1 + p_1 = s_1' + f_1$$

We introduce the sets of computational basis states for the qubits in the corresponding registers: we label these as $|\xi_0\rangle, |\pi_0\rangle, |\xi_0'\rangle, |\phi_0\rangle$ for the registers of size s_0, p_0, s_0', f_0 respectively, and $|\xi_1\rangle, |\pi_1\rangle, |\xi_1'\rangle, |\phi_1\rangle$ for s_1, p_1, s_1', f_1.

Now, without loss of generality, we can assume that $s_0 \geq s_1$, because if this were not the case, then we can implement $E_1 \oplus E_0$ and apply the direct sum's commutativity isomorphism by simply surrounding the circuit with Pauli X gates on the first "signal" qubit. Now, we have four cases to analyze.

Case 1: Suppose that $s_0' \geq s_1'$ and $p_0 \geq p_1$. For this case, let $k = \max\{0, f_1 - f_0\}$. Let $f = \max\{f_0, f_1\} = f_0 + k$. Then, consider the circuit U shown in Figure 26. First, to verify that the wire counts are correct:

$$1+s_0+p_0+k = 1+s_0+p_0+\max\{0, f_1-f_0\} = 1+s_0'+f_0+\max\{0, f_1-f_0\} = 1+s_0'+\max\{f_0, f_1\} = 1+s_0'+f$$

Now, to verify the correctness of the circuit:

$$\begin{aligned}
&\langle 0, \text{enc}(v_0'), 0| U |0, \text{enc}(v_0), 0, 0\rangle = \langle \text{enc}(v_0'), 0| U_0 |\text{enc}(v_0), 0\rangle = \\
&= \langle v_0'| E_0 |v_0\rangle = (\langle v_0'| \oplus 0)(E_0 \oplus E_1)(|v_0\rangle \oplus 0)
\end{aligned}$$

$$\langle 1, \text{enc}(v_1'), 0, 0| U |0, \text{enc}(v_0), 0, 0\rangle = 0 = (0 \oplus \langle v_1'|)(E_0 \oplus E_1)(|v_0\rangle \oplus 0)$$

$$\langle 0, \text{enc}(v_0'), 0| U |1, \text{enc}(v_1), 0, 0, 0, 0\rangle = 0 = (\langle v_0'| \oplus 0)(E_0 \oplus E_1)(0 \oplus |v_1\rangle)$$

$$\begin{aligned}
&\langle 1, \text{enc}(v_1'), 0, 0| U |1, \text{enc}(v_1), 0, 0, 0, 0\rangle = \\
&= \langle \text{enc}(v_1'), 0, 0| P^{(1)} \sum_{\xi_1, \pi_1} |\xi_1, 0, \pi_1, 0, 0\rangle \langle \xi_1, \pi_1| U_1 |\text{enc}(v_1), 0\rangle = \\
&= \langle \text{enc}(v_1'), 0, 0| \sum_{\xi_1, \pi_1} \sum_{\xi_1', \phi_1} |\xi_1', 0, 0, 0, \phi_1\rangle \langle \xi_1', \phi_1|\xi_1, \pi_1\rangle \langle \xi_1, \pi_1| U_1 |\text{enc}(v_1), 0\rangle = \\
&= \langle \text{enc}(v_1'), 0, 0| \sum_{\xi_1'} |\xi_1', 0, 0, 0, 0\rangle \langle \xi_1', 0| U_1 |\text{enc}(v_1), 0\rangle = \\
&= \langle \text{enc}(v_1'), 0| U_1 |\text{enc}(v_1), 0\rangle = \\
&= \langle \text{enc}(v_1'), 0| U_1 |\text{enc}(v_1), 0\rangle = \langle v_1'| E_1 |v_1\rangle = (0 \oplus \langle v_1'|)(E_0 \oplus E_1)(0 \oplus |v_1\rangle)
\end{aligned}$$

And now, to verify the preservation of encoding validity:

If a valid encoding of T_0 is given as input, the output will always be a valid encoding of T_0' since $s_0' \geq s_1'$. If a valid encoding of T_1 is given as input, observe that the register of size $s_0 - s_1$ must be in the $|0\rangle$ state. It is not affected by the application of U_1. Then, the output register of size $s_0' - s_1'$ can

only be formed from qubits in the input registers of sizes $s_0 - s_1$, $p_0 - p_1$, and k, and the latter two must also be $|0\rangle$ since they are part of the prep register. Thus U preserves encoding validity.

Now, for the adjoint $U^\dagger$, we consider the version of this circuit run in reverse, where the flag registers are now prep registers and vice versa. If a valid encoding of T_0' is given as input, the output will always be a valid encoding of T_0 since $s_0 \geq s_1$. If a valid encoding of T_1 is given as input, the register of size $s_0' - s_1'$ must be in the $|0\rangle$ state. The output register of size $s_0 - s_1$ can only be formed from qubits in that register and from the input register of size f, which is a prep register. Thus, $U^\dagger$ also preserves encoding validity.

Case 2: Suppose that $s_0' \geq s_1'$ and $p_0 \leq p_1$. For this case, let $k = \max\{0, f_1 - f_0 + p_0 - p_1\}$, and let $f = \max\{f_0 + p_1 - p_0, f_1\}$. Then, consider the circuit U shown in Figure 27. First, to verify that the wire counts are correct:

$$1 + s_0 + p_1 + k = 1 + s_0 + p_1 + \max\{0, f_1 - f_0 + p_0 - p_1\} =$$
$$= 1 + s_0' + f_0 - p_0 + p_1 + \max\{0, f_1 - f_0 + p_0 - p_1\} = 1 + s_0' + \max\{f_0 + p_1 - p_0, f_1\} = 1 + s_0' + f$$

Now, to verify the correctness of the circuit (we will now omit the obvious cases where the state of the first qubit does not match):

$$\begin{aligned}
&\left\langle 0, \mathrm{enc}(v_0'), 0\right| U \left|0, \mathrm{enc}(v_0), 0, 0, 0\right\rangle = \left\langle \mathrm{enc}(v_0'), 0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle v_0'\right| E_0 \left|v_0\right\rangle = (\left\langle v_0'\right| \oplus 0)(E_0 \oplus E_1)(\left|v_0\right\rangle \oplus 0) \\
&\left\langle 1, \mathrm{enc}(v_1'), 0, 0\right| U \left|1, \mathrm{enc}(v_1), 0, 0, 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0, 0\right| P^{(2)} \sum_{\xi_1, \pi_1} \left|\xi_1, 0, \pi_1, 0\right\rangle \left\langle \xi_1, 0, \pi_1, 0\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0, 0\right| \sum_{\xi_1, \pi_1} \sum_{\xi_1', \phi_1} \left|\xi_1', 0, 0, \phi_1\right\rangle \left\langle \xi_1, \phi_1 | \xi_1, \pi_1\right\rangle \left\langle \xi_1, 0, \pi_1, 0\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0, 0\right| \sum_{\xi_1'} \left|\xi_1', 0, 0, 0\right\rangle \left\langle \xi_1', 0\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \left\langle v_1'\right| E_1 \left|v_1\right\rangle = (0 \oplus \left\langle v_1'\right|)(E_0 \oplus E_1)(0 \oplus \left|v_1\right\rangle)
\end{aligned}$$

And now, to verify the preservation of encoding validity:

The encoding validity for T_0 and T_0' is clear as in the previous case. If a valid encoding of T_1 is given as input, the register of size $s_0' - s_1'$ is again formed only from the register of size $s_0 - s_1$ and prep registers (not the flag register since $f \geq f_1$). Similarly, for the adjoint circuit, the register of size $s_0 - s_1$ is only formed from the registers of size $s_0' - s_1'$ and f. So, the encoding validity is preserved.

Case 3: Suppose that $s_0' \leq s_1'$ and $p_0 \geq p_1$. For this case, let $k = s_1' - s_0'$, and let $f = f_0$. Then, consider the circuit U shown in Figure 28. First, to verify that the wire counts are correct:

$$1 + s_0 + p_0 + k = 1 + s_0' + f_0 + s_1' - s_0' = 1 + s_1' + f$$

Now, to verify the correctness of the circuit:

$$
\begin{aligned}
&\left\langle 0, \mathrm{enc}(v_0'), 0, 0\right| U \left|0, \mathrm{enc}(v_0), 0, 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0, 0\right| P_0^{(3)} \sum_{\xi_0,\pi_0} \left|\xi_0, \pi_0, 0\right\rangle \left\langle \xi_0, \pi_0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0, 0\right| \sum_{\xi_0,\pi_0} \sum_{\xi_0',\phi_0} \left|\xi_0', 0, \phi_0\right\rangle \left\langle \xi_0', \phi_0 \middle| \xi_0, \pi_0\right\rangle \left\langle \xi_0, \pi_0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0, 0\right| \sum_{\xi_0'} \left|\xi_0', 0, 0\right\rangle \left\langle \xi_0', 0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \left\langle v_0'\right| E_0 \left|v_0\right\rangle = (\left\langle v_0'\right| \oplus 0)(E_0 \oplus E_1)(\left|v_0\right\rangle \oplus 0) \\
&\left\langle 1, \mathrm{enc}(v_1'), 0\right| U \left|1, \mathrm{enc}(v_1), 0, 0, 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| P_1^{(3)} \sum_{\xi_1,\pi_1} \left|\xi_1, 0, \pi_1, 0, 0\right\rangle \left\langle \xi_1, \pi_1\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| \sum_{\xi_1,\pi_1} \sum_{\xi_1',\phi_1} \left|\xi_1', \phi_1, 0, 0, 0\right\rangle \left\langle \xi_1', \phi_1 \middle| \xi_1, \pi_1\right\rangle \left\langle \xi_1, \pi_1\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| \sum_{\xi_1'} \left|\xi_1', 0, 0, 0, 0\right\rangle \left\langle \xi_1', \phi_1\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \left\langle v_1'\right| E_1 \left|v_1\right\rangle = (0 \oplus \left\langle v_1'\right|)(E_0 \oplus E_1)(0 \oplus \left|v_1\right\rangle)
\end{aligned}
$$

The encoding validity preserving property follows by a similar argument as before. If a valid encoding of T_0 is given as input, the register of size $s_1' - s_0'$ exactly corresponds to the prep register of size k, which must be in the $|0\rangle$ state. If a valid encoding of T_1 is given as input, the encoding validity of the output is clear. For the adjoint circuit, if T_0' is given, the encoding validity is clear since $s_0 \geq s_1$. If T_1' is given the output register of size $s_0 - s_1$ is only formed from the prep registers. So, the encoding validity is preserved.

Case 4: Suppose that $s_0' \leq s_1'$ and $p_0 \leq p_1$. For this case, let $k = \max\{0, s_1' - s_0' + p_0 - p_1\}$, and let $f = s_0 + p_1 - s_1' + k$. Then, consider the circuit U shown in Figure 29. First, to verify that the wire counts are correct:

$$1 + s_0 + p_1 + k = 1 + s_0 + p_1 + f - s_0 - p_1 + s_1' = 1 + s_1' + f$$

Now, to verify the correctness of the circuit:

$$
\begin{aligned}
&\left\langle 0, \mathrm{enc}(v_0'), 0, 0\right| U \left|0, \mathrm{enc}(v_0), 0, 0, 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0, 0\right| P_0^{(4)} \sum_{\xi_0,\pi_0} \left|\xi_0, \pi_0, 0, 0\right\rangle \left\langle \xi_0, \pi_0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0, 0\right| \sum_{\xi_0,\pi_0} \sum_{\xi_0',\phi_0} \left|\xi_0', 0, 0, \phi_0\right\rangle \left\langle \xi_0', \phi_0\right|\left.\xi_0, \pi_0\right\rangle \left\langle \xi_0, \pi_0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0, 0\right| \sum_{\xi_0'} \left|\xi_0', 0, 0, 0\right\rangle \left\langle \xi_0', 0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_0'), 0\right| U_0 \left|\mathrm{enc}(v_0), 0\right\rangle = \left\langle v_0'\right| E_0 \left|v_0\right\rangle = (\left\langle v_0'\right| \oplus 0)(E_0 \oplus E_1)(\left|v_0\right\rangle \oplus 0) \\
&\left\langle 1, \mathrm{enc}(v_1'), 0\right| U \left|1, \mathrm{enc}(v_1), 0, 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| P_1^{(4)} \sum_{\xi_1,\pi_1} \left|\xi_1, 0, \pi_1, 0\right\rangle \left\langle \xi_1, \pi_1\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| \sum_{\xi_1,\pi_1} \sum_{\xi_1',\phi_1} \left|\xi_1', \phi_1, 0, 0\right\rangle \left\langle \xi_1', \phi_1\right|\left.\xi_1, \pi_1\right\rangle \left\langle \xi_1, \pi_1\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| \sum_{\xi_1'} \left|\xi_1', 0, 0, 0\right\rangle \left\langle \xi_1', \phi_1\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \\
&= \left\langle \mathrm{enc}(v_1'), 0\right| U_1 \left|\mathrm{enc}(v_1), 0\right\rangle = \left\langle v_1'\right| E_1 \left|v_1\right\rangle = (0 \oplus \left\langle v_1'\right|)(E_0 \oplus E_1)(0 \oplus \left|v_1\right\rangle)
\end{aligned}
$$

For the encoding validity: if a valid encoding of T_0 is given as input, the register of size $s_1' - s_0'$ is only formed from the prep registers of size $p_1 - p_0$ and k: we can confirm that

$$
\begin{aligned}
f &= s_0 + p_1 - s_1' + \max\{0, s_1' - s_0' + p_0 - p_1\} = s_0 + \max\{p_1 - s_1', -s_0' + p_0\} = \\
&= s_0' + f_0 - p_0 + \max\{p_1 - s_1', -s_0' + p_0\} = f_0 + \max\{p_1 - s_1' - p_0 + s_0', 0\} \geq f_0,
\end{aligned}
$$

so the flag register does not overlap with the $s_1' - s_0'$ register. If a valid encoding of T_1 is given as input, the encoding validity of the output is clear. For the adjoint: if a valid encoding of T_0' is given as input, the encoding validity is clear. If a valid encoding of T_1' is given as input, the register of size $s_0 - s_1$ is only formed from the prep registers, so the encoding validity is preserved.

This completes the analysis of all four cases, and thus completes the proof for the direct sum circuit construction. □

J Notation and Definitions for Binary Trees

First, we can define a binary tree $\mathcal{R}$ as either Leaf or $(\mathcal{R}_0, \mathcal{R}_1)$, where $\mathcal{R}_0$ and $\mathcal{R}_1$ are binary trees, the left and right children of $\mathcal{R}$.

We can define size and height functions as:

Definition J.1.

$$
\begin{aligned}
\mathrm{size}(\mathrm{Leaf}) &:= 1 \\
\mathrm{size}((\mathcal{R}_0, \mathcal{R}_1)) &:= \mathrm{size}(\mathcal{R}_0) + \mathrm{size}(\mathcal{R}_1) \\
\mathrm{height}(\mathrm{Leaf}) &:= 0 \\
\mathrm{height}((\mathcal{R}_0, \mathcal{R}_1)) &:= 1 + \max(\mathrm{height}(\mathcal{R}_0), \mathrm{height}(\mathcal{R}_1))
\end{aligned}
$$

Then, we can define the following:

Definition J.2 (Direct sum of operators (or states or spaces) along a binary tree).

$$\bigoplus_{j:\text{Leaf}} E_j := E_1$$

$$\bigoplus_{j:(\mathcal{R}_0,\mathcal{R}_1)} E_j := \left(\bigoplus_{j:\mathcal{R}_0} E_j\right) \oplus \left(\bigoplus_{j:\mathcal{R}_1} E_{j+\text{size}(\mathcal{R}_0)}\right)$$

As a shorthand, we can write

$$E^{\oplus\mathcal{R}} = \bigoplus_{j:\mathcal{R}} E.$$

Definition J.3 (Direct sum of operators (or states or spaces) along a binary tree, leveled to height h).

$$\bigoplus_{j:L(\text{Leaf},0)} E_j := E_1$$

$$\bigoplus_{j:L(\text{Leaf},h)} E_j := \left(\bigoplus_{j:L(\text{Leaf},h-1)} E_j\right) \oplus 0 \qquad (\text{where } h \in \mathbb{N},\, h > 0)$$

$$\bigoplus_{j:L((\mathcal{R}_0,\mathcal{R}_1),h)} E_j := \left(\bigoplus_{j:L(\mathcal{R}_0,h-1)} E_j\right) \oplus \left(\bigoplus_{j:L(\mathcal{R}_1,h-1)} E_{\text{size}(\mathcal{R}_0)}\right)$$

Note that the 0 in the above definition exists in the space $\{0\}$: thus, leveling operation is effectively an extension of the additive unit isomorphism. As a shorthand, we can write

$$\bigoplus_{j:L(\mathcal{R})} E_j := \bigoplus_{j:L(\mathcal{R},\text{height}(\mathcal{R}))} E_j,$$

and

$$E^{\oplus L(\mathcal{R})} = \bigoplus_{j:L(\mathcal{R})} E.$$

Summing along a leveled tree, as in Definition J.3, can be thought of as adding left children to all nodes in the tree, until they reach a specified height (which must be at least the height of the tree). This construction will be useful when constructing the new spanning circuit.

Definition J.4 (Indexed injection into a tree).

$$\text{inj}_1^{\text{Leaf}} |v\rangle = |v\rangle$$

$$\text{inj}_k^{(\mathcal{R}_0,\mathcal{R}_1)} |v\rangle = \begin{cases} \text{inj}_k^{\mathcal{R}_0} |v\rangle \oplus 0^{\oplus\mathcal{R}_1} & \text{if } 1 \le k \le \text{size}(\mathcal{R}_1) \\ 0^{\oplus\mathcal{R}_1} \oplus \text{inj}_{k-\text{size}(\mathcal{R}_0)}^{\mathcal{R}_1} |v\rangle & \text{otherwise.} \end{cases}$$

Definition J.5 (Indexed injection into a leveled tree).

$$\text{inj}_1^{L(\text{Leaf},0)} |v\rangle = |v\rangle$$

$$\text{inj}_1^{L(\text{Leaf},h)} |v\rangle = \text{inj}_1^{L(\text{Leaf},h-1)} |v\rangle \oplus 0 \qquad (\text{where } h \in \mathbb{N},\, h > 0)$$

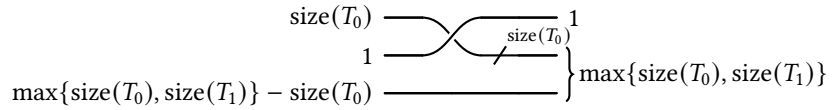
$$\text{inj}_k^{L((\mathcal{R}_0,\mathcal{R}_1),h)} |v\rangle = \begin{cases} \text{inj}_k^{L(\mathcal{R}_0,h-1)} |v\rangle \oplus 0^{\oplus L(\mathcal{R}_1,h-1)} & \text{if } 1 \le k \le \text{size}(\mathcal{R}_1) \\ 0^{\oplus L(\mathcal{R}_1,h-1)} \oplus \text{inj}_{k-\text{size}(\mathcal{R}_0)}^{L(\mathcal{R}_1,h-1)} |v\rangle & \text{otherwise.} \end{cases}$$

K Proofs of Correctness for Low-Level Compilation

In this section, we reproduce the proofs from Appendix H.1 in Voichick et al., with appropriate modifications. The constructions in this section correspond to primitive operations in Qunity's intermediate representation. Note, however, that this does not constitute a proof of correctness of the entire Qunity compilation procedure, as mathematically formalizing the circuit specification data structures, the process of circuit instantiation, and qubit allocation and recycling in registers is a complex task beyond the scope of this work. Note that we do not need the "validation" circuits introduced in Voichick et al. since our construction guarantees the preservation of valid encodings: this property is not stated explicitly in the following, but it is is clear for the circuits in this section as the direct sum (Appendix I) is the only truly nontrivial construction in terms of encoding validity.

Lemma K.1. *It is always possible to implement the direct sum injections* $[\![\texttt{left}_{T_0 \oplus T_1}]\!]$ *and* $[\![\texttt{right}_{T_0 \oplus T_1}]\!]$.

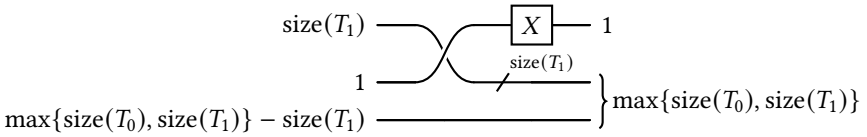
Proof. First, one can implement the operator $\texttt{left}_{T_0 \oplus T_1} : \mathcal{H}(T_0) \to \mathcal{H}(T_0) \oplus \mathcal{H}(T_1)$ using $1 + \max\{\text{size}(T_0), \text{size}(T_1)\} - \text{size}(T_0)$ prep wires and 0 flag wires with this circuit:



Using U to represent the qubit-based unitary implemented by this circuit, we can show that it meets the needed criteria:

$$
\begin{aligned}
&\left\langle \text{enc}(\texttt{left}_{T_0 \oplus T_1} v'), 0 \right| U \left| \text{enc}(v), 0, 0 \right\rangle \\
&= \langle 0, \text{enc}(v'), 0 | \, U \, | \text{enc}(v), 0, 0 \rangle \\
&= \langle 0, \text{enc}(v'), 0 | 0, \text{enc}(v), 0 \rangle \\
&= \langle \text{enc}(v') | \, \text{enc}(v) \rangle \\
&= \langle v' | v \rangle \\
&= (\langle v' | \oplus 0) \, (|v\rangle \oplus 0) \\
&= \langle \texttt{left}_{T_0 \oplus T_1} v' | \texttt{left}_{T_0 \oplus T_1} v \rangle \\
&\left\langle \text{enc}(\texttt{right}_{T_0 \oplus T_1} v'), 0 \right| U \left| \text{enc}(v), 0, 0 \right\rangle \\
&= \langle 1, \text{enc}(v'), 0 | \, U \, | \text{enc}(v), 0, 0 \rangle \\
&= \langle 1, \text{enc}(v'), 0 | 0, \text{enc}(v), 0 \rangle \\
&= 0 \\
&= (0 \oplus \langle v' |) \, (|v\rangle \oplus 0) \\
&= \langle \texttt{right}_{T_0 \oplus T_1} v' | \texttt{left}_{T_0 \oplus T_1} v \rangle
\end{aligned}
$$

Similarly, one can implement the operator $\texttt{right}_{T_0 \oplus T_1} : \mathcal{H}(T_1) \to \mathcal{H}(T_0) \oplus \mathcal{H}(T_1)$ using $1 + \max\{\text{size}(T_0), \text{size}(T_1)\} - \text{size}(T_1)$ prep wires and 0 flag wires like this:



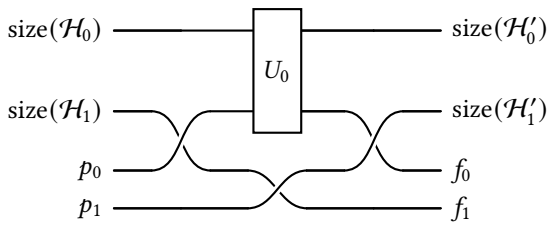
$$
\begin{aligned}
&\left\langle \mathrm{enc}(\mathtt{left}_{T_0 \oplus T_1} v'), 0 \right| U \left| \mathrm{enc}(v), 0, 0 \right\rangle \\
=\ & \langle 0, \mathrm{enc}(v'), 0 | \, U \, | \mathrm{enc}(v), 0, 0 \rangle \\
=\ & \langle 0, \mathrm{enc}(v'), 0 | 1, \mathrm{enc}(v), 0 \rangle \\
=\ & \langle \mathrm{enc}(v') | \, \mathrm{enc}(v) \rangle \\
=\ & 0 \\
=\ & (\langle v' | \oplus 0) \, (0 \oplus |v\rangle) \\
=\ & \langle \mathtt{left}_{T_0 \oplus T_1} v' | \mathtt{right}_{T_0 \oplus T_1} v \rangle \\
&\left\langle \mathrm{enc}(\mathtt{right}_{T_0 \oplus T_1} v'), 0 \right| U \left| \mathrm{enc}(v), 0, 0 \right\rangle \\
=\ & \langle 1, \mathrm{enc}(v'), 0 | \, U \, | \mathrm{enc}(v), 0, 0 \rangle \\
=\ & \langle 1, \mathrm{enc}(v'), 0 | 1, \mathrm{enc}(v), 0 \rangle \\
=\ & \langle \mathrm{enc}(v') | \, \mathrm{enc}(v) \rangle \\
=\ & \langle v' | v \rangle \\
=\ & (0 \oplus \langle v' |) \, (0 \oplus |v\rangle) \\
=\ & \langle \mathtt{right}_{T_0 \oplus T_1} v' | \mathtt{right}_{T_0 \oplus T_1} v \rangle
\end{aligned}
$$

□

When we construct circuits of these operators, we are implicitly using three things: identity operators (represented by bare wires), tensor products (represented by vertically-stacked gates), and operator composition (represented by horizontally-stacked gates). To justify this form of circuit diagram, we must show that these constructions are always possible. It is always possible to implement the identity operator $\mathbb{I} : \mathcal{H}(T) \to \mathcal{H}(T)$, by using an identity circuit with no prep or flag wires. The next two lemmas demonstrate that tensor products and compositions are possible.

Lemma K.2. *Suppose it is possible to implement the operators $E_0 : \mathcal{H}_0 \to \mathcal{H}_0'$ and $E_1 : \mathcal{H}_1 \to \mathcal{H}_1'$. Then it is possible to implement the operator $E_0 \otimes E_1 : \mathcal{H}_0 \otimes \mathcal{H}_1 \to \mathcal{H}_0' \otimes \mathcal{H}_1'$.*

Proof. Assume E_0 is implemented by U_0 with p_0 prep wires and f_0 flag wires, and assume that E_1 is implemented by U_1 with p_1 prep wires and f_1 flag wires. The following qubit circuit U then implements $E_0 \otimes E_1$ with $p_0 + p_1$ prep wires and $f_0 + f_1$ flag wires:

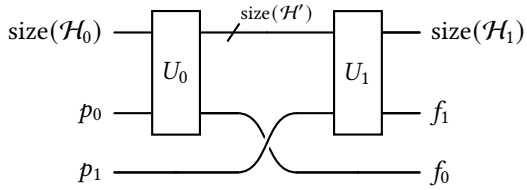


$$
\begin{aligned}
&\left\langle \mathrm{enc}(v_0'), \mathrm{enc}(v_1'), 0, 0 \right| U \left| \mathrm{enc}(v_0), \mathrm{enc}(v_1), 0, 0 \right\rangle \\
=\ & \left\langle \mathrm{enc}(v_0'), 0, \mathrm{enc}(v_1'), 0 \right| (U_0 \otimes U_1) \left| \mathrm{enc}(v_0), 0, \mathrm{enc}(v_1), 0 \right\rangle \\
=\ & \left\langle \mathrm{enc}(v_0'), 0 \right| U_0 \left| \mathrm{enc}(v_0), 0 \right\rangle \cdot \left\langle \mathrm{enc}(v_1'), 0 \right| U_1 \left| \mathrm{enc}(v_1), 0 \right\rangle \\
=\ & \left\langle v_0' \right| E_0 \left| v_0 \right\rangle \cdot \left\langle v_1' \right| E_1 \left| v_1 \right\rangle \\
=\ & \left\langle v_0', v_1' \right| (E_0 \otimes E_1) \left| v_0, v_1 \right\rangle
\end{aligned}
$$

□

Lemma K.3. *Suppose it is possible to implement the operators $E_0 : \mathcal{H}_0 \to \mathcal{H}'$ and $E_1 : \mathcal{H}' \to \mathcal{H}_1$. Then it is possible to implement the operator $E_1 E_0 : \mathcal{H}_0 \to \mathcal{H}_1$.*

Proof. Assume E_0 is implemented by $C(E_0) = U_0$ with p_0 prep wires and f_0 flag wires, and assume that E_1 is implemented by $C(E_1) = U_1$ with p_1 prep wires and f_1 flag wires. Let B' be a basis for $\mathcal{H}'$. The following qubit circuit U then implements $E_1 E_0$ with $p_0 + p_1$ prep wires and $f_0 + f_1$ flag wires:



$$
\begin{aligned}
& \langle \mathrm{enc}(v_1), 0, 0 |\, U \,| \mathrm{enc}(v_0), 0, 0 \rangle \\
= \; & \langle \mathrm{enc}(v_1), 0, 0 |\, (U_1 \otimes \mathbb{I})(\mathbb{I} \otimes \mathrm{SWAP})(U_0 \otimes \mathbb{I}) \,| \mathrm{enc}(v_0), 0, 0 \rangle \\
= \; & (\langle \mathrm{enc}(v_1), 0 |\, U_1 \otimes \langle 0 |) \, (\mathbb{I} \otimes \mathrm{SWAP}) \, (U_0 \,| \mathrm{enc}(v_0), 0 \rangle \otimes |0\rangle) \\
= \; & (\langle \mathrm{enc}(v_1), 0 |\, U_1) \, (\mathbb{I} \otimes |0\rangle\langle 0|) \, (U_0 \,| \mathrm{enc}(v_0), 0 \rangle) \\
= \; & \sum_{b \in \{0,1\}^{\mathrm{size}(\mathcal{H}')}} \langle \mathrm{enc}(v_1), 0 |\, U_1 | b, 0 \rangle\langle b, 0 | U_0 \,| \mathrm{enc}(v_0), 0 \rangle \\
= \; & \sum_{|v'\rangle \in B'} \langle \mathrm{enc}(v_1), 0 |\, C(E_1) |\, \mathrm{enc}(v'), 0 \rangle\langle \mathrm{enc}(v'), 0 | C(E_0) \,| \mathrm{enc}(v_0), 0 \rangle \\
= \; & \sum_{|v'\rangle \in B'} \langle v_1 |\, E_1 | v' \rangle\langle v' | E_0 \,| v_0 \rangle \\
= \; & \langle v_1 |\, E_1 \left(\sum_{v' \in B'} |v'\rangle\langle v'| \right) E_0 \,| v_0 \rangle \\
= \; & \langle v_1 |\, E_1 \mathbb{I} E_0 \,| v_0 \rangle \\
= \; & \langle v_1 |\, E_1 E_0 \,| v_0 \rangle
\end{aligned}
$$

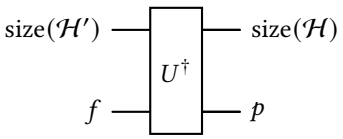
□

The adjoint is another useful construction that we will need in Appendix L, motivating the following lemma:

Lemma K.4. *Suppose it is possible to implement the operator $E : \mathcal{H} \to \mathcal{H}'$. Then it is possible to implement the operator $E^\dagger : \mathcal{H}' \to \mathcal{H}$.*

Proof. Assume E is implemented by U with p prep wires and f flag wires. Given the circuit for U, one can construct a circuit for $U^\dagger$ in the standard way: by taking the adjoint of each gate in the circuit and reversing the order. This circuit $U^\dagger$ then implements $E^\dagger$ with f prep wires and p flag

wires.



$$\begin{aligned}
&\langle \mathrm{enc}(v'), 0 |\, U^\dagger \,| \mathrm{enc}(v), 0 \rangle \\
&= (\langle \mathrm{enc}(v), 0 |\, U \,| \mathrm{enc}(v'), 0 \rangle)^* \\
&= (\langle v |\, E \,| v' \rangle)^* \\
&= \langle v' |\, E^\dagger \,| v \rangle
\end{aligned}$$

□

Lemma K.5. *Suppose it is possible to implement the operators $E_0 : \mathcal{H}_0 \to \mathcal{H}_0'$ and $E_1 : \mathcal{H}_1 \to \mathcal{H}_1'$. Then it is possible to implement the operator $E_0 \oplus E_1 : \mathcal{H}_0 \oplus \mathcal{H}_1 \to \mathcal{H}_0' \oplus \mathcal{H}_1'$.*

Proof. See Lemma I.1. □

It is standard to use the tensor product monoidally, implicitly using vector space isomorphisms $(\mathcal{H}_1 \otimes \mathcal{H}_2) \otimes \mathcal{H}_3 \approx \mathcal{H}_1 \otimes (\mathcal{H}_2 \otimes \mathcal{H}_3)$ and $\mathbb{C} \otimes \mathcal{H} \approx \mathcal{H} \approx \mathcal{H} \otimes \mathbb{C}$. Under the most general definition of the tensor product, these are isomorphisms rather than strict equality [24, Chapter 14], but these isomorphisms can typically be used implicitly. Our encoding ensures that these implicit isomorphisms do not require any low-level gates because $\mathrm{enc}(((v_1, v_2), v_3)) = \mathrm{enc}((v_1, (v_2, v_3)))$ and $\mathrm{enc}((), v)) = \mathrm{enc}(v) = \mathrm{enc}((v, ()))$.

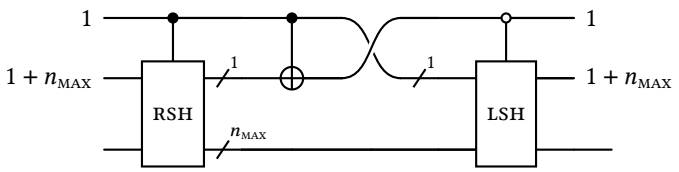
However, the associativity and distributivity isomorphisms for the direct sum *do* correspond to different encodings, so we must show how to implement them. Note that the additive unit isomorphisms were already implemented in Lemma K.1. They are $\texttt{left}_{T \oplus \texttt{Void}}$ and $\texttt{right}_{\texttt{Void} \oplus T}$, and their inverses are their adjoints as compiled in Lemma K.4. The following lemma demonstrates how to compile the associativity isomorphism:

Lemma K.6. *Let T_1, T_2, and T_3 be arbitrary types. Then it is possible to implement the direct sum's associativity isomorphism* assoc $: (\mathcal{H}(T_1) \oplus \mathcal{H}(T_2)) \oplus \mathcal{H}(T_3) \to \mathcal{H}(T_1) \oplus (\mathcal{H}(T_2) \oplus \mathcal{H}(T_3))$.

Proof. Define the following integers:

$$\begin{aligned}
n_{\mathrm{MAX}} &= \max\{\mathrm{size}(T_1), \mathrm{size}(T_2), \mathrm{size}(T_3)\} \\
p &= n_{\mathrm{MAX}} - \max\{\max\{\mathrm{size}(T_1), \mathrm{size}(T_2)\}, \mathrm{size}(T_3) - 1\} \\
f &= n_{\mathrm{MAX}} - \max\{\mathrm{size}(T_1) - 1, \max\{\mathrm{size}(T_2), \mathrm{size}(T_3)\}
\end{aligned}$$

We can implement "shifting" operations rsh and lsh via a series of swap gates that enacts a rotation permutation, so that $\mathrm{rsh}\,|\psi_1, \psi_2, \ldots, \psi_{n-1}, \psi_n\rangle = |\psi_n, \psi_1, \psi_2, \ldots, \psi_{n-1}\rangle$ and $\mathrm{lsh} = \mathrm{rsh}^{-1}$. The following qubit circuit U then implements the associator with the p prep wires and f flag wires.



We can show that this circuit has the desired behavior:

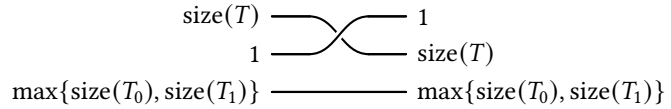
$$\begin{aligned}
& \left|\mathrm{enc}\left(\mathtt{left}_{(T_1\oplus T_2)\oplus T_3}\left(\mathtt{left}_{T_1\oplus T_2} v\right)\right), 0\right\rangle \\
= \;& |0, 0, \mathrm{enc}(v), 0\rangle \\
\mapsto^* \;& |0, \mathrm{enc}(v), 0, 0\rangle \\
= \;& \left|\mathrm{enc}(\mathtt{left}_{T_1\oplus(T_2\oplus T_3)} v), 0\right\rangle \\
& \left|\mathrm{enc}\left(\mathtt{left}_{(T_1\oplus T_2)\oplus T_3}\left(\mathtt{right}_{T_1\oplus T_2} v\right)\right), 0\right\rangle \\
= \;& |0, 1, \mathrm{enc}(v), 0\rangle \\
\mapsto^* \;& |1, 0, \mathrm{enc}(v), 0\rangle \\
= \;& \left|\mathrm{enc}\left(\mathtt{right}_{T_1\oplus(T_2\oplus T_3)}\left(\mathtt{left}_{T_2\oplus T_3} v\right)\right), 0\right\rangle \\
& \left|\mathrm{enc}\left(\mathtt{right}_{(T_1\oplus T_2)\oplus T_3} v\right), 0\right\rangle \\
= \;& |1, \mathrm{enc}(v), 0\rangle \\
\mapsto \;& |1, 0, \mathrm{enc}(v), 0\rangle \\
\mapsto \;& |1, 1, \mathrm{enc}(v), 0\rangle \\
= \;& \left|\mathrm{enc}\left(\mathtt{right}_{T_1\oplus(T_2\oplus T_3)}\left(\mathtt{right}_{T_2\oplus T_3} v\right)\right), 0\right\rangle
\end{aligned}$$

□

We can now implement the monoidal isomorphisms for the tensor product and direct sum. Both of these are *symmetric* monoidal categories, and the swap maps are straightforward to implement, using swap gates to implement $T_0 \otimes T_1 \cong T_1 \otimes T_0$, and using a single Pauli-X gate on the indicator qubit to implement $T_0 \oplus T_1 \cong T_1 \oplus T_0$. We will also need *distributivity* of the tensor product over the direct sum, part of the definition of a *bimonoidal* category [35] (sometimes known as a "rig category" [5]).

Lemma K.7. *Let T, T_0, and T_1 be arbitrary types. It is possible to implement the distributivity isomorphism* distr $: \mathcal{H}(T) \otimes (\mathcal{H}(T_0) \oplus \mathcal{H}(T_1)) \cong (\mathcal{H}(T) \otimes \mathcal{H}(T_0)) \oplus (\mathcal{H}(T) \otimes \mathcal{H}(T_1))$*, which acts like* $|v\rangle \otimes (|v_0\rangle \oplus |v_1\rangle) \mapsto |v, v_0\rangle \oplus |v, v_1\rangle$.

Proof. This can be done with no prep or flag wires:



It should be clear from the value encoding that this circuit has the correct behavior. □

In the high-level circuits, we will often use some transformation of the distr construction, for example:

- its adjoint $(\mathcal{H}(T) \otimes \mathcal{H}(T_0)) \oplus (\mathcal{H}(T) \otimes \mathcal{H}(T_1)) \cong \mathcal{H}(T) \otimes (\mathcal{H}(T_0) \oplus \mathcal{H}(T_1))$,
- its composition with swaps $(\mathcal{H}(T_0) \oplus \mathcal{H}(T_1)) \otimes \mathcal{H}(T) \cong (\mathcal{H}(T_0) \otimes \mathcal{H}(T)) \oplus (\mathcal{H}(T_1) \otimes \mathcal{H}(T))$,
- compositions of distributions $(\mathcal{H}(T_{00}) \oplus \mathcal{H}(T_{01})) \otimes (\mathcal{H}(T_{10}) \oplus \mathcal{H}(T_{11})) \cong (\mathcal{H}(T_{00}) \otimes \mathcal{H}(T_{10})) \oplus (\mathcal{H}(T_{00}) \otimes \mathcal{H}(T_{11})) \oplus (\mathcal{H}(T_{01}) \otimes \mathcal{H}(T_{10})) \oplus (\mathcal{H}(T_{01}) \otimes \mathcal{H}(T_{11}))$.

We will denote all of these as "distr," but the transformation being applied should always be clear from context.

Additional operators used in Qunity's intermediate representation include "context partition" and "context merge" operators. We maintain the invariant that when contexts are encoded into qubit

registers, the encodings of the variables are stored in lexicographic order. Therefore, implementing the isomorphism $\mathcal{H}(\Delta_0, \Delta_1) \to \mathcal{H}(\Delta_0) \otimes \mathcal{H}(\Delta_1)$ may require the use of SWAP gates. We use these operators implicitly in most of the constructions in Appendix L.

Lemma K.8 (pure error handling). *Suppose it is possible to implement a contraction $E : \mathcal{H} \to \mathcal{H}'$. Then it is possible to implement a norm-preserving operator $E_F : \mathcal{H} \to \mathcal{H}' \oplus \mathcal{H}_F$ for some "flag space" $\mathcal{H}_F$ such that $[\![\texttt{left}_{T \oplus T'}]\!]^\dagger E_F = E$.*

Proof. Assume that E is implemented by the unitary U with p prep wires and f flag wires. Then, since we assume encoding validity to be preserved, we can write

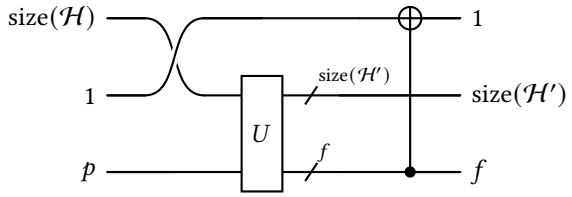
$$U \left|\mathrm{enc}(v), \texttt{0}\right\rangle = \left(\sum_{v' \in \mathbb{V}(T')} \left\langle v' \right| E \left| v \right\rangle \left|\mathrm{enc}(v'), \texttt{0}\right\rangle \right) + \left|\psi_v\right\rangle$$

for some $\left|\psi_v\right\rangle \in \mathcal{H}_\mathrm{F}$ with $(\mathbb{I} \otimes \left|\texttt{0}\right\rangle\!\left\langle\texttt{0}\right|) \left|\psi_v\right\rangle = 0$. We use E_F defined such that

$$E_\mathrm{F} \left| v \right\rangle = \left(\sum_{v' \in \mathbb{V}(T')} \left\langle v' \right| E \left| v \right\rangle \left| v' \right\rangle \right) \oplus \left|\psi_v\right\rangle = E \left| v \right\rangle \oplus \left|\psi_v\right\rangle .$$

This definition ensures that $[\![\texttt{left}_{T \oplus T'}]\!]^\dagger E_\mathrm{F} \left| v \right\rangle = [\![\texttt{left}_{T \oplus T'}]\!]^\dagger \left(E \left| v \right\rangle \oplus \left|\psi_v\right\rangle \right) = E \left| v \right\rangle$. See that $\left\| E \left| v \right\rangle \right\| = \left\| U \left| \mathrm{enc}\, v, \texttt{0} \right\rangle \right\| = 1$, so E_F is norm-preserving.

The following circuit implements E_F with $p + 1$ prep wires and no flag wires.



$$\begin{aligned}
\left|\mathrm{enc}(v), 0, 0\right\rangle &\mapsto \left|0, \mathrm{enc}(v), 0\right\rangle \\
&\mapsto \left|0\right\rangle \otimes U \left|\mathrm{enc}(v), 0\right\rangle \\
&= \left(\sum_{v' \in \mathbb{V}(T')} \left\langle v' \right| E \left| v \right\rangle \left|\texttt{0}, \mathrm{enc}\, v', \texttt{0}\right\rangle \right) + \left|\texttt{0}, \psi_v\right\rangle \\
&\mapsto \left(\sum_{v' \in \mathbb{V}(T')} \left\langle v' \right| E \left| v \right\rangle \left|\texttt{0}, \mathrm{enc}\, v', \texttt{0}\right\rangle \right) + \left|1, \psi_v\right\rangle \\
\left\langle 0, v', 0 \right| C(E_\mathrm{F}) \left|\mathrm{enc}(v), 0, 0\right\rangle &= \left\langle 0, v', 0 \right| \left(\sum_{v' \in \mathbb{V}(T')} \left\langle v' \right| E \left| v \right\rangle \left|\texttt{0}, \mathrm{enc}\, v', \texttt{0}\right\rangle \right) \\
&= \left(\left\langle v' \right| \oplus 0 \right) E_\mathrm{F} \left| v \right\rangle \\
\left\langle 1, v' \right| C(E_\mathrm{F}) \left|\mathrm{enc}(v), 0, 0\right\rangle &= \left\langle 1, v' \right| \left|1, \psi_v\right\rangle \\
&= \left(0 \oplus \left\langle v' \right| \right) E_\mathrm{F} \left| v \right\rangle
\end{aligned}$$

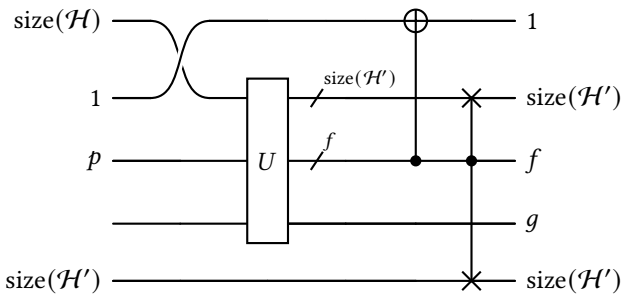
□

Lemma K.9 (mixed error handling). *Suppose it is possible to implement the completely positive trace-non-increasing linear superoperator $\mathcal{E} \in \mathcal{L}(\mathcal{L}(\mathcal{H}), \mathcal{L}(\mathcal{H}'))$. Then, it is possible to implement a*

completely positive trace-preserving linear superoperator $\mathrm{CPTP}(\mathcal{E}) \in \mathcal{L}(\mathcal{L}(\mathcal{H}), \mathcal{L}(\mathcal{H}' \oplus \mathbb{C}))$ *such that for all* $\rho \in \mathcal{L}(\mathcal{H})$,

$$\mathrm{CPTP}(\mathcal{E})(\rho) = \mathcal{E}(\rho) \oplus (\mathrm{tr}(\rho) - \mathrm{tr}(\mathcal{E}(\rho))).$$

Proof. Assuming $\mathcal{E}$ is implemented by the unitary U with p prep wires, f flag wires, and g garbage wires, the following circuit achieves the desired result with $p + \mathrm{size}(\mathcal{H}') + 1$ prep wires, no flag wires, and $f + \mathrm{size}(\mathcal{H}') + g$ garbage wires:



This circuit works by turning flag wires from $\mathcal{E}$ into garbage wires for $\mathrm{CPTP}(\mathcal{E})$. The first qubit is used as an indicator of failure, and in the event of failure, the output of U is treated as garbage and replaced with a fresh set of $|\mathtt{0}\rangle$ qubits from the prep register, as required by the bitstring encoding of sum types. Here, the "control" on the flag wires should be understood to apply the gate conditioned on *any* of the qubits on the control register being in the $|1\rangle$ state, which can be implemented using a control construct conditioned on *all* of the qubits on the control register being in the $|\mathtt{0}\rangle$ state followed by an uncontrolled gate.

To prove that this circuit superoperator C_{CPTP} correctly implements $\mathrm{CPTP}(\mathcal{E})$, we must show that

$$\left\langle v_1' \right| C_{\text{CPTP}}(|v_1\rangle\langle v_2|) \left| v_2' \right\rangle = \sum_b \left\langle \mathrm{enc}(v_1'), b \right| \mathcal{E}(|\,\mathrm{enc}(v_1), \mathtt{0}\rangle\langle \mathrm{enc}(v_2), \mathtt{0}|) \left| \mathrm{enc}(v_2'), b \right\rangle$$

for all $v_1, v_2 \in \mathbb{V}(T), v_1', v_2' \in \mathbb{V}(T' \oplus \mathtt{Unit})$.

We will consider three cases for v_1' and v_2'.

- First, suppose both are in the "success" (non-error) subspace, and see that

  $$(\langle v_1'| \oplus 0)\,\mathrm{CPTP}(\mathcal{E})(|v_1\rangle\langle v_2|)(|v_2'\rangle \oplus 0) = \left\langle v_1' \right| \mathcal{E}(|v_1\rangle\langle v_2|) \left| v_2' \right\rangle.$$

  To see that this equals

  $$\sum_b \left\langle \mathtt{0}, \mathrm{enc}(v_1'), b \right| C_{\text{CPTP}}(|\,\mathrm{enc}(v_1), \mathtt{0}\rangle\langle \mathrm{enc}(v_2), \mathtt{0}|) \left| \mathtt{0}, \mathrm{enc}(v_2'), b \right\rangle,$$

  see from the circuit that a $\mathtt{0}$ output on the first wire also implies a $\mathtt{0}$ output on the f segment of the output wires, as well as the final $\mathrm{size}(\mathcal{H}')$ section, so we really care about

  $$\sum_{b \in \{\mathtt{0},1\}^g} \left\langle \mathtt{0}, \mathrm{enc}(v_1'), \mathtt{0}, b, \mathtt{0} \right| C_{\text{CPTP}}(|\,\mathrm{enc}(v_1), \mathtt{0}\rangle\langle \mathrm{enc}(v_2), \mathtt{0}|) \left| \mathtt{0}, \mathrm{enc}(v_2'), \mathtt{0}, b, \mathtt{0} \right\rangle.$$

  The requirements placed on U ensure that this equality holds.
- Consider the case where exactly one of the two values is in the error subspace, for example $(\langle v_1'| \oplus 0)\,\mathrm{CPTP}(\mathcal{E})(|v_1\rangle\langle v_2|)(0 \oplus |v_2'\rangle) = 0$. Our circuit works correctly in this case because $\left\langle \mathtt{0}, \mathrm{enc}(v_1'), b \right| C_{\text{CPTP}}(|\,\mathrm{enc}(v_1), \mathtt{0}\rangle\langle \mathrm{enc}(v_2), \mathtt{0}|) \left| 1, \mathrm{enc}(v_2'), b \right\rangle$ is always zero, regardless of b. To see this, see that the f garbage bits are all zero if and only if the first indicator output bit is zero, so any setting of b would cause one of the two sides of the expression to vanish. In

other words, there is no superposition between the error and non-error subspaces because the discarded garbage collapses the state into one of the two.

- In the final case, both v_1' and v_2' are in the error subspace, so we must show that

$$\sum_b \langle 1, \mathtt{0}, b|\, C_{\text{CPTP}}(|\operatorname{enc}(v_1), \mathtt{0}\rangle\langle \operatorname{enc}(v_2), \mathtt{0}|)\, |1, \mathtt{0}, b\rangle = \operatorname{tr}(|v_1\rangle\langle v_2|) - \operatorname{tr}(\mathcal{E}(|v_1\rangle\langle v_2|)),$$

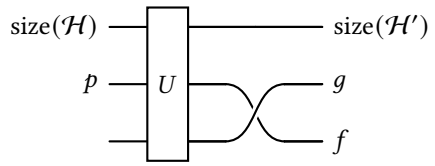
as we are encoding an error value as $\texttt{"1"} +\!\!+ \texttt{"0"}^{\operatorname{size}(\mathcal{H}')}$. This case is constrained by the first two, as it is the only possible value that would ensure that $\text{CPTP}(\mathcal{E})$ is trace-preserving. To verify that $\text{CPTP}(\mathcal{E})$ is trace-preserving, see that there are no flag wires and U cannot output invalid encodings without setting flag qubits, so there is no way for the trace to decrease.

□

Lemma K.10 (purification). *Suppose it is possible to implement the completely positive trace-non-increasing linear superoperator $\mathcal{E} \in \mathcal{L}(\mathcal{L}(\mathcal{H}), \mathcal{L}(\mathcal{H}'))$. Then, it is possible to implement a contraction $E \in \mathcal{L}(\mathcal{H}, \mathcal{H}' \otimes \mathcal{H}_G)$ for some "garbage" Hilbert space $\mathcal{H}_G$ with the following property: for any $\rho \in \mathcal{L}(\mathcal{H}), |\psi\rangle \in \mathcal{H}'$, there is some $\left|g_{\rho,\psi}\right\rangle \in \mathcal{H}_G$ such that*

$$\langle\psi|\, \mathcal{E}(\rho)\, |\psi\rangle = \left\langle g_{\rho,\psi}, \psi\right| E\rho E^\dagger \left|g_{\rho,\psi}, \psi\right\rangle$$

Proof. Assuming $\mathcal{E}$ is implemented by the unitary U with p prep wires, f flag wires, and g garbage wires, the following circuit achieves the desired result with p prep wires and f flag wires by setting $\mathcal{H}_G = \mathbb{C}^{2^g}$:



This circuit simply feeds the existing garbage wires into an additional output. □

Next, we consider some results about how to implement trace-non-increasing superoperators with low-level qubit-based unitaries.

Lemma K.11. *For any type T, it is possible to implement a superoperator $\mathcal{E} : \mathcal{L}(\mathcal{H}(T)) \to \mathcal{L}(\mathcal{H}(\texttt{Unit}))$ that computes the trace of its input, effectively discarding it.*

Proof. This gate is implemented with an empty (identity) circuit by setting $p = f = 0$, $g = \operatorname{size}(T)$.

$$\operatorname{size}(T) \text{ ── } g$$

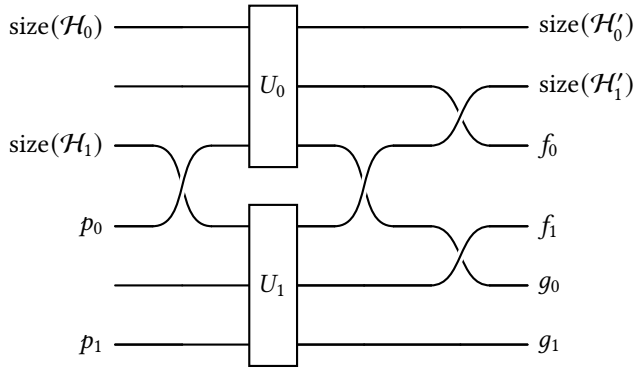
$$\sum_{g \in \{\mathtt{0},\mathtt{1}\}^{\operatorname{size}(T)}} \langle \operatorname{enc}(()), g|\, \mathbb{I}\rho\mathbb{I}^\dagger\, |\operatorname{enc}(()), g\rangle = \operatorname{tr}(\rho)$$

□

In Appendix L, we will also be constructing circuits from these trace-non-increasing superoperators. Again, we must justify this by demonstrating that tensor products and function composition are possible.

Lemma K.12. *Suppose it is possible to implement the superoperators $\mathcal{E}_0 : \mathcal{L}(\mathcal{H}_0) \to \mathcal{L}(\mathcal{H}_0')$ and $\mathcal{E}_1 : \mathcal{L}(\mathcal{H}_1) \to \mathcal{L}(\mathcal{H}_1')$. Then it is possible to implement the operator $\mathcal{E}_0 \otimes \mathcal{E}_1 : \mathcal{L}(\mathcal{H}_0 \otimes \mathcal{H}_1) \to \mathcal{L}(\mathcal{H}_0' \otimes \mathcal{H}_1')$.*

Proof. Assume E_0 is implemented by U_0 with p_0 prep wires, f_0 flag wires, and g garbage wires. Assume E_1 is implemented by U_1 with p_1 prep wires, f_1 flag wires, and g garbage wires. The following qubit circuit U then implements $E_0 \otimes E_1$ with $p = (p_0 + p_1)$ prep wires, $f = (f_0 + f_1)$ flag wires, and $g = (g_0 + g_1)$ garbage wires:



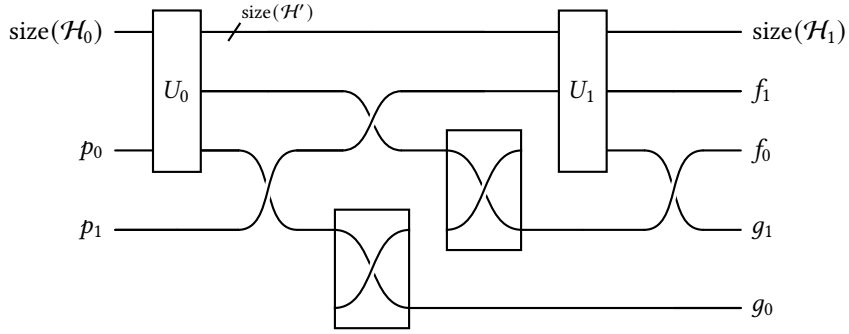
$$
\begin{aligned}
&\sum_{g_0 \in \{0,1\}^{g_0}} \sum_{g_1 \in \{0,1\}^{g_1}} \left\langle \mathrm{enc}(v'_{0,1}), \mathrm{enc}(v'_{1,1}), 0, g_0, g_1 \right| U \left(\rho_0 \otimes \rho_1 \otimes |0\rangle\langle 0|\right) \\
&\qquad \cdot U^\dagger \left| \mathrm{enc}(v'_{0,2}), \mathrm{enc}(v'_{1,2}), 0, g_0, g_1 \right\rangle \\
= &\sum_{g_0 \in \{0,1\}^{g_0}} \sum_{g_1 \in \{0,1\}^{g_1}} \left\langle \mathrm{enc}(v'_{0,1}), 0, g_0, \mathrm{enc}(v'_{1,1}), 0, g_1 \right| (U_0 \otimes U_1) \left(\rho_0 \otimes |0\rangle\langle 0| \otimes \rho_1 \otimes |0\rangle\langle 0|\right) \\
&\qquad \cdot (U_0 \otimes U_1)^\dagger \left| \mathrm{enc}(v'_{0,2}), 0, g_0, \mathrm{enc}(v'_{1,2}), 0, g_1 \right\rangle \\
= &\sum_{g_0 \in \{0,1\}^{g_0}} \left\langle \mathrm{enc}(v'_{0,1}), 0, g_0 \right| U_0 \left(\rho_0 \otimes |0\rangle\langle 0|\right) U_0^\dagger \left| \mathrm{enc}(v'_{0,2}), 0, g_0 \right\rangle \\
&\qquad \cdot \sum_{g_1 \in \{0,1\}^{g_1}} \left\langle \mathrm{enc}(v'_{1,1}), 0, g_1 \right| U_1(\rho_1 \otimes |0\rangle\langle 0|) U_1^\dagger \left| \mathrm{enc}(v'_{1,2}), 0, g_1 \right\rangle \\
= &\left\langle v'_{0,1} \right| \mathcal{E}_0(\rho_0) \left| v'_{0,2} \right\rangle \cdot \left\langle v'_{1,1} \right| \mathcal{E}_1(\rho_1) \left| v'_{1,2} \right\rangle \\
= &\left\langle v'_{0,1}, v'_{1,1} \right| (\mathcal{E}_0 \otimes \mathcal{E}_1)(\rho_0 \otimes \rho_1) \left| v'_{0,2}, v'_{1,2} \right\rangle
\end{aligned}
$$

□

Lemma K.13. *Suppose it is possible to implement the superoperators $\mathcal{E}_0 : \mathcal{L}(\mathcal{H}_0) \to \mathcal{L}(\mathcal{H}')$ and $\mathcal{E}_1 : \mathcal{L}(\mathcal{H}') \to \mathcal{L}(\mathcal{H}_1)$. Then it is possible to implement the operator $\mathcal{E}_1 \circ \mathcal{E}_0 : \mathcal{L}(\mathcal{H}_0) \to \mathcal{L}(\mathcal{H}_1)$.*

Proof. Assume E_0 is implemented by U_0 with p_0 prep wires, f_0 flag wires, and g_0 garbage wires. Assume E_1 is implemented by U_1 with p_1 prep wires, f_1 flag wires, and g_1 garbage wires. The following qubit circuit U then implements $E_0 \otimes E_1$ with $p_0 + p_1$ prep wires, $f_0 + f_1$ flag wires, and

$g_0 + g_1$ garbage wires:



As with the other compositions, we're grouping the flags from the two together and grouping the garbage from the two together. This diagram uses a couple of SWAP gates that do not correspond to physical gates but are just there to ensure wires in the diagram don't collide. This is necessary because of the way that we are using single wires to represent different numbers of qubits, for example the unitaries U_0 and U_1 would have the same number of input qubits as output qubits. □

LEMMA K.14 (TREE LEVELING OPERATOR). *Referring to the notation in Appendix J, let $\mathcal{R}$ be a binary tree of size n. Let $h \geq \text{height}(\mathcal{R})$, and let $\mathcal{H}_1, \ldots, \mathcal{H}_n$ be Hilbert spaces corresponding to Qunity types or contexts. It is possible to implement an operator*

$$\textsc{level}(\mathcal{R}; h; \mathcal{H}_1, \ldots, \mathcal{H}_n) : \bigoplus_{j:\mathcal{R}} \mathcal{H}_j \to \bigoplus_{j:L(\mathcal{R})} \mathcal{H}_j,$$

such that

$$\textsc{level}(\mathcal{R}; h; \mathcal{H}_1, \ldots, \mathcal{H}_n) \operatorname{inj}_j^{\mathcal{R}} |v\rangle = \operatorname{inj}_j^{L(\mathcal{R},h)} |v\rangle$$

PROOF. For the case where $\mathcal{R} = \text{Leaf}$, we can implement this using h prep wires and no flag wires with the following circuit U:

$$\begin{aligned}
&\langle 0, \text{enc}(v')| U |\text{enc}(v), 0\rangle = \\
&= \langle \text{enc}(v')|\text{enc}\, v\rangle = \\
&= \langle v'| \operatorname{inj}_1^{\mathcal{R}} \operatorname{inj}_1^{L(\mathcal{R},h)\dagger} \operatorname{inj}_1^{L(\mathcal{R},h)} \operatorname{inj}_1^{\mathcal{R}\dagger} |v\rangle = \\
&= \langle v'| \textsc{level}(\text{Leaf}; h; \mathcal{H}_1)^\dagger \textsc{level}(\text{Leaf}; h; \mathcal{H}_1) |v\rangle
\end{aligned}$$

Now, when $\mathcal{R} = (\mathcal{R}_0, \mathcal{R}_1)$, we can simply write

$$\textsc{level}((\mathcal{R}_0, \mathcal{R}_1); h; \mathcal{H}_1, \ldots, \mathcal{H}_n) = \textsc{level}(\mathcal{R}_0; h{-}1; \mathcal{H}_1, \ldots, \mathcal{H}_{\text{size}(\mathcal{R}_0)}) \oplus \textsc{level}(\mathcal{R}_1; h{-}1; \mathcal{H}_{\text{size}(\mathcal{R}_0)+1}, \ldots, \mathcal{H}_n),$$

where we take the direct sum of the operators using the construction in Appendix I. It is clear that this is correct from Definition J.3. □

The tree leveling operator allows us to take advantage of the way the encoding is structured and split a direct sum encoding into separate "index" and "data" registers, which will be used for constructing the orthogonality circuit in Appendix L.2.

L Proofs of Correctness for High-Level Compilation

In this section, we justify the correctness of the high-level stage of the compilation procedure, reproducing the original proofs from Voichick et al., with modifications from the addition of the new constructs and the changes described in Section 7.

L.1 Erasure Compilation

Lemma L.1 (erasure compilation). *Suppose* $(\Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T')$ *for all* $j \in \{1, \dots, n\}$ *and* $\text{erases}_{T'}(x; e'_1, \dots, e'_n)$ *is true for all* $x \in \text{dom}(\Delta)$. *Then, one can implement an operator*

$$[\![\text{erases}_{T'}(\Delta; e'_1, \dots, e'_n)]\!] : \mathcal{H}(\Delta) \otimes \mathcal{H}(T') \to \mathcal{H}(T')$$

with the following behavior for all $\sigma \in \mathbb{V}(\Gamma), \sigma_j \in \mathbb{V}(\Gamma_j), \tau \in \mathbb{V}(\Delta), \tau' \in \mathbb{V}(\Delta')$*:*

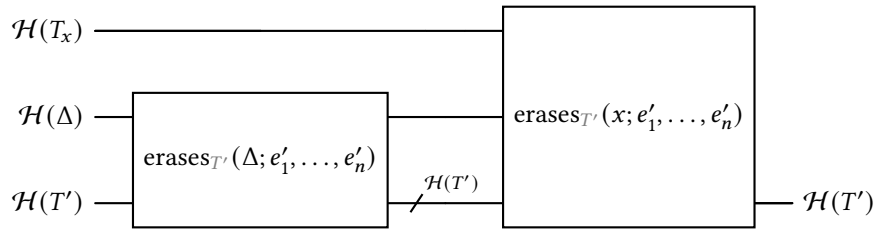
$$|\tau\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T']\!] \, |\tau, \tau'\rangle \mapsto [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel \Delta, \Delta' \vdash e'_j : T']\!] \, |\tau, \tau'\rangle$$

Proof. We construct the circuit by recursing on Δ. In the base case, $\Delta = \varnothing$ and an identity operator (empty circuit) suffices. Thus, we focus on the inductive case where our context is $(x : T_x, \Delta)$, assuming the inductive hypothesis that $[\![\text{erases}_{T'}(\Delta; e'_1, \dots, e'_n)]\!]$ is implementable with the behavior:

$$|\tau\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \| x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \mapsto [\![\sigma, \sigma_j : \Gamma, \Gamma_j \| x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle$$

(Note that x appears here even though this is the inductive hypothesis, which would normally be free of x. This is still a valid induction principle; we are effectively inducting on the number of variables that must be erased.) The problem is then reduced to implementing an operator $[\![\text{erases}_{T'}(x; e'_1, \dots, e'_n)]\!] : \mathcal{H}(T_x) \otimes \mathcal{H}(T') \to \mathcal{H}(T')$ with the following behavior:

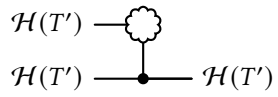
$$|v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \| x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \mapsto [\![\sigma, \sigma_j : \Gamma, \Gamma_j \| x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle$$



$$\begin{aligned}
&[\![\text{erases}_{T'}(x : T_x, \Delta; e'_1, \dots, e'_n)]\!] \\
: \quad &|x \mapsto v, \tau\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
\mapsto \; &|v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
\mapsto \; &[\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle
\end{aligned}$$

The rest of this proof constructs this gate $[\![\text{erases}_{T'}(x; e'_1, \dots, e'_n)]\!]$ by induction on the rule used to prove the erasure judgment.

E-Var: In this case, $e'_1 = \cdots = e'_n = x$ and $T' = T_x$. We know that $\Delta = \Delta' = \varnothing$ because these contexts must be relevant.



$$
\begin{aligned}
& [\![\mathrm{erases}_{T'}(x; e'_1, \ldots, e'_n)]\!] \\
: \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T' \vdash x : T']\!] \, |x \mapsto v\rangle \\
= \quad & |v\rangle \otimes |v\rangle \\
\mapsto \quad & |v\rangle \\
= \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T' \vdash x : T']\!] \, |x \mapsto v\rangle
\end{aligned}
$$

E-Gphase: In this case, the circuit produced by the inductive hypothesis already has the needed behavior.

$$
\begin{aligned}
& [\![\mathrm{erases}_{T'}(x; e'_1, \ldots, e'_{j-1}, e'_j \triangleright \mathsf{gphase}_T(r), e'_{j+1}, \ldots, e'_n)]\!] \\
: \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j \triangleright \mathsf{gphase}_T(r) : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
= \quad & |v\rangle \otimes e^{ir} [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
\mapsto \quad & e^{ir} [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
= \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash e'_j \triangleright \mathsf{gphase}_T(r) : T']\!] \, |x \mapsto v, \tau, \tau'\rangle
\end{aligned}
$$

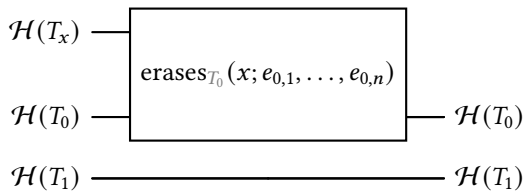
E-Ctrl: This rule allows us effectively to "inline" the right side of the `ctrl` expressions for the purpose of the erases judgment. Assume one of the expressions is of the following form:

$$
\texttt{ctrl}\ e\ {}_T\left\{ \begin{matrix} e_{j,1} \mapsto e'_{j,1} \\ \cdots \\ e_{j,m} \mapsto e'_{j,m} \end{matrix} \right\}_{T'}
$$

Then, we can use the fact that the semantics of `ctrl` is a linear combination of the semantics of its subexpressions:

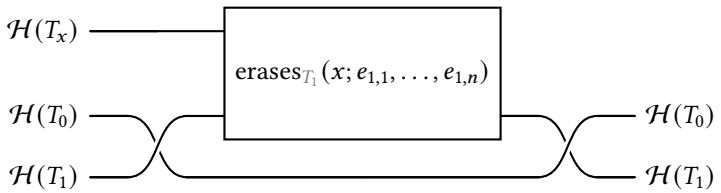
$$
\begin{aligned}
& [\![\mathrm{erases}_{T'}(x; e'_1, \ldots, e'_{j-1}, \texttt{ctrl}\ \ e\ {}_T\{\cdots\}_{T'}, e'_{j+1}, \ldots, e'_n)]\!] \\
: \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash \texttt{ctrl}\ \ e\ {}_T\{\cdots\}_{T'} : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
= \quad & \sum \cdots |v\rangle \otimes [\![\sigma, \sigma_{j,k} : \Gamma, \Gamma_{j,k} \parallel x : T_x, \Delta, \Delta' \vdash e'_{j,k} : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
\mapsto \quad & \sum \cdots [\![\sigma, \sigma_{j,k} : \Gamma, \Gamma_{j,k} \parallel x : T_x, \Delta, \Delta' \vdash e'_{j,k} : T']\!] \, |x \mapsto v, \tau, \tau'\rangle \\
= \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash \texttt{ctrl}\ \ e\ {}_T\{\cdots\}_{T'} : T']\!] \, |x \mapsto v, \tau, \tau'\rangle
\end{aligned}
$$

E-Pair0:



$$
\begin{aligned}
& [\![\mathrm{erases}_{T_0 \otimes T_1}(x; (e_{0,1}, e_{1,1}), \ldots, (e_{0,n}, e_{1,n}))]\!] \\
: \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash (e_{0,j}, e_{1,j}) : T_0 \otimes T_1]\!] \, |x \mapsto v, \tau, \tau'\rangle \\
= \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_0, \Delta_1, \Delta'_*, \Delta'_0, \Delta'_1 \vdash (e_{0,j}, e_{1,j}) : T_0 \otimes T_1]\!] \, \left|x \mapsto v, \tau_*, \tau_0, \tau_1, \tau'_*, \tau'_0, \tau'_1\right\rangle \\
= \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_0, \Delta'_*, \Delta'_0 \vdash e_{0,j} : T_0]\!] \left|x \mapsto v, \tau_*, \tau_0, \tau'_*, \tau'_0\right\rangle \\
& \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_1, \Delta'_*, \Delta'_1 \vdash e_{1,j} : T_1]\!] \left|x \mapsto v, \tau_*, \tau_1, \tau'_*, \tau'_1\right\rangle \\
\mapsto \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_0, \Delta'_*, \Delta'_0 \vdash e_{0,j} : T_0]\!] \left|x \mapsto v, \tau_*, \tau_0, \tau'_*, \tau'_0\right\rangle \\
& \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_1, \Delta'_*, \Delta'_1 \vdash e_{1,j} : T_1]\!] \left|x \mapsto v, \tau_*, \tau_1, \tau'_*, \tau'_1\right\rangle \\
= \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash (e_{0,j}, e_{1,j}) : T_0 \otimes T_1]\!] \, |x \mapsto v, \tau, \tau'\rangle
\end{aligned}
$$

E-Pair1:



$$
\begin{aligned}
& [\![\mathrm{erases}_{T_0 \otimes T_1}(x; (e_{0,1}, e_{1,1}), \ldots, (e_{0,n}, e_{1,n}))]\!] \\
: \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash (e_{0,j}, e_{1,j}) : T_0 \otimes T_1]\!] \, |x \mapsto v, \tau, \tau'\rangle \\
= \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_0, \Delta_1, \Delta'_*, \Delta'_0, \Delta'_1 \vdash (e_{0,j}, e_{1,j}) : T_0 \otimes T_1]\!] \, \left|x \mapsto v, \tau_*, \tau_0, \tau_1, \tau'_*, \tau'_0, \tau'_1\right\rangle \\
= \quad & |v\rangle \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_0, \Delta'_*, \Delta'_0 \vdash e_{0,j} : T_0]\!] \left|x \mapsto v, \tau_*, \tau_0, \tau'_*, \tau'_0\right\rangle \\
& \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_1, \Delta'_*, \Delta'_1 \vdash e_{1,j} : T_1]\!] \left|x \mapsto v, \tau_*, \tau_1, \tau'_*, \tau'_1\right\rangle \\
\mapsto \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_0, \Delta'_*, \Delta'_0 \vdash e_{0,j} : T_0]\!] \left|x \mapsto v, \tau_*, \tau_0, \tau'_*, \tau'_0\right\rangle \\
& \otimes [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta_*, \Delta_1, \Delta'_*, \Delta'_1 \vdash e_{1,j} : T_1]\!] \left|x \mapsto v, \tau_*, \tau_1, \tau'_*, \tau'_1\right\rangle \\
= \quad & [\![\sigma, \sigma_j : \Gamma, \Gamma_j \parallel x : T_x, \Delta, \Delta' \vdash (e_{0,j}, e_{1,j}) : T_0 \otimes T_1]\!] \, |x \mapsto v, \tau, \tau'\rangle
\end{aligned}
$$

We have thus demonstrated that a circuit with this semantics can always be constructed. □

L.2 The New Orthogonality Circuit

Lemma L.2. *Suppose that* $\mathrm{ortho}_T\,(e_1, \ldots, e_n)$ *holds, with tree structure* $\mathcal{R}$ *(Definition 7.1) and* $n > 0$. *Suppose that each* e_j *is typed using the pure expression typing judgment with no classical context and quantum context* Δ_j. *Then, it is possible to construct*

$$[\![\mathrm{ortho}_T\,(e_1, \ldots, e_n)]\!] : \mathcal{H}(T) \to \bigoplus_{j:\mathcal{R}} \mathcal{H}(\Delta_j),$$

such that

$$[\![\mathrm{ortho}_T\,(e_1, \ldots, e_n)]\!][\![e_j]\!] \left|\tau_j\right\rangle = \mathrm{inj}_j^{\mathcal{R}} \left|\tau_j\right\rangle.$$

Proof. For O-Void and O-Unit, this operator is simply the identity circuit on an empty register. For O-Var, it is also the identity circuit, since it maps $|v\rangle \mapsto |x \mapsto v\rangle$.

For O-IsoApp, we define

$$[\![\mathrm{ortho}_{T'}\,(fe_1, \ldots, fe_n)]\!] = [\![\mathrm{ortho}_T\,(e_1, \ldots, e_n)]\!][\![f]\!]^\dagger.$$

Then, we have that

$$\llbracket \mathrm{ortho}_{T'} \left(f e_1, \ldots, f e_n\right) \rrbracket \llbracket f e_j \rrbracket \left|\tau_j\right\rangle = \llbracket \mathrm{ortho}_{T} \left(e_1, \ldots, e_n\right) \rrbracket \llbracket f \rrbracket^\dagger \llbracket f \rrbracket \llbracket e_j \rrbracket \left|\tau_j\right\rangle =$$
$$= \llbracket \mathrm{ortho}_{T} \left(e_1, \ldots, e_n\right) \rrbracket \llbracket e_j \rrbracket \left|\tau_j\right\rangle = \mathrm{inj}_j^{\mathcal{R}} \left|\tau_j\right\rangle .$$

Here, we used the fact that f is an isometry to say that $\llbracket f \rrbracket^\dagger \llbracket f \rrbracket$ is the identity.

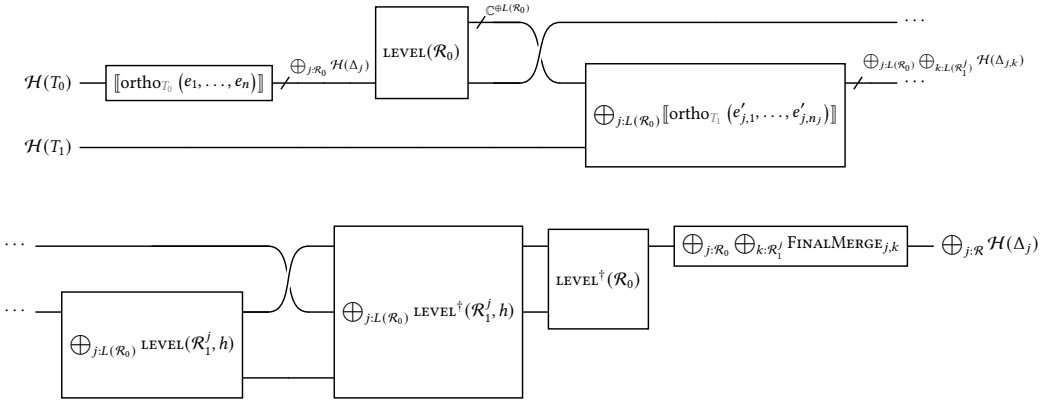
For O-Sum, we define

$$\llbracket \mathrm{ortho}_{T_0 \oplus T_1} \begin{pmatrix} \mathtt{left}_{T_0 \oplus T_1} e_1, \ldots, \mathtt{left}_{T_0 \oplus T_1} e_n, \\ \mathtt{right}_{T_0 \oplus T_1} e'_1, \ldots, \mathtt{right}_{T_0 \oplus T_1} e'_{n'} \end{pmatrix} \rrbracket = \llbracket \mathrm{ortho}_{T_0} \left(e_1, \ldots, e_n\right) \rrbracket \oplus \llbracket \mathrm{ortho}_{T_1} \left(e'_1, \ldots, e'_{n'}\right) \rrbracket.$$

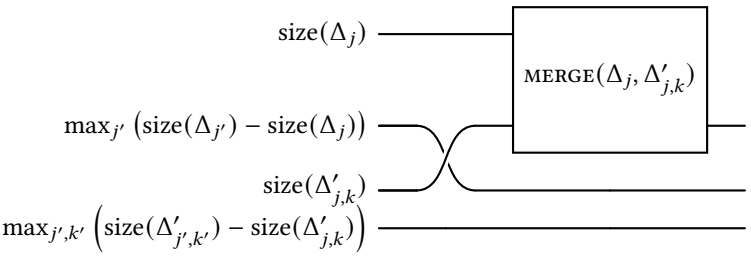
Suppose that the tree structures corresponding to $\mathrm{ortho}_{T_0} \left(e_1, \ldots, e_n\right)$ and $\mathrm{ortho}_{T_1} \left(e'_1, \ldots, e'_{n'}\right)$ are $\mathcal{R}_0$ and $\mathcal{R}_1$. This circuit acts as:

$$\llbracket \mathtt{left}_{T_0 \oplus T_1} e_j \rrbracket \left|\tau_j\right\rangle = \llbracket e_j \rrbracket \left|\tau_j\right\rangle \oplus 0 \mapsto \mathrm{inj}_j^{\mathcal{R}_0} \left|\tau_j\right\rangle \oplus 0^{\oplus \mathcal{R}_1} = \mathrm{inj}_j^{(\mathcal{R}_0, \mathcal{R}_1)} \left|\tau_j\right\rangle$$
$$\llbracket \mathtt{right}_{T_0 \oplus T_1} e'_j \rrbracket \left|\tau'_j\right\rangle = 0 \oplus \llbracket e'_j \rrbracket \left|\tau'_j\right\rangle \mapsto 0^{\oplus \mathcal{R}_1} \oplus \mathrm{inj}_j^{\mathcal{R}_0} \left|\tau_j\right\rangle = \mathrm{inj}_{\mathrm{size}(\mathcal{R}_0)+j}^{(\mathcal{R}_0, \mathcal{R}_1)} \left|\tau_j\right\rangle .$$

Now, for O-Pair: suppose that we have $\mathrm{ortho}_{T_0} \left(e_1, \ldots, e_n\right)$ holds with tree structure $\mathcal{R}_0$ where each e_j is typed with quantum context Δ_j, and for each j, we have $\mathrm{ortho}_{T_1} \left(e'_{j,1}, \ldots, e'_{j,n_j}\right)$ with tree structure $\mathcal{R}_1^j$, where each $e_{j,k}$ is typed with quantum context $\Delta'_{j,k}$. The tree structure of the combined sequence of pairs will be $\mathcal{R}$, which is obtained by replacing the jth leaf of $\mathcal{R}_0$ with a copy of $\mathcal{R}_1^j$. Now, the operator is defined by the following circuit:



where we define $h = \max_j \{\mathrm{height}(\mathcal{R}_1^j)\}$, we define $\mathrm{FinalMerge}_{j,k}$ (with a diagram drawn in the style of low-level circuits) as:



where the bottom two wires are marked as flag registers, since they contain zero-padding regions from direct sum encodings.

The LEVEL operator (Lemma K.14) is designed to separate a potentially uneven direct sum encoding structure into a separate *index register* and *data register*. This allows us to concatenate

the tree path from $\mathcal{R}_0$ and $\mathcal{R}_1^j$ into a single path in the combined tree, then merge the pieces of data from the two sides of the pair and put them in the correct place in $\mathcal{R}$. The action of the circuit proceeds as follows:

(1) First, applying $\text{ortho}_{T_0}\ (e_1, \ldots, e_n)$, we obtain the direct sum of the Δ_j over $\mathcal{R}_0$.
(2) After applying the leveling operator, we are able to partition our quantum register into two: the index register contains only the information about which path was taken on the tree (with trailing zeros where appropriate due to different-sized branches), and the data register corresponds to each of the Δ_j (with trailing zeros where appropriate due to different-sized contexts).
(3) We take a direct sum of the operators $[\![\text{ortho}_{T_1}\ (e'_{j,1}, \ldots, e'_{j,n_j})]\!]$ over the leveled tree $L(\mathcal{R}_0)$ and then take the direct sum of the leveling operators for each $\mathcal{R}_1^j$, where the leveling is done to the maximum height of all the $\mathcal{R}_1^j$. This creates a "stack" of the index registers of the trees: the first block of qubits encodes a branch in $\mathcal{R}_0$, possibly followed by some zeros, then the next block encodes a branch in $\mathcal{R}_1^j$ for the j corresponding to the path taken in the first block, possibly followed by more trailing zeros. The data register then contains all the $\Delta'_{j,k}$ starting at the same position.
(4) We put the stacked index registers on top and stack the data registers below them. Now, we need to eliminate the zeros interspersed between encodings and undo the separation into blocks. We apply the direct sum of the adjoints of the leveling operators for the $\mathcal{R}_1^j$, which removes the zeros between the index encodings and the combined data of Δ_j and $\Delta'_{j,k}$. Note that these leveling operators are not exactly the same as the ones before, since the data registers are now larger (we omit this in the notation in the circuit diagram).
(5) We then apply the adjoint of the leveling operator for $\mathcal{R}_0$. We now have a sum over the correct structure $\mathcal{R}$, but each leaf corresponds to two blocks of contexts, possibly separated by zeros.
(6) Applying the direct sum of the FinalMerge operators, we combine all the contexts in the correct way, obtaining the desired result.

Finally, for O-Sub: Since we assume $n > 0$ (we will treat $n = 0$ as a special case for the control flow constructs), at least one of the expressions before the application of O-Sub must be kept. If $\mathcal{R} = \text{Leaf}$, the circuit then must be the identity. Now, we define the circuit $U_{\mathcal{R}}$ recursively in terms of subtrees and selected indices. Suppose that $\mathcal{R} = (\mathcal{R}_0, \mathcal{R}_1)$. We can associate these subtrees with types T_0, T_1 that are formed as sum types taken along the binary trees, with the contexts at the leaves treated as product types (the encoding is identical). In the case where $\mathcal{R}_0$ has no selected indices, we define $U_{\mathcal{R}} = [\![\texttt{right}_{T_0 \oplus T_1}]\!]^\dagger$, and if $\mathcal{R}_1$ has no selected indices, we define $U_{\mathcal{R}} = [\![\texttt{left}_{T_0 \oplus T_1}]\!]^\dagger$. Otherwise, we just define $U_{\mathcal{R}} = U_{\mathcal{R}_0} \oplus U_{\mathcal{R}_1}$. It is clear that this operator specifically removes those subtrees that only contain discarded expressions.

□

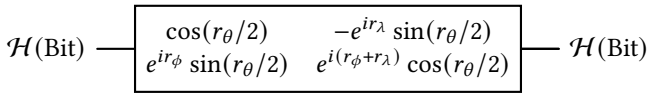
While the above circuit appears complicated, it is more efficient than the existing solution in many cases. In fact, if the expressions contain no variables in the first elements of the pairs, this entire circuit actually simplifies to the identity. This is because in that case, we do not need to do anything to combine the indices of the two trees, as the encodings already correspond to the desired structure.

L.3 Qunity Typing Judgment Compilation

Here we give the compiled circuits for all of the typing judgment cases and demonstrate algebraically that they are correct. In the following circuit diagrams, the "controlled cloud" is a "share" gate,

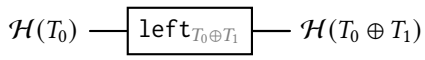
implemented simply as a series of CNOT gates between the given register and an ancilla register. Additionally, we put control symbols on wires associated with classical contexts, to indicate that any interaction with these wires only uses such share gates.

T-Gate: We assume that our low-level circuits contain these gates as primitives.



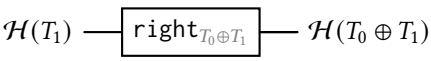
$$\begin{aligned}
&|\texttt{0}\rangle && \in \mathcal{H}(\text{Bit})\\
\mapsto\ &\cos(r_\theta/2)\,|\texttt{0}\rangle + e^{ir_\phi}\sin(r_\theta/2)\,|\texttt{1}\rangle && \in \mathcal{H}(\text{Bit})\\
&|\texttt{1}\rangle && \in \mathcal{H}(\text{Bit})\\
\mapsto\ &-e^{ir_\lambda}\sin(r_\theta/2)\,|\texttt{0}\rangle + e^{i(r_\phi+r_\lambda)}\cos(r_\theta/2)\,|\texttt{1}\rangle && \in \mathcal{H}(\text{Bit})
\end{aligned}$$

T-Left: This was already implemented in Appendix K.

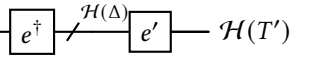


$$\begin{aligned}
&|v\rangle && \in \mathcal{H}(T_0)\\
\mapsto\ &|v\rangle \oplus 0
\end{aligned}$$

T-Right: This was already implemented in Appendix K.

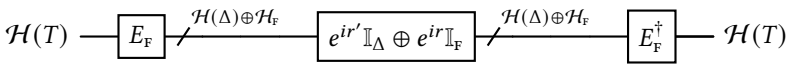


$$\begin{aligned}
&|v\rangle && \in \mathcal{H}(T_1)\\
\mapsto\ &0 \oplus |v\rangle
\end{aligned}$$

T-PureAbs: 

$$\begin{aligned}
&|v\rangle && \in \mathcal{H}(T)\\
\mapsto\ &[\![e]\!]^\dagger\,|v\rangle && \in \mathcal{H}(\Delta)\\
\mapsto\ &[\![e']\!][\![e]\!]^\dagger\,|v\rangle && \in \mathcal{H}(T')
\end{aligned}$$

T-Rphase: Let $E_{\text{F}} : \mathcal{H}(T) \to \mathcal{H}(\Delta) \oplus \mathcal{H}_{\text{F}}$ be the norm-preserving operator constructed from $[\![\varnothing \parallel \Delta \vdash e : T]\!]^\dagger$ using Lemma K.8. The compiled T-Rphase circuit then looks like this:



Using the fact that $[\![\varnothing \parallel \Delta \vdash e : T]\!]^\dagger = [\![\texttt{left}_{\Delta\oplus G}]\!]^\dagger E_F$,

$$
\begin{aligned}
& E_F^\dagger \left(e^{ir} \mathbb{I}_\Delta \oplus e^{ir'} \mathbb{I}_F \right) E_F \\
= \; & E_F^\dagger \left(e^{ir} [\![\texttt{left}_{\Delta\oplus G}]\!][\![\texttt{left}_{\Delta\oplus G}]\!]^\dagger + e^{ir'} [\![\texttt{right}_{\Delta\oplus G}]\!][\![\texttt{right}_{\Delta\oplus G}]\!]^\dagger \right) E_F \\
= \; & E_F^\dagger \left(e^{ir} [\![\texttt{left}_{\Delta\oplus G}]\!][\![\texttt{left}_{\Delta\oplus G}]\!]^\dagger + e^{ir'} (\mathbb{I} - [\![\texttt{left}_{\Delta\oplus G}]\!][\![\texttt{left}_{\Delta\oplus G}]\!]^\dagger) \right) E_F \\
= \; & e^{ir} E_F^\dagger [\![\texttt{left}_{\Delta\oplus G}]\!][\![\texttt{left}_{\Delta\oplus G}]\!]^\dagger E_F + e^{ir'} \left(\mathbb{I} - E_F^\dagger [\![\texttt{left}_{\Delta\oplus G}]\!][\![\texttt{left}_{\Delta\oplus G}]\!]^\dagger E_F \right) \\
= \; & e^{ir} [\![\varnothing \parallel \Delta \vdash e : T]\!][\![\varnothing \parallel \Delta \vdash e : T]\!]^\dagger + e^{ir'} \left(\mathbb{I} - [\![\varnothing \parallel \Delta \vdash e : T]\!][\![\varnothing \parallel \Delta \vdash e : T]\!]^\dagger \right)
\end{aligned}
$$

T-Pmatch:

This is explained in Section 7.2.

T-Channel: $\mathcal{H}(T) \longrightarrow \boxed{f} \longrightarrow \mathcal{H}(T')$

$$
\begin{aligned}
& |v\rangle\langle v'| && \in \mathcal{L}(\mathcal{H}(T)) \\
\mapsto \; & [\![f]\!]|v\rangle\langle v'|[\![f]\!]^\dagger && \in \mathcal{L}(\mathcal{H}(T'))
\end{aligned}
$$

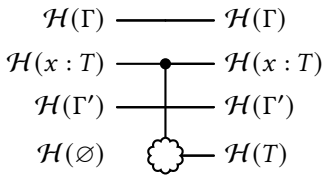
T-MixedAbs: $\mathcal{H}(T) \longrightarrow \boxed{e^\dagger} \xrightarrow{\mathcal{H}(\Delta)} \boxed{e'} \longrightarrow \mathcal{H}(T')$

$$
\begin{aligned}
& |v\rangle\langle v'| && \in \mathcal{L}(\mathcal{H}(T)) \\
\mapsto \; & [\![e]\!]^\dagger |v\rangle\langle v'| [\![e]\!] && \in \mathcal{L}(\mathcal{H}(\Delta)) \\
\mapsto \; & [\![e']\!] \left([\![e]\!]^\dagger |v\rangle\langle v'| [\![e]\!] \right) && \in \mathcal{L}(\mathcal{H}(T'))
\end{aligned}
$$

$$
\begin{array}{rl}
& \mathcal{H}(\Gamma) \longrightarrow \mathcal{H}(\Gamma) \\
\text{T-Unit:} & \mathcal{H}(\varnothing) \longrightarrow \mathcal{H}(\texttt{Unit})
\end{array}
$$

$$
\begin{aligned}
& |\sigma, \varnothing\rangle && \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(\varnothing) \\
= \; & |\sigma, ()\rangle && \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(\texttt{Unit})
\end{aligned}
$$

T-Cvar:

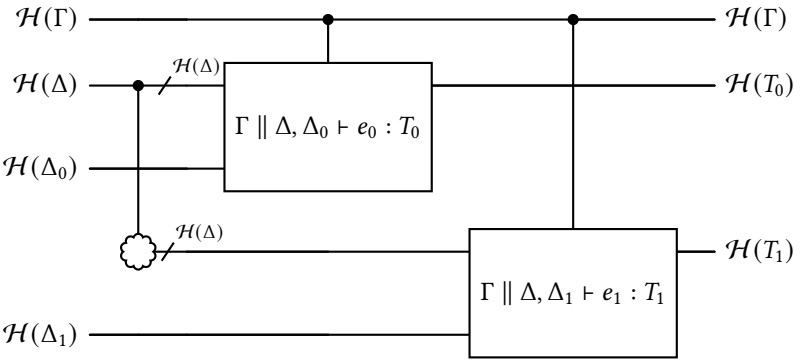


$$
\begin{aligned}
& |\sigma, x \mapsto v, \sigma'\rangle && \in \mathcal{H}(\Gamma, x : T, \Gamma') \\
\mapsto \; & |\sigma, x \mapsto v, \sigma', v\rangle && \in \mathcal{H}(\Gamma, x : T, \Gamma') \otimes \mathcal{H}(T)
\end{aligned}
$$

T-QVAR:
$$\frac{\mathcal{H}(\Gamma) \longrightarrow \mathcal{H}(\Gamma)}{\mathcal{H}(x : T) \longrightarrow \mathcal{H}(T)}$$

$$\begin{aligned} & |\sigma, x \mapsto v\rangle \in \mathcal{H}(\Gamma, x : T) \\ = \; & |\sigma, v\rangle \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(T) \end{aligned}$$

T-PUREPAIR:



$$\begin{aligned}
& |\sigma, \tau, \tau_0, \tau_1\rangle && \in \mathcal{H}(\Gamma, \Delta, \Delta_0, \Delta_1) \\
\mapsto \; & |\sigma, \tau, \tau_0, \tau, \tau_1\rangle && \in \mathcal{H}(\Gamma, \Delta, \Delta_0, \Delta, \Delta_1) \\
\mapsto \; & |\sigma\rangle \otimes [\![\sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0]\!] \, |\tau, \tau_0\rangle \otimes |\tau, \tau_1\rangle && \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(T_0) \otimes \mathcal{H}(\Delta, \Delta_1) \\
\mapsto \; & |\sigma\rangle \otimes [\![\sigma : \Gamma \parallel \Delta, \Delta_0 \vdash e_0 : T_0]\!] \, |\tau, \tau_0\rangle \otimes [\![\sigma : \Gamma \parallel \Delta, \Delta_1 \vdash e_1 : T_1]\!] \, |\tau, \tau_1\rangle && \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(T_0 \otimes T_1)
\end{aligned}$$

T-CTRL:

In the special case where $n = 0$, the semantics corresponds to $0 \in \mathcal{L}(\mathcal{H}(\Delta, \Delta')) \to \mathcal{H}(\texttt{Void})$, so this may be implemented by a circuit that sends all input qubits to the flag register.

Now, consider $n > 0$. The compiled circuit for T-CTRL uses modified versions of its subcircuits. We use Lemma K.10 to get a purified version of the circuit for e with semantics $[\![e]\!] : \mathcal{H}(\Gamma, \Delta) \to \mathcal{H}(T) \otimes \mathcal{H}_G$. Here, $\mathcal{H}_G$ is some "garbage" Hilbert space containing vectors

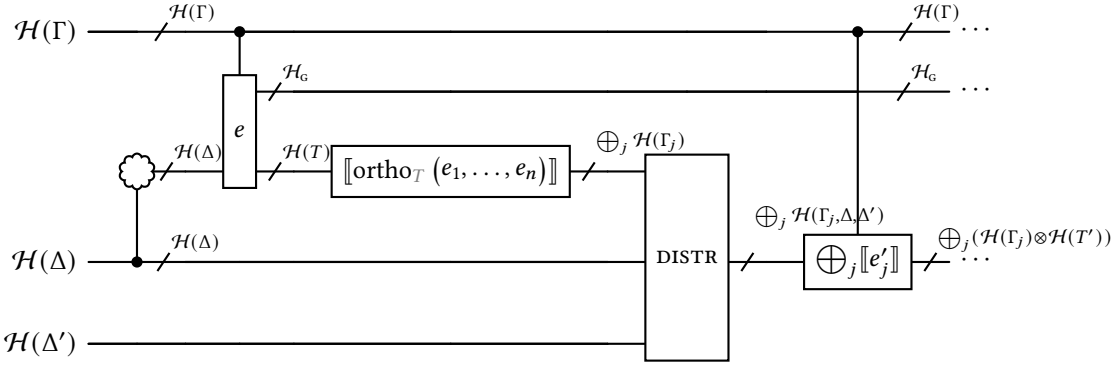
$$\left\{ \left| g_{\sigma,\tau,v} \right\rangle : \sigma \in \mathbb{V}(\Gamma), \tau \in \mathbb{V}(\Delta), v \in \mathbb{V}(T) \right\}$$

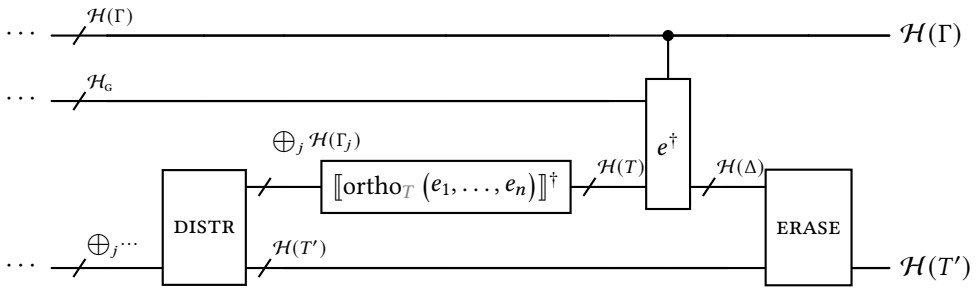
such that

$$[\![e]\!] \, |\sigma, \tau\rangle = \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v}, v \right\rangle$$

for all σ, τ, v.

The circuit below is too large to fit on a single page, so the dots denote where the two pieces must fit together. All direct sums below are to be understood as being taken over the tree $\mathcal{R}$ associated with the orthogonality judgment (Definition 7.1). We use the orthogonality circuit from Appendix L.2.


$\mathcal{H}(\Gamma)$
$\mathcal{H}(\Gamma)$
$\mathcal{H}(\Gamma)$
$\mathcal{H}_G$
$\mathcal{H}_G$
e
$\mathcal{H}(\Delta)$
$\mathcal{H}(T)$
$[\![\mathrm{ortho}_T\,(e_1,\ldots,e_n)]\!]$
$\bigoplus_j \mathcal{H}(\Gamma_j)$
$\mathcal{H}(\Delta)$
$\mathcal{H}(\Delta)$
DISTR
$\bigoplus_j \mathcal{H}(\Gamma_j,\Delta,\Delta')$
$\bigoplus_j [\![e'_j]\!]$
$\bigoplus_j (\mathcal{H}(\Gamma_j)\otimes\mathcal{H}(T'))$
$\mathcal{H}(\Delta')$

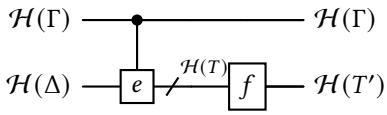

$\mathcal{H}(\Gamma)$
$\mathcal{H}(\Gamma)$
$\mathcal{H}_G$
$e^\dagger$
$\bigoplus_j \mathcal{H}(\Gamma_j)$
$[\![\mathrm{ortho}_T\,(e_1,\ldots,e_n)]\!]^\dagger$
$\mathcal{H}(T)$
$\mathcal{H}(\Delta)$
DISTR
ERASE
$\bigoplus_j \cdots$
$\mathcal{H}(T')$
$\mathcal{H}(T')$

$$
\begin{aligned}
& |\sigma, \tau, \tau'\rangle \\
& \in \mathcal{H}(\Gamma, \Delta, \Delta') \\
\mapsto\ & |\sigma, \tau, \tau, \tau'\rangle \\
& \in \mathcal{H}(\Gamma, \Delta, \Delta, \Delta') \\
\mapsto\ & |\sigma\rangle \otimes [\![e]\!] |\sigma, \tau\rangle \otimes |\tau, \tau'\rangle \\
=\ & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v}, v \right\rangle \otimes |\tau, \tau'\rangle \\
& \in \mathcal{H}(\Gamma) \otimes \mathcal{H}_{\mathrm{G}} \otimes \mathcal{H}(T) \otimes \mathcal{H}(\Delta, \Delta') \\
\mapsto\ & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \otimes \left(\bigoplus_{j:\mathcal{R}} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger |v\rangle \cdot [\![e_j]\!]^\dagger |v\rangle \right) \otimes |\tau, \tau'\rangle \\
=\ & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \otimes \left(\bigoplus_{j:\mathcal{R}} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger |v\rangle \cdot \left\langle \sigma_j \right| [\![e_j]\!]^\dagger |v\rangle \cdot \left| \sigma_j \right\rangle \right) \otimes |\tau, \tau'\rangle \\
=\ & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \otimes \left(\bigoplus_{j:\mathcal{R}} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger |v\rangle \cdot \left| \sigma_j \right\rangle \right) \otimes |\tau, \tau'\rangle \\
& \in \mathcal{H}(\Gamma) \otimes \mathcal{H}_{\mathrm{G}} \otimes \bigoplus_{j:\mathcal{R}} \mathcal{H}(\Gamma_j) \otimes \mathcal{H}(\Delta, \Delta') \\
\mapsto\ & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \otimes \left(\bigoplus_{j:\mathcal{R}} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger |v\rangle \cdot \left| \sigma_j, \tau, \tau' \right\rangle \right) \\
& \in \mathcal{H}(\Gamma) \otimes \mathcal{H}_{\mathrm{G}} \otimes \bigoplus_{j:\mathcal{R}} \left(\mathcal{H}(\Gamma_j, \Delta, \Delta') \right) \\
\mapsto\ & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \otimes \left(\bigoplus_{j:\mathcal{R}} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger |v\rangle \cdot [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \right)
\end{aligned}
$$

$$
\begin{aligned}
& |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \\
& \qquad \otimes \left(\bigoplus_{j:\mathcal{R}} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot \left| \sigma_j, v' \right\rangle \right) \\
= \; & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot \mathrm{inj}_j \left| \sigma_j, v' \right\rangle \right) \\
& \quad \in \mathcal{H}(\Gamma) \otimes \mathcal{H}_{\mathrm{G}} \otimes \bigoplus_{j:\mathcal{R}} \left(\mathcal{H}(\Gamma_j) \otimes \mathcal{H}(T') \right) \\
\mapsto \; & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot \mathrm{inj}_j^{\mathcal{R}} \left| \sigma_j \right\rangle \otimes \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot |v'\rangle \right) \\
& \quad \in \mathcal{H}(\Gamma) \otimes \mathcal{H}_{\mathrm{G}} \otimes \bigoplus_{j:\mathcal{R}} \mathcal{H}(\Gamma_j) \otimes \mathcal{H}(T') \\
\mapsto \; & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \cdot \left| g_{\sigma,\tau,v} \right\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot |v\rangle \otimes \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot |v'\rangle \right) \\
& \quad \in \mathcal{H}(\Gamma) \otimes \mathcal{H}_{\mathrm{G}} \otimes \mathcal{H}(T) \otimes \mathcal{H}(T') \\
\mapsto \; & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot [\![e]\!]^\dagger \left| g_{\sigma,\tau,v}, v \right\rangle \otimes \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot |v'\rangle \right) \\
& \quad \in \mathcal{H}(\Gamma, \Delta) \otimes \mathcal{H}(T')
\end{aligned}
$$

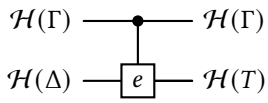
$$
\begin{aligned}
& |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot \sum_{\tau_\star} \langle \sigma, \tau_\star | \, [\![e]\!]^\dagger \left| g_{\sigma,\tau,v}, v \right\rangle \cdot |\tau_\star\rangle \otimes \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot |v'\rangle \right) \\
& \in \mathcal{H}(\Gamma, \Delta) \otimes \mathcal{H}(T') \\
= \; & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \, \langle \sigma, \tau | \, [\![e]\!]^\dagger \left| g_{\sigma,\tau,v}, v \right\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \cdot |\tau\rangle \otimes \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot |v'\rangle \right) \\
& \in \mathcal{H}(\Gamma, \Delta) \otimes \mathcal{H}(T') \\
\mapsto \; & |\sigma\rangle \otimes \sum_{v \in \mathbb{V}(T)} \left\langle g_{\sigma,\tau,v}, v \right| [\![e]\!] \, |\sigma, \tau\rangle \, \langle \sigma, \tau | \, [\![e]\!]^\dagger \left| g_{\sigma,\tau,v}, v \right\rangle \\
& \qquad \otimes \left(\sum_{j=1}^{n} \sum_{\sigma_j \in \mathbb{V}(\Gamma_j)} \left\langle \sigma_j \right| [\![e_j]\!]^\dagger \, |v\rangle \otimes \sum_{v' \in \mathbb{V}(T')} \left\langle \sigma_j, v' \right| [\![e'_j]\!] \left| \sigma_j, \tau, \tau' \right\rangle \cdot |v'\rangle \right) \\
& \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(T')
\end{aligned}
$$

T-PureApp:

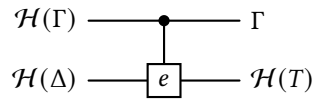


$$
\begin{aligned}
& |\sigma, \tau\rangle \in \mathcal{H}(\Gamma, \Delta) \\
\mapsto \; & |\sigma\rangle \otimes [\![e]\!] \, |\tau\rangle \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(T) \\
\mapsto \; & |\sigma\rangle \otimes [\![f]\!][\![e]\!] \, |\tau\rangle \in \mathcal{H}(\Gamma) \otimes \mathcal{H}(T')
\end{aligned}
$$

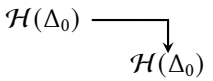
T-Mix:



$$
\begin{aligned}
& |\sigma, \tau\rangle\langle\sigma, \tau'| \in \mathcal{L}(\mathcal{H}(\Gamma, \Delta)) \\
\mapsto \; & |\sigma\rangle\langle\sigma| \otimes [\![e]\!] |\tau\rangle\langle\tau'| [\![e]\!]^\dagger \in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \mathcal{L}(\mathcal{H}(T))
\end{aligned}
$$

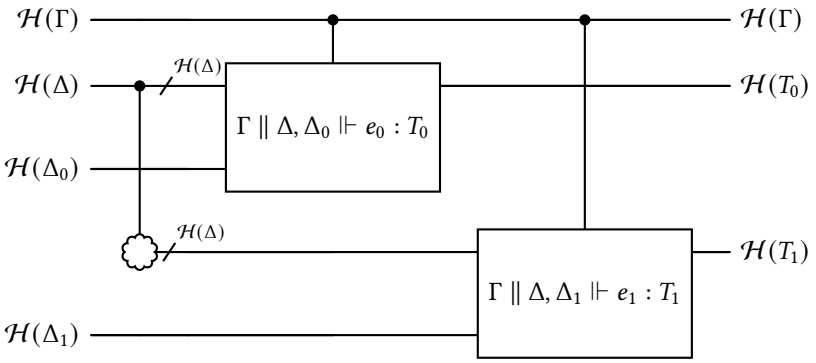


T-Discard:



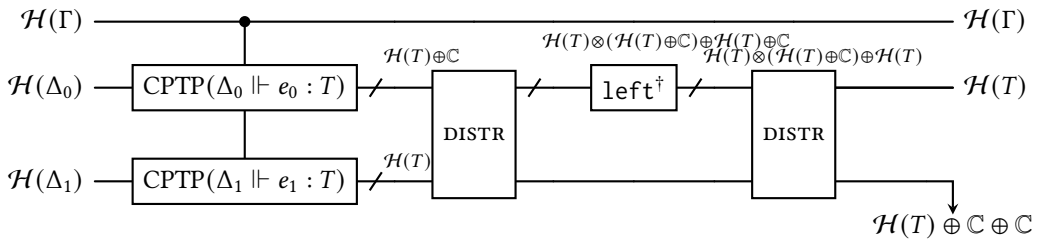
$$|\sigma, \tau, \tau_0\rangle\langle\sigma, \tau', \tau_0'| \in \mathcal{L}(\mathcal{H}(\Gamma, \Delta, \Delta_0))$$
$$\mapsto |\sigma\rangle\langle\sigma| \otimes \mathrm{tr}(|\tau_0\rangle\langle\tau_0'|)[\![e]\!](|\tau\rangle\langle\tau'|) \in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \mathcal{L}(\mathcal{H}(T))$$

T-MixedPair:



$$\begin{aligned}
&|\sigma, \tau, \tau_0, \tau_1\rangle\langle\sigma, \tau', \tau_0', \tau_1'| && \in \mathcal{L}(\mathcal{H}(\Gamma, \Delta, \Delta_0, \Delta_1)) \\
\mapsto\ &|\sigma, \tau, \tau_0, \tau, \tau_1\rangle\langle\sigma, \tau', \tau_0', \tau', \tau_1'| \\
=\ &|\sigma\rangle\langle\sigma| \otimes |\tau, \tau_0\rangle\langle\tau', \tau_0'| \otimes |\tau, \tau_1\rangle\langle\tau', \tau_1'| && \in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \mathcal{L}(\mathcal{H}(\Delta, \Delta_0)) \otimes \mathcal{L}(\mathcal{H}(\Delta, \Delta_1)) \\
\mapsto\ &|\sigma\rangle\langle\sigma| \otimes [\![e_0]\!]\left(|\tau, \tau_0\rangle\langle\tau', \tau_0'|\right) \otimes [\![e_1]\!]\left(|\tau, \tau_1\rangle\langle\tau', \tau_1'|\right) && \in \mathcal{L}(\mathcal{H}(\Gamma) \otimes \mathcal{H}(T_0 \otimes T_1))
\end{aligned}$$

T-Try:



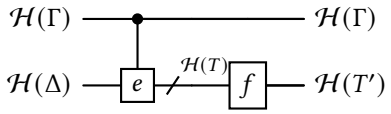
For brevity, define:

$$\rho_0' \coloneqq [\![\sigma : \Gamma \parallel \Delta_0 \Vdash e_0 : T]\!](\rho_0)$$
$$\rho_1' \coloneqq [\![\sigma : \Gamma \parallel \Delta_1 \Vdash e_1 : T]\!](\rho_1)$$

Then, the circuit acts as follows:

$$
\begin{aligned}
&\rho_0 \otimes \rho_1 \\
\mapsto\ &\left(\rho_0' \oplus (\operatorname{tr}(\rho_0) - \operatorname{tr}(\rho_0'))\right) \otimes \left(\rho_1' \oplus (\operatorname{tr}(\rho_1) - \operatorname{tr}(\rho_1'))\right) \\
\mapsto\ &\rho_0' \otimes \left(\rho_1' \oplus (\operatorname{tr}(\rho_1) - \operatorname{tr}(\rho_1'))\right) \oplus (\operatorname{tr}(\rho_0) - \operatorname{tr}(\rho_0'))\left(\rho_1' \oplus (\operatorname{tr}(\rho_1) - \operatorname{tr}(\rho_1'))\right) \\
\mapsto\ &\rho_0' \otimes \left(\rho_1' \oplus (\operatorname{tr}(\rho_1) - \operatorname{tr}(\rho_1'))\right) \oplus (\operatorname{tr}(\rho_0) - \operatorname{tr}(\rho_0'))\rho_1' \\
\mapsto\ &\rho_0' \otimes \left(\rho_1' \oplus (\operatorname{tr}(\rho_1) - \operatorname{tr}(\rho_1')) \oplus 0\right) + \rho_1'\left(0 \oplus 0 \oplus (\operatorname{tr}(\rho_0) - \operatorname{tr}(\rho_0'))\right) \\
\mapsto\ &\operatorname{tr}(\rho_1)\,\rho_0' + (\operatorname{tr}(\rho_0) - \operatorname{tr}(\rho_0'))\rho_1'
\end{aligned}
$$

T-MixedApp:



$$
\begin{aligned}
&|\sigma, \tau\rangle\langle\sigma, \tau'| \in \mathcal{L}(\mathcal{H}(\Gamma, \Delta)) \\
\mapsto\ &|\sigma\rangle\langle\sigma| \otimes [\![e]\!]\,(|\tau\rangle\langle\tau'|) \in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \mathcal{L}(\mathcal{H}(T)) \\
\mapsto\ &|\sigma\rangle\langle\sigma| \otimes [\![f]\!]\,([\![e]\!](|\tau\rangle\langle\tau'|)) \in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \mathcal{L}(\mathcal{H}(T'))
\end{aligned}
$$

T-Match:

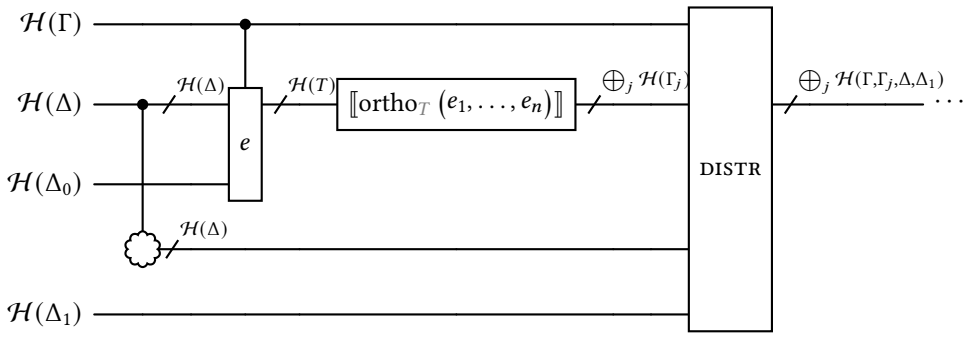
This construction is similar to that of T-Ctrl, but it does not perform uncomputation and instead discards quantum data. We use purified versions of the circuits for the e_j', obtaining $\textsc{purify}(e_j') : \mathcal{H}(\Gamma, \Gamma_j, \Delta, \Delta_1) \to \mathcal{H}(\Gamma, \Gamma_j) \otimes \mathcal{H}(T') \otimes \mathcal{H}_j$. We will still write it as $[\![e_j']\!]$ where it is clear from context we are referring to the purification. Here, $\mathcal{H}_j$ is a "garbage Hilbert space" containing vectors

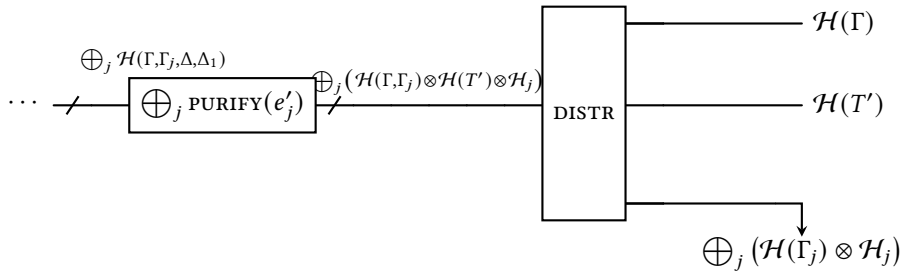
$$\left\{\left|g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\right\rangle : \sigma \in \mathbb{V}(\Gamma), \sigma_j \in \mathbb{V}(\Gamma_j), \tau \in \mathbb{V}(\Delta), \tau_1 \in \mathbb{V}(\Delta_1), v' \in \mathbb{V}(T')\right\},$$

such that

$$[\![e_j']\!]\left|\sigma, \sigma_j, \tau, \tau_1\right\rangle = \sum_{v' \in \mathbb{V}(T')} \left\langle v', g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\right| [\![e_j']\!]\left|\sigma, \sigma_j, \tau, \tau_1\right\rangle \cdot \left|v', g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\right\rangle.$$

Note that unlike in T-Ctrl, we do *not* purify e. As for T-Ctrl, all direct sums are to be understood as being taken over the tree $\mathcal{R}$ associated with the orthogonality judgment.





$$
\begin{aligned}
&|\sigma,\tau,\tau_0,\tau_1\rangle\langle\sigma,\tau',\tau'_0,\tau'_1| \\
&\quad\in \mathcal{L}(\mathcal{H}(\Gamma,\Delta,\Delta_0,\Delta_1)) \\
\mapsto\ &|\sigma,\tau,\tau_0,\tau,\tau_1\rangle\langle\sigma,\tau',\tau'_0,\tau',\tau'_1| \\
&\quad\in \mathcal{L}(\mathcal{H}(\Gamma,\Delta,\Delta_0,\Delta,\Delta_1)) \\
\mapsto\ &|\sigma\rangle\langle\sigma| \otimes [\![e]\!]\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau'_0|\right) \otimes |\tau_0,\tau,\tau_1\rangle\langle\tau'_0,\tau',\tau'_1| \\
=\ &|\sigma\rangle\langle\sigma| \otimes \sum_{v,w\in\mathbb{V}(T)} \langle v|\left([\![e]\!]\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau'_0|\right)\right)|w\rangle \cdot |v\rangle\langle w| \otimes |\tau,\tau_1\rangle\langle\tau',\tau'_1| \\
&\quad\in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \mathcal{L}(\mathcal{H}(T)) \otimes \mathcal{L}(\mathcal{H}(T)) \otimes \mathcal{L}(\mathcal{H}(\Delta,\Delta_1)) \\
\mapsto\ &|\sigma\rangle\langle\sigma| \otimes \sum_{v,w\in\mathbb{V}(T)} \langle v|\left([\![e]\!]\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau'_0|\right)\right)|w\rangle \cdot \\
&\qquad\left(\sum_{j,k}\sum_{\substack{\sigma_j\in\mathbb{V}(\Gamma_j)\\ \sigma_k\in\mathbb{V}(\Gamma_k)}} \left\langle\sigma_j\right| [\![e_j]\!]^\dagger |v\rangle \langle w| [\![e_j]\!] \left|\sigma_k\right\rangle \cdot \text{inj}_j^{\mathcal{R}} [\![e_j]\!]^\dagger |v\rangle \langle w| [\![e_k]\!] \text{inj}_k^{\mathcal{R}\dagger}\right) \otimes |\tau,\tau_1\rangle\langle\tau',\tau'_1| \\
=\ &|\sigma\rangle\langle\sigma| \otimes \sum_{v,w\in\mathbb{V}(T)} \langle v|\left([\![e]\!]\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau'_0|\right)\right)|w\rangle \cdot \\
&\qquad\left(\sum_{j,k}\sum_{\substack{\sigma_j\in\mathbb{V}(\Gamma_j)\\ \sigma_k\in\mathbb{V}(\Gamma_k)}} \left\langle\sigma_j\right| [\![e_j]\!]^\dagger |v\rangle \langle w| [\![e_j]\!] \left|\sigma_k\right\rangle \cdot \text{inj}_j^{\mathcal{R}} |\sigma_j\rangle\langle\sigma_k| \text{inj}_k^{\mathcal{R}\dagger}\right) \otimes |\tau,\tau_1\rangle\langle\tau',\tau'_1| \\
&\quad\in \mathcal{L}(\mathcal{H}(\Gamma)) \otimes \bigoplus_{j:\mathcal{R}} \mathcal{L}(\mathcal{H}(\Gamma_j)) \otimes \mathcal{L}(\mathcal{H}(\Delta,\Delta_1)) \\
\mapsto\ &\sum_{v,w\in\mathbb{V}(T)} \langle v|\left([\![e]\!]\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau'_0|\right)\right)|w\rangle \cdot \\
&\qquad\left(\sum_{j,k}\sum_{\substack{\sigma_j\in\mathbb{V}(\Gamma_j)\\ \sigma_k\in\mathbb{V}(\Gamma_k)}} \left\langle\sigma_j\right| [\![e_j]\!]^\dagger |v\rangle \langle w| [\![e_j]\!] \left|\sigma_k\right\rangle \cdot \text{inj}_j^{\mathcal{R}} |\sigma,\sigma_j,\tau,\tau_1\rangle\langle\sigma,\sigma_k,\tau',\tau'_1| \text{inj}_k^{\mathcal{R}\dagger}\right) \\
&\quad\in \bigoplus_{j:\mathcal{R}} \mathcal{L}(\mathcal{H}(\Gamma,\Gamma_j,\Delta,\Delta_1))
\end{aligned}
$$

$$\begin{aligned}
&\mapsto \sum_{v,w\in\mathbb{V}(T)} \sum_{v',w'\in\mathbb{V}(T')} \langle v|\left(\llbracket e\rrbracket\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau_0'|\right)\right)|w\rangle \cdot \\
&\qquad \cdot \sum_{j,k} \sum_{\substack{\sigma_j\in\mathbb{V}(\Gamma_j)\\ \sigma_k\in\mathbb{V}(\Gamma_k)}} \left\langle\sigma_j\right|\llbracket e_j\rrbracket^\dagger|v\rangle\langle w|\llbracket e_j\rrbracket|\sigma_k\rangle \cdot \left\langle v', g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\right|\llbracket e_j'\rrbracket\left|\sigma,\sigma_j,\tau,\tau_1\right\rangle \cdot \\
&\qquad \cdot \left\langle\sigma,\sigma_k,\tau',\tau_1'\right|\llbracket e_k'\rrbracket^\dagger\left|w', g_{k,\sigma,\sigma_k,\tau',\tau_1',w'}\right\rangle \cdot \mathrm{inj}_j^{\mathcal{R}}|\sigma,\sigma_j,v',g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\rangle\langle\sigma,\sigma_k,w',g_{k,\sigma,\sigma_k,\tau',\tau_1',w'}|\,\mathrm{inj}_k^{\mathcal{R}\dagger} \\
&\quad \in \bigoplus_{j:\mathcal{R}} \mathcal{L}(\mathcal{H}(\Gamma,\Gamma_j)\otimes\mathcal{H}(T')\otimes\mathcal{H}_j) \\
&\mapsto |\sigma\rangle\langle\sigma|\otimes \sum_{v,w\in\mathbb{V}(T)} \sum_{v',w'\in\mathbb{V}(T')} \langle v|\left(\llbracket e\rrbracket\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau_0'|\right)\right)|w\rangle \cdot \\
&\qquad \cdot \sum_{j,k} \sum_{\substack{\sigma_j\in\mathbb{V}(\Gamma_j)\\ \sigma_k\in\mathbb{V}(\Gamma_k)}} \left\langle\sigma_j\right|\llbracket e_j\rrbracket^\dagger|v\rangle\langle w|\llbracket e_j\rrbracket|\sigma_k\rangle \cdot \left\langle v', g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\right|\llbracket e_j'\rrbracket\left|\sigma,\sigma_j,\tau,\tau_1\right\rangle \cdot \\
&\qquad \cdot \left\langle\sigma,\sigma_k,\tau',\tau_1'\right|\llbracket e_k'\rrbracket^\dagger\left|w', g_{k,\sigma,\sigma_k,\tau',\tau_1',w'}\right\rangle \cdot |v'\rangle\langle w'|\otimes \mathrm{inj}_j^{\mathcal{R}}|\sigma_j,g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\rangle\langle\sigma_k,g_{k,\sigma,\sigma_k,\tau',\tau_1',w'}|\,\mathrm{inj}_k^{\mathcal{R}\dagger} \\
&\quad \in \mathcal{L}(\mathcal{H}(\Gamma))\otimes\mathcal{L}(\mathcal{H}(T'))\otimes\bigoplus_{j:\mathcal{R}}\mathcal{L}(\mathcal{H}(\Gamma_j)\otimes\mathcal{H}_j) \\
&\mapsto |\sigma\rangle\langle\sigma|\otimes \sum_{v,w\in\mathbb{V}(T)} \sum_{v',w'\in\mathbb{V}(T')} \langle v|\left(\llbracket e\rrbracket\left(|\sigma,\tau,\tau_0\rangle\langle\sigma,\tau',\tau_0'|\right)\right)|w\rangle \cdot \\
&\qquad \cdot \sum_{j=1}^{n} \sum_{\sigma_j\in\mathbb{V}(\Gamma_j)} \left\langle\sigma_j\right|\llbracket e_j\rrbracket^\dagger|v\rangle\langle w|\llbracket e_j\rrbracket\left|\sigma_j\right\rangle \cdot \left\langle v', g_{j,\sigma,\sigma_j,\tau,\tau_1,v'}\right|\llbracket e_j'\rrbracket\left|\sigma,\sigma_j,\tau,\tau_1\right\rangle \cdot \\
&\qquad \cdot \left\langle\sigma,\sigma_j,\tau',\tau_1'\right|\llbracket e_j'\rrbracket^\dagger\left|w', g_{j,\sigma,\sigma_j,\tau',\tau_1',w'}\right\rangle \cdot |v'\rangle\langle w'| \\
&= |\sigma\rangle\langle\sigma|\otimes \sum_{v\in\mathbb{V}(T)} \langle v|\left(\llbracket\sigma:\Gamma \parallel \Delta,\Delta_0 \Vdash e:T\rrbracket\left(|\tau,\tau_0\rangle\left\langle\tau',\tau_0'\right|\right)\right)|v\rangle \cdot \sum_{j=1}^{n}\sum_{\sigma_j\in\mathbb{V}(\Gamma_j)} \left\langle\sigma_j\right|\llbracket\varnothing:\varnothing \parallel \Gamma_j\vdash e_j:T\rrbracket^\dagger|v\rangle \cdot \\
&\qquad \cdot \llbracket\sigma,\sigma_j:\Gamma,\Gamma_j \parallel \Delta,\Delta_1\Vdash e_j'\rrbracket\left(|\tau,\tau_1\rangle\langle\tau',\tau_1'|\right) \\
&\quad \in \mathcal{L}(\mathcal{H}(\Gamma))\otimes\mathcal{L}(\mathcal{H}(T'))
\end{aligned}$$