Proto-Quipper with Dynamic Lifting

PENG FU, Dalhousie University, Canada
KOHEI KISHIDA, University of Illinois at Urbana-Champaign, USA
NEIL J. ROSS, Dalhousie University, Canada
PETER SELINGER, Dalhousie University, Canada

Quipper is a functional programming language for quantum computing. Proto-Quipper is a family of languages aiming to provide a formal foundation for Quipper. In this paper, we extend Proto-Quipper-M with a construct called *dynamic lifting*, which is present in Quipper. By virtue of being a circuit description language, Proto-Quipper has two separate runtimes: circuit generation time and circuit execution time. Values that are known at circuit generation time are called *parameters*, and values that are known at circuit execution time are called *states*. Dynamic lifting is an operation that enables a state, such as the result of a measurement, to be lifted to a parameter, where it can influence the generation of the next portion of the circuit. As a result, dynamic lifting enables Proto-Quipper programs to interleave classical and quantum computation. We describe the syntax of a language we call Proto-Quipper-Dyn. Its type system uses a system of modalities to keep track of the use of dynamic lifting. We also provide an operational semantics, as well as an abstract categorical semantics for dynamic lifting based on enriched category theory. We prove that both the type system and the operational semantics are sound with respect to our categorical semantics. Finally, we give some examples of Proto-Quipper-Dyn programs that make essential use of dynamic lifting.

CCS Concepts: • **Theory of computation** → **Type theory**; **Program semantics**.

Additional Key Words and Phrases: Quipper, Proto-Quipper, quantum programming languages, dynamic lifting, categorical semantics

ACM Reference Format:
Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2023. Proto-Quipper with Dynamic Lifting. *Proc. ACM Program. Lang.* 7, POPL, Article 11 (January 2023), 26 pages. https://doi.org/10.1145/3571204

1 INTRODUCTION

1.1 Quipper and Proto-Quipper

Quipper is a functional programming language for quantum computing [Green et al. 2013a,b]. The overall aim of Quipper is to allow quantum algorithms to be specified at a level of abstraction that is similar to how the algorithm might be described in a research paper, and to compile this down to the level of individual quantum gates, producing a logical quantum circuit. Quipper has been used to program a set of nontrivial algorithms from the quantum computing literature, and it has been used to generate quantum circuits consisting of trillions of gates. As a circuit description language, Quipper shares some of the traits of hardware description languages. In particular, it has two notions of runtime: The first of these is *circuit generation time*. This is when a Quipper program is run to generate a quantum circuit. The second is *circuit execution time*. This is when a quantum circuit is executed by a quantum computer or a simulator.

Authors' addresses: Peng Fu, Dalhousie University, Canada, frank-fu@dal.ca; Kohei Kishida, University of Illinois at Urbana-Champaign, USA, kkishida@illinois.edu; Neil J. Ross, Dalhousie University, Canada, neil.jr.ross@dal.ca; Peter Selinger, Dalhousie University, Canada, selinger@mathstat.dal.ca.

2475-1421/2023/1-ART11
https://doi.org/10.1145/3571204

Quipper is a practical language, implemented as an embedded language in Haskell. As such, it lacks formal foundations such as operational and denotational semantics. This motivates the development of Proto-Quipper, a family of experimental languages that aim to provide formal semantics for fragments of Quipper. Proto-Quipper-S features a linear type system with subtyping as well as an operational semantics [Ross 2015]. Proto-Quipper-M has a linear type system without subtyping, but with a sound categorical semantics in addition to its operation semantics [Rios and Selinger 2018]. More recently, Proto-Quipper-D was introduced, which features a type system with linear dependent types as well as a fibrational categorical semantics [Fu et al. 2020b,a].

1.2 Dynamic Lifting and the Interaction of the Two Runtimes

Proto-Quipper, like Quipper, distinguishes two runtimes. Moreover, Proto-Quipper gives a formal account of parameters and states. A *parameter* is a value that is known at circuit generation time, such as a boolean value for an if-then-else expression. A *state* is a value that is only known at circuit execution time, such as the actual state of a qubit or classical bit in a circuit. The type system of Proto-Quipper reflects this distinction. Among the types, there is a subset of *parameter types*, such as **Nat** and **Bool**, whose elements can be duplicated and discarded. There is also a subset of *state types*, such as **Qubit** and **Bit**, which are linear so that their elements cannot in general be duplicated or discarded. One of the fundamental design decisions of Proto-Quipper is that parameter types and state types belong to the same universe of types, so that one can form compound types that are part parameter and part state. An example of this is the type $\mathbf{Bool} \otimes \mathbf{Qubit}$, whose elements are pairs of a boolean (a parameter) and a qubit (a state). Another example is the type of lists of qubits. Here, the length of the list is a parameter (known at circuit generation time), but the actual qubits in the list are states (known at circuit execution time). In this way, Proto-Quipper differs, e.g., from QWire, an embedded quantum circuit description language in which parameters and states belong to separate universes [Paykin et al. 2017].

In Quipper, the two runtimes can interact with each other. A priori, it is clear that states can depend on parameters. For example, we can initialize a qubit based on a boolean parameter, simply by inserting a gate at circuit generation time to initialize the qubit in one state or another. The opposite direction is more complicated. Usually, circuit execution happens *after* circuit generation, and in this case, it is clear that a state cannot be converted to a parameter. However, there are some quantum algorithms that require circuit generation and circuit execution to be interleaved. Here, a state, such as the outcome of a measurement in a circuit, may be used to inform the generation of the next part of the circuit. To enable such interleaving, Quipper provides a construct called *dynamic lifting*, which enables a state to be lifted to a parameter in certain situations. For example, dynamic lifting permits the result of a measurement, which is a state of type **Bit**, to be lifted to a parameter of type **Bool**. It is important to note that dynamic lifting is an *expensive* operation, as it requires control to pass from circuit evaluation time back to circuit generation time. This requires the real-time quantum computer to put all of its active qubits into long-term storage while spending an indeterminate amount of time awaiting further instructions from the classical computer in charge of circuit generation.

Dynamic lifting is important because it can be used to express quantum algorithms that require interleaving circuit execution time and circuit generation time. While there are many quantum algorithms that do not require such interleaving, there are some that do. An example is *magic state distillation* [Bravyi and Kitaev 2005]. Here, the goal is to prepare a qubit in some target state. We start with a large number of qubits, say n of them, each of which is a rough approximation of the target state. We then apply a probabilistic “distillation” procedure which yields on average, say, $n/4$ qubits that are better approximations of the target state; the remaining qubits are wasted. By repeated distillation steps, we eventually wind up with a small number of qubits that are excellent approximations of the target state. In such a situation, dynamic lifting is essential because after each distillation step, we must throw away the wasted qubits, but we do not know ahead of time which ones (or indeed, how many) there will be. Thus which future gates will be applied depends on the outcomes of previous measurements. With the help of dynamic lifting, these algorithms can be naturally expressed as functions in the programming language.

The concept of dynamic lifting is different from measurement, and the two should not be confused. Measurement is merely a gate in a circuit, turning a quantum bit (a state) into a classical bit (also a state). Dynamic lifting is an operation of the programming language, turning a classical bit (a state) into a boolean (a parameter).

1.3 A Type System for Dynamic Lifting

Previous versions of Proto-Quipper lacked dynamic lifting. Modeling dynamic lifting is a challenging problem. To better understand the issues involved, it is useful to know that there are two things that can be done with circuits in Proto-Quipper. On the one hand, circuits can be run on a quantum device. On the other hand, circuits can be *boxed*. A boxed circuit is a data structure that contains a circuit that has *already been generated*, i.e., an actual list of gates, rather than merely instructions for how to generate such a list. As a result, a boxed circuit can be used as a building block for all kinds of things. In the simplest case, it can be re-used in the construction of other circuits. But it can also be inspected and manipulated in other ways, such as by applying gate transformations (systematically replacing gates by other gates), by adding things like error correction, or by rewriting the circuit to simplify it, among many other possibilities. Boxed circuits can also be reversed, which is used in many quantum algorithms, for example to uncompute ancillas. The ability to box circuits is crucial to Quipper's ability to express algorithms at a natural level of abstraction, because algorithms are often described in terms of meta-operations on circuits.

Now it is clear that dynamic lifting only makes sense in the context of a circuit that is actually being executed, rather than one that is merely being boxed. We will keep track of this in the programming language by adding a modality to the type system and a corresponding monad to the semantics. The modality should be thought of as denoting "boxability". For example, a function of type $\mathbf{Qubit} \multimap_1 \mathbf{Qubit}$ represents a circuit that can be boxed or executed, i.e., that does not use dynamic lifting, whereas a function of type $\mathbf{Qubit} \multimap_0 \mathbf{Qubit}$ represents a quantum operation that can only be executed but not boxed.

Before we can describe the operational or denotational semantics of Proto-Quipper-Dyn, we must be more precise about what we mean by a "circuit". We must also specify what it means to "execute" a circuit. There are many different notions of circuits, differing, for example, in which collection of gates is provided. Rather than specializing to one of these, we take a more general point of view: a *circuit* is simply a morphism in a small symmetric monoidal category $\mathbf{M}$, which we assume to be given ahead of time, but otherwise arbitrary (subject to some properties). Similarly, for the execution of circuits, we assume given another small symmetric monoidal category $\mathbf{Q}$ of *quantum operations*. Conceptually, the morphisms of $\mathbf{M}$ are *syntactic* entities; thus, $\mathbf{M}$ is typically a category that is free generated (say by a collection of gates). On the other hand, we think of the morphisms of $\mathbf{Q}$ as *physical* operations, which can be performed on a quantum computer. The categories $\mathbf{M}$ and $\mathbf{Q}$ have the same objects, and there is a symmetric monoidal *interpretation functor* $J : \mathbf{M} \to \mathbf{Q}$.

Operationally, dynamic lifting is an operation that reads the state of a bit in $\mathbf{Q}$, and returns a boolean value. Since a bit state can be the result of a measurement, the read operation for dynamic lifting is nondeterministic, i.e., it can return different boolean values with probabilities governed by measurements. The nondeterministic nature of the dynamic lifting suggests that it should be modeled as a monadic operation [Moggi 1991].

We therefore conceptualize the types of Proto-Quipper-Dyn as the objects of a single category $\mathbf{A}$, with a monad $T : \mathbf{A} \to \mathbf{A}$, called the *dynamic lifting monad*. This will be done in such a way that $\mathbf{M}$ is fully embedded in $\mathbf{A}$, and $\mathbf{Q}$ is fully embedded in the Kleisli category $Kl_T(\mathbf{A})$, in a way that makes the following diagram commute.

$$\begin{tikzcd}
\mathbf{M} \arrow[r, hook, "\psi"] \arrow[d, "J"] & \mathbf{A} \arrow[d, "E"] \\
\mathbf{Q} \arrow[r, hook, "\phi"] & Kl_T(\mathbf{A})
\end{tikzcd}$$

Here, J is the given interpretation functor, and E is the canonical functor from $\mathbf{A}$ to $Kl_T(\mathbf{A})$. We then model dynamic lifting as a map $\mathsf{dynlift} : \mathbf{Bit} \to T\mathbf{Bool} \in Kl_T(\mathbf{A})$ such that the following diagram commutes.

$$\begin{tikzcd}
 & \mathbf{Bit} \arrow[d, "\mathsf{dynlift}"] \\
\mathbf{Bool} \arrow[ur, "\mathsf{init}"] \arrow[r, "\eta"] & T\mathbf{Bool}
\end{tikzcd}$$

Note that dynamic lifting is a morphism of the Kleisli category; this makes sense because it is essentially a side-effecting read operation. More generally, any computation that potentially uses dynamic lifting will have type $A \to TB$.

As mentioned above, our type system must also distinguish quantum circuits that are being executed from quantum circuits that are being boxed. Naturally, since the latter may not use dynamic lifting, they are maps in the category $\mathbf{A}$ while the former are maps in the Kleisli category $Kl_T(\mathbf{A})$. As a practical matter for programmer convenience, it would be awkward to have T as an explicit type constructor that must be mentioned everywhere in the program. Instead, we use a system of modalities to keep track of the dynamic lifting monad T. More specifically, we annotate a typing judgment with a modality, i.e., $\Gamma \vdash_\alpha M : A$, where $\alpha \in \{0, 1\}$. When $\alpha = 0$, it means that the term M represents a morphism $[\![\Gamma]\!] \to T[\![A]\!]$ in the Kleisli category $Kl_T(\mathbf{A})$. When $\alpha = 1$, it means that the term M represents a morphism $[\![\Gamma]\!] \to [\![A]\!]$ in $\mathbf{A}$. An example of the typing rule for dynamic lifting is the following (where $\mathsf{Meas} : \mathbf{Qubit} \to \mathbf{Bit}$ represents the measurement gate).

$$\frac{\ell : \mathbf{Qubit} \vdash_1 \mathsf{Meas}(\ell) : \mathbf{Bit}}{\ell : \mathbf{Qubit} \vdash_0 \mathsf{dynlift}(\mathsf{Meas}(\ell)) : \mathbf{Bool}}$$

If we have a quantum circuit $\mathbf{Qubit} \to \mathbf{Bit}$, it can be run by a quantum computer and the measurement result of type $\mathbf{Bit}$ will be lifted to a parameter of type $\mathbf{Bool}$. Note that the dynlift operation sets the modality of the typing judgment to 0, and as a result, we have a map $\mathbf{Qubit} \to T\mathbf{Bool}$ in the Kleisli category. The use of modalities in our type system ensures that the term $\mathsf{Meas}(\ell)$ can be turned into a boxed circuit, whereas it will be a compile time typing error to try to box the term $\mathsf{dynlift}(\mathsf{Meas}(\ell))$.

1.4 Operational Semantics

Next, let us take a look at the operational semantics of Proto-Quipper-Dyn. In previous versions of Proto-Quipper, the operational semantics used configurations of the form (C, M), where C is the circuit being currently constructed, and M is a term. On the other hand, in the quantum lambda calculus [Selinger and Valiron 2009], which is not a circuit construction language but intended to run directly on a quantum computer, the operational semantics used configurations of the form (Q, M), where Q is the current quantum state and M is a term.

In a sense, Proto-Quipper-Dyn is a combination of these prior languages: it is a language for circuit construction (via the *boxing* operation), but it is also a language for running quantum operations (as otherwise dynamic lifting would not be possible). Consequently, our operational semantics uses *both* kinds of configurations: those of the form (Q, M) are only used for top-level computations that actually run on a quantum device, and those of the form (C, M) are used during boxing. These two kinds of configurations correspond closely to the two runtimes, since configuration of the form (C, M) are used for circuit construction and those of the form (Q, M) are used for circuit execution. They also correspond to the two categories $\mathbf{M}$ and $\mathbf{Q}$.

Consequently, the evaluation rules take two different forms. Evaluation at circuit generation time takes the form $(C, M) \Downarrow (C', V)$, where C is a circuit. The type system ensures that such an evaluation does not involve dynamic lifting, so it can be done entirely with a classical computer and the evaluation is deterministic. On the other hand, evaluation at circuit execution time takes the form $(Q, M) \Downarrow \sum_i p_i(Q_i, V_i)$, where Q represents a quantum state. Since M can use dynamic lifting, the result of such an evaluation rule is probabilistic, with outcome (Q_i, V_i) happening with probability p_i.

1.5 Related Work

A common misunderstanding is that dynamic lifting means performing measurements during the execution of a quantum program. That would not be a new feature; indeed, the ability to perform on-the-fly measurements was already present in the earliest quantum programming languages, such as [Ömer 1998; Selinger 2004; Selinger and Valiron 2009]. Rather, dynamic lifting is an operation that only makes sense in the context of a *circuit description language*, where circuits are not executed during the circuit generation phase. Dynamic lifting is the transfer of information from the circuit execution environment back to the circuit generation environment. Therefore, in the following discussion of related work, we do not include comparisons with most papers on languages that include measurement but do not have separate circuit generation and circuit execution times.

One of the features of the present work, and of Proto-Quipper in general, is that it works with the standard notion of quantum circuits [Nielsen and Chuang 2002], which are basically lists of gates, or more precisely, gates that have been composed using the laws of symmetric monoidal categories. By contrast, some of the other notions of dynamic lifting that appear in the literature not only add features to the programming language, but also to the generated circuits themselves. Relatedly, one of the features that makes boxed circuits useful in Proto-Quipper is that they are actual data structures. Here, by a “data structure”, we mean data that can be queried, for example via a case distinction or pattern matching. This is different from a “thunk”, such as a lambda abstraction, which represents a suspended computation. Some of the alternative notions of dynamic lifting that appear in the literature make dynamic lifting part of the circuit language, allowing circuits containing dynamic lifting to be boxed. This turns circuits into thunks.

In recent work, [Lee et al. 2021] extended Proto-Quipper with a version of dynamic lifting. They work with a single runtime modeled by a category of *quantum channels*, which are generalizations of quantum circuits with a notion of branching for measurement results. A quantum channel is a list of gates like a quantum circuit, with the important exception that if the current gate is a measurement, the list has two tails, one for each possible measurement outcome. Consequently, the channels of Lee et al. must either be implemented as thunks, or as data structures that are exponentially large. The main difference with our work is that in our setting, dynamic lifting ensures that boxed circuits are data structures that contain only one branch (namely, the one corresponding to the actual measurement result when the circuit is run), whereas in Lee et al.’s setting, either all branches are evaluated, or the circuit is a thunk.

Another version of Proto-Quipper incorporating a form of dynamic lifting was proposed by [Colledan and Dal Lago 2022]. Their language uses a very general version of dynamic lifting, which is even more general than the one present in the Quipper language, and allows for measurements to be conditional on the outcomes of prior measurements. As a consequence, the output *type* of their circuit can depend on the outcomes of the measurements specified in the computation. They also work with a single runtime where dynamic lifting is part of their generalized notion of quantum circuits. While this alternative notion of dynamic lifting is interesting in its own right, their language does not come equipped with a denotational semantics.

QWire [Paykin et al. 2017] is a quantum programming language that also supports dynamic lifting. 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. QWire has a denotational semantics for the circuit language, but not for the host language. Dynamic lifting is part of the syntax in the circuit language. Therefore QWire's notion of quantum circuits differs from Proto-Quipper's notion. Besides dynamic lifting, QWire also has a notion of *static lifting* in the form of a "run" function. This allows measuring all the qubits in a circuit, returning boolean values to the host language, without leaving any unmeasured quantum state. By contrast, Proto-Quipper-Dyn does not require a run function, since it does not have separate host and circuit languages. All circuits that are not constructed inside a box are automatically executed, and dynamic lifting can be used to bring measurement results into the control flow of the language.

We use enriched category theory to describe our categorical model for dynamic lifting. There are some existing works that also use enriched categories in the context of quantum programming languages. For example, [Lindenhovius et al. 2018] use CPO-enrichment to model a version of Proto-Quipper-M with recursion. The main difference between our model and that of Lindenhovius et al. is that our model accounts for dynamic lifting while their model accounts for recursion. [Rennela and Staton 2020] give a categorical model for a QWire-like language that also uses enriched categories. Their language allows boxing a circuit that uses dynamic lifting, which is quite different from how boxing works in Proto-Quipper. As a result, circuits in their setting are thunks and not data structures. Also, in Rennela and Staton's EWire language, the host language does not include wire types such as a type of qubits, whereas Proto-Quipper does not have separate host and circuit languages, and includes all types in a single language. Consequently, Proto-Quipper has a linear type system, whereas the EWire host language does not. This difference is also reflected in the model: in Rennela and Staton's semantics, programs are interpreted in a cartesian-closed category, whereas in our model, they are interpreted in a monoidal category.

The fact that Proto-Quipper has two distinct runtimes (circuit generation time and circuit execution time) suggests a possible connection to another computational paradigm that also has multiple runtimes, namely *multi-staged computation* [MetaOCaml 2020; Taha and Sheard 2000]. However, there are some important differences. One of them is that multi-staged computation, such as in MetaML [Taha and Sheard 2000], deals with potentially many levels, but all of the levels share the same operations and the same hardware; the primary purpose of staging is to precisely orchestrate the order in which operations are evaluated. On the other hand, the main purpose of dynamic lifting in Proto-Quipper is to interleave computations from two different hardware models. The "meta-language" of Proto-Quipper terms has almost nothing in common with the "object language" of circuits. Each of the two stages has its own distinct operations: classical expressions and control flow for the meta-language, and gates and measurements for the object language. In particular, quantum circuits are not just code for expressions of the meta-language.

Finally, we will mention the quantum programming language Silq [Bichsel et al. 2020]. Like Proto-Quipper-Dyn, Silq also uses modalities to keep track of the use of certain operations. For example, the modality "mfree" in Silq is used to indicate whether a computation uses measurement. The difference is that Silq is not a circuit description language, so it does not have a notion of boxed circuits or dynamic lifting.

1.6 Contributions

In this paper, we describe the syntax and type system of an extension of Proto-Quipper with dynamic lifting, called Proto-Quipper-Dyn. The type system uses a system of modalities to keep track of the use of dynamic lifting. We also provide an operational semantics, using two different kinds of configurations to model circuit generation time and circuit execution time. We further provide an abstract categorical semantics for this language, in which dynamic lifting is modeled by a map $\mathbf{Bit} \to T\mathbf{Bool}$, where T is a monad encapsulating circuit execution. By an "abstract" categorical semantics, we mean that we only state the properties that a categorical model must satisfy to give a sound interpretation of the language, without constructing an actual concrete example of such a model. We give such a concrete model in a companion paper [Fu et al. 2022a].

The rest of the paper is organized as follows: In Section 2, we briefly recall the basics of enriched category theory, and then we give an axiomatization of a general categorical semantics for dynamic lifting. In Section 3, we define a type system for dynamic lifting that uses a system of modalities. We then show how a typing judgment with modalities is interpreted as a morphism in our categorical model. In Section 4, we define a call-by-value big-step operational semantics for our language. We show that the operational semantics satisfies type preservation and that the type system guarantees error freeness. We also show that the operational semantics is sound with respect to the enriched categorical semantics. In Section 5, we give some applications of dynamic lifting in Proto-Quipper-Dyn. We finish the paper with some concluding remarks in Section 6.

2 AN ENRICHED CATEGORICAL SEMANTICS FOR DYNAMIC LIFTING

In this section we will give a general categorical semantics for dynamic lifting. Our categorical semantics is based on enriched categories, which are generalizations of ordinary categories. In enriched categories, instead of hom-sets, one works with hom-objects, which are objects in a monoidal category.

Definition 2.1. Let $\mathcal{V}$ be a monoidal category. A *$\mathcal{V}$-enriched category* $\mathbf{A}$ (or *$\mathcal{V}$-category* for short) is given by the following:

- A class of objects, also denoted $\mathbf{A}$.
- For any $A, B \in \mathbf{A}$, an object $\mathbf{A}(A, B)$ in $\mathcal{V}$.
- For any $A \in \mathbf{A}$, a morphism in $u_A : I \to \mathbf{A}(A, A)$ in $\mathcal{V}$, called the *identity* on A.
- For any $A, B, C \in \mathbf{A}$, a morphism $c_{A,B,C} : \mathbf{A}(A, B) \otimes \mathbf{A}(B, C) \to \mathbf{A}(A, C)$ in $\mathcal{V}$, called *composition.*
- The composition and identity morphisms must satisfy suitable diagrams in $\mathcal{V}$ (see [Borceux 1994; Kelly 1982]).

Remarks.
- Many concepts from non-enriched category theory 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 to [Borceux 1994; Kelly 1982] for comprehensive introductions.
- In the rest of this paper, when we speak of a map $f : A \to B$ in a $\mathcal{V}$-enriched category $\mathbf{A}$, we mean a morphism of the form $f : I \to \mathbf{A}(A, B)$ in $\mathcal{V}$. Furthermore, when $g : B \to C$ is another map in $\mathbf{A}$, we write $g \circ f : A \to C$ as a shorthand for

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

- A $\mathcal{V}$-enriched category $\mathbf{A}$ gives rise to an ordinary category $V(\mathbf{A})$, called the *underlying category*[1] of $\mathbf{A}$, where the objects of $V(\mathbf{A})$ are objects of $\mathbf{A}$ and a hom-set is defined as

[1] V stands for "underlying" because the letter U serves another purpose in this paper.

$V(\mathbf{A})(A,B) := \mathcal{V}(I, \mathbf{A}(A,B))$ for any $A, B \in V(\mathbf{A})$. Similarly, a $\mathcal{V}$-functor $F : \mathbf{A} \to \mathbf{B}$ gives rise to an ordinary functor $VF : V(\mathbf{A}) \to V(\mathbf{B})$ and a $\mathcal{V}$-natural transformation $\alpha : F \to G$ gives rise to an ordinary natural transformation $V\alpha : VF \to VG$.

Ordinary symmetric monoidal categories can be generalized to enriched categories as well.

Definition 2.2. Let $\mathcal{V}$ be a symmetric monoidal category. A $\mathcal{V}$-category $\mathbf{A}$ is symmetric monoidal if it is equipped with the following:

- There is an object $I \in \mathbf{A}$ called the *tensor unit*. For any $A, B \in \mathbf{A}$, there is an object $A \otimes B \in \mathbf{A}$. Moreover, for any $A_1, A_2, B_1, B_2 \in \mathbf{A}$, there is a morphism
$$\mathrm{Tensor} : \mathbf{A}(A_1, B_1) \otimes \mathbf{A}(A_2, B_2) \to \mathbf{A}(A_1 \otimes A_2, B_1 \otimes B_2)$$
in $\mathcal{V}$. The tensor product is a bifunctor in the sense that $\mathrm{Tensor} \circ (u_A \otimes u_B) = u_{A\otimes B}$ for the identity maps $u_A, u_B, u_{A\otimes B}$, and the following diagram commutes for any $A_1, A_2, B_1, B_2, C_1, C_2 \in \mathbf{A}$.
$$\begin{tikzcd}
\mathbf{A}(A_1,B_1) \otimes \mathbf{A}(A_2,B_2) \otimes \mathbf{A}(B_1,C_1) \otimes \mathbf{A}(B_2,C_2) \arrow[r, "c\otimes c"] \arrow[d, "\mathrm{Tensor}\otimes\mathrm{Tensor}"] & \mathbf{A}(A_1,C_1) \otimes \mathbf{A}(A_2,C_2) \arrow[d, "\mathrm{Tensor}"] \\
\mathbf{A}(A_1 \otimes A_2, B_1 \otimes B_2) \otimes \mathbf{A}(B_1 \otimes B_2, C_1 \otimes C_2) \arrow[r, "c"] & \mathbf{A}(A_1 \otimes A_2, C_1 \otimes C_2)
\end{tikzcd}$$
- There are the following $\mathcal{V}$-natural isomorphisms in $\mathbf{A}$ and they satisfy the same coherence diagrams for symmetric monoidal categories.
$$l_A : I \otimes A \to A$$
$$r_A : A \otimes I \to A$$
$$\gamma_{A,B} : A \otimes B \to B \otimes A$$
$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$

If the $\mathcal{V}$-category $\mathbf{A}$ is symmetric monoidal, then its underlying category $V(\mathbf{A})$ is symmetric monoidal. For any maps $f : A_1 \to B_1, g : A_2 \to B_2$ in $\mathbf{A}$, we write the map $f \otimes g : A_1 \otimes A_2 \to B_1 \otimes B_2$ as a shorthand for the following composition.
$$I \xrightarrow{f\otimes g} \mathbf{A}(A_1, B_1) \otimes \mathbf{A}(A_2, B_2) \xrightarrow{\mathrm{Tensor}} \mathbf{A}(A_1 \otimes A_2, B_1 \otimes B_2)$$

2.1 An Axiomatization of Enriched Categorical Models of Dynamic Lifting

In the following, we assume $\mathcal{V}$ to be a cartesian closed category with coproducts. For any $A, B \in \mathcal{V}$, we write $A \times B$ for the cartesian product, $A \Rightarrow B$ for the exponential object, and $1 \in \mathcal{V}$ for the terminal object. Since $\mathcal{V}$ is cartesian closed, it is self-enriched, i.e., $\mathcal{V}$ is a $\mathcal{V}$-category where the hom-objects are defined by $\mathcal{V}(A, B) := A \Rightarrow B$.

We will now focus on defining a $\mathcal{V}$-enriched category $\mathbf{A}$ that models dynamic lifting. We give a sequence of definitions that specify a sequence of properties (a)-(h), which will culminate in Definition 2.8 of a model for Proto-Quipper with dynamic lifting.

Definition 2.3. A $\mathcal{V}$-category $\mathbf{A}$ is a *linear-non-linear programming language model* if

(a) $\mathbf{A}$ has coproducts and is symmetric monoidal closed, i.e., it is symmetric monoidal and there is a $\mathcal{V}$-adjunction $- \otimes A \dashv A \multimap -$ for each $A \in \mathbf{A}$.

(b) $\mathbf{A}$ is equipped with a $\mathcal{V}$-adjunction
$$p : \mathcal{V} \to \mathbf{A} \dashv \flat : \mathbf{A} \to \mathcal{V}$$
such that p is a strong monoidal $\mathcal{V}$-functor.

Remarks.

- The requirement that $\mathbf{A}$ has coproducts and is symmetric monoidal closed implies that it can model function types and sum types in a functional programming language. Moreover, since $- \otimes A$ is a left adjoint $\mathcal{V}$-functor for any $A \in \mathbf{A}$, it preserves the coproducts, so the tensor products distribute over coproducts in $\mathbf{A}$.
- The adjunction in (b) is often called a *linear-non-linear adjunction* [Benton 1995]. Here, the assumption that p is a strong monoidal $\mathcal{V}$-functor means that there exist isomorphisms $e : I \to p1$ and $m : pX \otimes pY \to p(X \times Y)$ making some diagrams commute (see [Fu et al. 2022b, Appendix A]).
- Since p is strong monoidal and $\mathcal{V}$ is cartesian, for any $X \in \mathcal{V}$, there are maps $\mathrm{discard}_X : pX \to I$ and $\mathrm{dup}_X : pX \to pX \otimes pX$ in $\mathbf{A}$. Moreover, for any map $f : X \to Y$ in $\mathcal{V}$, we have the following in $\mathbf{A}$.
$$\mathrm{dup}_Y \circ pf = (pf \otimes pf) \circ \mathrm{dup}_X$$
We call objects of the form $pX \in \mathbf{A}$ *parameter objects*, since they can be duplicated and discarded. For example, $\mathbf{Bool} := I + I \cong p1 + p1 \cong p(1+1)$ is a parameter object.
- For any $X \in \mathcal{V}, B \in \mathbf{A}$, we write δ for the isomorphism $\delta : \mathbf{A}(pX, B) \cong \mathcal{V}(X, \flat B)$, and force_B for the counit $\mathrm{force}_B : p\flat B \to B$.

Definition 2.4. A *convex space* is a set X equipped with a *convex sum* operation, which assigns to any $x, y \in X$ and $p, q \in [0,1]$ such that $p + q = 1$ an element $px + qy \in X$, subject to certain standard conditions, which are detailed in [Fu et al. 2022b, Appendix B]. A category is *enriched in convex spaces* if each hom-set is equipped with the structure of a convex space, and moreover, composition is *bilinear* with respect to convex sum, i.e., $(pf + qg) \circ h = p(f \circ h) + q(g \circ h)$ and $h \circ (pf + qg) = p(h \circ f) + q(h \circ g)$.

As mentioned in the introduction, Proto-Quipper-Dyn is parameterized by two (ordinary) small categories $\mathbf{M}$ and $\mathbf{Q}$ of *circuits* and *quantum operations*, respectively. We now specify the properties that these categories must satisfy.

Assumption 2.5. We assume that we are given two small symmetric monoidal categories $\mathbf{M}$ and $\mathbf{Q}$, satisfying the following properties:

(1) $\mathbf{M}$ and $\mathbf{Q}$ have the same objects, including a distinguished object called $\mathbf{Bit}$. The category $\mathbf{M}$ has distinguished morphisms zero, one : $I \to \mathbf{Bit}$.
(2) $\mathbf{Q}$ has a coproduct $\mathbf{Bit} = I + I$, and the tensor product in $\mathbf{Q}$ distributes over this coproduct.
(3) There exists a given strict symmetric monoidal functor $J : \mathbf{M} \to \mathbf{Q}$ that is the identity on objects and $J(\mathrm{zero}) = \mathrm{inj}_1 : I \to I + I$, $J(\mathrm{one}) = \mathrm{inj}_2 : I \to I + I$. We call J the *interpretation functor*.
(4) The category $\mathbf{Q}$ is enriched in convex spaces.
(5) For any $A \in \mathbf{Q}$, and $f : I \to \mathbf{Bit} \otimes A \in \mathbf{Q}$, we have $f = p_1(\mathrm{inj}_1 \otimes f_1) + p_2(\mathrm{inj}_2 \otimes f_2)$, where $\mathrm{inj}_1, \mathrm{inj}_2 : I \to I + I$ and $p_1, p_2 \in [0,1]$ are uniquely determined real numbers such that $p_1 + p_2 = 1$. When $p_i \neq 0$, the map $f_i : I \to A$ is also unique.

The categories $\mathbf{M}$ and $\mathbf{Q}$ are not only used in the categorical semantics, but also in the operational semantics of Proto-Quipper-Dyn (i.e., to run the program, we must know what a circuit is and what a quantum operation is). Therefore, these categories should be regarded as given as part of the language specification, rather than as a degree of freedom in the semantics. On the other hand, nothing in the operational or denotational semantics depends on particular properties of $\mathbf{M}$ and $\mathbf{Q}$ other than properties (1)–(5) above. Therefore, Proto-Quipper-Dyn can handle a wide variety of possible circuit models and physical execution models.

In practice, the category $\mathbf{M}$ will be a category of quantum circuits and the category $\mathbf{Q}$ will be a category of quantum operations. These categories will typically have additional objects, such as **Qubit** and perhaps **Qutrit**, and additional morphisms, such as $H : \mathbf{Qubit} \to \mathbf{Qubit}$ and $\mathsf{Meas} : \mathbf{Qubit} \to \mathbf{Bit}$. Assumption (5) means that any morphism with domain I and a bit state in its codomain is a convex sum of two morphisms. This property is used in the rule for dynamic lifting in the operational semantics.

Definition 2.6. Suppose the $\mathcal{V}$-enriched category $\mathbf{A}$ is a linear-non-linear programming model. We say it supports *box-unbox operations* if the following hold.

- **(c)** There is a fully faithful embedding $\psi : \mathbf{M} \overset{\psi}{\hookrightarrow} V(\mathbf{A})$ and ψ is strong monoidal.
- **(d)** Let $\mathcal{S}$ denote the set of objects in the image of ψ. For any $S, U \in \mathcal{S}$, there is an isomorphism

$$\flat(S \multimap U) \overset{e}{\cong} \mathbf{A}(S, U).$$

Condition (c) implies that there is a circuit subcategory in $\mathbf{A}$. Using condition (d), we define $\mathsf{box} = p(e)$ and $\mathsf{unbox} = p(e^{-1})$, and there is an isomorphism $p\flat(S \multimap U) \overset{\mathsf{box/unbox}}{\cong} p\mathbf{A}(S, U)$. Elements of $p\mathbf{A}(S, U)$ correspond to boxed circuits with input S and output U.

If a $\mathcal{V}$-enriched category $\mathbf{A}$ satisfies (a)–(d), then it is a model for Proto-Quipper *without* dynamic lifting. For example, the **Set**-enriched category $\overline{\overline{\mathbf{M}}}$ in [Rios and Selinger 2018] is such a model. To support dynamic lifting, we define the following monad to account for the category **Q**.

Definition 2.7. Let $\mathbf{A}$ be a symmetric monoidal $\mathcal{V}$-category and let $T : \mathbf{A} \to \mathbf{A}$ be a $\mathcal{V}$-monad on $\mathbf{A}$. We say T is a *dynamic lifting monad* if the following hold.

- **(e)** T is a commutative strong $\mathcal{V}$-monad. For any $A, B \in \mathbf{A}$, we write $t_{A,B} : A \otimes TB \to T(A \otimes B)$ for the strength and $s_{A,B} : TA \otimes B \to T(A \otimes B)$ for the costrength.
- **(f)** Let $V(\mathbf{A})$ be the underlying category of $\mathbf{A}$, let VT be the underlying monad of T, and let $Kl_{VT}(V(\mathbf{A}))$ be the Kleisli category of VT. The Kleisli category $Kl_{VT}(V(\mathbf{A}))$ is enriched in convex spaces.
- **(g)** There are the following fully faithful embeddings:

$$\mathbf{M} \overset{\psi}{\hookrightarrow} V(\mathbf{A}),$$

$$\mathbf{Q} \overset{\phi}{\hookrightarrow} Kl_{VT}(V(\mathbf{A})).$$

These embedding functors are strong monoidal, and ϕ preserves the convex sum. Moreover, the following diagram commutes for any $S, U \in \mathbf{M}$.

$$\begin{tikzcd}
\mathbf{M}(S,U) \arrow[r, "\psi_{S,U}"] \arrow[d, "J_{S,U}"] & V(\mathbf{A})(S,U) \arrow[d, "E_{S,U}"] \\
\mathbf{Q}(S,U) \arrow[r, "\phi_{S,U}"] & Kl_{VT}(V(\mathbf{A}))(S,U)
\end{tikzcd}$$

Here $E : V(\mathbf{A}) \to Kl_{VT}(V(\mathbf{A}))$ is the the functor such that $E(A) = A$ and $E(f) = \eta \circ f$.
- **(h)** There are maps $\mathsf{dynlift} : \mathbf{Bit} \to T\mathbf{Bool}$ and $\mathsf{init} : \mathbf{Bool} \to \mathbf{Bit}$ in $\mathbf{A}$ such that the following diagram commutes.

$$\begin{tikzcd}
 & \mathbf{Bit} \arrow[d, "\mathsf{dynlift}"] \\
\mathbf{Bool} \arrow[ur, "\mathsf{init}"] \arrow[r, "\eta"] & T\mathbf{Bool}
\end{tikzcd}$$

Remarks.
- The objects of the Kleisli category $Kl_{VT}(V(\mathbf{A}))$ are the same as the objects of $\mathbf{A}$, and the hom-set is given by $Kl_{VT}(V(\mathbf{A}))(A, B) := V(\mathbf{A})(A, VTB) = \mathcal{V}(1, \mathbf{A}(A, TB))$ for any $A, B \in \mathbf{A}$. Moreover, $\mathcal{V}(1, \mathbf{A}(A, TB)) = \mathcal{V}(1, Kl_T(\mathbf{A})(A, B)) = V(Kl_T(\mathbf{A}))(A, B)$.

- Note that in condition (f), we are not taking a $\mathcal{V}$-enriched Kleisli category of the $\mathcal{V}$-monad T, but just an ordinary Kleisli category of the ordinary monad VT. Thus, the Kleisli category is not $\mathcal{V}$-enriched. However, we do require it to be enriched in convex spaces, which amounts to requiring the existence of additional operations on its hom-sets, in the sense of Definition 2.4.
- Since T is a commutative strong $\mathcal{V}$-monad, VT is a commutative strong (ordinary) monad. Therefore the Kleisli category $Kl_{VT}(V(\mathbf{A}))$ is monoidal. For any $f : A_1 \to VTB_1$ and $g : A_2 \to VTB_2$ in $Kl_{VT}(V(\mathbf{A}))$, we define $f \otimes g \in Kl_{VT}(V(\mathbf{A}))(A_1 \otimes A_2, B_1 \otimes B_2)$ to be the following

$$A_1 \otimes A_2 \xrightarrow{f\otimes g} VTB_1 \otimes VTB_2 \xrightarrow{s} VT(B_1 \otimes VTB_2) \xrightarrow{Tt} VTVT(B_1 \otimes B_2) \xrightarrow{\mu} VT(B_1 \otimes B_2).$$

- Condition (g) expresses the requirement that the enriched category $\mathbf{A}$ must combine both categories $\mathbf{M}$ and $\mathbf{Q}$, i.e., they are subcategories of $V(\mathbf{A})$ and its Kleisli category, respectively. Thus $\mathbf{A}$ has both quantum circuits and quantum operations. The commutative diagram implies that a circuit in $\mathbf{A}$ can be used as a quantum operation.
- Since $\psi(S) = \phi(S)$ for any $S \in \mathbf{M}, \mathbf{Q}$, we define $\mathbf{Bit} = \psi(\mathbf{Bit}) = \phi(\mathbf{Bit}) \in \mathbf{A}$.
- Condition (h) gives a categorical characterization of dynamic lifting. The map dynlift is not in the image of ϕ or ψ, and therefore it is neither a quantum circuit nor a quantum operation.

Definition 2.8. We say a $\mathcal{V}$-enriched category $\mathbf{A}$ is a *model for Proto-Quipper with dynamic lifting* if it satisfies (a)–(h).

We have now axiomatized a general categorical model for Proto-Quipper with dynamic lifting. In [Fu et al. 2022a], we give a construction of a concrete model based on *biset-enrichment* that satisfies (a)-(h). In the rest of this paper, we will be focusing on showing this abstract categorical model $\mathbf{A}$ is sound with respect to the type system and the operational semantics.

3 A TYPE SYSTEM FOR DYNAMIC LIFTING

In this section, we present the syntax of Proto-Quipper-Dyn and a type system for dynamic lifting. Our typing judgments have the form $\Gamma \vdash_\alpha M : A$, where $\alpha ::= 0 \mid 1$ is a modality used to keep track of dynamic lifting. When $\alpha = 1$, the term M is guaranteed not to perform any dynamic lifting operations while it is being reduced to a value. Such computations can therefore be carried out at circuit generation time. When $\alpha = 0$, M may invoke dynamic lifting so the evaluation of M needs to be performed at circuit execution time.

Definition 3.1 (Syntax). The syntax of Proto-Quipper-Dyn is in Figure 1.

The modality α appears in the linear function type $A \multimap_\alpha B$ and the linear exponential type $!_\alpha A$. This is because the values of $A \multimap_\alpha B$ and $!_\alpha A$ are *thunks* and we use the modality α in the types to keep track of the dynamic lifting within the thunks. $\mathbf{Circ}(S, U)$ denotes a type of circuits with input S and output U. The values of this type are boxed quantum circuits. They can be further manipulated by meta-operations such as circuit reversal, circuit iteration, or printing; these operations are treated as constants in the language, i.e., we do not fix a particular set of such operations, but assume that they would be defined in a standard library that comes with any particular instance of Proto-Quipper-Dyn.

The terms of our language are similar to the ones from [Rios and Selinger 2018], with the addition of a term construct for dynamic lifting dynlift M, which will be evaluated to a boolean value. The term c ranges over constants such as booleans, natural numbers, and built-in functions. A term of parameter type can be duplicated or discarded. A value of simple type corresponds to a state. Our language and semantics can accommodate coproducts (sum types), but we elide the treatment here for the sake of simplicity.

Modality	α, β	$::=$	$0 \mid 1$
Types	A, B	$::=$	$\mathbf{Unit} \mid \mathbf{Qubit} \mid \mathbf{Bit} \mid \mathbf{Bool} \mid !_\alpha A \mid A \multimap_\alpha B \mid \mathbf{Circ}(S, U) \mid A \otimes B$
Parameter Types	P, R	$::=$	$\mathbf{Unit} \mid \mathbf{Nat} \mid !_\alpha A \mid \mathbf{Circ}(S, U) \mid P \otimes R$
Simple Types	S, U	$::=$	$\mathbf{Unit} \mid \mathbf{Qubit} \mid \mathbf{Bit} \mid S \otimes U$
Terms	M, N	$::=$	$c \mid x \mid \lambda x.M \mid M\ N \mid \mathsf{Unit} \mid (a, C, b) \mid \mathsf{apply}(M, N) \mid \mathsf{force} M$
			$\mid \mathsf{lift}\ M \mid \mathsf{box}\ U\ M \mid (M, N) \mid \mathsf{let}\ (x, y) = N\ \mathsf{in}\ M \mid \mathsf{dynlift}\ M$
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	$::=$	$x \mid \ell \mid \lambda x.M \mid \mathsf{lift}\ M \mid (a, C, b) \mid (V, V') \mid \mathsf{Unit}$
Circuits	$C, \mathcal{D} : \Sigma \to \Sigma'$		

Fig. 1. The syntax for Proto-Quipper-Dyn

$$\frac{}{\Phi, x : A \vdash_1 x : A}\ \mathit{var} \qquad \frac{}{\ell : \mathbf{Qubit}|\mathbf{Bit} \vdash_1 \ell : \mathbf{Qubit}|\mathbf{Bit}}\ \mathit{label}$$

$$\frac{\Gamma_1 \vdash_{\alpha_1} M : A \multimap_\beta B \quad \Gamma_2 \vdash_{\alpha_2} N : A}{\Gamma_1 + \Gamma_2 \vdash_{\alpha_1 \& \alpha_2 \& \beta} MN : B}\ \mathit{app} \qquad \frac{\Gamma, x : A \vdash_\alpha M : B}{\Gamma \vdash_1 \lambda x.M : A \multimap_\alpha B}\ \mathit{lambda}$$

$$\frac{\Phi \vdash_\alpha M : A}{\Phi \vdash_1 \mathsf{lift}\ M : !_\alpha A}\ \mathit{lift} \qquad \frac{\Gamma \vdash_\beta M : !_\alpha A}{\Gamma \vdash_{\alpha \& \beta} \mathsf{force}\ M : A}\ \mathit{force}$$

$$\frac{\Gamma \vdash_\alpha M : !_1(S \multimap_1 U)}{\Gamma \vdash_\alpha \mathsf{box}\ S\ M : \mathbf{Circ}(S, U)}\ \mathit{box} \qquad \frac{\Gamma_1 \vdash_\alpha M : \mathbf{Circ}(S, U) \quad \Gamma_2 \vdash_\beta N : S}{\Gamma_1 + \Gamma_2 \vdash_{\alpha \& \beta} \mathsf{apply}(M, N) : U}\ \mathit{apply}$$

$$\frac{\begin{array}{cc}\Sigma_1 \vdash_1 a : S & \Sigma_2 \vdash_1 b : U \\ C : \Sigma_1 \to \Sigma_2 & \end{array}}{\Phi \vdash_1 (a, C, b) : \mathbf{Circ}(S, U)}\ \mathit{circ} \qquad \frac{\Gamma \vdash_\alpha M : \mathbf{Bit}}{\Gamma \vdash_0 \mathsf{dynlift}\ M : \mathbf{Bool}}\ \mathit{dynlift}$$

$$\frac{\Gamma_1 \vdash_{\alpha_1} M : A \quad \Gamma_2 \vdash_{\alpha_2} N : B}{\Gamma_1 + \Gamma_2 \vdash_{\alpha_1 \& \alpha_2} (M, N) : A \otimes B}\ \mathit{pair} \qquad \frac{\Gamma_1, x : A, y : B \vdash_{\alpha_1} M : C \quad \Gamma_2 \vdash_{\alpha_2} N : A \otimes B}{\Gamma_1 + \Gamma_2 \vdash_{\alpha_1 \& \alpha_2} \mathsf{let}\ (x, y) = N\ \mathsf{in}\ M : C}\ \mathit{let}$$

Fig. 2. The typing rules for Proto-Quipper-Dyn

We make a distinction between variables and labels. A label ℓ corresponds to a wire in a circuit, or to an address of a bit or qubit state. Consequently, a label is a value that can only have type **Bit** or **Qubit**. Labels can only be renamed, not substituted. Every label context Σ has an obvious interpretation $[\![\Sigma]\!]$ in the category **M** as a tensor of the appropriate sequence of the objects **Qubit** and **Bit**. We write $\mathcal{D} : \Sigma \to \Sigma'$ to denote a quantum circuit, i.e., a morphism $\mathcal{D} : [\![\Sigma]\!] \to [\![\Sigma']\!]$.

Definition 3.2 (Typing). The typing rules are in Figure 2.

We write $\alpha \ \&\ \beta$ for the boolean conjunction of α and β so that, e.g., $0 \ \&\ 1 = 0$. If $\Gamma_1 = \Phi, \Gamma_1'$ and $\Gamma_2 = \Phi, \Gamma_2'$, we write $\Gamma_1 + \Gamma_2$ for $\Phi, \Gamma_1', \Gamma_2'$.

In the *var* rule, we require a parameter context Φ. In the *lift* and *lambda* rules, the modality α is moved to the type and the current modality (i.e., modality in the conclusion) is set to 1. This is because the lift and lambda terms are values, and values do not perform dynamic lifting. In fact, all values have modality 1.

In elimination rules such as *app* and *force*, the modality in the type affects the current modality of the typing judgment through boolean conjunction. This is related to how the evaluations are performed for these terms. For example, when evaluating the term MN, we will first evaluate M, then evaluate N and finally perform a beta-reduction. Thus, the evaluation of MN could perform dynamic lifting of $\alpha_1 = 0$, $\alpha_2 = 0$, or $\beta = 0$. Consequently, the modality for the typing judgment of MN is the boolean conjunction of all these related modalities.

By the *dynlift* rule, an application of dynamic lifting sets the current modality to 0, signifying that a dynamic lifting is performed. In the *box* rule, a term M can only be boxed into a circuit if it has type $!_1(S \multimap_1 U)$. This ensures that the value of M (denoted by V) does not use dynamic lifting. Thus, when evaluating the term $(\mathrm{box}\, S\ V)$, a dynamic lifting cannot occur. This prevents a class of runtime errors in Quipper that are caused by boxing functions that use dynamic lifting.

In the *apply* rule, depending on the modality $\alpha_1 \,\&\, \alpha_2$, the term $\mathrm{apply}(M, N)$ either appends the quantum circuit M to N, which is done at circuit generation time, or applies the quantum operation M to N, which is done at circuit execution time. The *circ* rule defines a well-typed quantum circuit. In practice, we often assume that a set of well-typed quantum gates is provided as pre-defined constants of the language, so that the programmer does not need to use the *circ* rule.

The following lemma shows that a value can only have modality 1 and, in particular, that the free variables of a parameter must come from a parameter context.

Lemma 3.3. *If* $\Gamma \vdash_\alpha V : B$*, then* $\alpha = 1$*. Moreover, if* $\Gamma \vdash_\alpha V : P$*, then* $\alpha = 1$ *and* $\Gamma = \Phi$*.*

The following lemma shows that the type system has the usual substitution property.

Lemma 3.4 (Substitution). *If* $\Gamma_1, x : A, \Gamma_1' \vdash_\alpha M : B$ *and* $\Gamma_2 \vdash_1 V : A$*, then* $\Gamma_1, \Gamma_1', \Gamma_2 \vdash_\alpha [V/x]M : B$*.*

3.1 Interpretation of the Typing Rules

The modality is a syntactic device to track the dynamic lifting monad T. We will interpret $\Gamma \vdash_1 M : A$ as a map $[\![\Gamma]\!] \longrightarrow [\![A]\!]$ in $\mathbf{A}$, and $\Gamma \vdash_0 M : A$ as a map $[\![\Gamma]\!] \longrightarrow T[\![A]\!]$. The modalities in types such as $A \multimap_\alpha B$ and $!_\alpha A$ also indicate occurrences of the dynamic lifting monad T.

Definition 3.5. We interpret types as objects in $\mathbf{A}$.

$$\begin{array}{lcl}
[\![A \multimap_1 B]\!] & = & [\![A]\!] \multimap [\![B]\!] \\
[\![A \multimap_0 B]\!] & = & [\![A]\!] \multimap T[\![B]\!] \\
[\![A \otimes B]\!] & = & [\![A]\!] \otimes [\![B]\!] \\
[\![!_1 A]\!] & = & p\flat[\![A]\!] \\
[\![!_0 A]\!] & = & p\flat T[\![A]\!] \\
[\![\mathbf{Circ}(S, U)]\!] & = & p\mathbf{A}([\![S]\!], [\![U]\!]) \\
[\![\mathbf{Bool}]\!] & = & p(1 + 1) \\
[\![\mathbf{Bit}]\!] & = & \mathbf{Bit} \\
[\![\mathbf{Qubit}]\!] & = & \mathbf{Qubit}
\end{array}$$

For a parameter type P, there exists $X \in \mathcal{V}$ such that $[\![P]\!] = pX$. For a simple type S, there exists $Y \in \mathbf{M}$ such that $[\![S]\!] = \psi Y$. We call objects of the form ψY *simple objects*. We write $\alpha[\![A]\!]$ to mean $T[\![A]\!]$ if $\alpha = 0$, otherwise it is $[\![A]\!]$. We interpret a context Γ as a tensor product of all objects in Γ (denoted by $[\![\Gamma]\!]$). The interpretation of parameter context $[\![\Phi]\!]$ is a parameter object and the interpretation of a label context $[\![\Sigma]\!]$ is a simple object. Without loss of generality, we assume that if $[\![\Sigma]\!] = [\![\Sigma']\!]$, then $\Sigma = \Sigma'$ (this condition can always be ensured by making additional isomorphic copies of objects, if necessary).

The interpretation of typing judgements is defined as follows.

Definition 3.6 (Interpretation)**.** To each valid typing judgement $\Gamma \vdash_\alpha M : A$, we associate a map $[\![M]\!] : [\![\Gamma]\!] \to \alpha[\![A]\!]$ in $\mathbf{A}$, called its *interpretation*. Note that $[\![M]\!]$ here is an abbreviation for $[\![\Gamma \vdash_\alpha M : A]\!]$.

The interpretation is defined by induction on the derivation of $\Gamma \vdash_\alpha M : A$. Here we show a few cases, the rest are in [Fu et al. 2022b, Appendix C].

- Case
$$\frac{\Gamma \vdash_\alpha M : \mathbf{Bit}}{\Gamma \vdash_0 \mathsf{dynlift}\, M : \mathbf{Bool}.}$$
By induction hypothesis, we have $[\![M]\!] : [\![\Gamma]\!] \to \alpha[\![\mathbf{Bit}]\!]$. If $\alpha = 1$, we define $[\![\mathsf{dynlift}\, M]\!]$ by
$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![\mathbf{Bit}]\!] \xrightarrow{\mathsf{dynlift}} T[\![\mathbf{Bool}]\!].$$
If $\alpha = 0$, we define $[\![\mathsf{dynlift}\, M]\!]$ by
$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} T[\![\mathbf{Bit}]\!] \xrightarrow{T\,\mathsf{dynlift}} TT[\![\mathbf{Bool}]\!] \xrightarrow{\mu} T[\![\mathbf{Bool}]\!].$$
- Case
$$\frac{\Gamma, x : A \vdash_\alpha M : B}{\Gamma \vdash_1 \lambda x.M : A \multimap_\alpha B.}$$
By induction hypothesis, we have $[\![M]\!] : [\![\Gamma]\!] \otimes [\![A]\!] \to \alpha[\![B]\!]$. Using monoidal closedness, we define $[\![\lambda x.M]\!] := \mathsf{curry}([\![M]\!]) : [\![\Gamma]\!] \to [\![A]\!] \multimap \alpha[\![B]\!]$.
- Case
$$\frac{\Phi, \Gamma_1 \vdash_{\alpha_1} M : A \multimap_\beta B \quad \Phi, \Gamma_2 \vdash_{\alpha_2} N : A}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha_1 \& \alpha_2 \& \beta} MN : B.}\ \mathit{app}$$
Here we only consider the case where $\alpha_1 = \alpha_2 = \beta = 0$. The other cases are similar. By induction hypothesis, we have morphisms $[\![M]\!] : [\![\Phi]\!] \otimes [\![\Gamma_1]\!] \to T([\![A]\!] \multimap T[\![B]\!])$ and $[\![N]\!] : [\![\Phi]\!] \otimes [\![\Gamma_2]\!] \to T[\![A]\!]$. Thus we define $[\![MN]\!]$ to be the following.

$$\begin{aligned}[\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] &\xrightarrow{\mathsf{dup} \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!]} [\![\Phi]\!] \otimes [\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] \xrightarrow{[\![M]\!] \otimes [\![N]\!]} T([\![A]\!] \multimap T[\![B]\!])) \otimes T[\![A]\!] \\ &\xrightarrow{t} T(T([\![A]\!] \multimap T[\![B]\!]) \otimes [\![A]\!]) \xrightarrow{Ts} TT(([\![A]\!] \multimap T[\![B]\!]) \otimes [\![A]\!]) \\ &\xrightarrow{\mu} T(([\![A]\!] \multimap T[\![B]\!]) \otimes [\![A]\!]) \xrightarrow{T\epsilon} TT[\![B]\!] \xrightarrow{\mu} T[\![B]\!].\end{aligned}$$

- Case
$$\frac{\Phi \vdash_\alpha M : A}{\Phi \vdash_1 \mathsf{lift}\, M : !_\alpha A.}$$
By induction hypothesis, we have $[\![M]\!] : [\![\Phi]\!] = pX \to \alpha[\![A]\!]$ for some $X \in \mathcal{V}$. By the $\mathcal{V}$-adjunction $p \dashv \flat$, we have $\delta[\![M]\!] : X \to \flat\alpha[\![A]\!]$. So we define $[\![\mathsf{lift}\, M]\!] := p\delta[\![M]\!] : pX \to p\flat\alpha[\![A]\!]$.
- Case
$$\frac{\Gamma \vdash_\beta M : !_\alpha A}{\Gamma \vdash_{\alpha \& \beta} \mathsf{force}\, M : A.}$$

We only consider the case where $\alpha = \beta = 0$, the other cases are similar. By induction hypothesis, we have a map $[\![M]\!] : [\![\Gamma]\!] \to Tp\flat T[\![A]\!]$. Since there is a $\mathcal{V}$-natural transformation $\mathsf{force} : p\flat T[\![A]\!] \to T[\![A]\!]$, we define $[\![\mathsf{force} M]\!]$ by

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} Tp\flat T[\![A]\!] \xrightarrow{T\mathsf{force}} TT[\![A]\!] \xrightarrow{\mu} T[\![A]\!].$$

- Case

$$\frac{\Gamma \vdash_\alpha M : !_1(S \multimap_1 U)}{\Gamma \vdash_\alpha \mathsf{box}\ S\ M : \mathbf{Circ}(S, U).}$$

Here we only consider the case $\alpha = 1$. By induction hypothesis, we have $[\![M]\!] : [\![\Gamma]\!] \to p\flat([\![S]\!] \multimap [\![U]\!])$. We define $[\![\mathsf{box}\, SM]\!]$ by

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} p\flat([\![S]\!] \multimap [\![U]\!]) \xrightarrow{\mathsf{box}} p\mathbf{A}([\![S]\!], [\![U]\!]).$$

- Case

$$\frac{\Phi, \Gamma_1 \vdash_\alpha M : \mathbf{Circ}(S, U) \quad \Phi, \Gamma_2 \vdash_\beta N : S}{\Phi, \Gamma_1, \Gamma_2 \vdash_{\alpha \& \beta} \mathsf{apply}(M, N) : U}\ \mathit{apply}$$

Here we only consider the case $\alpha = \beta = 0$. By induction hypothesis, we have $[\![M]\!] : [\![\Gamma_1]\!] \to Tp\mathbf{A}([\![S]\!], [\![U]\!])$ and $[\![N]\!] : [\![\Gamma_2]\!] \to T[\![S]\!]$. Thus we define $[\![\mathsf{apply}(M, N)]\!]$ by

$$\begin{aligned}
[\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] &\xrightarrow{\mathsf{dup} \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!]} [\![\Phi]\!] \otimes [\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] \\
&\xrightarrow{[\![M]\!] \otimes [\![N]\!]} Tp\mathbf{A}([\![S]\!], [\![U]\!]) \otimes T[\![S]\!] \\
&\xrightarrow{t} T(Tp\mathbf{A}([\![S]\!], [\![U]\!]) \otimes [\![S]\!]) \\
&\xrightarrow{Ts} TT(p\mathbf{A}([\![S]\!], [\![U]\!]) \otimes [\![S]\!]) \\
&\xrightarrow{\mu} T(p\mathbf{A}([\![S]\!], [\![U]\!]) \otimes [\![S]\!]) \\
&\xrightarrow{T((\mathsf{force} \circ \mathsf{unbox}) \otimes [\![S]\!])} T(([\![S]\!] \multimap [\![U]\!]) \otimes [\![S]\!]) \\
&\xrightarrow{T\epsilon} T[\![U]\!].
\end{aligned}$$

Our interpretation of the typing rules satisfies the usual semantics substitution theorem. The details of the proof are in [Fu et al. 2022b, Appendix D].

Theorem 3.7 (Substitution). *If* $\Phi, \Gamma_1, x : A, \Gamma_2 \vdash_\alpha M : B$ *and* $\Phi, \Gamma_3 \vdash_1 V : A$, *then*

$$[\![[V/x]M]\!] = [\![M]\!] \circ ([\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![V]\!] \otimes [\![\Gamma_2]\!]) \circ (\mathsf{dup} \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] \otimes [\![\Gamma_3]\!]) : [\![\Phi, \Gamma_1, \Gamma_2, \Gamma_3]\!] \to \alpha[\![B]\!].$$

The next two theorems show that values of parameter type are in the image of functor p, and that values of simple types are isomorphisms.

Theorem 3.8. *If* $\Phi \vdash_1 V : P$, *then* $[\![V]\!] = pf : pX \to pY$ *for some* $f : X \to Y \in \mathcal{V}$ *such that* $[\![\Phi]\!] = pX, [\![P]\!] = pY$.

Theorem 3.9. *Suppose* $\Sigma \vdash_1 V : S$, *then* $[\![V]\!] : [\![\Sigma]\!] \to [\![S]\!]$ *is an isomorphism in* $\mathbf{A}$.

Since the embedding $\psi : \mathbf{M} \hookrightarrow V(\mathbf{A})$ is fully faithful, $[\![V]\!] : [\![\Sigma]\!] \to [\![S]\!]$ is also an isomorphism in $\mathbf{M}$.

4 OPERATIONAL SEMANTICS AND SOUNDNESS

In this section, we will specify an operational semantics for Proto-Quipper-Dyn and show that it is sound with respect to the $\mathcal{V}$-enriched categorical model $\mathbf{A}$ for dynamic lifting.

We distinguish two kinds of evaluation in order to model Quipper's two runtimes. The evaluation rules for circuit generation time will work with morphisms in $\mathbf{M}$, i.e., quantum circuits. On the other hand, the evaluation rules for circuit execution time will work with morphisms in $\mathbf{Q}$, i.e., quantum operations. Because of the embeddings $\psi : \mathbf{M} \hookrightarrow V(\mathbf{A})$ and $\phi : \mathbf{Q} \hookrightarrow Kl_{VT}(V(\mathbf{A}))$, we are able to interpret the configurations for these two runtimes as maps in the $\mathcal{V}$-enriched category $\mathbf{A}$.

4.1 Operational Semantics for Circuit Generation Time

First of all, we specify the meaning of appending circuits in the category $\mathbf{M}$.

Definition 4.1 (Circuit append). Suppose $C : \Sigma \to \Sigma_1, \Sigma_2$ and $\mathcal{D} : \Sigma_1' \to \Sigma_3$ are morphisms in $\mathbf{M}$ and there are typing judgments $\Sigma_1 \vdash_1 V : S$ and $\Sigma_1' \vdash_1 V' : S$. We define $\text{append}(\mathcal{D}, C, V', V)$ to be the following morphism in $\mathbf{M}$.

$$((\mathcal{D} \circ [\![V']\!]^{-1} \circ [\![V]\!]) \otimes [\![\Sigma_2]\!]) \circ C : \Sigma \to \Sigma_3, \Sigma_2$$

Thus $\text{append}(\mathcal{D}, C, V', V)$ is the result of appending the circuit $\mathcal{D}$ to C by connecting the interfaces V' and V. The following are evaluation rules for circuit generation time, where the underlying states are given by morphisms in $\mathbf{M}$.

Definition 4.2 (Circuit generation time evaluation).

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

$$\frac{(C_1, M) \Downarrow (C_2, (a, \mathcal{D}, b)) \quad (C_2, N) \Downarrow (C_3, V) \quad \text{append}(\mathcal{D}, C_3, a, V) = C'}{(C_1, \mathsf{apply}(M, N)) \Downarrow (C', b)}\ \textit{apply}$$

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

$$\frac{(C, M) \Downarrow (C', \mathsf{lift}\ M') \quad \text{gen}(S) = a \quad (\mathrm{Id}_S, M'\ a) \Downarrow (\mathcal{D}, b)}{(C, \mathsf{box}\ S\ M) \Downarrow (C', (a, \mathcal{D}, b))}\ \textit{box}$$

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

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

In the rule *box*, we use $\text{gen}(S) = a$ to mean that the a is a fresh simple term of type S. Note that the evaluation of $(C, M) \Downarrow (C', V)$ does not account for dynamic lifting, and the underlying states are circuits. So it is the same set of evaluation rules as in [Rios and Selinger 2018]. The evaluation comes with the following notion of configuration.

Definition 4.3 (Well-typed circuit configuration). We write $\Sigma \vdash (C, M) : A; \Sigma'$ to mean there exists Σ'' such that $C : \Sigma \to \Sigma', \Sigma''$ and $\Sigma'' \vdash_1 M : A$.

A well-typed circuit configuration requires a typed term with modality 1, i.e., $\vdash_1 M : A$. It is a runtime error if a term with dynamic lifting is encountered when using the evaluation rules in Definition 4.2. Our type system and the following type preservation theorem ensures that this can not happen.

Theorem 4.4. *If $\Sigma \vdash (C, M) : A; \Sigma'$ and $(C, M) \Downarrow (C', V)$, then $\Sigma \vdash (C', V) : A; \Sigma'$.*

In the following we define the interpretation $[\![C, M]\!]$ as a map in the $\mathcal{V}$-category $\mathbf{A}$.

Definition 4.5. Suppose $\Sigma \vdash (C, M) : A; \Sigma'$. We have maps $\psi C : [\![\Sigma]\!] \to [\![\Sigma']\!] \otimes [\![\Sigma'']\!]$ and $[\![M]\!] : [\![\Sigma'']\!] \to [\![A]\!]$ in $\mathbf{A}$. We define $[\![C, M]\!]$ as follows:

$$[\![\Sigma]\!] \xrightarrow{\psi C} [\![\Sigma']\!] \otimes [\![\Sigma'']\!] \xrightarrow{[\![\Sigma']\!] \otimes [\![M]\!]} [\![\Sigma']\!] \otimes [\![A]\!].$$

The following theorem shows that the evaluation rules for circuit generation time are sound with respect to the categorical model $\mathbf{A}$. Since in this case dynamic lifting cannot occur, the proof is similar to the one in [Rios and Selinger 2018].

Theorem 4.6. *If* $\Sigma \vdash (C, M) : A; \Sigma'$ *and* $(C, M) \Downarrow (C', V)$, *then* $[\![C, M]\!] = [\![C', V]\!]$.

4.2 Operational Semantics for Circuit Execution Time

Since dynamic lifting requires the ability to access the states in $\mathbf{Q}$, we first define the concepts of *state* and *addresses.*

Definition 4.7 (State and addresses). For any object $S \in \mathbf{Q}$, a *state* is a morphism $Q : I \to S \in \mathbf{Q}$. We write $\mathrm{addr}(Q) = \Sigma$ if $\phi(S) = [\![\Sigma]\!]$, we call Σ the *addresses* of Q. (Recall that we have, for convenience and without loss of generality, assumed that the interpretation function $[\![-]\!]$ is one-to-one on label contexts).

We often write $Q : I \to \Sigma \in \mathbf{Q}$ for $Q : I \to S$, where $\phi(S) = [\![\Sigma]\!]$. The following *read* operation will be used to define the operational semantics for dynamic lifting.

Definition 4.8 (Read operation). Suppose $\mathrm{addr}(Q) = \Sigma, \ell : \mathbf{Bit}$ and $Q = p_1(Q_1 \otimes \mathrm{inj}_1) + p_2(Q_2 \otimes \mathrm{inj}_2)$, where $\mathrm{addr}(Q_1) = \mathrm{addr}(Q_2) = \Sigma$ and $p_1, p_2 \in [0, 1]$ and $p_1 + p_2 = 1$. We define a formal sum $\mathrm{read}(Q, \ell) = p_1(Q_1, \mathsf{False}) + p_2(Q_2, \mathsf{True})$, where $\mathsf{False}, \mathsf{True} : \mathbf{Bool}$.

Note that by the last condition in Assumption 2.5, we know that $Q = p_1(Q_1 \otimes \mathrm{inj}_1) + p_2(Q_2 \otimes \mathrm{inj}_2) : I \to \Sigma \otimes \mathbf{Bit}$ for some essentially uniquely determined $Q_1, Q_2 : I \to \Sigma$, and $p_1, p_2 \in [0, 1]$ such that $p_1 + p_2 = 1$. The only time Q_i is not uniquely determined is when $p_i = 0$, but in this case, it will turn out that the Q_i does not matter since it corresponds to a branch of computation taken with probability zero. In this case, we can just make some fixed but arbitrary choice for Q_i. So the read operation makes the information of the probabilities p_1, p_2 and the states Q_1, Q_2 available.

In the following, we define the circuit execution time counterpart of Definition 4.1. It specifies the meaning of updating a quantum state by applying a quantum circuit, where the identity-on-object interpretation functor $J : \mathbf{M} \to \mathbf{Q}$ is needed for the definition.

Definition 4.9. Suppose $Q : I \to \Sigma_1, \Sigma_2$ is a morphism in $\mathbf{Q}$, and $C : \Sigma_1' \to \Sigma_3$ is a morphism in $\mathbf{M}$, and there are typing judgements $\Sigma_1 \vdash V : S$ and $\Sigma_1' \vdash V' : S$. We define $\mathrm{operate}(C, Q, V', V)$ to be the following map in $\mathbf{Q}$.

$$(J(C \circ [\![V']\!]^{-1} \circ [\![V]\!]) \otimes [\![\Sigma_2]\!]) \circ Q : I \to \Sigma_3, \Sigma_2$$

We now we define the operational semantics for circuit execution time. The underlying states of the evaluation are the states in $\mathbf{Q}$. The evaluation is of the form $(Q, M) \Downarrow \sum_{i \in [n]} p_i(Q_i, V_i)$. Its intuitive meaning is that the configuration (Q, M) can be reduced to (Q_i, V_i) with probability p_i. The notation $\sum_{i \in [n]} p_i(Q_i, V_i)$ is a short hand for the formal sum $p_1(Q_1, V_1) + \ldots + p_n(Q_n, V_n)$, and we assume $\sum_{i \in [n]} p_i = 1$. We write $[n] = \{1, \ldots, n\}$.

Definition 4.10 (Operational semantics for circuit execution time).

$$\frac{\begin{array}{c}(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, \lambda x.M'_i)\\ (Q_i,N) \Downarrow \sum_{j\in[m]} q_{i,j}(Q'_{i,j}, V_{i,j})\\ (Q'_{i,j}, [V_{i,j}/x]M'_i) \Downarrow \sum_{k\in[l]} s_{i,j,k}(Q''_{i,j,k}, V'_{i,j,k})\end{array}}{(Q, MN) \Downarrow \sum_{(i,j,k)\in[n]\times[m]\times[l]} p_i q_{i,j} s_{i,j,k}(Q''_{i,j,k}, V'_{i,j,k})} \qquad \frac{\begin{array}{c}(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, \mathsf{lift}\, M'_i)\\ (Q_i,M'_i) \Downarrow \sum_{j\in[m]} q_{i,j}(Q'_{i,j}, V_{i,j})\end{array}}{(Q, \mathsf{force} M) \Downarrow \sum_{(i,j)\in[n]\times[m]} p_i q_{i,j}(Q'_{i,j}, V_{i,j})}$$

$$\frac{\begin{array}{c}(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, (a_i, \mathcal{D}_i, b_i))\\ (Q_i,N) \Downarrow \sum_{j\in[m]} q_{i,j}(Q'_{i,j}, V_{i,j})\\ \mathrm{operate}(\mathcal{D}_i, Q'_{i,j}, a_i, V_{i,j}) = Q''_{i,j}\end{array}}{(Q, \mathsf{apply}(M,N)) \Downarrow \sum_{(i,j)\in[n]\times[m]} p_i q_{i,j}(Q''_{i,j}, b_i)}\ \mathit{apply} \qquad \frac{\begin{array}{c}(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, \mathsf{lift}\, M'_i)\\ \mathrm{gen}(S) = a\\ (\mathrm{Id}_S, M'_i\ a) \Downarrow (\mathcal{D}_i, b_i)\end{array}}{(Q, \mathsf{box}\ S\ M) \Downarrow \sum_{i\in[n]} p_i(Q_i, (a, \mathcal{D}_i, b_i))}\ \mathit{box}$$

$$\frac{\begin{array}{c}(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, \ell_i)\\ \mathrm{read}(Q_i, \ell_i) = q_{i,1}(Q'_{i,1}, a_{i,1}) + q_{i,2}(Q'_{i,2}, a_{i,2})\end{array}}{(Q, \mathsf{dynlift}\, M) \Downarrow \sum_{(i,j)\in[n]\times[2]} p_i q_{i,j}(Q'_{i,j}, a_{i,j})}\ \mathit{dynlift}$$

$$\frac{(Q,N) \Downarrow \sum_{i\in[n]} p_i(Q'_i, (V_i, V'_i)) \quad (Q'_i, [V_i/x, V'_i/y]M) \Downarrow \sum_{j\in m}(Q''_{i,j}, V''_{i,j})}{(Q, \mathsf{let}\ (x,y) = N\ \mathsf{in}\ M) \Downarrow \sum_{(i,j)\in[n]\times[m]}(Q''_{i,j}, V''_{i,j})}$$

$$\frac{\begin{array}{c}(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, V_i)\\ (Q_i,N) \Downarrow \sum_{j\in[m]} q_{i,j}(Q'_{i,j}, V'_{i,j})\end{array}}{(Q, (M,N)) \Downarrow \sum_{i,j\in[n]\times[m]}(Q'_{i,j}, (V_i, V'_{i,j}))}$$

In the *apply* rule, we use *operate* instead of *append*, which allows a quantum circuit to be applied as a quantum operation. In the *dynlift* rule, for each (Q_i, ℓ_i), we apply the operation $\mathrm{read}(Q_i, \ell_i)$, which gives rise to two possible outcomes $(Q'_{i,1}, a_{i,1})$, $(Q'_{i,2}, a_{i,2})$ with probabilities $q_{i,1}, q_{i,2}$, where $a_{i,1}, a_{i,2}$: **Bool** and $a_{i,1} \neq a_{i,2}$. This is the only rule that gives rise to probabilistic results in the evaluation. In the *box* rule, the evaluation of $(\mathrm{Id}_S, M'_i a)$ uses the rules defined in Definition 4.2, so it is performed at circuit generation time.

We now define a well-typed configuration for evaluating a term under a quantum state.

Definition 4.11 (Well-typed configuration). We write $\vdash_\alpha (Q,M) : A; \Sigma'$ to mean there exists Σ'' such that $\Sigma'' \vdash_\alpha M : A$, and $\mathrm{addr}(Q) = \Sigma'', \Sigma'$.

Since the evaluation rules in Definition 4.10 account for dynamic lifting, the above configuration allows the term M to have modality 0. The operational semantics defined in Definition 4.10 is type-safe in the following sense.

Theorem 4.12. *If* $\vdash_\alpha (Q,M) : A; \Sigma'$ *and* $(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, V_i)$, *then* $\vdash_1 (Q_i, V_i) : A; \Sigma'$ *for all* $i \in [n]$.

Theorem 4.13. *If* $\vdash_1 (Q,M) : A; \Sigma'$ *and* $(Q,M) \Downarrow \sum_{i\in[n]} p_i(Q_i, V_i)$, *then* $n = 1$. *In other words, we actually have* $(Q,M) \Downarrow (Q', V)$.

In the following we interpret a well-typed configuration $\vdash_\alpha (Q,M) : A; \Sigma'$ as a map in the Kleisli category $Kl_T(\mathbf{A})$.

Definition 4.14. Suppose $\vdash_\alpha (Q,M) : A; \Sigma'$. We have $\phi Q : I \to T(\llbracket \Sigma'_1 \rrbracket \otimes \llbracket \Sigma'' \rrbracket \otimes \llbracket \Sigma'_2 \rrbracket)$ and $\llbracket M \rrbracket : \llbracket \Sigma'' \rrbracket \to \alpha \llbracket A \rrbracket$ in $\mathbf{A}$. We define $\llbracket Q, M \rrbracket$ by:

- If $\alpha = 1$, then

$$I \xrightarrow{\phi Q} T(\llbracket \Sigma'_1 \rrbracket \otimes \llbracket \Sigma'' \rrbracket \otimes \llbracket \Sigma'_2 \rrbracket) \xrightarrow{T(\llbracket \Sigma'_1 \rrbracket \otimes \llbracket M \rrbracket \otimes \llbracket \Sigma'_2 \rrbracket)} T(\llbracket \Sigma'_1 \rrbracket \otimes \llbracket A \rrbracket \otimes \llbracket \Sigma'_2 \rrbracket).$$

- If $\alpha = 0$, then

$$I \xrightarrow{\phi Q} T(\llbracket \Sigma_1' \rrbracket \otimes \llbracket \Sigma'' \rrbracket \otimes \llbracket \Sigma_2' \rrbracket) \xrightarrow{T(\llbracket \Sigma_1' \rrbracket \otimes \llbracket M \rrbracket \otimes \llbracket \Sigma_2' \rrbracket)} T(\llbracket \Sigma_1' \rrbracket \otimes T\llbracket A \rrbracket \otimes \llbracket \Sigma_2' \rrbracket)$$

$$\xrightarrow{T(t \otimes \llbracket \Sigma_2' \rrbracket)} T(T(\llbracket \Sigma_1' \rrbracket \otimes \llbracket A \rrbracket) \otimes \llbracket \Sigma_2' \rrbracket) \xrightarrow{Ts} TT(\llbracket \Sigma_1' \rrbracket \otimes \llbracket A \rrbracket \otimes \llbracket \Sigma_2' \rrbracket) \xrightarrow{\mu} T(\llbracket \Sigma_1' \rrbracket \otimes \llbracket A \rrbracket \otimes \llbracket \Sigma_2' \rrbracket).$$

The following theorem shows that the operational semantics in Definition 4.10 is sound with respect to the semantic model **A**.

Theorem 4.15 (Soundness). *If* $\vdash_\alpha (Q, M) : A; \Sigma'$, *and* $(Q, M) \Downarrow \sum_{i \in [n]} p_i(Q_i, V_i)$, *then*

$$\llbracket Q, M \rrbracket = \sum_{i \in [n]} p_i \llbracket Q_i, V_i \rrbracket : I \to T(\llbracket A \rrbracket \otimes \llbracket \Sigma' \rrbracket).$$

Proof sketch. The proof is by induction on the evaluation rules. Here we focus on the case for dynamic lifting. Please see [Fu et al. 2022b, Appendix E] for the proofs of the other cases.

Suppose $\mathrm{addr}(Q) = \Sigma'', \Sigma'$, and $\Sigma'' \vdash_0 M : A$, and

$$\frac{\Sigma'' \vdash_1 M : \mathbf{Bit}}{\Sigma'' \vdash_0 \mathsf{dynlift}\, M : \mathbf{Bool}.}$$

Consider the following.

$$\frac{(Q, M) \Downarrow (Q', \ell) \qquad \mathsf{read}(Q', \ell) = q_1(Q_1', \mathsf{False}) + q_2(Q_2', \mathsf{True})}{(Q, \mathsf{dynlift}\, M) \Downarrow q_1(Q_1', \mathsf{False}) + q_2(Q_2', \mathsf{True})}$$

Since $\mathsf{read}(Q', \ell) = q_1(Q_1', \mathsf{False}) + q_2(Q_2', \mathsf{True})$ implies that $Q' = q_1(Q_1' \otimes \mathrm{inj}_1) + q_2(Q_2' \otimes \mathrm{inj}_2)$ in **Q**, we have the following in **A**.

$$\phi Q' = q_1(\mu \circ Tt \circ s \circ (\phi Q_1' \otimes \phi(\mathrm{inj}_1))) + q_2(\mu \circ Tt \circ s \circ (\phi Q_2' \otimes \phi(\mathrm{inj}_2))),$$

where $\phi Q' : I \to T(\mathbf{Bit} \otimes \llbracket \Sigma' \rrbracket)$, and $\phi Q_1', \phi Q_2' : I \to T\llbracket \Sigma' \rrbracket$, and $\phi(\mathrm{inj}_1), \phi(\mathrm{inj}_2) : I \to T\mathbf{Bit}$. Note that by condition (g), we have $\phi(\mathrm{inj}_1) = \eta \circ \mathsf{init} \circ \llbracket \mathsf{False} \rrbracket$ and $\phi(\mathrm{inj}_2) = \eta \circ \mathsf{init} \circ \llbracket \mathsf{True} \rrbracket$. We need to show that

$$\llbracket Q, \mathsf{dynlift}\, M \rrbracket = q_1(T(\llbracket \mathsf{False} \rrbracket \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q_1') + q_2(T(\llbracket \mathsf{True} \rrbracket \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q_2').$$

By induction hypothesis, we have $\llbracket Q, M \rrbracket = \llbracket Q', \ell \rrbracket$, i.e., $T(\llbracket M \rrbracket \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q = \phi Q'$. Thus

$$\begin{aligned}
\llbracket Q, \mathsf{dynlift}\, M \rrbracket &= \mu \circ Ts \circ T((\mathsf{dynlift} \circ \llbracket M \rrbracket) \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q \\
&= \mu \circ Ts \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \circ T(\llbracket M \rrbracket \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q \\
&= \mu \circ Ts \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q' \\
&= \mu \circ Ts \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \\
&\circ (q_1(\mu \circ Tt \circ s \circ (\phi Q_1' \otimes \phi(\mathrm{inj}_1))) + q_2(\mu \circ Tt \circ s \circ (\phi Q_2' \otimes \phi(\mathrm{inj}_2)))) \\
&= q_1(\mu \circ Ts \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \circ \mu \circ Tt \circ s \circ (\phi Q_1' \otimes \phi(\mathrm{inj}_1))) \\
&+ q_2(\mu \circ T\sigma \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \circ \mu \circ Tt \circ \sigma \circ (\phi Q_2' \otimes \phi(\mathrm{inj}_2))).
\end{aligned}$$

We just need to show

$$\begin{aligned}
T(\llbracket \mathsf{False} \rrbracket \otimes \llbracket \Sigma' \rrbracket) \circ \phi Q_1' &= \mu \circ Ts \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \circ \mu \circ Tt \circ s \circ (\phi Q_1' \otimes \phi(\mathrm{inj}_1)) \\
&= \mu \circ Ts \circ T(\mathsf{dynlift} \otimes \llbracket \Sigma' \rrbracket) \circ \mu \circ Tt \circ s \circ (\phi Q_1' \otimes (\eta \circ \mathsf{init} \circ \llbracket \mathsf{False} \rrbracket))
\end{aligned}$$

This is true because of the commutative diagram in Figure 3. □

$$\begin{tikzcd}
I \arrow[d, "\phi Q'_1"] & & & \\
T[\![\Sigma']\!] \arrow[d, "\lambda^{-1}"] \arrow[dr, "T\lambda^{-1}"] & & & \\
I \otimes T[\![\Sigma']\!] \arrow[r, "t"] \arrow[d, "{[\![\mathsf{False}]\!] \otimes T[\![\Sigma']\!]}"] & T(I \otimes [\![\Sigma']\!]) \arrow[d, "{T([\![\mathsf{False}]\!] \otimes [\![\Sigma']\!])}"] & & \\
\mathbf{Bool} \otimes T[\![\Sigma']\!] \arrow[r, "t"] \arrow[d, "\mathsf{init} \otimes T[\![\Sigma']\!]"] & T(\mathbf{Bool} \otimes [\![\Sigma']\!]) \arrow[rr, bend left, "\mathrm{Id}"] \arrow[rrd, bend left, "\eta"] \arrow[rd, "T(\eta \otimes [\![\Sigma']\!])"] \arrow[d, "T(\mathsf{init} \otimes [\![\Sigma']\!])"] & & T(\mathbf{Bool} \otimes [\![\Sigma']\!]) \\
\mathbf{Bit} \otimes T[\![\Sigma']\!] \arrow[r, "t"] \arrow[d, "\eta \otimes T[\![\Sigma']\!]"'] & T(\mathbf{Bit} \otimes [\![\Sigma']\!]) \arrow[r, "T(\mathsf{dynlift} \otimes [\![\Sigma']\!])"'] \arrow[ldd, "T(\eta \otimes [\![\Sigma']\!])"'] \arrow[lddd, bend left, "\eta"] & T(T\mathbf{Bool} \otimes [\![\Sigma']\!]) \arrow[r, "Ts"] & TT(\mathbf{Bool} \otimes [\![\Sigma']\!]) \arrow[u, "\mu"] \\
T\mathbf{Bit} \otimes T[\![\Sigma']\!] \arrow[d, "t"] & & & \\
T(T\mathbf{Bit} \otimes [\![\Sigma']\!]) \arrow[d, "Ts"] & & & \\
TT(\mathbf{Bit} \otimes [\![\Sigma']\!]) \arrow[r, "\mu"] & T(\mathbf{Bit} \otimes [\![\Sigma']\!]) \arrow[uuu, "\mathrm{Id}"] & &
\end{tikzcd}$$

Fig. 3. A commutative diagram from the proof of Theorem 4.15

Remark. In practice, a closed term M is always evaluated with the initial configuration (Id_I, M), where $\mathrm{Id}_I : I \longrightarrow I$ is a state in $\mathbf{Q}$. When M has modality 0, we would need access to a quantum computer/simulator in order to evaluate (Id_I, M) and each run of (Id_I, M) could give a different value. When M has modality 1, the evaluation of (Id_I, M) is deterministic, i.e., the top-level quantum state is updated in a deterministic fashion. In this case, instead of performing the quantum operations, we could also just generate a list of gates, which can be done entirely in a classical computer.

5 DYNAMIC LIFTING IN PROTO-QUIPPER

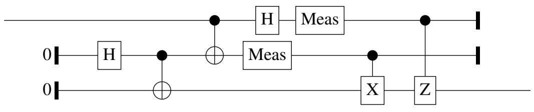
While typing judgments and certain types ($A \multimap_\alpha B$ and $!_\alpha A$) are annotated with a modality, information about this modality is meant to be hidden from the programmer unless an error occurs. For example, if one attempt to box a function which uses dynamic lifting, the type checker will raise a modality error. As a result, such programming errors are caught at compile time in Proto-Quipper-Dyn, whereas they are only caught at runtime in Quipper.

The modality inference can readily be integrated into bi-directional type checking, which uses a pair of recursively defined functions for type checking and type inference [Pierce and Turner 2000]. To work with modalities, the type checking function not only takes a term and a type as inputs, but also the current modality of the typing judgment. For example, when checking a term $\lambda x.M$ against a type $A \multimap_\alpha B$ with current modality β, the type checking function first ensures that the current modality β is 1, then extends the current typing environment with $x : A$ and recursively checks the term M against the type B, with the modality α. The type inference function takes a term as input and outputs the inferred type as well as the inferred modality. For example, when inferring the type for a term MN, the type inference function first infers a type $A \multimap_\alpha B$ and a modality β for M, and then it infers a type A and a modality γ for the term N, so the type inference function will return the type B and the inferred modality $\alpha \,\&\, \beta \,\&\, \gamma$.

We now discuss several Proto-Quipper-Dyn programs that make use of dynamic lifting. An experimental implementation of Proto-Quipper-Dyn is available from https://gitlab.com/frank-peng-fu/dpq-remake, and the programs in Listings 1–6 have been tested with that implementation. Note that several of the following example programs make use of recursion. While we do not formally treat recursion in this paper, it is included in the prototype implementation.

5.1 Quantum Teleportation

The following circuit implements a one-qubit quantum teleportation protocol.



This circuit is generated by the following Proto-Quipper-Dyn programs.

```
alice1 : !(Qubit -> Qubit -> Bit *
    Bit)
alice1 a q =
  let (a, q) = CNot a q
      q = H q
  in (Meas a, Meas q)

bob1 : !(Qubit -> Bit -> Bit -> Qubit
    )
bob1 q x y =
  let (q, x) = C_X q x
      (q, y) = C_Z q y
      _ = Discard x
      _ = Discard y
  in q
```

Listing 1. Alice and Bob circuits

```
bell00 : !(Unit -> Qubit * Qubit)
bell00 u =
  let a = Init0 ()
      b = Init0 ()
  in CNot b (H a)

tele1 : !(Qubit -> Qubit)
tele1 q =
  let (b, a) = bell00 ()
      (x, y) = alice1 a q
      z = bob1 b x y
  in z

boxTele : Circ(Qubit, Qubit)
boxTele = box Qubit tele1
```

Listing 2. Teleportation circuit

As can be seen in Listings 1 and 2, the modality information is not visible to the programmer. Because the programs in Listings 1 and 2 do not use dynamic lifting, the modalities in the fully annotated types are all 1. For example, the fully annotated type of tele1 is $!_1(\mathbf{Qubit} \multimap_1 \mathbf{Qubit})$. We can therefore box tele1 into a quantum circuit. The evaluation of boxTele occurs on a classical computer and generates the circuit diagram above.

For comparison, let us consider the following Proto-Quipper-Dyn programs that implement quantum teleportation using dynamic lifting.

```
alice2 : !(Qubit -> Qubit -> Bool *
    Bool)
alice2 a q =
  let (a, q) = CNot a q
      q = H q
  in (dynlift (Meas a), dynlift (Meas
       q))

bob2 : !(Qubit -> Bool -> Bool ->
    Qubit)
bob2 q x y =
  let q = if x then QNot q else q
      q = if y then ZGate q else q
  in q
```

Listing 3. Alice and Bob functions

```
tele2 : !(Qubit -> Qubit)
tele2 q =
  let (b, a) = bell00 ()
      (x, y) = alice2 a q
      z = bob2 b x y
  in z

-- The following will raise an error
boxAttempt : Circ(Qubit, Qubit)
boxAttempt = box Qubit tele2

test : Bool
test =
  dynlift (Meas (tele2 (Init0 ())))
```

Listing 4. Teleportation function

As before, the code in Listings 3 and 4 contains no modality annotations. In the alice2 function, dynamic lifting is used right after the measurement gate Meas : **Qubit** $\to$ **Bit**. Accordingly, the fully annotated type of alice2 is $!_1$(**Qubit** $\multimap_1$ **Qubit** $\multimap_0$ **Bool** $*$ **Bool**). The bob2 function then uses if-then-else expressions to decide whether to apply the gates QNot and ZGate, rather than applying the bit-controlled gates C_X and C_Z, as in the bob1 function in Listing 1.

The tele2 function calls the bob2 function with the booleans provided by the alice2 function. Hence, the tele2 function implicitly uses dynamic lifting. Its fully annotated type is $!_1$(**Qubit** $\multimap_0$ **Qubit**). Because of the modality inference, the type checker will issue a typing error for the boxAttempt function. According to the typing rule for *box*, the box Qubit function requires an argument of type $!_1$(**Qubit** $\multimap_1$ **Qubit**), which is distinct from the type of tele2. This error is sensible because the tele2 function does not correspond to a circuit.

The test function applies tele2 to an input qubit in the $|0\rangle$ state. The output value of test should then be False with probability 1. Note that the evaluation of test requires access to a quantum computer or a simulator.

5.2 Magic State Distillation

Magic states are quantum states that can be used, in conjunction with Clifford gates, to perform universal quantum computing fault tolerantly [Bravyi and Kitaev 2005]. For example, there is a standard method to implement a T gate using the magic state $(|0\rangle + e^{\frac{\pi i}{4}}|1\rangle)/\sqrt{2}$, along with Clifford gates and measurements. This enables the application of any operation from the Clifford+T gate set, a well-known universal set of quantum gates [Nielsen and Chuang 2002].

The process of producing a magic state such as $(|0\rangle + e^{\frac{\pi i}{4}}|1\rangle)/\sqrt{2}$ from several imperfect states is called *magic state distillation* [Bravyi and Kitaev 2005]. In order to distill a magic state $|M\rangle$, one first prepares several qubits in a state that approximates $|M\rangle$ up to an error rate ϵ. A carefully designed quantum circuit is then applied to these qubits and some of them are measured. If all of the measurement results are 0, then the remaining qubits are guaranteed to be in a state that approximates $|M\rangle$ up to an improved error rate $\epsilon' < \epsilon$. If any one of the measurement results is 1, then all of the qubits are discarded and the entire process is restarted. In practice, several rounds of distillation are required to obtain a state that approximates $|M\rangle$ up to an acceptable error rate.

A Proto-Quipper-Dyn implementation of Bravyi and Kitaev's distillation algorithm is given in Listing 5. In the distill function, we first apply a five-qubit error correction circuit fiveQubits to the inputs, then measure the qubits and, through dynamic lifting, promote the resulting bits to

```
distill : ! (Qubit * Qubit * Qubit *  Qubit * Qubit -> Maybe Qubit)
distill input =
  let (a1, a2, a3, a4, a5) = fiveQubits input
      a1' = dynlift (Meas a1)
      a2' = dynlift (Meas a2)
      a3' = dynlift (Meas a3)
      a4' = dynlift (Meas a4)
  in if a1' || a2' || a3' || a4'
     then let a = dynlift (Meas a5) in Nothing
     else Just a5

distillation : ! (Nat -> Qubit)
distillation n =
  case n of
    Z -> prepMixedState ()
    S n' ->
     let q1 = distillation n'
         q2 = distillation n'
         q3 = distillation n'
         q4 = distillation n'
         q5 = distillation n'
     in
     case distill (q1, q2, q3, q4, q5) of
       Nothing -> distillation n
       Just q -> q
```

Listing 5. Bravyi and Kitaev's algorithm

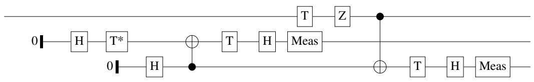
booleans. If all of the booleans are False, the distillation was successful and we return the remaining qubit. Otherwise the distillation failed, so we discard the unmeasured qubit and return nothing. Dynamic lifting is essential for defining the distill function because in this case the if-then-else expression cannot be implemented as a circuit.

The distillation function performs n rounds of magic state distillation. The function prepMixed-State prepares an initial imperfect state. The distillation function is a recursive function that assumes five successful distillations from the previous round and then applies the distill function to the resulting qubits. If that function returns a qubit, the n-th round of distillation was successful, otherwise it will restart the whole process.

5.3 Repeat-Until-Success

The *repeat-until-success* paradigm provides a technique to apply a unitary that cannot be implemented exactly, at the cost of potentially running the same circuit multiple times. In order to apply a non-Clifford+T gate N to a target qubit $|\phi\rangle$, one first initializes several ancillary qubits before applying a well-chosen Clifford+T circuit C to the target and the ancillas and measuring the ancillas. If all of the measurement results are 0, the target qubit is guaranteed to be in the state $N|\phi\rangle$. Otherwise, a correction is applied to the target to return it to its initial state and the process is repeated.

Consider the following circuit used in [Paetznick and Svore 2014] to illustrate the implementation of the gate $V_3 = \frac{I+2iZ}{\sqrt{5}}$ using the repeat-until-success method.



The top wire is the target qubit, while the wires below it are the ancillas. We apply a sequence of gates (H, H, T^*, $CNOT$, T, and H) to the ancillas before measuring the first ancilla. If, as we assume here, the measurement result is 0, then we apply a sequence of gates (T, Z, $CNOT$, T, and H) to the target qubit and the second ancilla before measuring the second ancilla. Assuming, again, that the measurement result is 0, we then know that the target qubit is in the desired state. Note that the circuit above is not a representation of the entire repeat-until-success protocol. Instead, it is the circuit constructed in the event that both measurement results are 0 (which can be shown to occur with probability 5/8). If the measurements yield different results, the circuit constructed by the repeat-until-success protocol is different. For example, if the result of the second measurement is 1, a Z gate must be applied to the target qubit to return it to its initial state.

Listing 6 gives a precise description of the implementation of V_3 in Proto-Quipper-Dyn.

```
v3 : !(Qubit -> Qubit)
v3 q =
  let a1 = tgate_inv (H (Init0 ()))
      a2 = H (Init0 ())
      (a1, a2) = CNot a1 a2
      a1 = H (TGate a1)
  in if dynlift (Meas a1)
     then
       let _ = Discard (Meas a2)
       in v3 q
     else let q = ZGate (TGate q)
              (a2, q) = CNot a2 q
              a2 = H (TGate a2)
          in if dynlift (Meas a2)
             then v3 (ZGate q)
             else q
```

Listing 6. A repeat-until-success example

Once again, dynamic lifting plays an essential role here. Note that the v3 function has type $!_1(\mathbf{Qubit} \multimap_0 \mathbf{Qubit})$; it is a quantum computation, rather than a quantum circuit.

6 CONCLUSION

We have given an axiomatization of an enriched categorical semantics for Proto-Quipper with dynamic lifting. We defined a type system with a modality to keep track of functions that use dynamic lifting. The main benefit of our type system is that it statically ensures that the boxing operation can only be applied to a function that does not use dynamic lifting. We also gave an operational semantics for dynamic lifting. The operational semantics models both circuit generation and circuit execution. We also defined an abstract categorical semantics for this language and proved that the type system and the operational semantics are sound with respect to it. Lastly, we gave some examples of quantum algorithms that rely on dynamic lifting.

There are many things left to be done. One of them is how to combine dynamic lifting with dependent types and/or recursion. At this point, we have a model for dynamic lifting [Fu et al. 2022a], but it does not support dependent types or recursion. We also have a model for Proto-Quipper with dependent types [Fu et al. 2020a], but it does not support dynamic lifting or recursion. Finally, Lindenhovius et al. have a model for Proto-Quipper with recursion [Lindenhovius et al. 2018], but it does not support dynamic lifting or dependent types. We do not think that adding recursion to Proto-Quipper-Dyn would create fundamental difficulties at the level of the syntax, type system, or operational semantics (although languages with recursion are usually better handled by small-step operational semantics rather than the big-step semantics we considered here). On the other hand, finding a concrete denotational semantics gets more complicated the more features are included in the programming language, and how to add dependent types or recursion to a semantics of Proto-Quipper-Dyn is an open problem.

ACKNOWLEDGEMENTS

We thank the referees for their thoughtful comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Air Force Office of Scientific Research under Award No. FA9550-21-1-0041.

REFERENCES

Nick Benton. 1995. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *Proceedings of the 8th Workshop on Computer Science Logic, CSL'94, Selected Papers (Springer Lecture Notes in Computer Science 933).* 121–135.

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

Francis Borceux. 1994. *Handbook of Categorical Algebra, Volume 2: Categories and Structures.* Cambridge University Press.

Sergey Bravyi and Alexei Kitaev. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A* 71, 2 (2005), 022316.

Andrea Colledan and Ugo Dal Lago. 2022. On dynamic lifting and effect typing in circuit description languages (extended version). (2022). Available from arXiv:2202.07636.

Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2020b. 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).* Springer, 153–168. https://doi.org/10.1007/978-3-030-52482-1_9 Also available from arXiv:2005.08396.

Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2022a. A biset-enriched categorical model for Proto-Quipper with dynamic lifting. (April 2022). To appear in *QPL 2022.* Available from arXiv:2204.13039.

Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2022b. Proto-Quipper with dynamic lifting. (2022). Available from arXiv:2204.13041.

Peng Fu, Kohei Kishida, and Peter Selinger. 2020a. 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.* 440–453. https://doi.org/10.1145/3373718.3394765 Also available from arXiv:2004.13472.

Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013a. 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).* Springer, 110–124. https://doi.org/10.1007/978-3-642-38986-3_10 Also available from arXiv:1304.5485.

Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013b. 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)).* 333–342. https://doi.org/10.1145/2499370.2462177 Also available from arXiv:1304.3390.

G. M. Kelly. 1982. *Basic concepts of enriched category theory.* Lecture Notes of the London Mathematical Society, Vol. 64. Cambridge University Press.

Dongho Lee, Valentin Perrelle, Benoît Valiron, and Zhaowei Xu. 2021. Concrete categorical model of a quantum circuit description language with measurement. In *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2021 (LIPIcs, Vol. 213),* Mikolaj Bojanczyk and Chandra Chekuri (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 51:1–51:20. https://doi.org/10.4230/LIPIcs.FSTTCS.2021.51

Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. 2018. 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* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 659–668. https://doi.org/10.1145/3209108.3209196

MetaOCaml 2020. MetaOCaml – an OCaml dialect for multi-stage programming. https://okmij.org/ftp/ML/MetaOCaml.html. Accessed: 2022-10-05.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.

Michael A. Nielsen and Isaac L. Chuang. 2002. *Quantum Computation and Quantum Information*. Cambridge University Press.

Bernhard Ömer. 1998. *A Procedural Formalism for Quantum Computing*. Master's thesis. Department of Theoretical Physics, Technical University of Vienna. http://tph.tuwien.ac.at/~oemer/qcl.html

Adam Paetznick and Krysta M. Svore. 2014. Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. *Quantum Information and Computation* 14, 15–16 (2014), 1277–1301. https://doi.org/10.26421/QIC14.15-16-2 Also available from arXiv:1311.1074.

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (ACM SIGPLAN Notices, Vol. 52)*. ACM, 846–858.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

Mathys Rennela and Sam Staton. 2020. Classical control, quantum circuits and linear logic in enriched category theory. *Logical Methods in Computer Science* 16, 1 (2020), 30:1–24. https://doi.org/10.23638/LMCS-16(1:30)2020

Francisco Rios and Peter Selinger. 2018. 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)*. 164–178. https://doi.org/10.4204/EPTCS.266.11 Also available from arXiv:1706.02630.

Neil J. Ross. 2015. *Algebraic and logical methods in quantum computation*. Ph. D. Dissertation. Dalhousie University, Department of Mathematics and Statistics. Available from arXiv:1510.02198.

Peter Selinger. 2004. Towards a quantum programming language. *Mathematical Structures in Computer Science* 14, 4 (2004), 527–586.

Peter Selinger and Benoît Valiron. 2009. Quantum lambda calculus. In *Semantic Techniques in Quantum Computation*, Simon Gay and Ian Mackie (Eds.). Cambridge University Press, Chapter 4, 135–172.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0 PEPM'97.

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