Proto-Quipper with Reversing and Control

Peng Fu[1], Kohei Kishida[2], Neil J. Ross[3], and Peter Selinger[3]

[1] University of South Carolina, USA
[2] University of Illinois Urbana-Champaign, USA
[3] Dalhousie University, Canada

Abstract. The quantum programming language Quipper supports circuit operations such as reversing and control, which allows programmers to control and reverse certain quantum circuits. In addition to these two operations, Quipper provides a function called *with-computed*, which can be used to program circuits of the form $g; f; g^\dagger$. The latter is a common pattern in quantum circuit design. One benefit of using with-computed, as opposed to constructing the circuit $g; f; g^\dagger$ directly from g, f, and $g^\dagger$, is that it facilitates an important optimization. Namely, if the resulting circuit is later controlled, only the circuit f in the middle needs to be controlled; the circuits g and $g^\dagger$ need not even be controllable.
In this paper, we formalize a semantics for reversible and controllable circuits, using a dagger symmetric monoidal category $\mathbf{R}$ to interpret reversible circuits, and a new notion we call a *controllable category* $\mathbf{N}$ to interpret controllable circuits. The controllable category $\mathbf{N}$ encompasses the control and with-computed operations in Quipper. We extend the language Proto-Quipper with reversing, control and the with-computed operation. Since not all circuits are reversible and/or controllable, we use a type system with modalities to track reversibility and controllability. This generalizes the modality of Fu-Kishida-Ross-Selinger 2023. We give an abstract categorical semantics for reversing, control and with-computed, and show that the type system and operational semantics are sound with respect to this semantics. Lastly, we construct a concrete model using a generalization of *biset enrichment* from Fu-Kishida-Ross-Selinger 2022.

Keywords: Proto-Quipper· Quipper· Control· Reversing· With-computed· Quantum Circuits· Categorical Semantics.

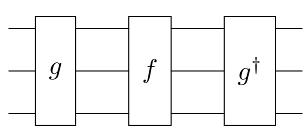
1 Introduction

The goal of this paper is to devise a strongly typed, functional programming language that can formally treat the quantum operations of reversing and control. Many quantum algorithms take essential advantage of reversing, control, and their associated operation "with-computed". At the same time, some essential operations, such as state preparations and measurements, can make circuits irreversible or uncontrollable. It is therefore desirable to equip quantum programming languages with a type system that can prevent the user from trying to reverse the irreversible or to control the uncontrollable. Some recently proposed quantum programming languages (e.g., Qiskit and Cirq) also include the concepts of reversibility and controllability. However, since these languages are imperative, errors of reversibility or controllability are treated as run-time exceptions (see [11,15]). As to the with-computed operation, we are only aware of one language besides Quipper that has a similar operation, namely ProjectQ [14,26]. However, ProjectQ is not a strongly-typed language. In this paper, we incorporate reversing, control, and with-computed into the language *Proto-Quipper*.

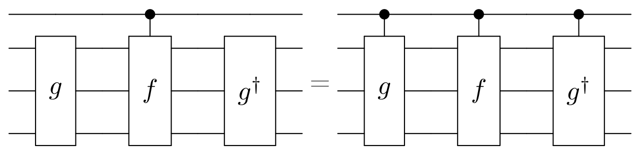
Proto-Quipper is a family of programming languages that aims to provide a formal syntax and semantics for various features of the embedded quantum programming language Quipper [12,13]. Like Quipper, Proto-Quipper is a quantum circuit description language. When a Proto-Quipper program is run, it outputs a quantum circuit; this generated circuit can later be executed on a quantum computer. We refer to these two runtimes as "circuit generation time" and "circuit execution time", respectively. There are various errors that programmers can introduce by attempting the impossible—e.g., to duplicate linear data, to apply circuit operations to a program that is not a circuit, etc.—but since Proto-Quipper is strongly typed, it can catch such errors at compile time, rather than at circuit generation time or circuit execution time. Previous research on Proto-Quipper includes providing a syntax and semantics for circuit boxing [22,23], supporting recursion [18], combining linear types and dependent types [7,10], and incorporating dynamic lifting [4,8,9,17]. This paper defines a new variant Proto-Quipper-C, which extends the type system of Proto-Quipper with reversibility and controllability.

1.1 Reversing, control, and with-computed

In this paper, we address the issue of how to formally treat reversing, control, and the with-computed operations by extending Proto-Quipper. In Quipper, the `controlled` function takes a Quipper circuit as input, and returns the controlled version of the circuit. The `reverse` function takes the adjoint of an input circuit. The program (`with_computed` $g\,f$) first performs a circuit g, then a circuit f, and finally the adjoint of g.



The *with-computed* circuit pattern is very common in quantum computing. For example, Bennett's method for constructing reversible classical circuits [1] uses this pattern to initialize ancillary qubits and uncompute them; and the quantum Fourier transform (QFT) addition [6] conjugates a series of controlled rotation gates by the QFT. This pattern is used so often that Quipper implements the following optimization: when controlling a quantum circuit $g; f; g^\dagger$ generated by the with-computed function, it suffices to control the circuit f.



In fact, the circuits g and $g^\dagger$ need not even be controllable. As long as the circuit f is controllable, we are able to control the circuit $g; f; g^\dagger$ by directly controlling f. Thus the left hand side of the above circuit identity is more general to the right hand side. It is also more efficient, since it uses a smaller number of controlled gates than the circuit on the right. Note that in quantum computing, a controlled gate may require substantially more resources than the non-controlled version. For example, a common way of measuring resources is the *T-count*, i.e., the number of T-gates required to implement a gate. While a T-gate itself has T-count 1, a controlled T-gate has a T-count of at least 5.

1.2 Incorporating reversing, control and with-computed in Proto-Quipper

To incorporate reversing, control, and with-computed in Proto-Quipper, our starting point is to annotate each gate with flags that indicate its reversibility and controllability. So when controlling or reversing a circuit, we can first check if all the gates in the circuit are controllable or reversible. If they are, then we proceed to control or reverse each gate in the circuit. If there is a gate that is not controllable or reversible, then the operation fails.

To handle the with-computed operation, we could define $\mathsf{withComputed}(g, f)$ directly as the composition $g; f; g^\dagger$. But this has the drawback that when the circuit g is not controllable, it would not be possible to control $g; f; g^\dagger$. This prevents a large class of circuits from being controllable, e.g., quantum circuits that are constructed via Bennett's method [1].

Our solution is to view $\mathsf{withComputed}$ as a special programming language construct, rather than as the composition $g; f; g^\dagger$. It satisfies, for example, the following identity (up to equivalence of circuits):

$$\mathsf{control}_S(\mathsf{withComputed}(g, f)) = \mathsf{withComputed}(g \otimes \mathrm{id}_S, \mathsf{control}_S(f)) \quad (1)$$

This identity states that controlling $\mathsf{withComputed}(g, f)$ is the same thing as first controlling f, and then combining it with $g \otimes \mathrm{id}_S$ using the with-computed construct. Here, $g \otimes \mathrm{id}_S$ means stacking identity wires of type S on top of g.

Now let us give a semantic description for $\mathsf{withComputed}$. We assume that we are given a symmetric monoidal category $\mathbf{M}$ of quantum circuits. We further assume that we are given a monoidal subcategory $\mathbf{R}$ of $\mathbf{M}$ such that $\mathbf{R}$ is dagger symmetric monoidal, i.e., if $g \in \mathbf{R}(A, B)$, then $g^\dagger \in \mathbf{R}(B, A)$. So $\mathbf{R}$ represents the category of reversible quantum circuits, where reversing a circuit corresponds to applying the dagger functor $(-)^\dagger : \mathbf{R}(A, B) \to \mathbf{R}(B, A)$. To interpret control and $\mathsf{withComputed}$, we require an additional dagger symmetric monoidal category $\mathbf{N}$

of *controllable circuits*. It has same set of objects as $\mathbf{R}$, and is equipped with two operations, namely $\mathsf{control}_S : \mathbf{N}(A, A) \to \mathbf{N}(A \otimes S, A \otimes S)$ and $\mathsf{withComputed} : \mathbf{R}(A, B) \times \mathbf{N}(B, B) \to \mathbf{N}(A, A)$. The functions $\mathsf{control}_S$ and withComputed must be assumed to satisfy equation (1), as well as several other properties (which we will state in more detail in Section 3). For example, here are two more properties that withComputed satisfies:

$$\mathsf{withComputed}(\mathrm{id}, f) = f \tag{2}$$

$$\mathsf{withComputed}(g_1; g_2, f) = \mathsf{withComputed}(g_1, \mathsf{withComputed}(g_2, f)) \tag{3}$$

We can interpret withComputed in the category $\mathbf{R}$ by an identity-on-objects, dagger symmetric monoidal functor $G : \mathbf{N} \to \mathbf{R}$ such that

$$G(\mathsf{withComputed}(g, f)) = g; G(f); g^\dagger.$$

Because of the functor G, we can regard every controllable circuit as a reversible circuit. However, note that $\mathbf{N}$ is not a subcategory of $\mathbf{R}$, as the functor G is not in general faithful: it is possible for two circuits, such as $\mathsf{withComputed}(\hat{g}, f)$ and $g; f; g^\dagger$, where $G(g) = \hat{g}$, to be different in $\mathbf{N}$ (as they behave differently when controlled), but equal in $\mathbf{R}$.

1.3 Modalities for reversing and control

In Quipper, controlling an uncontrollable circuit or reversing a non-reversible circuit will give rise to runtime errors. We want to incorporate reversing and control in Proto-Quipper in such a way that erroneously controlling or reversing a circuit is detected as a typing error at compile time. We achieve this by introducing a notion of modality $\alpha \in \{0, 1, 2\}$. Here, the modality 2 is associated with circuits that are both controllable and reversible, for example a Hadamard gate. The modality 1 is associated with circuits that are reversible but not controllable, for example an initialization gate. The modality 0 is associated with circuits that are neither controllable nor reversible, for example a measurement gate.

We can use modalities to annotate the type of a circuit. For example, the values of the type $\mathbf{Circ}_\alpha(S, U)$ are circuits with input type S, output type U and a modality annotation α. One important feature of modalities is that they are composable. Suppose we want to compose a circuit $\mathbf{Circ}_\alpha(S, U)$ with a circuit $\mathbf{Circ}_\beta(U, S')$. The resulting circuit will have type $\mathbf{Circ}_{\alpha\wedge\beta}(S, S')$, where $\alpha \wedge \beta = \min(\alpha, \beta)$. This means that as long as we know the modalities for the basic gate sets, we can devise a type system to track the modalities of the circuits constructed from basic gates. Moreover, the reversing, control, and withcomputed operations can be given the following types:

$$\begin{array}{lll}
\mathsf{control}_S & : \mathbf{Circ}_2(U, U) \to \mathbf{Circ}_2(U \otimes S, U \otimes S). & \\
\mathsf{reverse} & : \mathbf{Circ}_\alpha(U, S) \to \mathbf{Circ}_\alpha(S, U), & \text{where } \alpha > 0. \\
\mathsf{withComputed} & : \mathbf{Circ}_\alpha(U, S) \times \mathbf{Circ}_2(S, S) \to \mathbf{Circ}_2(U, U), & \text{where } \alpha > 0.
\end{array}$$

Note that withComputed requires its first argument to be at least reversible, and the resulting circuit is again controllable.

Our modal type system makes it possible to detect errors, like controlling a uncontrollable circuit, at compile time. From the programmer's perspective, it is not necessary to specify modalities explicitly in the source code, as they can be inferred automatically by the type checker.

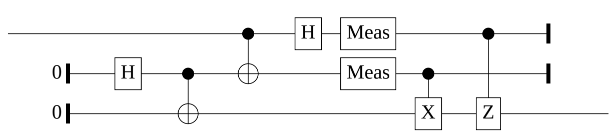
1.4 Detecting reversing and controlling errors with modalities

Consider the following Proto-Quipper program that implements *quantum teleportation* [19].

```
tele : !(Qubit -> Qubit)
tele q =
  let (a, b) = bell00 ()
      (cq, cb) = bellMeas (q, b)
      (a, cb) = C_X a cb
      (a, cq) = C_Z a cq
      _ = Discard cq
      _ = Discard cb
  in a

teleCirc : Circ(Qubit, Qubit)
teleCirc = box Qubit tele
```

The `bell00` function prepares a Bell state and `bellMeas` performs Bell measurement. The gates `C_X` and `C_Z` are classically controlled gates such that the second bit is the classical control bit. The operation `box` turns the circuit generating function `tele` of type `!(Qubit -> Qubit)` into a circuit `teleCirc` of type `Circ(Qubit, Qubit)`, which is the following.



Even though teleportation conceptually implements an identity function, it is not reversible nor controllable, due to the presence of state preparations, measurements, and the discarding of states. So if programmers try to reverse or control `teleCirc`, the type checker reports a typing error. For example, consider the program below.

```
rteleCirc : Circ(Qubit, Qubit)
rteleCirc = reverse teleCirc
```

It gives rise to the following typing error in Proto-Quipper-C.

```
Can't resolve modality.
When checking
  teleCirc,
the type
  Circ{1, 0, 0}(Qubit, Qubit)
indicates that it is not reversible,
but it is expected to be reversible.
```

Here the notation `{1,0,0}` is an encoding of the modality that means the circuit is boxable, but it is not controllable nor reversible. The modalities are not directly visible to the programmer and they only show up in an error message when a modality constraint is violated.

1.5 Related work

The reverse, control and with-computed operations we consider in this paper originally appeared in Quipper [13]. Our contributions are the formal study of these operations via a type system, operational semantics, and categorical semantics. Other than Quipper, the closest related work is our earlier papers [8] and [9]. In this work, we gave a type system, operational semantics, and categorical semantics for Proto-Quipper with *dynamic lifting*. Dynamic lifting allows interleaving circuit execution time with circuit generation time. Our type system ensures that a boxable quantum circuit does not use dynamic lifting. This is done by using a modality to distinguish quantum circuits from quantum computations. We also constructed a categorical semantics based on biset-enrichment. In the present paper, we give a similar type system, but with multiple modalities. We also generalize our categorical construction from biset-enrichment to *triset-enrichment*. For the operational semantics, we only consider the circuit evaluation for reversing, control and with-computed; we do not consider dynamic lifting in this paper. We believe this work complements our earlier work in the sense that the controllability and reversibility can work together with the notion of boxability.

QWire [20] is a quantum programming language that also supports circuit reversal. Unlike Proto-Quipper, QWire has a host language and a circuit language. The host language describes the computation of the classical computer, while the circuit language describes the computation of the quantum computer. Safe circuit reversing can be defined in QWire in the host language via pattern matching on circuits. In our work, it is ensured by the type system. To our knowledge, there are no concepts corresponding to the control and with-computed operations in QWire.

The quantum programming language Silq [2] has annotations **qfree** and **mfree**. These are annotations for types and typing judgments to denote classical functions and functions that do not use measurement, respectively. Silq's core language supports reversing, and the control operation is supported by an if-then-else construct. The main difference between Silq and Proto-Quipper is that Proto-Quipper is primarily a circuit description language. Silq is not a circuit description language and does not have a notion of boxed circuits. As a result, the circuit optimization shown in equation (1) is not presented in Silq.

In our construction of a concrete model for reversing and control in Section 6, we use enriched category theory. There are some recent works that also use enriched categories to model the semantics of quantum programming languages. For example, [18] use a CPO-enriched categorical model to interpret a version of Proto-Quipper with recursion. [21] give a categorical model for a QWire-like language that also uses enriched categories. As far as we know, none of these works studied the concepts of reversing, control and with-computed.

1.6 Contributions

In this paper, we formalize a notion of controllable category and give an axiomatization of an abstract categorical model for Proto-Quipper with reversing, control, and the with-computed operation. We extend the type system of [9] with modalities for reversing and control. We furthermore define an operational semantics and show that it is sound with respect to the categorical semantics. Lastly, we give a construction of a concrete categorical model that generalizes the biset-enrichment construction that was first developed in [8].

The rest of the paper is organized as follows: In Section 2, we give more examples of how reversing, control and with-computed are used in practice. In Section 3, we define a notion of controllable category and give an axiomatization of a categorical semantics for reversing and control. In Section 4, we define a type system that features the use of modalities. We then show how a typing judgment with modalities can be interpreted as a morphism in our categorical model. In Section 5, we define a call-by-value big-step operational semantics for our language. We show that the operational semantics is sound with respect to the categorical semantics. In Section 6, we construct a concrete categorical model. We finish the paper with concluding remarks in Section 7.

2 Reversing, control and with-computed in Proto-Quipper

In this section, we give examples of programs using reversing, control, and with-computed. We have implemented the type system and operational semantics (available from `https://gitlab.com/frank-peng-fu/dpq-remake`) and the programs in this section have been tested with this implementation.

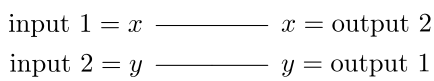
The reversing, control, and with-computed operations have the following types.

```
reverse : forall (a b : Type) ->
          (Simple a, Simple b) => Circ(a, b) -> Circ(b, a)
control : {a s : Type} ->
          (Simple s, Simple a) => Circ(a, a) -> Circ(a * s, a * s)
withComputed : forall (a b : Type) ->
          (Simple a, Simple b) => Circ(a, b) -> Circ(b, b) -> Circ(a, a)
```

The tensor product is represented by `*`. The type class constraint `(Simple a)` ensures that the variable `a` is a simple type.

2.1 Controlling a permutation circuit

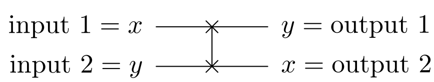
In Proto-Quipper, there are two ways to represent the swap operation. One way is by permuting the logical order of the circuit outputs without using any gates, as in the following diagram.



The above circuit is generated by the following Proto-Quipper program.

```
f : !(Qubit * Qubit -> Qubit * Qubit)
f input = let (x, y) = input in (y, x)
```

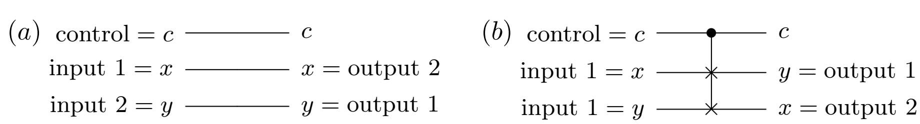
The other way is by using an explicit swap gate, as in the following diagram.



The corresponding Proto-Quipper program is the following.

```
g : !(Qubit * Qubit -> Qubit * Qubit)
g input = let (x, y) = input in Swap x y
```

These circuits are semantically equivalent: each of them sends (x, y) to (y, x). However, care must be taken when controlling these circuits, since controlling them naively may produce the following non-equivalent circuits.



While the second circuit is correct, the first one is not. Indeed, the first circuit will send input 1 to output 2 independently of the state of the control qubit. This is not the correct behavior, since the swapping should only take place when the control qubit is in the state $|1\rangle$. When the control qubit is in the state $|0\rangle$, the functions f and g should both behave like the identity function on `Qubit * Qubit`.

Therefore, the programming language must be aware not only of the circuit, but also of the implicit permutation of any qubits performed in the language. If such an operation is controlled, explicit swap gates must sometimes be inserted. Proto-Quipper-C does this correctly, whereas the original Quipper implementation did not.

For example, consider the following program.

```
permuteCirc : Circ(Qubit * Qubit * Qubit * Qubit,
                   Qubit * Qubit * Qubit * Qubit)
permuteCirc =
  boxCirc $ \ input -> let (x, y, z, w) = input in (w, y, x, z)
```

Since `permuteCirc` only permutes the input qubits and does nothing else, the generated circuit has no gates. It is shown in Fig. 1. The following program is the controlled version of `permuteCirc`.

```
cpermuteCirc : Circ(Qubit * Qubit * Qubit * Qubit * Qubit,
                    Qubit * Qubit * Qubit * Qubit * Qubit)
cpermuteCirc = control permuteCirc
```

The printed circuit for `cpermuteCirc` is in Fig. 2.

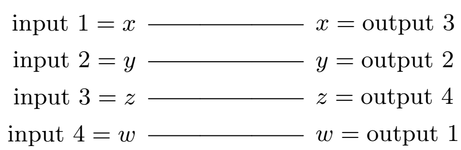


Fig. 1. Circuit for `permuteCirc`

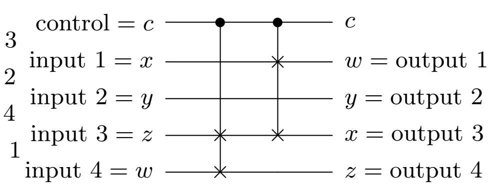
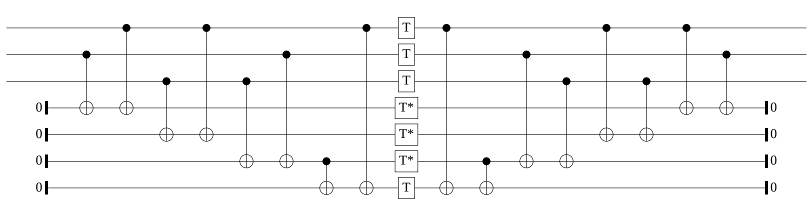


Fig. 2. Circuit for `cpermuteCirc`

Thus, in the implementation, we do not insert any swap gates when the qubit variables are permuted in the programming language. We do, however, track permutations of the variables and insert necessary swap gates when we are controlling a circuit.

2.2 Controlling a CCZ gate

It is well-known that a CCZ gate can be implemented by 7 T-gates with T-depth one [25]. See the following circuit.



We can use with-computed to define the above circuit in Proto-Quipper.

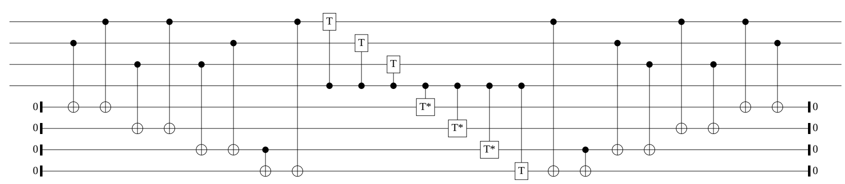
```
my_ccz : Circ(Qubit * Qubit * Qubit, Qubit * Qubit * Qubit)
my_ccz = withComputed box_cnot_circuit box_parallel_T
```

The function `box_parallel_T` generates the 7 parallel T gates in the middle of the circuit and the function `box_cnot_circuit` generates the initialization gates and the cnot circuits before the parallel T gates. Their definitions are available in Appendix D.

We can add an extra control to the CCZ circuit by the following program.

```
ctrl_ccz : Circ(Qubit * Qubit * Qubit * Qubit,
                 Qubit * Qubit * Qubit * Qubit)
ctrl_ccz = control my_ccz
```

The above program generates the following circuit.



We can see that only the T-gates in the middle are controlled. If we manually program the CCZ gate without using the with-computed operation, we will get an error when trying to control it.

```
cnot_circuit_rev :
  !(Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit
    -> Qubit * Qubit * Qubit)
cnot_circuit_rev = unbox (reverse (boxCirc cnot_circuit))

my_ccz' : Circ(Qubit * Qubit * Qubit, Qubit * Qubit * Qubit)
my_ccz' =
  boxCirc $ \input -> cnot_circuit_rev (parallel_T (cnot_circuit input))

-- The following gives rise to a typing error.
ctrl_my_ccz' : Circ(Qubit * Qubit * Qubit * Qubit,
                    Qubit * Qubit * Qubit * Qubit)
ctrl_my_ccz' = control my_ccz'
```

In the above program, `my_ccz'` is defined by manual composition, where `cnot_circuit_rev` is the reverse version of `cnot_circuit`. We will get a typing error when controlling `my_ccz'`. This is because according to the semantics of the control, we have to control all the gates in the circuit `my_ccz'`, which includes the non-controllable initialization gates. So in order to control the CCZ circuit that has initialization gates, we need to use the with-computed operation to construct the circuit.

3 Categorical semantics for reversing, control and with-computed

3.1 The categorical semantics for the modalities

The type system in this paper is closely related to the type system we proposed in [9]. In that work, a typing judgment takes the form of $\Gamma \vdash_\alpha M : A$, where $\alpha \in \{0,1\}$. The modality $\alpha = 1$ indicates that the term M is *boxable*, whereas modality $\alpha = 0$ indicates that the term M is not boxable. Semantically, boxable quantum circuits are interpreted as the morphisms of a category $\mathbf{M}$, whereas general (not necessarily boxable) circuits are interpreted as morphisms of a category $\mathbf{Q}$. These categories are equipped with an identity-on-objects functor $J : \mathbf{M} \to \mathbf{Q}$ which forgets that a circuit is boxable. From this data, in [9] we constructed a single symmetric monoidal category $\mathbf{A}$ with a monad T and a commutative diagram

$$\begin{array}{ccc} \mathbf{M} & \hookrightarrow & \mathbf{A} \\ \downarrow{\scriptstyle J} & & \downarrow \\ \mathbf{Q} & \hookrightarrow & Kl_T(\mathbf{A}). \end{array}$$

Here, $\mathbf{A} \to Kl_T(\mathbf{A})$ is the canonical functor from $\mathbf{A}$ to the Kleisli category of T, and the functors $\mathbf{M} \hookrightarrow \mathbf{A}$ and $\mathbf{Q} \hookrightarrow Kl_T(\mathbf{A})$ are full and faithful embeddings. In this setting, a typing judgment $\Gamma \vdash_1 M : A$ is interpreted as a morphism $[\![\Gamma]\!] \to [\![A]\!]$ in $\mathbf{A}$, which corresponds to a morphism in $\mathbf{M}$ via the embedding; and $\Gamma \vdash_0 M : A$ is interpreted as a morphism $[\![\Gamma]\!] \to T[\![A]\!]$ in the Kleisli category $Kl_T(\mathbf{A})$, which corresponds to a morphism in $\mathbf{Q}$.

Using an analogous approach, we consider typing judgments $\Gamma \vdash_\alpha M : A$ where $\alpha \in \{0,1,2\}$. Recall from Section 1.2 that we are given symmetric monoidal categories $\mathbf{M}$, $\mathbf{R}$, and $\mathbf{N}$ of general circuits, reversible circuits, and controllable circuits, respectively, with functors $G : \mathbf{N} \to \mathbf{R}$ and $I : \mathbf{R} \hookrightarrow \mathbf{M}$. In the semantics, these three categories are combined into a single category $\mathbf{A}$, and the modalities $\alpha = 0,1,2$ are modeled by monads T_0, T_1, and T_2 on $\mathbf{A}$, where T_2 is the identity monad. This is done in such a way that the following diagram commutes.

$$\begin{array}{ccc} \mathbf{N} & \hookrightarrow & \mathbf{A} \\ \downarrow{\scriptstyle G} & & \downarrow \\ \mathbf{R} & \hookrightarrow & Kl_{T_1}(\mathbf{A}) \\ \downarrow{\scriptstyle I} & & \downarrow \\ \mathbf{M} & \hookrightarrow & Kl_{T_0}(\mathbf{A}) \end{array}$$

Then a typing judgment $\Gamma \vdash_\alpha M : A$ is interpreted as a morphism $[\![M]\!] : [\![\Gamma]\!] \to T_\alpha[\![A]\!]$ in the category $\mathbf{A}$. Thus a morphism $[\![\Gamma]\!] \to [\![A]\!]$ corresponds to a morphism in the controllable category $\mathbf{N}$, and $[\![\Gamma]\!] \to T_1[\![A]\!]$ corresponds to a morphism in the category $\mathbf{R}$, and $[\![\Gamma]\!] \to T_0[\![A]\!]$ corresponds to a morphism in the category $\mathbf{M}$.

We anticipate that it would be easy to combine our modalities for reversing and control with the modality for dynamic lifting of [9]. To add dynamic lifting, one would simply add a fourth modality and one more monad T, and extend the diagram with one more square:

$$
\begin{array}{ccc}
\mathbf{N} & \hookrightarrow & \mathbf{A} \\
\downarrow{\scriptstyle G} & & \downarrow \\
\mathbf{R} & \hookrightarrow & Kl_{T_1}(\mathbf{A}) \\
\downarrow{\scriptstyle I} & & \downarrow \\
\mathbf{M} & \hookrightarrow & Kl_{T_0}(\mathbf{A}) \\
\downarrow{\scriptstyle J} & & \downarrow \\
\mathbf{Q} & \hookrightarrow & Kl_{T}(\mathbf{A})
\end{array}
$$

For the sake of simplicity, we will only focus on reversing and control in this paper.

3.2 A category of controllable circuits

Before we can define a programming language for building quantum circuits, we must specify what a quantum circuit is. However, there exist many different classes of circuits; for example, they differ by what data they can manipulate (only qubits, or also classical bits, and/or more exotic objects like qutrits), what the built-in gates are (for example, the Clifford+T gate set, Toffoli+Hadamard, rotations by arbitrary angles), whether or not qubit initialization and termination is supported, whether measurement is considered as a gate, and so on. Therefore, rather than tailoring our programming language to a specific class of circuits, we make both the language and its operational and denotational semantics *parametric* on a given class of circuits. This is analogous to making a classical programming language parametric on some signature of built-in operations. For us, quantum circuits are abstractly given as the morphisms of monoidal categories with certain properties, which we now specify.

We start from a given symmetric monoidal category $\mathbf{M}$ of quantum circuits. In practice, the objects of $\mathbf{M}$ are generated from wire types such as **Bit** and **Qubit**, and the morphisms are generated from a finite set of gates. In the following, we define what we mean by a reversible subcategory of $\mathbf{M}$. Recall that a dagger symmetric monoidal category is a symmetric monoidal category equipped with a contravariant, identity-on-objects, involutive functor such that for all morphism $f : A \to B$, we have $f^\dagger : B \to A$. Moreover, the dagger functor is required to be compatible with the symmetric monoidal structure [24].

Definition 1. *By a* reversible subcategory *of* $\mathbf{M}$*, we mean a dagger symmetric monoidal category* $\mathbf{R}$*, together with an identity-on-objects, faithful, symmetric* *(strong) monoidal functor* $I : \mathbf{R} \to \mathbf{M}$. *We usually regard* I *as an inclusion functor, i.e., we regard* $\mathbf{R}$ *as a subcategory of* $\mathbf{M}$.

A morphism f of $\mathbf{R}$ is called *unitary* if $f \circ f^\dagger = \mathrm{id}$ and $f^\dagger \circ f = \mathrm{id}$. Note that we do not require all morphisms of $\mathbf{R}$ to be unitary. Thus, our notion of reversible morphism means a morphism that has an adjoint, not necessarily an inverse.

The dagger functor provides a semantics for the operation of reversing a quantum circuit. In order to capture the notion of control and with-computed, we define a notion of controllable category.

Definition 2. *Let* $\mathbf{R}$ *be a dagger symmetric monoidal category. By a* controllable category *of* $\mathbf{R}$*, we mean a dagger symmetric monoidal category* $\mathbf{N}$ *with the same objects as* $\mathbf{R}$*, and equipped with the following structure:*

(a) *For all* $A, B \in \mathbf{N}$*, a function* $- \bullet - : \mathbf{R}(B, A) \times \mathbf{N}(A, A) \to \mathbf{N}(B, B)$ *(also denoted by* $\mathsf{withComputed}$*) such that:*

$$\mathrm{id} \bullet h = h$$

$$(g_1; g_2) \bullet h = g_1 \bullet (g_2 \bullet h).$$

$$(g_1 \otimes g_2) \bullet (h_1 \otimes h_2) = (g_1 \bullet h_1) \otimes (g_2 \bullet h_2)$$

$$(g \bullet h)^\dagger = g \bullet h^\dagger$$

(b) *For all* $S, A \in \mathbf{N}$*, a function* $\mathsf{control}_S : \mathbf{N}(A, A) \to \mathbf{N}(A \otimes S, A \otimes S)$ *such that the following hold:*

$$\mathsf{control}_S(\mathrm{id}_A) = \mathrm{id}_{A \otimes S}$$

$$\begin{CD} A \otimes I @>{\mathsf{control}_I(h)}>> A \otimes I \\ @V{\rho}VV @VV{\rho}V \\ A @>{h}>> A \end{CD}$$

$$\begin{CD} A \otimes (B \otimes C) @>{\mathsf{control}_{B \otimes C}(h)}>> A \otimes (B \otimes C) \\ @V{\alpha}VV @VV{\alpha}V \\ (A \otimes B) \otimes C @>{\mathsf{control}_C(\mathsf{control}_B(h))}>> (A \otimes B) \otimes C \end{CD}$$

$$\begin{CD} A \otimes (B \otimes C) @>{\mathsf{control}_{B \otimes C}(h)}>> A \otimes (B \otimes C) \\ @V{A \otimes \gamma}VV @VV{A \otimes \gamma}V \\ A \otimes (C \otimes B) @>{\mathsf{control}_{C \otimes B}(h)}>> A \otimes (C \otimes B). \end{CD}$$

(c) *An identity-on-objects, strict dagger symmetric monoidal functor* $G : \mathbf{N} \to \mathbf{R}$ *such that*

$$G(g \bullet h) = g; G(h); g^\dagger$$

Remarks.

- The functor G gives an intended interpretation for the with-computed function. It captures the common quantum computation pattern $g; h; g^{\dagger}$.
- The functor G preserves the equalities of condition (a). For example, $G(id \bullet h) = id; G(h); id^{\dagger} = G(h)$. Note that we do not require $g \bullet \mathrm{id}_A = \mathrm{id}_B$, because this would imply $g; g^{\dagger} = g; \mathrm{id}_A; g^{\dagger} = G(g \bullet \mathrm{id}_A) = G(\mathrm{id}_B) = \mathrm{id}_B$, which is not an assumption we make for $\mathbf{R}$. For similar reasons, we do not require $g \bullet (h_1; h_2) = (g \bullet h_1); (g \bullet h_2)$.
- In (b), one might have expected us to also require the following properties. However, in the intended interpretation, these properties do not hold "on the nose" (up to literal equality of circuits) because controlling a circuit can insert additional controlled swap gates, which may end up in different places on the left- and right-hand side. Instead, the properties only hold up to equivalence of circuits. That is, if there is a given interpretation functor $J : \mathbf{M} \to \mathbf{Q}$ as in Section 3.1, we can define $f \sim g$ to mean $J(I(G(f))) = J(I(G(g)))$. In that case, it makes sense to require the following properties.

$$\mathsf{control}_S(h; g) \sim \mathsf{control}_S(h); \mathsf{control}_S(g), \textit{where } g, h \in \mathbf{N}(A, A).$$

$$\mathsf{control}_S(h^{\dagger}) \sim \mathsf{control}_S(h)^{\dagger}$$

$$\mathsf{control}_S(g \bullet h) \sim (g \otimes \mathrm{id}_S) \bullet \mathsf{control}_S(h)$$

  The last equation formalizes the idea that to control a with-computed circuit $g \bullet h$, it suffices to control the circuit h in the middle.

An intended model for $\mathbf{N}$ can be constructed as follows: Let $\mathbf{M}$ be a symmetric monoidal category of syntactic quantum circuit diagrams, with objects such as **Qubit** and **Bit**, and morphisms that are freely generated by some set of primitive gates. Suppose, as is usual in quantum circuits, that some distinguished subset of the primitive gates is reversible, i.e., to each such reversible primitive gate G, another primitive gate $G^{\dagger}$ has been associated. Let $\mathbf{R}$ be the subcategory of $\mathbf{M}$ that is generated by the reversible gates. Also suppose that certain distinguished ones of the reversible gates are controllable, i.e., for a controllable primitive gate $G : A \to A$, there an associated controllable primitive gate $CG : A \otimes \mathbf{Qubit} \to A \otimes \mathbf{Qubit}$. Let $\mathbf{N}$ be a category with the same objects as $\mathbf{R}$. The morphisms in $\mathbf{N}$ are freely generated by generators of two colors, say red and green. Each reversible primitive gate is a red generator of $\mathbf{N}$, and each controllable primitive gate is a green generator of $\mathbf{N}$. (Therefore, there are two different colored versions of each primitive controllable gate in $\mathbf{N}$). All morphisms in $\mathbf{N}$ are reversible, and reversing does not change color. The control function adds controls to the green generators. The with-computed function takes any morphism g of $\mathbf{R}$ and any morphism f of $\mathbf{N}$, and produces $g; f; g^{\dagger}$, where g and $g^{\dagger}$ have been colored red. The functor G is the forgetful functor that forgets the colors.

3.3 A categorical semantics for reversing and control

We now assume that we are given three *small* categories $\mathbf{M}$, $\mathbf{R}$, and $\mathbf{N}$ as specified in the previous section. The categorical semantics, type system and the operational semantics are all parameterized by the categories $\mathbf{N}, \mathbf{R}$ and $\mathbf{M}$. These three categories lack the necessary structures to interpret a programming language. We therefore define a category $\mathbf{A}$ below, in which the typing rules can be interpreted as morphisms.

Definition 3. *A category $\mathbf{A}$ is a model for Proto-Quipper with reversing, control and with-computed if it is equipped with the following structure.*

(a) *$\mathbf{A}$ is symmetric monoidal closed, i.e., it is symmetric monoidal and there is an adjunction $- \otimes A \dashv A \multimap -$ for any $A \in \mathbf{A}$. We write $\epsilon : (A \multimap B) \otimes A \to B$ for the counit of this adjunction.*

(b) *$\mathbf{A}$ has coproducts. Note that the tensor distributes over coproducts, because $- \otimes A$ is a left adjoint.*

(c) *There is an adjunction $p \vdash \flat : \mathbf{Set} \to \mathbf{A}$ where p is a strong monoidal functor, i.e., $p(1) \cong I$ and $p(X \times Y) \cong p(X) \otimes p(Y)$.*

(d) *$\mathbf{A}$ is equipped with idempotent commutative strong monads T_0 and T_1 such that $T_0T_1 \cong T_0 \cong T_1T_0$. More specifically, we require the natural maps $\eta_{T_0B} : T_0B \to T_1T_0B$ and $T_0\eta_B : T_0B \to T_0T_1B$ to be isomorphisms. Here, by idempotent monad we mean that $\mu_B : T^2B \to TB$ is an isomorphism. We write $s : TA \otimes B \to T(A \otimes B)$ for the strength.*

(e) *There are full and faithful embeddings $\mathbf{N} \hookrightarrow \mathbf{A}$, $\mathbf{R} \hookrightarrow Kl_{T_1}(\mathbf{A})$, and $\mathbf{M} \hookrightarrow Kl_{T_0}(\mathbf{A})$. These embedding functors are strong monoidal. Moreover, the following diagram commutes for all $S, U \in \mathbf{M}$.*

$$\begin{array}{ccc}
\mathbf{N}(S,U) & \xrightarrow{\cong} & \mathbf{A}(S,U) \\
\downarrow{\scriptstyle G_{SU}} & & \downarrow{\scriptstyle E} \\
\mathbf{R}(S,U) & \xrightarrow{\cong} & Kl_{T_1}(\mathbf{A})(S,U) \\
\downarrow{\scriptstyle I_{S,U}} & & \downarrow{\scriptstyle L} \\
\mathbf{M}(S,U) & \xrightarrow{\cong} & Kl_{T_0}(\mathbf{A})(S,U)
\end{array}$$

Remarks:

- Conditions (a) and (b) are necessary to support higher-order functions in the language. All models of Proto-Quipper supports conditions (a)-(c).
- Because $p : \mathbf{Set} \to \mathbf{A}$ is a left adjoint and is strong monoidal, we can deduce that
$$p(X) \cong p(\sum_{x \in X} 1) \cong \sum_{x \in X} p(1) \cong \sum_{x \in X} I.$$
Due to the adjunction $p \dashv \flat$, we also have
$$\mathbf{Set}(X, \flat B) \cong \mathbf{A}(p(X), B) \cong \mathbf{A}(\sum_{x \in X} I, B) \cong \prod_{x \in X} \mathbf{A}(I, B) \cong \mathbf{Set}(X, \mathbf{A}(I, B)).$$
Therefore by Yoneda's principle, we have $\flat(B) \cong \mathbf{A}(I, B)$.

– In condition (d), the monad T_0 is intended to represent (via its Kleisli category) circuits that are neither reversible nor controllable, and the monad T_1 represents a circuit that is reversible but not controllable. To facilitate composition of morphisms from different Kleisli categories, we require the condition $T_1T_0 \cong T_0T_1 \cong T_0$, i.e., if a non-controllable circuit is composed with a non-reversible circuit, then the resulting circuit should be neither controllable nor reversible.
– Condition (e) expresses the idea that morphisms of ground type in the various Kleisli categories correspond to morphisms in $\mathbf{M}$, $\mathbf{R}$, and $\mathbf{N}$.
– In condition (e), the functors $E : \mathbf{A} \to Kl_{T_1}(\mathbf{A})$ and $L : Kl_{T_1}(\mathbf{A}) \to Kl_{T_0}(\mathbf{A})$ are the canonical identity-on-objects functors. Specifically, for any $f \in \mathbf{A}(A, B)$, we have $E(f) = \eta^{T_1} \circ f \in Kl_{T_1}(\mathbf{A})(A, B)$, and for any $f : A \to T_1B \in Kl_{T_1}(\mathbf{A})$, we set $L(f)$ to be

$$A \xrightarrow{f} T_1B \xrightarrow{\eta^{T_0}} T_0T_1B \xrightarrow{(T_0\eta_B)^{-1}} T_0B.$$

It can be shown that L is a well-defined symmetric monoidal functor.
– For $\beta \in \{2, 1, 0\}$, we write $\mathbf{D}^\beta$ for $\mathbf{N}$ if $\beta = 2$; $\mathbf{R}$ if $\beta = 1$; and $\mathbf{M}$ if $\beta = 0$. We also write T_β for the corresponding monad $T_2 = \mathrm{id}$, T_1, and T_0. We write $\alpha \wedge \beta = \min(\alpha, \beta)$.
– For any $S, U \in \mathrm{obj}(\mathbf{M})$, the map $\mathsf{unbox} : \mathbf{D}^\beta(S, U) \to \flat(S \multimap T_\beta U)$ is defined by

$$\mathbf{D}^\beta(S, U) \xrightarrow{\cong} \mathbf{A}(S, T_\beta U) \xrightarrow{\text{curry}} \mathbf{A}(I, S \multimap T_\beta U) \xrightarrow{\cong} \flat(S \multimap T_\beta U).$$

– For any $S, U \in \mathrm{obj}(\mathbf{M})$, the map $\mathsf{box} : \flat T_\alpha(S \multimap T_\beta U) \to \mathbf{D}^{\alpha\wedge\beta}(S, U)$ is defined by the following commutative diagram.

$$\begin{array}{ccc}
\flat T_\alpha(S \multimap T_\beta U) & \xrightarrow{\mathsf{box}} & \mathbf{D}^{\alpha\wedge\beta}(S, U) \\
\downarrow\cong & & \\
\mathbf{Set}(1, \flat T_\alpha(S \multimap T_\beta U)) & & \cong\uparrow \\
\downarrow\cong & & \\
\mathbf{A}(I, T_\alpha(S \multimap T_\beta U)) & \xrightarrow{k} & \mathbf{A}(S, T_{\alpha\wedge\beta}U)
\end{array}$$

For any $m \in \mathbf{A}(I, T_\alpha(S \multimap T_\beta U))$, the map $k(m)$ is given by the following composition.

$$S \xrightarrow{m\otimes S} T_\alpha(S \multimap T_\beta U) \otimes S \xrightarrow{s} T_\alpha((S \multimap T_\beta U) \otimes S) \xrightarrow{T_\alpha\epsilon} T_\alpha T_\beta U \xrightarrow{\cong} T_{\alpha\wedge\beta}U$$

– Note that box is the inverse of unbox when $\alpha = 2$.

In Section 6, we will construct a concrete category $\mathbf{A}$ based on $\mathbf{M}, \mathbf{N}, \mathbf{R}$ that will satisfy the above conditions.

4 A type system for reversing, control and with-computed

In this section, we define the syntax and type system of Proto-Quipper-C, a version of Proto-Quipper with support for reversing, control and with-computed. We will show that our type system is sound with respect to the categorical semantics by interpreting typing judgments as morphisms in the category **A** from Definition 3.

We first describe the syntax of the language and the type system. We note that the syntax of our term language mentions morphisms from the categories **M**, **N**, and **R**; however, it is still a *syntax*, not a semantics. Since the purpose of the language is to compute quantum circuits, there must be some terms in the language that represent circuits. We represent such terms as $\mathsf{circ}(C)$, where C is a morphism in the appropriate category. This is effectively the same as adding a constant symbol for every possible circuit.

Definition 4 (Syntax).

Modalities	α, β	$::= 2 \mid 1 \mid 0$
Types	A, B	$::= \mathbf{Unit} \mid \mathbf{Qubit} \mid \mathbf{Bit} \mid \mathbf{Bool} \mid !_\alpha A \mid A \multimap_\alpha B$
		$\mid \mathbf{Circ}_\alpha(S, U) \mid A \otimes B$
Parameter Types	P, R	$::= \mathbf{Unit} \mid \mathbf{Bool} \mid !_\alpha A \mid \mathbf{Circ}_\alpha(S, U) \mid P \otimes R$
Simple Types	S, U	$::= \mathbf{Unit} \mid \mathbf{Qubit} \mid \mathbf{Bit} \mid S \otimes U$
Terms	M, N	$::= c \mid x \mid \lambda x.M \mid MN \mid \mathsf{Unit} \mid \mathsf{circ}(C) \mid \mathsf{apply}(M, N)$
		$\mid \mathsf{lift}\ M \mid \mathsf{box}_U M \mid (M, N) \mid \mathsf{let}\ (x, y) = N\ \mathsf{in}\ M$
		$\mid \mathsf{force}\, M \mid \mathsf{reverse}\, M \mid \mathsf{control}_S M$
		$\mid \mathsf{withComputed}\, M\, N$
Simple Terms	a, b	$::= \ell \mid \mathsf{Unit} \mid (a, b)$
Contexts	Γ	$::= \cdot \mid x : A, \Gamma \mid \ell : \mathbf{Qubit}, \Gamma \mid \ell : \mathbf{Bit}, \Gamma$
Parameter contexts	Φ	$::= \cdot \mid x : P, \Phi.$
Label Contexts	Σ	$::= \cdot \mid \ell : \mathbf{Qubit}, \Sigma \mid \ell : \mathbf{Bit}, \Sigma$
Values	V	$::= x \mid \ell \mid \lambda x.M \mid \mathsf{lift}\ M \mid \mathsf{circ}(C) \mid (V, V') \mid \mathsf{unit}$
Circuits	C, D	$\in \mathbf{N}([\![S]\!], [\![U]\!]) \mid \mathbf{R}([\![S]\!], [\![U]\!]) \mid \mathbf{M}([\![S]\!], [\![U]\!])$

The syntax is similar to the one specified in [9], except for the highlighted terms and types. The symbol c is used for constant symbols, such as True and False for the type **Bool**, as well as other appropriate constants. For space reasons, we omit a comprehensive treatment of sum types. The values of *parameter types* can be duplicated or discarded, whereas the values *simple types* are *simple terms*, which are resources. They cannot be freely duplicated nor discarded. A simple term consists of a tuple of labels, which are pointers to places in a quantum circuit where gates can be attached. Unlike variables, labels cannot be substituted, and they can only have type **Qubit** or **Bit**. We call a typing context that contains only labels a *label context* (denoted by Σ). We call a typing context that contains only parameter types a *parameter context* (denoted by Φ). The modality is represented by a number in the lattice $2 > 1 > 0$, where $\alpha = 0$ indicates non-reversible and non-controllable; $\alpha = 1$ indicates reversible but not controllable; and $\alpha = 2$ indicates controllable and reversible. The modality in the type $!_\alpha A$ means that when its value is forced, it may append a gate that has modality α to the then-current circuit. Similarly, the modality in the type $A \multimap_\alpha B$ indicates that when its value is applied to an argument, it may append a gate that has modality α. We also add a modality to the circuit type $\mathbf{Circ}_\alpha(S, U)$, indicating the corresponding property (e.g., reversibility) of its circuit values. We include the terms $\mathsf{reverse}\,M$, $\mathsf{control}_S M$, and $\mathsf{withComputed}\,M\,N$ for reversing, control and the with-computed operation. For a simple type S, we write $[\![S]\!]$ to denote the object corresponding to S in the category $\mathbf{M}$ (note that $\mathrm{obj}(\mathbf{M}) = \mathrm{obj}(\mathbf{N}) = \mathrm{obj}(\mathbf{R})$). The term $\mathsf{circ}(C)$ is a value of a circuit type, where C is a morphism from $\mathbf{N}$, $\mathbf{R}$ or $\mathbf{M}$.

Recall that we write $\alpha \wedge \beta$ to denote the greatest lower bound, or $\min(\alpha, \beta)$. And we write $\mathbf{D}^\alpha$ to mean $\mathbf{M}$ if $\alpha = 0$, and $\mathbf{R}$ if $\alpha = 1$, and $\mathbf{N}$ if $\alpha = 2$. The following is the definition of the typing rules.

Definition 5 (Typing rules).

$$\frac{}{\Phi, x : A \vdash_2 x : A}\ \mathit{var} \qquad \frac{}{\ell : \mathbf{Qubit} \mid \mathbf{Bit} \vdash_2 \ell : \mathbf{Qubit} \mid \mathbf{Bit}}\ \mathit{label}$$

$$\frac{\Gamma \vdash_\beta M : !_\alpha A}{\Gamma \vdash_{\alpha\wedge\beta} \mathsf{force}\ M : A}\ \mathit{force} \qquad \frac{\Phi, \Gamma_1 \vdash_{\alpha_1} M : A \quad \Phi, \Gamma_2 \vdash_{\alpha_2} N : B}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha_1\wedge\alpha_2} (M, N) : A \otimes B}\ \mathit{pair}$$

$$\frac{\Gamma, x : A \vdash_\alpha M : B}{\Gamma \vdash_2 \lambda x.M : A \multimap_\alpha B}\ \mathit{lambda} \qquad \frac{\Phi, \Gamma_1 \vdash_{\alpha_1} M : A \multimap_\beta B \quad \Phi, \Gamma_2 \vdash_{\alpha_2} N : A}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha_1\wedge\alpha_2\wedge\beta} MN : B}\ \mathit{app}$$

$$\frac{\Gamma \vdash_\alpha M : !_\beta(S \multimap_\gamma U)}{\Gamma \vdash_\alpha \mathsf{box}_S\ M : \mathbf{Circ}_{\beta\wedge\gamma}(S, U)}\ \mathit{box} \qquad \frac{\Phi, \Gamma_1 \vdash_\alpha M : \mathbf{Circ}_\gamma(S, U) \quad \Phi, \Gamma_2 \vdash_\beta N : S}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha\wedge\beta\wedge\gamma} \mathsf{apply}(M, N) : U}\ \mathit{apply}$$

$$\frac{C \in \mathbf{D}^\alpha([\![S]\!], [\![U]\!])}{\Phi \vdash_2 \mathsf{circ}(C) : \mathbf{Circ}_\alpha(S, U)}\ \mathit{circ} \qquad \frac{\Gamma \vdash_\alpha M : \mathbf{Circ}_2(U, U)}{\Gamma \vdash_\alpha \mathsf{control}_S M : \mathbf{Circ}_2(U \otimes S, U \otimes S)}$$

$$\frac{\Phi \vdash_\alpha M : A}{\Phi \vdash_2 \mathsf{lift}\,M : !_\alpha A}\ \mathit{lift} \qquad \frac{\Phi, \Gamma_1, x : A, y : B \vdash_{\alpha_1} M : C \quad \Phi, \Gamma_2 \vdash_{\alpha_2} N : A \otimes B}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha_1\wedge\alpha_2} \mathsf{let}\ (x, y) = N\ \mathsf{in}\ M : C}$$

$$\frac{\Gamma \vdash_\alpha M : \mathbf{Circ}_\gamma(S, U) \quad \gamma > 0}{\Gamma \vdash_\alpha \mathsf{reverse}\,M : \mathbf{Circ}_\gamma(U, S)} \qquad \frac{\begin{array}{c}\Phi, \Gamma_1 \vdash_\alpha M : \mathbf{Circ}_\gamma(U, S)\ \gamma > 0 \\ \Phi, \Gamma_2 \vdash_\beta N : \mathbf{Circ}_2(S, S)\end{array}}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha\wedge\beta} \mathsf{withComputed}\,M\,N : \mathbf{Circ}_2(U, U)}$$

Remarks:

- Typing judgments are of the form $\Gamma \vdash_\alpha M : A$. Here, the modality α asserts that when evaluating the term M, a gate of modality α may be appended to the current circuit.
- Since a value cannot append any gates, it does not change the current circuit state. Therefore values always have modality 2, i.e., $\Gamma \vdash_2 V : A$.
- The *lambda* and *lift* rules store the modality α of M in the respective types $A \multimap_\alpha B$ and $!_\alpha A$.

- In the rules *app*, *force*, and *apply*, the modality in the types $!_\alpha A$ and $A \multimap_\alpha B$ affects the modality of the current term.
- A controllable circuit always has the type $\mathbf{Circ}_2(S,S)$ for some simple type S. It cannot have type $\mathbf{Circ}_\alpha(S,T)$ where $\alpha \neq 2$ or $S \neq T$.
- The typing rule for with-computed requires the term N to be a controllable circuit, whereas the term M only needs to be reversible. The resulting term $\mathsf{withComputed}\, M\, N$ is again a controllable circuit.
- In the typing rule *box*, the modalities β, γ jointly determine the modality of the circuit type.
- We often write $\Sigma \vdash a : S$ for $\Sigma \vdash_2 a : S$. Moreover, for $\Sigma \vdash a : S$, there is an obvious interpretation $[\![a]\!] : [\![\Sigma]\!] \to [\![S]\!]$ as an isomorphism in $\mathbf{M}$, $\mathbf{R}$ or $\mathbf{N}$.
- In the *circ* rule, the modality of the Circ-type depends on which category the circuit is coming from. Since C is a morphism in $\mathbf{M}$, $\mathbf{R}$ or $\mathbf{N}$, it is a fairly low-level rule. Programmers usually do not need to write the circuit value explicitly. Rather, they can use a set of builtin gates, which may be bound to identifiers in a pre-loaded library, to write programs of a circuit type.

4.1 Interpretation

We will interpret the typing rules in the model $\mathbf{A}$ from Definition 3. The following is the interpretation for types.

Definition 6.

$$\begin{aligned}
[\![\mathbf{Qubit}]\!] &= \mathbf{Qubit} \\
[\![A \otimes B]\!] &= [\![A]\!] \otimes [\![B]\!] \\
[\![\mathbf{Circ}_\alpha(S,U)]\!] &= p\mathbf{D}^\alpha([\![S]\!],[\![U]\!]) \\
[\![!_\alpha A]\!] &= p\flat T_\alpha [\![A]\!] \\
[\![A \multimap_\alpha B]\!] &= [\![A]\!] \multimap T_\alpha [\![B]\!]
\end{aligned}$$

We will interpret a typing context Γ as a tensor product of all of its objects (denoted by $[\![\Gamma]\!]$). Each valid typing judgment $\Gamma \vdash_\alpha M : A$ will be interpreted as a morphism in the Kleisli category of T_α. The interpretation of typing judgements is given by the following definition.

Definition 7. *For $\Gamma \vdash_\alpha M : A$, we define $[\![M]\!] : [\![\Gamma]\!] \to T_\alpha[\![A]\!]$ in $\mathbf{A}$ by induction on the typing rules. Here, we only give some important cases. The remaining typing rules can be interpreted similarly.*

- *Case*

$$\frac{\Gamma \vdash_\alpha M : !_{\alpha_1}(S \multimap_{\alpha_2} U)}{\Gamma \vdash_\alpha \mathsf{box}_S M : \mathbf{Circ}_{\alpha_1 \wedge \alpha_2}(S,U)}$$

We define $[\![\mathsf{box}_S M]\!]$ to be

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} T_\alpha p\flat T_{\alpha_1}([\![S]\!] \multimap T_{\alpha_2}[\![U]\!]) \xrightarrow{T_\alpha p\mathsf{box}} T_\alpha p\mathbf{D}^{\alpha_1 \wedge \alpha_2}([\![S]\!],[\![U]\!]).$$

– *Case*

$$\frac{C \in \mathbf{D}^{\alpha}(\llbracket S \rrbracket, \llbracket U \rrbracket)}{\Phi \vdash_2 \mathsf{circ}(C) : \mathbf{Circ}_{\alpha}(S, U)}$$

Since $C : \llbracket S \rrbracket \to \llbracket U \rrbracket$ *is a morphism in* $\mathbf{D}^{\alpha}$*, we define* $\llbracket \Phi \vdash_2 \mathsf{circ}(C) : \mathbf{Circ}_{\alpha}(S, U) \rrbracket$ *as the following composition, where* $1_C : 1 \to \mathbf{D}^{\alpha}(\llbracket S \rrbracket, \llbracket U \rrbracket)$ *such that* $1_C(*) = C$ *and* discard $= p!_X, !_X \in \mathbf{Set}(X, 1)$*.*

$$\llbracket \Phi \rrbracket \xrightarrow{\mathsf{discard}} p1 \xrightarrow{p1_C} p\mathbf{D}^{\alpha}(\llbracket S \rrbracket, \llbracket U \rrbracket).$$

– *Case*

$$\frac{\Gamma \vdash_{\alpha} M : \mathbf{Circ}_2(U, U)}{\Gamma \vdash_{\alpha} \mathsf{control}_S M : \mathbf{Circ}_2(U \otimes S, U \otimes S)}$$

By the induction hypothesis, we have $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \to T_{\alpha} p\mathbf{N}(\llbracket S \rrbracket, \llbracket S \rrbracket)$*. Using the function* $\mathsf{control}_{\llbracket S \rrbracket} : \mathbf{N}(\llbracket U \rrbracket, \llbracket U \rrbracket) \to \mathbf{N}(\llbracket U \rrbracket \otimes \llbracket S \rrbracket, \llbracket U \rrbracket \otimes \llbracket S \rrbracket)$*, we define* $\llbracket \mathsf{control}_S M \rrbracket$ *to be the following.*

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} T_{\alpha} p\mathbf{N}(\llbracket U \rrbracket, \llbracket U \rrbracket) \xrightarrow{T_{\alpha} p(\mathsf{control}_{\llbracket S \rrbracket})} T_{\alpha} p\mathbf{N}(\llbracket U \rrbracket \otimes \llbracket S \rrbracket, \llbracket U \rrbracket \otimes \llbracket S \rrbracket)$$

– *Case*

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\alpha} M : \mathbf{Circ}_{\gamma}(U, S) \ \gamma > 0 \\ \Gamma_2 \vdash_{\beta} N : \mathbf{Circ}_2(S, S)\end{array}}{\Gamma_1, \Gamma_2 \vdash_{\alpha \wedge \beta} \mathsf{withComputed}\, M N : \mathbf{Circ}_2(U, U)}$$

Suppose $\alpha = \beta = \gamma = 2$ *and* $\Gamma_1 \vdash_2 M : \mathbf{Circ}_2(U, S)$*. By the induction hypothesis, we have*

$$pG_{U,S} \circ \llbracket M \rrbracket : \llbracket \Gamma_1 \rrbracket \to p\mathbf{N}(\llbracket U \rrbracket, \llbracket S \rrbracket) \to p\mathbf{R}(\llbracket U \rrbracket, \llbracket S \rrbracket),$$
$$\llbracket N \rrbracket : \llbracket \Gamma_2 \rrbracket \to p\mathbf{N}(\llbracket S \rrbracket, \llbracket S \rrbracket).$$

Using the function $\mathsf{withComputed} : \mathbf{R}(\llbracket U \rrbracket, \llbracket S \rrbracket) \times \mathbf{N}(\llbracket S \rrbracket, \llbracket S \rrbracket) \to \mathbf{N}(\llbracket U \rrbracket, \llbracket U \rrbracket)$*, we define* $\llbracket \mathsf{withComputed}\, M N \rrbracket$ *to be the following.*

$$\llbracket \Gamma_1 \rrbracket \otimes \llbracket \Gamma_2 \rrbracket \xrightarrow{(pG_{U,S} \circ \llbracket M \rrbracket) \otimes \llbracket N \rrbracket} p\mathbf{R}(\llbracket U \rrbracket, \llbracket S \rrbracket) \otimes p\mathbf{N}(\llbracket S \rrbracket, \llbracket S \rrbracket)$$
$$\xrightarrow{p(\mathsf{withComputed})} p\mathbf{N}(\llbracket U \rrbracket, \llbracket U \rrbracket)$$

Suppose $\alpha = \beta = 2$*,* $\gamma = 1$ *and* $\Gamma_1 \vdash_2 M : \mathbf{Circ}_1(U, S)$*. By the induction hypothesis, we have*

$$\llbracket M \rrbracket : \llbracket \Gamma_1 \rrbracket \to p\mathbf{R}(\llbracket U \rrbracket, \llbracket S \rrbracket),$$
$$\llbracket N \rrbracket : \llbracket \Gamma_2 \rrbracket \to p\mathbf{N}(\llbracket S \rrbracket, \llbracket S \rrbracket).$$

We define $\llbracket \mathsf{withComputed}\, M N \rrbracket$ *to be the following.*

$$\llbracket \Gamma_1 \rrbracket \otimes \llbracket \Gamma_2 \rrbracket \xrightarrow{\llbracket M \rrbracket \otimes \llbracket N \rrbracket} p\mathbf{R}(\llbracket U \rrbracket, \llbracket S \rrbracket) \otimes p\mathbf{N}(\llbracket S \rrbracket, \llbracket S \rrbracket) \xrightarrow{p(\mathsf{withComputed})} p\mathbf{N}(\llbracket U \rrbracket, \llbracket U \rrbracket)$$

The remaining cases (e.g. when $\alpha, \beta \neq 2$*) are similar.*

Theorem 1. *If* $\Sigma \vdash_2 a : S$*, then* $\llbracket a \rrbracket : \llbracket \Sigma \rrbracket \to \llbracket S \rrbracket$ *is an isomorphism.*

Proof. This is by induction on the derivation of $\Sigma \vdash_2 a : S$.

Since $\mathbf{N} \hookrightarrow \mathbf{A}$ and $\llbracket \Sigma \rrbracket, \llbracket S \rrbracket \in \mathbf{N}$, we have $\llbracket a \rrbracket \in \mathbf{A}(\llbracket \Sigma \rrbracket, \llbracket S \rrbracket) \cong \mathbf{N}(\llbracket \Sigma \rrbracket, \llbracket S \rrbracket) = \mathbf{R}(\llbracket \Sigma \rrbracket, \llbracket S \rrbracket) = \mathbf{M}(\llbracket \Sigma \rrbracket, \llbracket S \rrbracket)$.

5 Operational semantics

In this section, we will define a big-step, call-by-value operational semantics for Proto-Quipper-C. Like the syntax and the type system, the operational semantics is parameterized by the triple of categories **M**, **R**, and **N**. Specifically, the operational semantics is defined on configurations that are pairs (C, M), where M is a term and $C : [\![S]\!] \to [\![\Sigma]\!]$ is the current circuit state, which is a morphism in **M**, **R** or **N**.

5.1 The operational semantics

Definition 8 (Operational semantics).

$$\frac{(C_1, M) \Downarrow (C_2, \lambda x.M') \quad (C_2, N) \Downarrow (C_3, V) \quad (C_3, [V/x]M') \Downarrow (C_4, V')}{(C_1, MN) \Downarrow (C_4, V')}\ \textit{app}$$

$$\frac{(C, M) \Downarrow (C', \mathsf{lift}\ M') \quad (C', M') \Downarrow (C'', V)}{(C, \mathsf{force}\ M) \Downarrow (C'', V)}\ \textit{force}$$

$$\frac{(C, N) \Downarrow (C', (V_1, V_2)) \quad (C', [V_1/x, V_2/y]M) \Downarrow (C'', V)}{(C, \mathsf{let}\ (x, y) = N\ \mathsf{in}\ M) \Downarrow (C'', V)}\ \textit{let}$$

$$\frac{(C, M) \Downarrow (C', V_1) \quad (C', N) \Downarrow (C'', V_2)}{(C, (M, N)) \Downarrow (C'', (V_1, V_2))}\ \textit{pair}$$

$$\frac{(C, M) \Downarrow (C', \mathsf{lift}\ M') \quad \mathrm{gen}(S) = (a, \Sigma) \quad ([\![a]\!]^\dagger, M'\ a) \Downarrow (D, b)}{(C, \mathsf{box}_S\ M) \Downarrow (C', \mathsf{circ}([\![b]\!] \circ D))}\ \textit{box}$$

$$\frac{(C_1, M) \Downarrow (C_2, \mathsf{circ}(D)) \quad (C_2, N) \Downarrow (C_3, V) \quad \mathrm{gen}(\mathrm{codomain}(D)) = (b, \Sigma) \quad C' = \mathrm{append}(C_3, D, V, b)}{(C_1, \mathsf{apply}(M, N)) \Downarrow (C', b)}\ \textit{apply}$$

$$\frac{(C, M) \Downarrow (C', \mathsf{circ}(D))}{(C, \mathsf{reverse}\, M) \Downarrow (C', \mathsf{circ}(D^\dagger))}\ \textit{rev}$$

$$\frac{(C, M) \Downarrow (C', \mathsf{circ}(D))}{(C, \mathsf{control}_S M) \Downarrow (C', \mathsf{circ}(\mathsf{control}_{[\![S]\!]} D))}\ \textit{ctrl}$$

$$\frac{(C', M) \Downarrow (C'', \mathsf{circ}(D_1)) \quad (C, N) \Downarrow (C', \mathsf{circ}(D_2)) \quad \alpha = \mathrm{mode}(D_1)}{(C, \mathsf{withComputed}\, M\, N) \Downarrow (C'', \mathsf{circ}(G^\alpha(D_1) \bullet D_2))}\ \textit{wc}$$

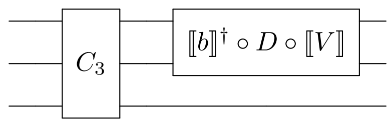
Remarks.

- The *app*, *force*, *let* and *pair* rules are standard. They do not directly modify the underlying circuit state.
- In the *box* rule, we write $\mathrm{gen}(S) = (a, \Sigma)$ to indicate the generation of a set of fresh labels a such that $\Sigma \vdash a : S$ and $[\![a]\!] : [\![\Sigma]\!] \to [\![S]\!]$. Suppose the codomain of D is $[\![\Sigma']\!]$ and $\Sigma' \vdash b : U$. Thus

$$[\![S]\!] \xrightarrow{D} [\![\Sigma']\!] \xrightarrow{[\![b]\!]} [\![U]\!]$$

  is a morphism in **M**, **R** or **N**.

- In the *wc* rule, we write $\text{mode}(f) = 0$ if a morphism f belongs to the category $\mathbf{M}$, $\text{mode}(f) = 1$ if f belongs to the category $\mathbf{R}$, and $\text{mode}(f) = 2$ if f belongs to the category $\mathbf{N}$.
- In the *apply* rule, suppose $\text{domain}(D) = [\![S]\!]$ and $\text{codomain}(D) = [\![U]\!]$. So $\text{gen}(U) = (b, \Sigma)$ implies that $\Sigma \vdash b : U$ and $[\![b]\!] : [\![\Sigma]\!] \to [\![U]\!]$. Note that V and S uniquely determine a label context Σ_1', which is a subset of labels in the codomain of C_3. We write $C' = \text{append}(C_3, D, V, b)$ to mean that C' is obtained via composing $([\![b]\!]^\dagger \circ D \circ [\![V]\!]) : [\![\Sigma_1']\!] \to [\![\Sigma]\!]$ with $C_3 : [\![S]\!] \to [\![\Sigma_1']\!] \otimes [\![\Sigma_2']\!]$ (illustrated by the following diagram).



  This composition is always possible, even when they belong to different categories. E.g., if $([\![b]\!]^\dagger \circ D \circ [\![V]\!]) \in \mathbf{N}$ and $C_3 \in \mathbf{M}$, we can apply the functor $I \circ G$ to $([\![b]\!]^\dagger \circ D \circ [\![V]\!])$ before the composition. Thus $\text{mode}(C') = \text{mode}(C_3) \wedge \text{mode}([\![b]\!]^\dagger \circ D \circ [\![V]\!])$.
- In the *rev* and *ctrl* rules, the morphism D is reversed/controlled using the reverse/control function from the category it belongs to. If D is not a morphism in $\mathbf{R}$ or $\mathbf{N}$, respectively, then it will cause a runtime error. The type preservation property (Theorem 2) will ensure that there are no such runtime errors.
- In the *wc* rule, if $\alpha = 2$, then $G^\alpha = G : \mathbf{N} \to \mathbf{R}$, and if $\alpha = 1$ then $G^\alpha = \text{id} : \mathbf{R} \to \mathbf{R}$. All the other cases are runtime errors, which are prevented by our type system via type preservation (Theorem 2).

In order to formulate type preservation, we first define a notion of *well-typed configuration.*

Definition 9 (Well-typed configuration). *We define a well-typed configuration* $\vdash_{\alpha\wedge\beta} (C, M) : A; \Sigma'$ *to mean:* $\text{mode}(C) = \alpha$, $C \in \mathbf{D}^\alpha([\![S]\!], [\![\Sigma'']\!] \otimes [\![\Sigma']\!])$ *and* $\Sigma'' \vdash_\beta M : A$. *We also write* $\text{mode}(M) = \beta$.

Theorem 2 (Type preservation). *Suppose* $\vdash_{\alpha\wedge\beta} (C, M) : A; \Sigma'$ *and* $(C, M) \Downarrow (C', V)$*, where* $\text{mode}(C) = \alpha, \text{mode}(M) = \beta$. *We have* $\vdash_{\alpha\wedge\beta} (C', V) : A; \Sigma'$, *where* $\text{mode}(C') = \alpha \wedge \beta$ *and* $\text{mode}(V) = 2$.

Proof. The proof is by induction on the derivation of $(C, M) \Downarrow (C', V)$.

Example. Consider the configuration

$$(\text{id}_I, \mathsf{box}_S\, \mathsf{lift}(\lambda x.\mathbf{let}(a_1, a_2) = x\ \mathbf{in}\ \mathsf{apply}(\mathsf{circ}(\mathsf{CNOT}), (a_2, a_1)))),$$

where $S = \mathbf{Qubit} \otimes \mathbf{Qubit}$ and $\mathsf{CNOT} : \mathbf{Qubit} \otimes \mathbf{Qubit} \to \mathbf{Qubit} \otimes \mathbf{Qubit}$ is a morphism in $\mathbf{N}$, where the second qubit of the CNOT is the control qubit. Using the *box* rule, we first call gen(S), which returns a pair of fresh labels (ℓ_1, ℓ_2) and a label context $\Sigma = \ell_1 : \mathbf{Qubit}, \ell_2 : \mathbf{Qubit}$. Then we evaluate

$$([\![(\ell_1,\ell_2)]\!]^\dagger, (\lambda x.\mathbf{let}(a_1,a_2) = x \ \mathbf{in}\ \mathsf{apply}(\mathsf{circ}(\mathsf{CNOT}),(a_2,a_1))))(\ell_1,\ell_2))$$

to

$$(([\![(\ell_1,\ell_2)]\!]^\dagger, \mathsf{apply}(\mathsf{circ}(\mathsf{CNOT}),(\ell_2,\ell_1))).$$

Then by the *apply* rule, it is evaluated to

$$(([\![(\ell_3,\ell_4)]\!]^\dagger \circ \mathsf{CNOT} \circ [\![(\ell_2,\ell_1)]\!] \circ [\![(\ell_1,\ell_2)]\!]^\dagger), (\ell_3,\ell_4)),$$

where gen(codomain(CNOT)) = gen(S) returns fresh labels (ℓ_3, ℓ_4) and a label context $\Sigma' = \ell_3 : \mathbf{Qubit}, \ell_4 : \mathbf{Qubit}$. Therefore by the *box* rule, we eventually obtain

$$(\mathrm{id}_I, \mathsf{circ}(([\![(\ell_3,\ell_4)]\!] \circ [\![(\ell_3,\ell_4)]\!]^\dagger \circ \mathsf{CNOT} \circ [\![(\ell_2,\ell_1)]\!] \circ [\![(\ell_1,\ell_2)]\!]^\dagger)).$$

Note that $[\![(\ell_3,\ell_4)]\!] \circ [\![(\ell_3,\ell_4)]\!]^\dagger = \mathrm{id}_S$ and $[\![(\ell_2,\ell_1)]\!] \circ [\![(\ell_1,\ell_2)]\!]^\dagger = \gamma : \mathbf{Qubit} \otimes \mathbf{Qubit} \to \mathbf{Qubit} \otimes \mathbf{Qubit}$. Thus, the final value is

$$(\mathrm{id}_I, \mathsf{circ}(\mathsf{CNOT} \circ \gamma)).$$

Remark. The morphism γ comes from the symmetric monoidal structure on $\mathbf{N}$, and its effect is to switch two wires without inserting an explicit swap gate (it is, for example, the same as the interpretation of the function `f` in Section 2.1). Only if we later control the circuit $\mathsf{circ}(\mathsf{CNOT} \circ \gamma)$, this morphism γ will be replaced by a controlled swap gate, just like circuit (b) in Section 2.1).

5.2 Soundness of operational semantics

We now prove that the operational semantics is sound with respect to the categorical model $\mathbf{A}$ (Definition 3). To do so, we first interpret a well-typed configuration as a morphism in $\mathbf{A}$.

Definition 10. *Suppose* $\vdash_{\alpha\wedge\beta} (C, M) : A; \Sigma'$*, where* $\mathrm{mode}(C) = \alpha, C \in \mathbf{D}^\alpha([\![S]\!], [\![\Sigma'']\!] \otimes [\![\Sigma']\!])$ *and* $\Sigma'' \vdash_\beta M : A$*. We define* $[\![(C,M)]\!] : [\![S]\!] \to T_{\alpha\wedge\beta}([\![A]\!] \otimes [\![\Sigma']\!])$ *to be the following composition (note that* $T_\alpha \circ T_\beta \cong T_{\alpha\wedge\beta}$*):*

$$[\![S]\!] \xrightarrow{C} T_\alpha([\![\Sigma'']\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_\alpha([\![M]\!]\otimes[\![\Sigma']\!])} T_\alpha(T_\beta[\![A]\!] \otimes [\![\Sigma']\!])$$

$$\xrightarrow{T_\alpha s} T_\alpha T_\beta([\![A]\!] \otimes [\![\Sigma']\!]) \xrightarrow{\cong} T_{\alpha\wedge\beta}([\![A]\!] \otimes [\![\Sigma']\!]).$$

Since we have the embedding $\mathbf{D}^\alpha \hookrightarrow Kl_{T_\alpha}(\mathbf{A})$, the morphism $C : [\![S]\!] \to [\![\Sigma'']\!] \otimes [\![\Sigma']\!] \in \mathbf{D}^\alpha$ corresponds to a morphism $[\![S]\!] \to T_\alpha([\![\Sigma'']\!] \otimes [\![\Sigma']\!])$ in $\mathbf{A}$, which is also denoted by C.

Theorem 3 (Soundness of the evaluation). *If* $\vdash_{\alpha\wedge\beta} (C, M) : A; \Sigma'$ *and* $(C, M) \Downarrow (C', V)$*, then* $[\![(C,M)]\!] = [\![(C',V)]\!]$*.*

Proof. The proof is by induction on $(\mathcal{C}, M) \Downarrow (\mathcal{C}', V)$. Here we only prove the cases for control and with-computed. See Appendix B for more details.

– Case

$$\frac{(C, M) \Downarrow (C', \mathsf{circ}(D))}{(C, \mathsf{control}_S M) \Downarrow (C', \mathsf{circ}(\mathsf{control}_{[\![S]\!]} D))}\ \mathit{ctrl}$$

- Suppose $C : [\![S]\!] \to [\![\Sigma'']\!] \otimes [\![\Sigma']\!] \in \mathbf{N}$ and $\Sigma'' \vdash_2 M : \mathbf{Circ}_2(U, U)$. By the induction hypothesis and type preservation (Theorem 2), we have $[\![(C, M)]\!] = [\![(C', \mathsf{circ}(D))]\!]$ and $\Sigma \vdash_2 (C', \mathsf{circ}(D)) : \mathbf{Circ}_2(U, U) : \Sigma'$, where $\vdash_2 \mathsf{circ}(D) : \mathbf{Circ}_2(U, U)$. So

$$[\![S]\!] \xrightarrow{C} [\![\Sigma'']\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![M]\!] \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!]$$
$$= [\![S]\!] \xrightarrow{C'} I \otimes [\![\Sigma']\!] \xrightarrow{p1_D \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!].$$

The above equality implies the following.

$$[\![S]\!] \xrightarrow{C} [\![\Sigma'']\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![M]\!] \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!]$$
$$\xrightarrow{p(\mathsf{control}_{[\![S]\!]}) \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!] \otimes [\![S]\!], [\![U]\!] \otimes [\![S]\!]) \otimes [\![\Sigma']\!]$$
$$= [\![S]\!] \xrightarrow{C'} I \otimes [\![\Sigma']\!] \xrightarrow{p1_D \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!]$$
$$\xrightarrow{p(\mathsf{control}_{[\![S]\!]}) \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!] \otimes [\![S]\!], [\![U]\!] \otimes [\![S]\!]) \otimes [\![\Sigma']\!].$$

Thus $[\![(C, \mathsf{control}_S M)]\!] = [\![(C', \mathsf{circ}(\mathsf{control}_{[\![S]\!]}(D)))]\!]$.
- Now suppose $C \in \mathbf{M}([\![S]\!], [\![\Sigma'']\!] \otimes [\![\Sigma']\!])$ and $\Sigma'' \vdash_1 M : \mathbf{Circ}_2(U, U)$. By the induction hypothesis, we have $[\![(C, M)]\!] = [\![(C', \mathsf{circ}(D))]\!]$, i.e.,

$$[\![S]\!] \xrightarrow{C} T_0([\![\Sigma'']\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![M]\!] \otimes [\![\Sigma']\!])} T_0(T_1 p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0 s} T_0(p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!])$$
$$= [\![S]\!] \xrightarrow{C'} T_0(I \otimes [\![\Sigma']\!]) \xrightarrow{T_0(p1_D \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!]).$$

The above equality implies the following.

$$[\![S]\!] \xrightarrow{C} T_0([\![\Sigma'']\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![M]\!] \otimes [\![\Sigma']\!])} T_0(T_1 p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0 s} T_0(p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(p(\mathsf{control}_{[\![S]\!]}) \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![U]\!] \otimes [\![S]\!], [\![U]\!] \otimes [\![S]\!]) \otimes [\![\Sigma']\!])$$
$$= [\![S]\!] \xrightarrow{C'} T_0(I \otimes [\![\Sigma']\!]) \xrightarrow{T_0(p1_D \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(p(\mathsf{control}_{[\![S]\!]}) \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![U]\!] \otimes [\![S]\!], [\![U]\!] \otimes [\![S]\!]) \otimes [\![\Sigma']\!]).$$

By the naturality of the strength s, we have

$$[\![(C, \mathsf{control}_S M)]\!] = [\![(C', \mathsf{circ}(D))]\!].$$

– Case

$$\frac{\begin{array}{c}(C, M) \Downarrow (C', \mathsf{circ}(D_1)) \\ (C', N) \Downarrow (C'', \mathsf{circ}(D_2)) \\ \alpha = \mathrm{mode}(D_1)\end{array}}{(C, \mathsf{withComputed}\, M\, N) \Downarrow (C'', \mathsf{circ}(G^{\alpha}(D_1) \bullet D_2))}\ wc$$

- Suppose $\alpha = 1$, i.e., $G = \mathrm{id}$, $\mathrm{mode}(C) = 2$, $C \in \mathbf{N}([\![S]\!], [\![\Sigma_1'']\!] \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!])$, $\Sigma_1'' \vdash_2 N : \mathbf{Circ}_2(S', S')$ and $\Sigma_2'' \vdash_2 M : \mathbf{Circ}_1(U, S')$. By the induction hypothesis, we have $[\![(C, M)]\!] = [\![(C', \mathsf{circ}(D_1))]\!]$ and $[\![(C', N)]\!] = [\![(C'', \mathsf{circ}(D_2))]\!]$. Thus we have

$$[\![S]\!] \xrightarrow{C} [\![\Sigma_1'']\!] \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![M]\!] \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!]} p\mathbf{R}([\![U]\!], [\![S']\!]) \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!]$$

$$= [\![S]\!] \xrightarrow{C'} I \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!] \xrightarrow{p1_{D_1} \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!]} p\mathbf{R}([\![U]\!], [\![S']\!]) \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!]$$

and

$$[\![S]\!] \xrightarrow{C'} [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![N]\!] \otimes [\![\Sigma']\!]} p\mathbf{N}([\![S']\!], [\![S']\!]) \otimes [\![\Sigma']\!]$$

$$= [\![S]\!] \xrightarrow{C''} I \otimes [\![\Sigma']\!] \xrightarrow{p1_{D_2} \otimes [\![\Sigma']\!]} p\mathbf{N}([\![S']\!], [\![S']\!]) \otimes [\![\Sigma']\!].$$

We want to show $[\![(C, \mathsf{withComputed}\, M\, N)]\!] = [\![(C'', \mathsf{circ}(D_1 \bullet D_2))]\!]$, i.e.,

$$[\![S]\!] \xrightarrow{C} [\![\Sigma_1'']\!] \otimes [\![\Sigma_2'']\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![M]\!] \otimes [\![N]\!] \otimes [\![\Sigma']\!]}$$

$$p\mathbf{R}([\![U]\!], [\![S]\!]) \otimes p\mathbf{N}([\![S]\!], [\![S]\!]) \otimes [\![\Sigma']\!] \xrightarrow{p(\mathsf{withComputed}) \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!]$$

$$= [\![S]\!] \xrightarrow{C''} I \otimes [\![\Sigma']\!] \xrightarrow{p1_{D_1 \bullet D_2} \otimes [\![\Sigma']\!]} p\mathbf{N}([\![U]\!], [\![U]\!]) \otimes [\![\Sigma']\!].$$

This is the case by the induction hypotheses and $\mathsf{withComputed} \circ (1_{D_1} \otimes 1_{D_2}) = 1_{D_1 \bullet D_2}$.

6 A concrete model for reversing and control

In this section, we will show how to use the categories **N**, **R**, and **M** to construct a category **A** that satisfies the conditions of Definition 3. Our construction is based on enriched categories (see Appendix A for some background on enrichment). We first define a category of *trisets*.

Definition 11. *The category of* trisets *is defined as follows:*

– *An object X is a tuple $(X_0, X_1, X_2, f_1 : X_1 \to X_0, f_2 : X_2 \to X_1)$, where X_0, X_1, X_2 are sets and f_1, f_2 are functions.*

– *A morphism from* $(X_0, X_1, X_2, f_1, f_2)$ *to* $(Y_0, Y_1, Y_2, g_1, g_2)$ *is a commutative diagram of the following form.*

$$\begin{array}{ccc} X_2 & \xrightarrow{h_2} & Y_2 \\ \downarrow{\scriptstyle f_2} & & \downarrow{\scriptstyle g_2} \\ X_1 & \xrightarrow{h_1} & Y_1 \\ \downarrow{\scriptstyle f_1} & & \downarrow{\scriptstyle g_1} \\ X_0 & \xrightarrow{h_0} & Y_0 \end{array}$$

Remarks.

– Let **3** denote the diagram $0 \to 1 \to 2$ and let **2** denote the diagram $0 \to 1$. The category of trisets is the presheaf category $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$. We refer to the objects of $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$ as *trisets*. We often write $A = (X_0, X_1, X_2)$ for the triset $(X_0, X_1, X_2, f_1, f_2)$. In this case, we further write A_0 for X_0, A_1 for X_1, etc.
– In analogy with trisets, we call objects in the presheaf category $\mathbf{Set}^{\mathbf{2}^{\mathrm{op}}}$ *bisets*. Objects in $\mathbf{Set}^{\mathbf{2}^{\mathrm{op}}}$ are of the form $(X_0, X_1, f : X_1 \to X_0)$ and morphisms in $\mathbf{Set}^{\mathbf{2}^{\mathrm{op}}}$ are commutative squares.
– Since $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$ is a presheaf category, it is cartesian closed, and thus self-enriched. That is, the hom-object $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}(A, B)$ in $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$ is the exponential $A \Rightarrow B$, where $A \Rightarrow B$ is the triset given by $(A \Rightarrow B)_0 = \mathbf{Set}(A_0, B_0)$, $(A \Rightarrow B)_1 = \mathbf{Set}^{\mathbf{2}^{\mathrm{op}}}((A_0, A_1), (B_0, B_1))$, and $(A \Rightarrow B)_2 = \mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}(A, B)$.
– We often write $\mathcal{V}_3$ for $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$, $\mathcal{V}_2$ for $\mathbf{Set}^{\mathbf{2}^{\mathrm{op}}}$, and $\mathcal{V}_1$ for **Set**.
– We write $|\mathcal{V}_3|(A, B)$ for the *set* of morphisms from the triset A to the triset B, to distinguish it from the *triset* of morphisms, which is written $\mathcal{V}_3(A, B)$.
– The category $\mathcal{V}_2$ is also $\mathcal{V}_3$ enriched, since $\mathcal{V}_2(A, B)$ corresponds to the triset

$$(\mathcal{V}_2(A, B)_0, \mathcal{V}_2(A, B)_1, \mathcal{V}_2(A, B)_1).$$

Similarly, **Set** is $\mathcal{V}_3$-enriched because $\mathbf{Set}(A, B)$ corresponds to the triset

$$(\mathbf{Set}(A, B), \mathbf{Set}(A, B), \mathbf{Set}(A, B)).$$

Definition 12. *We define the following functors.*

– $U_0 : \mathbf{Set}^{\mathbf{3}^{\mathrm{op}}} \to \mathbf{Set}$*, defined by* $U_0(X_0, X_1, X_2) = X_0$.
– $\Delta_0 : \mathbf{Set} \to \mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$*, defined by* $\Delta_0(X) = (X_0, X_0, X_0)$.
– $U_1 : \mathbf{Set}^{\mathbf{3}^{\mathrm{op}}} \to \mathbf{Set}^{\mathbf{2}^{\mathrm{op}}}$*, defined by* $U_1(X_0, X_1, X_2) = (X_0, X_1)$.
– $\Delta_1 : \mathbf{Set}^{\mathbf{2}^{\mathrm{op}}} \to \mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$*, defined by* $\Delta_1(X_0, X_1) = (X_0, X_1, X_1)$.

We have adjunctions $U_0 \dashv \Delta_0$ and $U_1 \dashv \Delta_1$. These adjunctions give rise to monads $T_0 = \Delta_0 U_0$ and $T_1 = \Delta_1 U_1$ on $\mathbf{Set}^{\mathbf{3}^{\mathrm{op}}}$. The action of these monads on objects is given below.

$$T_1(X_0, X_1, X_2, f_1, f_2) = (X_0, X_1, X_1, f_1, \mathrm{id}).$$

$$T_0(X_0, X_1, X_2, f_1, f_2) = (X_0, X_0, X_0, \mathrm{id}, \mathrm{id}).$$

It is easy to verify that both T_0 and T_1 are idempotent monads and satisfy $T_1T_0 \overset{\eta^{T_1}}{\cong} T_0 \overset{T_0\eta^{T_1}}{\cong} T_0T_1$.

Definition 13. *We define the triset-enriched category* $\mathbf{D}$ *by:*

- $\mathrm{obj}(\mathbf{D}) = \mathrm{obj}(\mathbf{M}) = \mathrm{obj}(\mathbf{R}) = \mathrm{obj}(\mathbf{N})$.
- *For any* $A, B \in \mathbf{D}$*, the hom-object* $\mathbf{D}(A, B)$ *is the following triset, where* $G : \mathbf{N} \to \mathbf{R}$ *and* $I : \mathbf{R} \to \mathbf{M}$ *are inclusion functors.*

$$\begin{array}{c} \mathbf{N}(A,B) \\ \big\downarrow{\scriptstyle G_{AB}} \\ \mathbf{R}(A,B) \\ \big\downarrow{\scriptstyle I_{AB}} \\ \mathbf{M}(A,B) \end{array}$$

The enriched category $\mathbf{D}$ is symmetric monoidal, but not closed. For this reason, we define the following category $\overline{\mathbf{D}}$ via the enriched Yoneda embedding.

Definition 14. *We define* $\overline{\mathbf{D}}$ *to be the triset-enriched functor category* $\mathcal{V}_3^{\mathbf{D}^{\mathrm{op}}}$.

The monads T_1, T_0 and the functors $U_0, U_1, \varDelta_0, \varDelta_1$ are $\mathcal{V}_3$-enriched. Hence, they can all be lifted to $\overline{\mathbf{D}}$.

Definition 15. *We define the following* $\mathcal{V}_3$*-enriched functors.*

$$\begin{array}{ll}
\overline{T}_0 : \overline{\mathbf{D}} \to \overline{\mathbf{D}} & \text{defined by } \overline{T}_0(F) = T_0 \circ F, \\
\overline{T}_1 : \overline{\mathbf{D}} \to \overline{\mathbf{D}} & \text{defined by } \overline{T}_1(F) = T_1 \circ F, \\
\overline{U}_0 : \overline{\mathbf{D}} \to \mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}} & \text{defined by } \overline{U}_0(F) = U_0 \circ F, \\
\overline{U}_1 : \overline{\mathbf{D}} \to \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}} & \text{defined by } \overline{U}_1(F) = U_1 \circ F, \\
\overline{\varDelta}_0 : \mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}} \to \overline{\mathbf{D}} & \text{defined by } \overline{\varDelta}_0(F) = \varDelta_0 \circ F, \\
\overline{\varDelta}_1 : \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}} \to \overline{\mathbf{D}} & \text{defined by } \overline{\varDelta}_1(F) = \varDelta_1 \circ F.
\end{array}$$

Note that $\overline{U}_0 \dashv \overline{\varDelta}_0$, $\overline{U}_1 \dashv \overline{\varDelta}_1$, $\overline{T}_0 = \overline{\varDelta}_0\overline{U}_0$, and $\overline{T}_1 = \overline{\varDelta}_1\overline{U}_1$.

Definition 16. *We define* $\mathbf{C}$ *to be the biset-enriched category whose objects are those of* $\mathbf{M}$ *and whose hom-objects are given, for any* $A, B \in \mathbf{C}$*, by the biset* $\mathbf{C}(A, B) = (\mathbf{M}(A, B), \mathbf{R}(A, B), j_{AB})$*. We further define the biset-enriched functor category* $\overline{\mathbf{C}} = \mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}}$.

Consider a $\mathcal{V}_3$-functor $H : \mathbf{D}^{\mathrm{op}} \to \mathcal{V}_3$. For every object $A \in \mathbf{D}$, there is an object $H(A) \in \mathcal{V}_3$, and for every $A, B \in \mathbf{D}$, there is the following morphism in

$\mathcal{V}_3$.

$$\begin{array}{ccc} \mathbf{N}(B,A) & \xrightarrow{H_2} & \mathcal{V}_3(HA,HB)_2 \\ \downarrow & & \downarrow \\ \mathbf{R}(B,A) & \xrightarrow{H_1} & \mathcal{V}_3(HA,HB)_1 \\ \downarrow & & \downarrow \\ \mathbf{M}(B,A) & \xrightarrow{H_0} & \mathcal{V}_3(HA,HB)_0 \end{array}$$

Therefore, H induces a $\mathcal{V}_2$-functor $H^1 : \mathbf{C}^{\mathrm{op}} \to \mathcal{V}_2$ such that $H^1 A = (HA)_1 \in \mathcal{V}_2$ and $H^1_{AB} = (H_0, H_1) : \mathbf{C}(B,A) \to \mathcal{V}_2(H^1 A, H^1 B)$. Similarly, H induces a $\mathcal{V}_1$-functor $H^0 : \mathbf{M}^{\mathrm{op}} \to \mathcal{V}_1$ such that $H^0 A = (HA)_0$ and $H^0_{AB} = H_0 : \mathbf{M}(B,A) \to \mathcal{V}_1(H^0 A, H^0 B)$.

Theorem 4. *(a) The $\mathcal{V}_3$-monads $\overline{T}_0, \overline{T}_1$ are idempotent, strong commutative monads, and*

$$\overline{T}_0 \circ \overline{T}_1 \overset{(\overline{T}_0 \eta^{\overline{T}_1})^{-1}}{\cong} \overline{T}_0 \overset{\eta^{\overline{T}_1}}{\cong} \overline{T}_1 \circ \overline{T}_0.$$

(b) $\mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}} \cong \mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}}$.
(c) $\mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}} \cong \mathcal{V}_1^{\mathbf{M}^{\mathrm{op}}}$.

Proof. (**a**) By the definition of $\overline{T}_0$ and $\overline{T}_1$, the fact that T_0 and T_1 are idempotent, and the fact that $T_1 T_0 \cong T_0 \cong T_0 T_1$. The proof of commutative strength is similar to the one in [8].

(**b**) We first define a $\mathcal{V}_3$-functor $\Omega : \mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}} \to \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}}$. Let $F \in \mathbf{C}^{\mathrm{op}} \to \mathcal{V}_2$. We can define $\Omega(F) = \hat{F} \in \mathbf{D}^{\mathrm{op}} \to \mathcal{V}_2$, where $\hat{F}(A) = F(A)$ and $\hat{F}_{AB} : \mathbf{D}(B,A) \to \mathcal{V}_2(F(A), F(B))$ is given by $F_{AB} : \mathbf{C}(B,A) \to \mathcal{V}_2(F(A), F(B))$ via the following diagram.

$$\begin{array}{ccc} \mathbf{N}(B,A) & \xrightarrow{F^1_{AB} \circ G_{BA}} & \mathcal{V}_2(FA,FB)_1 \\ \downarrow{\scriptstyle G_{BA}} & & \downarrow{\scriptstyle \mathrm{id}} \\ \mathbf{R}(B,A) & \xrightarrow{F^1_{AB}} & \mathcal{V}_2(FA,FB)_1 \\ \downarrow & & \downarrow \\ \mathbf{M}(B,A) & \xrightarrow{F^0_{AB}} & \mathcal{V}_2(FA,FB)_0 \end{array}$$

For all $F, H : \mathbf{C}^{\mathrm{op}} \to \mathcal{V}_2$, and for all $A \in \mathbf{C}$(or $\mathbf{D}$), we have $\mathcal{V}_2(FA, HA) = \mathcal{V}_2(\hat{F}(A), \hat{H}(A))$, which induces a map $\Omega_{FH} : \mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}}(F,H) \to \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}}(\hat{F}, \hat{H})$. We now define the $\mathcal{V}_3$-functor $\Omega^{-1} : \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}} \to \mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}}$. For any $F \in \mathbf{D}^{\mathrm{op}} \to \mathcal{V}_2$. We can define $\Omega^{-1}(F) = F^1 \in \mathbf{C}^{\mathrm{op}} \to \mathcal{V}_2$. For any $A \in \mathbf{C}$(or $\mathbf{D}$), we have $\mathcal{V}_2(FA, HA) \cong \mathcal{V}_2(F^1 A, H^1 A)$. Therefore there is a map $\Omega^{-1}_{FH} : \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}}(F,H) \to \mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}}(F^1, H^1)$.

(**c**) Similar to the argument in (b).

The following theorem shows that Kleisli morphisms in $\overline{\mathbf{D}}$ correspond to morphisms in $\mathcal{V}_2^{\mathbf{C}^{\mathrm{op}}}$ and $\mathcal{V}_1^{\mathbf{M}^{\mathrm{op}}}$.

Theorem 5. *(a) For any $H, L \in \overline{\mathbf{D}}$, we have $\overline{\mathbf{D}}(H, \overline{T}_1 L) \cong \mathcal{V}_2^{\mathbf{C}^{op}}(H^1, L^1)$.*
(b) For any $H, L \in \overline{\mathbf{D}}$, we have $\overline{\mathbf{D}}(H, \overline{T}_0 L) \cong \mathcal{V}_1^{\mathbf{M}^{op}}(H^0, L^0)$.

Proof. (**a**) $\overline{\mathbf{D}}(H, \overline{T}_1 L) = \overline{\mathbf{D}}(H, \overline{\Delta}_1 \overline{U}_1 L) \cong \mathcal{V}_2^{\mathbf{D}^{op}}(\overline{U}_1 H, \overline{U}_1 L) \cong \mathcal{V}_2^{\mathbf{C}^{op}}(H^1, L^1)$.
(**b**) $\overline{\mathbf{D}}(H, \overline{T}_0 L) = \overline{\mathbf{D}}(H, \overline{\Delta}_0 \overline{U}_0 L) \cong \mathcal{V}_1^{\mathbf{D}^{op}}(\overline{U}_0 H, \overline{U}_0 L) \cong \mathcal{V}_1^{\mathbf{M}^{op}}(H^0, L^0)$.

We write $V(\overline{\mathbf{D}})$ for the underlying (ordinary) category of $\overline{\mathbf{D}}$. The objects of $V(\overline{\mathbf{D}})$ are the same as those of $\overline{\mathbf{D}}$. A morphism $f : A \to B \in V(\overline{\mathbf{D}})$ is an element in the set $|\mathcal{V}_3|(1, \overline{\mathbf{D}}(A, B))$. Similarly, for a V_3-functor such as $\overline{T}_0 : \overline{\mathbf{D}} \to \overline{\mathbf{D}}$, we write $V\overline{T}_0 : V(\overline{\mathbf{D}}) \to V(\overline{\mathbf{D}})$ for the underlying functor of $\overline{T}_0$.

Theorem 6. *There are strong monoidal embedding functors $\phi_2 : \mathbf{N} \hookrightarrow V(\overline{\mathbf{D}}), \phi_1 : \mathbf{R} \hookrightarrow Kl_{V\overline{T}_1} V(\overline{\mathbf{D}}), \phi_0 : \mathbf{M} \hookrightarrow Kl_{V\overline{T}_0} V(\overline{\mathbf{D}})$ such that the following diagram commutes.*

$$\begin{array}{ccc}
\mathbf{N} & \xhookrightarrow{\phi_2} & V(\overline{\mathbf{D}}) \\
\downarrow G & & \downarrow E_1 \\
\mathbf{R} & \xhookrightarrow{\phi_1} & Kl_{V\overline{T}_1}(V(\overline{\mathbf{D}})) \\
\downarrow I & & \downarrow E_0 \\
\mathbf{M} & \xhookrightarrow{\phi_0} & Kl_{V\overline{T}_0}(V(\overline{\mathbf{D}}))
\end{array}$$

Note that $E_1(A) = A, E_1(f) = \eta^{V\overline{T}_1} \circ f$, and $E_0(A) = A, E_0(f) = \eta^{V\overline{T}_0} \circ f$.

Proof. See Appendix C

We are now ready the state the following main theorem.

Theorem 7. *The category $\mathbf{A} = V(\overline{\mathbf{D}})$ is a model for Proto-Quipper with reversing and control in the sense of Definition 3 (a)-(e).*

Proof. (**a**) $V(\overline{\mathbf{D}})$ is symmetric monoidal closed because of $\overline{\mathbf{D}}$, where tensor products are given by Day's convolution [5].
(**b**) $V(\overline{\mathbf{D}})$ has coproducts because $\overline{\mathbf{D}}$ is the enriched Yoneda embedding of $\mathbf{D}$.
(**c**) We define $p(X) = \sum_{x \in X} I : \mathbf{Set} \to V(\overline{\mathbf{D}})$ and $\flat(A) = V(\overline{\mathbf{D}})(I, A) : V(\overline{\mathbf{D}}) \to \mathbf{Set}$. We then have the adjunction $p \dashv \flat$ because

$$\begin{aligned}
V(\overline{\mathbf{D}})(pX, B) = \mathcal{V}_3(1, \overline{\mathbf{D}}(pX, B)) &\cong \mathcal{V}_3(1, \overline{\mathbf{D}}(\sum_{x \in X} I, B)) \\
&\cong \mathcal{V}_3(1, \prod_{x \in X} \overline{\mathbf{D}}(I, B)) \cong \prod_{x \in X} \mathcal{V}_3(1, \overline{\mathbf{D}}(I, B)) \\
&\cong \mathbf{Set}(X, V(\overline{\mathbf{D}})(I, B)) = \mathbf{Set}(X, \flat(B))
\end{aligned}$$

(**d**) By Theorem 4 (a).
(**e**) By Theorem 6.

7 Conclusion

In this paper, we showed how to extend Proto-Quipper with reversing, control, and the with-computed operation. Our language is parameterized by three categories $\mathbf{M}$, $\mathbf{R}$, and $\mathbf{N}$, which correspond to general quantum circuits, reversible circuits, and controllable circuits, respectively. We defined a type system that uses modalities to distinguish the different types of circuits. We provided an operational and a denotational semantics for the language; the latter takes the form of an abstract categorical model in which our modalities are represented by monads. We proved that the operational semantics is sound with respect to the categorical model. We also constructed a concrete categorical model from the given categories $\mathbf{N}, \mathbf{R}$ and $\mathbf{M}$, using triset-enriched categories. Lastly, we gave some examples of reversing, control and with-computed in Proto-Quipper.

There are many possible directions for future work. For example, in this paper, we only considered the modalities of reversibility and controllability. But it seems that our construction of the type system and its categorical semantics would easily generalize from a three element set to an arbitrary poset of modalities. Introducing additional modalities might be useful for characterizing additional properties of gates. Although we have implemented a type inference algorithm, we did not formally study its properties. Another challenge for future work is to combine modalities with dependent types. Although our software implementation does support both modalities and dependent types, we have not considered a formal semantics for combining them.

References

1. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development **17**(6), 525–532 (1973)
2. Bichsel, B., Baader, M., Gehr, T., Vechev, M.: 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. p. 286–300. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). `https://doi.org/10.1145/3385412.3386007`, `https://doi.org/10.1145/3385412.3386007`
3. Borceux, F.: Handbook of Categorical Algebra, Volume 2: Categories and Structures. Cambridge University Press (1994)
4. Colledan, A., Dal Lago, U.: On dynamic lifting and effect typing in circuit description languages (extended version) (2022), available from `arXiv:2202.07636`
5. Day, B.: On closed categories of functors. In: Reports of the Midwest Category Seminar IV. Springer Lecture Notes in Mathematics, vol. 137, pp. 1–38 (1970)
6. Draper, T.G.: Addition on a quantum computer. arXiv preprint quant-ph/0008033 (2000)
7. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In: Proceedings of the 12th International Conference on Reversible Computation, RC 2020, Oslo, Norway. Lecture Notes in Computer Science, vol. 12227, pp. 153–168. Springer (2020). `https://doi.org/10.1007/978-3-030-52482-1_9`, also available from `arXiv:2005.08396`

8. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A biset-enriched categorical model for Proto-Quipper with dynamic lifting. In: Gogioso, S., Hoban, M. (eds.) Proceedings 19th International Conference on Quantum Physics and Logic, QPL 2022, Wolfson College, Oxford, UK, 27 June - 1 July 2022. EPTCS, vol. 394, pp. 302–342 (2022). https://doi.org/10.4204/EPTCS.394.16, https://doi.org/10.4204/EPTCS.394.16
9. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: Proto-Quipper with dynamic lifting. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). https://doi.org/10.1145/3571204, https://doi.org/10.1145/3571204
10. Fu, P., Kishida, K., Selinger, P.: Linear dependent type theory for quantum programming languages. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2020, Saarbrücken, Germany. pp. 440–453 (2020). https://doi.org/10.1145/3373718.3394765, also available from arXiv:2004.13472
11. Google: Cirq Software Reference (Accessed October 9, 2024), https://quantumai.google/reference/python/cirq/ControlledOperation
12. Green, A., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in Quipper. In: Proceedings of the 5th International Conference on Reversible Computation, RC 2013, Victoria, British Columbia. Lecture Notes in Computer Science, vol. 7948, pp. 110–124. Springer (2013). https://doi.org/10.1007/978-3-642-38986-3_10, also available from arXiv:1304.5485
13. Green, A., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle. ACM SIGPLAN Notices, vol. 48(6), pp. 333–342 (Jun 2013). https://doi.org/10.1145/2499370.2462177, also available from arXiv:1304.3390
14. Häner, T., Steiger, D.S., Svore, K., Troyer, M.: A software methodology for compiling quantum programs. Quantum Science and Technology **3**(2), 020501 (2018)
15. IBM: IBM quantum documentation, QuantumCircuit class (Accessed October 9, 2024), https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.QuantumCircuit#control
16. Kelly, G.M.: Basic concepts of enriched category theory, Lecture Notes of the London Mathematical Society, vol. 64. Cambridge University Press (1982)
17. Lee, D., Perrelle, V., Valiron, B., Xu, Z.: Concrete categorical model of a quantum circuit description language with measurement. In: Bojanczyk, M., Chekuri, C. (eds.) 41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021. LIPIcs, vol. 213, pp. 51:1–51:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.FSTTCS.2021.51
18. Lindenhovius, B., Mislove, M., Zamdzhiev, V.: Enriching a linear/non-linear lambda calculus: A programming language for string diagrams. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. p. 659–668. LICS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209196, https://doi.org/10.1145/3209108.3209196
19. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2002)
20. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: a core language for quantum circuits. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of

Programming Languages. ACM SIGPLAN Notices, vol. 52, pp. 846–858. ACM (2017)
21. Rennela, M., Staton, S.: Classical control, quantum circuits and linear logic in enriched category theory. Logical Methods in Computer Science **16**(1), 30:1–24 (2020). `https://doi.org/10.23638/LMCS-16(1:30)2020`
22. Rios, F., Selinger, P.: A categorical model for a quantum circuit description language. Extended abstract. In: Proceedings of the 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen. Electronic Proceedings in Theoretical Computer Science, vol. 266, pp. 164–178 (2018). `https://doi.org/10.4204/EPTCS.266.11`, also available from `arXiv:1706.02630`
23. Ross, N.J.: Algebraic and logical methods in quantum computation. Ph.D. thesis, Dalhousie University, Department of Mathematics and Statistics (2015), available from `arXiv:1510.02198`
24. Selinger, P.: Dagger compact closed categories and completely positive maps. Electronic Notes in Theoretical computer science **170**, 139–163 (2007)
25. Selinger, P.: Quantum circuits of T-depth one. Physical Review A **87**(4), 042302 (2013)
26. Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: an open source software framework for quantum computing. Quantum **2**, 49 (2018)

A Background on enriched categories

Definition 17. *Let $\mathcal{V}$ be a monoidal category. A $\mathcal{V}$-enriched category* $\mathbf{B}$ *is given by the following:*

- *A class of objects, also denoted* $\mathbf{B}$.
- *For any $A, B \in \mathbf{B}$, an object $\mathbf{B}(A, B)$ in $\mathcal{V}$.*
- *For any $A \in \mathbf{B}$, a morphism $u_A : I \to \mathbf{B}(A, A)$ in $\mathcal{V}$, called the* identity *on A.*
- *For any $A, B, C \in \mathbf{B}$, a morphism $c_{A,B,C} : \mathbf{B}(A, B) \otimes \mathbf{B}(B, C) \to \mathbf{B}(A, C)$ in $\mathcal{V}$, called* composition.
- *The composition and identity morphisms must satisfy suitable diagrams in $\mathcal{V}$ (see e.g., [16] and [3]).*

Remarks.

- Many concepts from the theory of non-enriched categories can be generalized to the enriched setting. For example, $\mathcal{V}$-functors, $\mathcal{V}$-natural transformations, $\mathcal{V}$-adjunctions, and the $\mathcal{V}$-Yoneda embedding are all straightforward generalizations of their non-enriched counterparts. We refer the reader to [16] and [3] for comprehensive introductions. Symmetric monoidal categories can also be generalized to the enriched setting.
- When we speak of a map $f : A \to B$ in a $\mathcal{V}$-enriched category $\mathbf{B}$, we mean a morphism of the form $f : I \to \mathbf{B}(A, B)$ in $\mathcal{V}$. Furthermore, when $g : B \to C$ is also a map in $\mathbf{B}$, we write $g \circ f : A \to C$ as a shorthand for

$$I \xrightarrow{f \otimes g} \mathbf{B}(A, B) \otimes \mathbf{B}(B, C) \xrightarrow{c} \mathbf{B}(A, C).$$

- A $\mathcal{V}$-enriched category $\mathbf{B}$ gives rise to an ordinary (non-enriched) category $V(\mathbf{B})$, called the *underlying category* of $\mathbf{B}$. The objects of $V(\mathbf{B})$ are the objects of $\mathbf{B}$ and the hom-sets of $V(\mathbf{B})$ are defined as $V(\mathbf{B})(A, B) = \mathcal{V}(I, \mathbf{B}(A, B))$, for any $A, B \in V(\mathbf{B})$. Similarly, a $\mathcal{V}$-functor $F : \mathbf{B} \to \mathbf{B}$ gives rise to a functor $VF : V(\mathbf{B}) \to V(\mathbf{B})$ and a $\mathcal{V}$-natural transformation $\alpha : F \to G$ gives rise to a natural transformation $V\alpha : VF \to VG$.

B Proof of Theorem 3

Theorem 8 (Soundness of the evaluation). *If $\vdash_{\alpha \wedge \beta} (C, M) : A; \Sigma'$ and $(C, M) \Downarrow (C', V)$, then $[\![(C, M)]\!] = [\![(C', V)]\!]$.*

Proof. – Case

$$\frac{\begin{array}{c}(C, M) \Downarrow (C', \mathsf{lift}\ M') \\ \mathrm{gen}(S) = (a, \Sigma'') \\ ([\![a]\!]^\dagger, M'\ a) \Downarrow (D, b)\end{array}}{(C, \mathsf{box}_S M) \Downarrow (C', \mathsf{circ}([\![b]\!] \circ D))}\ box$$

Suppose $\vdash_2 (C, \mathsf{box}_S M) : \mathbf{Circ}_0(S, U); \Sigma'$. This implies that $C \in \mathbf{N}([\![S]\!], [\![\Sigma_1']\!] \otimes [\![\Sigma_2']\!])$ and $\Sigma_1' \vdash_2 \mathsf{box}_S M : \mathbf{Circ}_0(S, U)$. Thus $\Sigma_1' \vdash_2 M : !_0(S \multimap_2 U)$,

$C' \in \mathbf{N}([\![S]\!], [\![\Sigma_2']\!])$, and $\vdash_0 M' : S \multimap_2 U$. By the induction hypothesis, $[\![(C, M)]\!] = [\![(C', \mathsf{lift}\, M')]\!]$ and $[\![([\![a]\!]^\dagger, M'a)]\!] = [\![(D, b)]\!]$. Thus we have

$$[\![S]\!] \xrightarrow{C} [\![\Sigma_1']\!] \otimes [\![\Sigma_2']\!] \xrightarrow{[\![M]\!] \otimes [\![\Sigma_2']\!]} p\flat T_0([\![S]\!] \multimap [\![U]\!]) \otimes [\![\Sigma_2']\!]$$

$$= [\![S]\!] \xrightarrow{C'} I \otimes [\![\Sigma_2']\!] \xrightarrow{p\delta[\![M']\!] \otimes [\![\Sigma_2']\!]} p\flat T_0([\![S]\!] \multimap [\![U]\!]) \otimes [\![\Sigma_2']\!]$$

and

$$I \otimes [\![S]\!] \xrightarrow{I \otimes [\![a]\!]^\dagger} I \otimes [\![\Sigma'']\!] \xrightarrow{[\![M']\!] \otimes [\![a]\!]} T_0([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!]$$
$$\xrightarrow{s} T_0([\![S]\!] \multimap [\![U]\!] \otimes [\![S]\!]) \xrightarrow{T_0 \epsilon} T_0[\![U]\!]$$

$=$

$$I \otimes [\![S]\!] \xrightarrow{[\![M']\!] \otimes [\![S]\!]} T_0([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!]$$
$$\xrightarrow{s} T_0([\![S]\!] \multimap [\![U]\!] \otimes [\![S]\!]) \xrightarrow{T_0 \epsilon} T_0[\![U]\!]$$

$\overset{(*)}{=}$

$$[\![S]\!] \xrightarrow{D} T_0[\![\Sigma_3]\!] \xrightarrow{T_0[\![b]\!]} T_0[\![U]\!].$$

Moreover, $[\![C, \mathsf{box}_S M]\!] =$

$$[\![S]\!] \xrightarrow{C} [\![\Sigma_1']\!] \otimes [\![\Sigma_2']\!] \xrightarrow{[\![M]\!] \otimes [\![\Sigma_2']\!]} p\flat T_0([\![S]\!] \multimap [\![U]\!]) \otimes [\![\Sigma_2']\!]$$
$$\xrightarrow{p\mathsf{box}} p\mathbf{M}([\![S]\!], [\![U]\!]) \otimes [\![\Sigma_2']\!]$$

$=$

$$[\![S]\!] \xrightarrow{C'} I \otimes [\![\Sigma_2']\!] \xrightarrow{p\delta[\![M']\!] \otimes [\![\Sigma_2']\!]} p\flat T_0([\![S]\!] \multimap [\![U]\!]) \otimes [\![\Sigma_2']\!]$$
$$\xrightarrow{p\mathsf{box}} p\mathbf{M}([\![S]\!], [\![U]\!]) \otimes [\![\Sigma_2']\!].$$

So we just need to show

$$I \xrightarrow{p\delta[\![M']\!]} p\flat T_0([\![S]\!] \multimap [\![U]\!]) \xrightarrow{p\mathsf{box}} p\mathbf{M}([\![S]\!], [\![U]\!]) = I \xrightarrow{p1_{[\![b]\!] \circ D}} p\mathbf{M}([\![S]\!], [\![U]\!]).$$

In other words, we need to show $[\![b]\!] \circ D = \mathsf{box}(\delta[\![M']\!])$. It suffices to show they are equal in $\mathbf{A}$. This is true by definition of box and $(*)$.

– Case

$$\frac{\begin{array}{c}(C_1, M) \Downarrow (C_2, \mathsf{circ}(D)) \\ (C_2, N) \Downarrow (C_3, V) \\ \mathrm{gen}(\mathrm{codomain}(D)) = (b, \Sigma''') \\ C' = \mathrm{append}(C_3, D, V, b)\end{array}}{(C_1, \mathsf{apply}(M, N)) \Downarrow (C', b)} \; \mathit{apply}$$

Suppose $\Sigma \vdash_0 (C_1, \mathsf{apply}(M,N)) : U; \Sigma'$, where $C_1 \in \mathbf{R}([\![S]\!], [\![\Sigma'']\!] \otimes [\![\Sigma']\!])$, $\Sigma'' \vdash_0 \mathsf{apply}(M,N) : U$, $\Sigma''_1 \vdash_0 M : \mathbf{Circ}_1(S,U)$ and $\Sigma''_2 \vdash_1 N : S$ and $\Sigma'' = \Sigma''_1 \otimes \Sigma''_2$. By the induction hypothesis, we know that $[\![(C_1, M)]\!] = [\![(C_2, \mathsf{circ}(D))]\!]$ and $[\![(C_2, N)]\!] = [\![(C_3, V)]\!]$. Thus

$$\begin{aligned} [\![S]\!] &\xrightarrow{C_1} T_1([\![\Sigma''_1]\!] \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_1([\![M]\!] \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!])} T_1(T_0 p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_1 s} T_1 T_0(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{\cong} T_0(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \end{aligned}$$

$=$

$$[\![S]\!] \xrightarrow{C_2} T_0([\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0(p1_D \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!])$$

and

$$\begin{aligned} [\![S]\!] &\xrightarrow{C_2} T_0([\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![N]\!] \otimes [\![\Sigma']\!])} T_0(T_1[\![S]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_0 s} T_0 T_1([\![S]\!] \otimes [\![\Sigma']\!]) \xrightarrow{\cong} T_0([\![S]\!] \otimes [\![\Sigma']\!]) \end{aligned}$$

$=$

$$[\![S]\!] \xrightarrow{C_3} T_0([\![\Sigma'''_2]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![V]\!] \otimes [\![\Sigma']\!])} T_0([\![S]\!] \otimes [\![\Sigma']\!]).$$

We want to show that $[\![(C_1, \mathsf{apply}(M,N))]\!] = [\![(C', b)]\!]$, where $C' = \mathrm{append}(C_3, D, V, b)$ is the following morphism in $\mathbf{M}$.

$$\begin{aligned} [\![S]\!] &\xrightarrow{C_3} [\![\Sigma'''_2]\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![V]\!] \otimes [\![\Sigma']\!]} [\![S]\!] \otimes [\![\Sigma']\!] \\ &\xrightarrow{G(D) \otimes [\![\Sigma']\!]} [\![U]\!] \otimes [\![\Sigma']\!] \xrightarrow{[\![b]\!]^\dagger \otimes [\![\Sigma']\!]} [\![\Sigma''']\!] \otimes [\![\Sigma']\!]. \end{aligned}$$

Since $\mathbf{M} \hookrightarrow Kl_{T_0}(\mathbf{A})$ and $\mathbf{N} \hookrightarrow \mathbf{A}$, the corresponding morphism in A is

$$\begin{aligned} [\![S]\!] &\xrightarrow{C_3} T_0([\![\Sigma'''_2]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![V]\!] \otimes [\![\Sigma']\!])} T_0([\![S]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_0(D \otimes [\![\Sigma']\!])} T_0([\![U]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![b]\!]^\dagger \otimes [\![\Sigma']\!])} T_0([\![\Sigma''']\!] \otimes [\![\Sigma']\!]). \end{aligned}$$

We have $RHS =$

$$[\![S]\!] \xrightarrow{C_3} T_0([\![\Sigma'''_2]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0([\![V]\!] \otimes [\![\Sigma']\!])} T_0([\![S]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0(D \otimes [\![\Sigma']\!])} T_0([\![U]\!] \otimes [\![\Sigma']\!])$$

and $LHS =$

$$\begin{aligned} [\![S]\!] &\xrightarrow{C_1} T_1([\![\Sigma''_1]\!] \otimes [\![\Sigma''_2]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_1([\![M]\!] \otimes [\![N]\!] \otimes [\![\Sigma']\!])} T_1(T_0 p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes T_1[\![S]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_1 s} T_1 T_0(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes T_1[\![S]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0 s} T_0 T_1(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_0(\mathsf{unbox} \otimes [\![S]\!] \otimes [\![\Sigma']\!])} T_0(p\flat([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!]) \\ &\xrightarrow{T_0(\mathsf{force} \otimes [\![S]\!] \otimes [\![\Sigma']\!])} T_0(([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0(\epsilon \otimes [\![\Sigma']\!])} T_0([\![U]\!] \otimes [\![\Sigma']\!]) \end{aligned}$$

=

$$[\![S]\!] \xrightarrow{C_2} T_0([\![\Sigma_2'']\!] \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(p1_D \otimes [\![N]\!] \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes T_1[\![S]\!] \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0 s} T_0T_1(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(\mathsf{unbox} \otimes [\![S]\!] \otimes [\![\Sigma']\!])} T_0(p\flat([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(\mathsf{force} \otimes [\![S]\!] \otimes [\![\Sigma']\!])} T_0(([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0(\epsilon \otimes [\![\Sigma']\!])} T_0([\![U]\!] \otimes [\![\Sigma']\!])$$

=

$$[\![S]\!] \xrightarrow{C_3} T_0([\![\Sigma_2''']\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0(p1_D \otimes [\![V]\!] \otimes [\![\Sigma']\!])} T_0(p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(\mathsf{unbox} \otimes [\![S]\!] \otimes [\![\Sigma']\!])} T_0(p\flat([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!])$$
$$\xrightarrow{T_0(\mathsf{force} \otimes [\![S]\!] \otimes [\![\Sigma']\!])} T_0(([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \otimes [\![\Sigma']\!]) \xrightarrow{T_0(\epsilon \otimes [\![\Sigma']\!])} T_0([\![U]\!] \otimes [\![\Sigma']\!]).$$

So we just need to show

$$[\![S]\!] \xrightarrow{D} [\![U]\!]$$

=

$$I \otimes [\![S]\!] \xrightarrow{1_D} p\mathbf{N}([\![S]\!], [\![U]\!]) \otimes [\![S]\!] \xrightarrow{p\,\mathsf{unbox} \otimes [\![S]\!]} p\flat([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!]$$
$$\xrightarrow{\mathsf{force} \otimes [\![S]\!]} ([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \xrightarrow{\epsilon} [\![U]\!].$$

This is true because $LHS =$

$$I \otimes [\![S]\!] \xrightarrow{\mathrm{curry}(D) \otimes [\![S]\!]} ([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \xrightarrow{\epsilon} [\![U]\!]$$

and $RHS =$

$$I \otimes [\![S]\!] \xrightarrow{p(\delta \mathsf{curry}(D)) \otimes [\![S]\!]} p\flat([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!]$$
$$\xrightarrow{\mathsf{force} \otimes [\![S]\!]} ([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!] \xrightarrow{\epsilon} [\![U]\!]$$

and we have $\mathsf{force} \circ p\delta = \mathrm{id}$.

C Proof of Theorem 6

Theorem 9. *There are strong monoidal embedding functors $\phi_2 : \mathbf{N} \hookrightarrow V(\overline{\mathbf{D}}), \phi_1 : \mathbf{R} \hookrightarrow Kl_{V\overline{T}_1} V(\overline{\mathbf{D}}), \phi_0 : \mathbf{M} \hookrightarrow Kl_{V\overline{T}_0} V(\overline{\mathbf{D}})$ such that the following diagram com-*

mutes.

$$
\begin{array}{ccc}
\mathbf{N} & \overset{\phi_2}{\hookrightarrow} & V(\overline{\mathbf{D}}) \\
\downarrow{\scriptstyle G} & & \downarrow{\scriptstyle E_1} \\
\mathbf{R} & \overset{\phi_1}{\hookrightarrow} & Kl_{V\overline{T}_1}(V(\overline{\mathbf{D}})) \\
\downarrow{\scriptstyle j} & & \downarrow{\scriptstyle E_0} \\
\mathbf{M} & \overset{\phi_0}{\hookrightarrow} & Kl_{V\overline{T}_0}(V(\overline{\mathbf{D}}))
\end{array}
$$

Note that $E_1(A) = A, E_1(f) = \eta^{V\overline{T}_1} \circ f$, *and* $E_0(A) = A, E_0(f) = \eta^{V\overline{T}_0} \circ f$.

Proof. The functor $\phi_2 : \mathbf{N} \hookrightarrow V(\overline{\mathbf{D}})$ is given by

$$\mathbf{N} \overset{\cong}{\to} V(\mathbf{D}) \xrightarrow{Vy_2} V(\overline{\mathbf{D}}),$$

where $y_2 : \mathbf{D} \hookrightarrow \overline{\mathbf{D}}$ is the enriched Yoneda embedding. Since Vy_2 is a strong monoidal functor, ϕ_2 is strong monoidal.

To define the functor $\phi_1 : \mathbf{R} \hookrightarrow Kl_{V\overline{T}_1}(V(\overline{\mathbf{D}}))$, we first define $\phi_1(A) = \mathbf{D}(-, A)$ for any $A \in \mathbf{R}$ on objects. On morphisms, we define $\phi_1 : \mathbf{R}(A, B) \to Kl_{V\overline{T}_1}(V(\overline{\mathbf{D}}))(\phi_1(A), \phi_1(B))$ by the following composition of isomorphisms:

$$
\begin{aligned}
\mathbf{R}(A,B) &\overset{\cong}{\to} V(\mathbf{C})(A,B) \overset{Vy_1}{\to} V(\overline{\mathbf{C}})(y_1 A, y_1 B) \\
&\overset{=}{\to} V(\mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}})(\widehat{y_1 A}, \widehat{y_1 B}) \\
&\overset{=}{\to} V(\mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}})(\overline{U_1}\mathbf{D}(-,A), \overline{U_1}\mathbf{D}(-,B)) \\
&\overset{\cong}{\to} V(\overline{\mathbf{D}})(\mathbf{D}(-,A), \overline{T_1}\mathbf{D}(-,B)) \\
&\overset{=}{\to} Kl_{V\overline{T}_1}(V(\overline{\mathbf{D}}))(\mathbf{D}(-,A), \mathbf{D}(-,B)),
\end{aligned}
$$

where $y_1 : \mathbf{C} \hookrightarrow \overline{\mathbf{C}}$ is the enriched-Yoneda embedding. Here ϕ_1 is strong monoidal because $\overline{U}_1$ and y_1 are strong monoidal.

To define the functor $\phi_0 : \mathbf{M} \hookrightarrow Kl_{V\overline{T}_0}(V(\overline{\mathbf{D}}))$, we first define $\phi_0(A) = \mathbf{D}(-, A)$ on objects. On morphisms, we define $\phi_0 : \mathbf{M}(A, B) \to Kl_{\overline{T}_0}(V(\overline{\mathbf{D}}))(\phi_0(A), \phi_0(B))$ by the following composition of isomorphisms:

$$
\begin{aligned}
\mathbf{M}(A,B) &\overset{y_0}{\to} \mathcal{V}_1^{\mathbf{M}^{\mathrm{op}}}(y_0 A, y_0 B) \\
&\overset{=}{\to} \mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}}(\widehat{y_0 A}, \widehat{y_0 B})) \\
&\overset{=}{\to} \mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}}(\overline{U}_0\mathbf{D}(-,A), \overline{U}_0\mathbf{D}(-,B)) \\
&\overset{\cong}{\to} \mathcal{V}_3^{\mathbf{D}^{\mathrm{op}}}(\mathbf{D}(-,A), \overline{T}_0\mathbf{D}(-,B)) \\
&\overset{=}{\to} Kl_{V\overline{T}_0}(V(\overline{\mathbf{D}}))(\mathbf{D}(-,A), \mathbf{D}(-,B)),
\end{aligned}
$$

where $y_0 : \mathbf{M} \hookrightarrow \mathcal{V}_1^{\mathbf{M}^{\mathrm{op}}}$ is the Yoneda embedding. The functor ϕ_1 is strong monoidal because $\overline{U}_0$ and y_0 are strong monoidal.

Now we need to show that the functor diagram commutes. Since G, j, E_1, and E_0 are identity on objects, we just need to show that the following diagram commutes for any $S, U \in \mathbf{N}$.

$$\begin{array}{ccc}
\mathbf{N}(S,U) & \xrightarrow{\phi_2} & V(\overline{\mathbf{D}})(S,U) \\
\downarrow{\scriptstyle G} & & \downarrow{\scriptstyle E_1} \\
\mathbf{R}(S,U) & \xrightarrow{\phi_1} & Kl_{V\overline{T}_1}(V(\overline{\mathbf{D}}))(S,U) \\
\downarrow{\scriptstyle j} & & \downarrow{\scriptstyle E_0} \\
\mathbf{M}(S,U) & \xrightarrow{\phi_0} & Kl_{V\overline{T}_0}(V(\overline{\mathbf{D}}))(S,U)
\end{array}$$

Let $f \in \mathbf{N}(S,U)$, $\Omega_1 : \overline{\mathbf{C}} \xrightarrow{\cong} \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}}$ (Theorem 4 (b)) and $\theta_1 : \mathcal{V}_2^{\mathbf{D}^{\mathrm{op}}}(\overline{U}_1 F, \overline{U}_1 G) \xrightarrow{\cong} \overline{\mathbf{D}}(F, \overline{T_1} G)$. We write $(f, G(f), jG(f))$ for the corresponding map of f in $V(\mathbf{D})(S,U)$. We have the following:

$$\begin{aligned}
\phi_1^{-1} E_1(\phi_2(f)) &= V y_1^{-1} V \Omega_1^{-1} V \theta_1^{-1} V \eta^{\overline{T}_1} V y_2(f, G(f), jG(f)) \\
&= V y_1^{-1} V \Omega_1^{-1} V \overline{U}_1 V y_2(f, G(f), jG(f)) \\
&= V y_1^{-1} V \Omega_1^{-1} V \overline{U}_1 V \mathbf{D}(-, (f, G(f), jG(f))) \\
&= V y_1^{-1} V \mathbf{C}(-, (G(f), jG(f))) \\
&= G(f).
\end{aligned}$$

Let $g \in \mathbf{R}(S,U)$, $\Omega_0 : \mathcal{V}_1^{\mathbf{M}^{\mathrm{op}}} \xrightarrow{\cong} \mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}}$ (Theorem 4 (c)) and $\theta_0 : \mathcal{V}_1^{\mathbf{D}^{\mathrm{op}}}(\overline{U}_0 F, \overline{U}_0 G) \xrightarrow{\cong} \overline{\mathbf{D}}(F, \overline{T_0} G)$. We write $(g, j(g))$ for the corresponding map of g in $V(\mathbf{C})(S,U)$. We have the following:

$$\begin{aligned}
\phi_0^{-1} E_0 \phi_1(g) &= V y_0^{-1} V \Omega_0^{-1} V \theta_0^{-1} V \eta^{\overline{T}_0} V \theta_1 V \Omega_1 V y_1(g, j(g)) \\
&= V y_0^{-1} V \Omega_0^{-1} V \overline{U}_0 V \theta_1 V \Omega_1 V y_1(g, j(g)) \\
&= V y_0^{-1} V \Omega_0^{-1} V \overline{U}_0 V \theta_1 V \Omega_1 V \mathbf{C}(-, (g, j(g))) \\
&= V y_0^{-1} \mathbf{M}(-, j(g)) \\
&= j(g).
\end{aligned}$$

D Programs for CCZ

```
cnot_circuit : !(Qubit * Qubit * Qubit ->
                   Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit)
cnot_circuit input =
  let (x, y, z) = input
```

```
      t0 = Init0 ()
      t1 = Init0 ()
      t2 = Init0 ()
      t3 = Init0 ()
      (t0, y) = CNot t0 y
      (t0, x) = CNot t0 x
      (t1, z) = CNot t1 z
      (t1, x) = CNot t1 x
      (t2, z) = CNot t2 z
      (t2, y) = CNot t2 y
      (t3, t2) = CNot t3 t2
      (t3, x) = CNot t3 x
  in (x, y, z, t0, t1, t2, t3)

box_cnot_circuit : Circ(Qubit * Qubit * Qubit,
                        Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit)
box_cnot_circuit = boxCirc cnot_circuit

parallel_T : !(Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit ->
                 Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit)
parallel_T input =
  let (a0, a1, a2, a3, a4, a5, a6) = input
  in (TGate a0, TGate a1, TGate a2,
        TGate_Inv a3, TGate_Inv a4, TGate_Inv a5, TGate a6)

box_parallel_T : Circ(Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit,
                       Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit)
box_parallel_T = boxCirc parallel_T

my_ccz : Circ(Qubit * Qubit * Qubit, Qubit * Qubit * Qubit)
my_ccz = withComputed box_cnot_circuit box_parallel_T

cnot_circuit_rev : !(Qubit * Qubit * Qubit * Qubit * Qubit * Qubit * Qubit ->
                      Qubit * Qubit * Qubit)
cnot_circuit_rev = unbox (reverse (boxCirc cnot_circuit))

my_ccz' : Circ(Qubit * Qubit * Qubit, Qubit * Qubit * Qubit)
my_ccz' =
  boxCirc $ \ input ->
              cnot_circuit_rev (parallel_T (cnot_circuit input))

-- The following gives rise to a typing error.
ctrl_my_toffoli' : Circ(Qubit * Qubit * Qubit * Qubit,
                          Qubit * Qubit * Qubit * Qubit)
ctrl_my_toffoli' = control my_ccz'
```